# Microservices

*with Docker, Flask, and React*

Michael Herman

Powered By

testdriven.io

# Table of Contents

# Microservices with Docker, Flask, and React


```json
{
  "version": "2.2.0",
  "author": "Michael Herman",
  "email": "michael@mherman.org",
  "website": "https://testdriven.io",
  "copyright": "Copyright 2018 Michael Herman. All rights reserved."
}
```


> Have questions? Run into issues? Want feedback on your code? Shoot an email to `michael@mherman.org`. Cheers!

# Changelog

## 2.2

*Overall:*

1. Upgraded to the latest versions of-
    - Python
    - Node and NPM
    - Docker, Docker Compose, and Docker Machine
    - React
    - Swagger UI
2. Added directory structure overviews to the end of each part
3. Corrected grammar, rewrote confusing sections

*Part 1:*

1. Replaced Flask-Script with the Flask CLI tool
2. Added info about using a *.dockerignore* file

*Part 2:*

1. Added the Flask Debug Toolbar extension
2. Updated React Snapshot testing
3. Added multistage Docker build to the React app
4. Replaced the pushstate server with Nginx for the production React Dockerfile
5. Added a Docker data volume for the React app

*Parts 5, 6, and 7:*

1. Added info on how to set up a new Amazon IAM user
2. Updated all AWS images

## 2.1

*Added Part 7*:

1. Refactored the AWS Lambda function
2. Added type checking via PropTypes
3. Introduced a `scores` service
4. Refactored a number of React components

## 2.0

*Overall:*

1. Simplified the overall project structure
2. Added full-text search
3. Upgraded to latest versions of Docker and Docker Compose file version
4. Added lots and lots of screenshots
5. Upgraded to the latest versions of Python and Node
6. Updated the development workflow so that all development work is done within the Docker containers
7. Updated the test script
8. Upgraded to TestCafe v0.18.2 for the e2e tests
9. Upgraded to OpenAPI 3.0 (based on the original Swagger 2.0 specification)

*Client:*

1. Upgraded to React v16
2. Upgraded Bootstrap 3 to 4
3. Added auto-reload to the Docker container to speed up the development process
4. Added client-side React tests with Jest and Enzyme

*Server:*

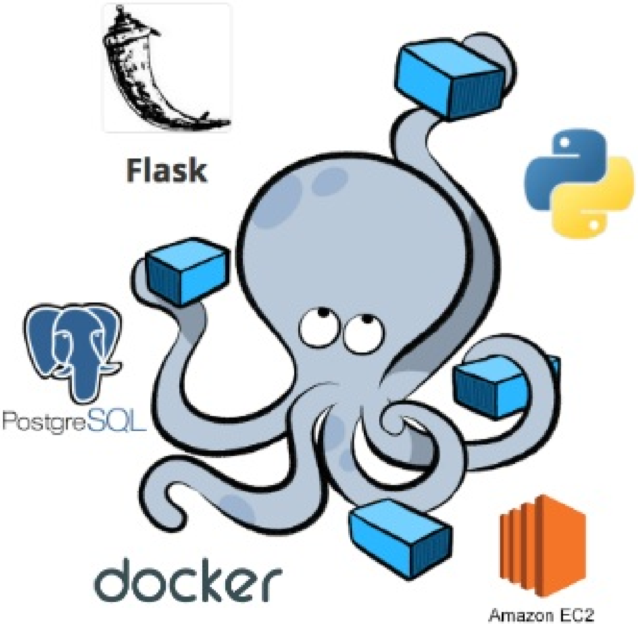
1. Refactored portions of the Flask APIs, adding a `serialize` method to the models
2. Refactored Flask error handlers to clean up the views
3. Added caching with Flask-Cache
4. Mocked `time.sleep` in the test suite

*Orchestration and Deployment:*

1. Revamped Parts 5 and 6
2. Reviewed ECS Service Task Placement Strategy
3. Added an AWS Billing Alarm
4. Added info on using Docker cache to speed up Travis CI builds
5. Added basic IAM and Route 53 setup info

# Part 1

In this first part, you'll learn how to quickly spin up a reproducible development environment with *Docker* to create a *RESTful API* powered by *Python*, *Postgres*, and the *Flask* web framework. After the app is up and running locally, you'll learn how to deploy it to an *Amazon EC2* instance.




## Prerequisites

This is not a beginner course. It's designed for the advanced-beginner - someone with at least six-months of web development experience. Before beginning, you should have some familiarity with the following topics. Refer to the resources for more info:

| Topic | Resource |
| --- | --- |
| Docker | Get started with Docker |
| Docker Compose | Get started with Docker Compose |
| Docker Machine | Docker Machine Overview |
| Flask | Flaskr TDD |

## Objectives

By the end of this part, you will be able to...

1. Develop a RESTful API with Flask
2. Practice test driven development
3. Configure and run services locally with Docker, Docker Compose, and Docker Machine
4. Utilize volumes to mount your code into a container

5. Run unit and integration tests inside a Docker container
6. Enable services running in different containers to talk to one another
7. Work with Python and Flask running inside a Docker Container
8. Install Flask, Ngnix, and Gunicorn on an Amazon EC2 instance

## App

Check out the live apps, running on EC2 -

1. Production
2. Staging

You can also test out the following endpoints...

| Endpoint | HTTP Method | CRUD Method | Result |
|---|---|---|---|
| /users | GET | READ | get all users |
| /users/:id | GET | READ | get single user |
| /users | POST | CREATE | add a user |
| /users/ping | GET | READ | sanity check |

> The `/users` POST endpoint is restricted as of part 3.

Essentially, the app is running in three containers - Flask, Postgres, and Nginx. At the end of this first part, you will have the above app completed and deployed. We'll add authentication and a number of other services in the subsequent parts.

Finished code for part 1: https://github.com/testdrivenio/testdriven-app-2.2/releases/tag/part1

## Dependencies

You will use the following dependencies in part 1:

1. Python v3.6.4
2. Flask v0.12.2
3. Docker v17.12.0-ce
4. Docker Compose v1.18.0
5. Docker Machine v0.13.0
6. Docker Compose file v3.4
7. Flask-SQLAlchemy v2.3.2
8. psycopg2 v2.7.3.2
9. Flask-Testing v0.6.2
10. Gunicorn v19.7.1
11. Nginx v1.13.8
12. Bootstrap 4.0.0

# Microservices

Microservice architecture provides a means of breaking apart large applications into smaller services that interact and communicate with each another. Communication between the services usually happens over a network connection through HTTP calls (request/response). Web sockets, message queues and remote procedure calls (RPC) can also be used to connect standalone components (pub/sub).

Each individual service focuses on a single task, generally separated by business unit, and is governed by its RESTful contact.

The goal of this course is to detail one approach to developing an application in the microservice fashion. It's less about the *why* and more about the *how*. Microservices are hard. They present a number of challenges and issues that are very difficult to solve. Keep this in mind before you start breaking apart your monolith.

## Pros

### Separation of Concerns

With a clear separation between services, developers are free to focus on their own areas of expertise, like languages, frameworks, dependencies, tools, and build pipelines.

For example, a front-end JavaScript engineer could develop the client-facing views without ever having to understand the underlying code in the back-end API. He or she is free to use the languages and frameworks of choice, only having to communicate with the back-end via AJAX requests to consume the RESTful API. Put another way, developers can treat a service like a black box since services communicate via APIs. The actual implementation and complexity are hidden.

That said, it's a good idea to create some organization-wide standards to help ensure each team can work and function together - like code quality and style checking, code reviews, API design.

Clear separation means that errors are mostly localized to the service that the developer is working on. So, you can assign a junior developer to a less critical service so that way if she or he brings down that service, the remainder of the application is not affected.

Less coupling also makes scaling easier since each service can be deployed separately. It also helps to eliminate one team having to wait on another team to finish up work that another team may be dependent on.

### Smaller Code Bases

Smaller code bases tend to be easier to understand since you do not have to grasp the entire system. This, along with the necessity for solid API design, means that applications in a microservice stack are generally faster to develop and easier to test, refactor, and scale.

### Accelerated Feedback Loops

With microservices, developers often own the entire lifecycle of the app, from inception to delivery. Instead of aligning teams with a particular set of technologies - like client ui, server-side, etc. - teams are more product-focused, responsible for delivering the application to the customers themselves. Because of this, they have much more visibility into how the application is being used in the real-world. This speeds up the feedback loop, making it easier to fix bugs and iterate.

## Cons

### Design Complexity

Deciding to split off a piece of your application into a microservice is no easy task. It's often much easier to refactor it into a separate module within the overall monolith rather than splitting it out.

Once you split out a service there is no going back.

### Network Complexity

With a monolith, generally everything happens in a single process so you don't have to make very many calls to other services. As you break out pieces of your application into microservices, you'll find that you'll now have to make a network call when before you could just call a function.

This can cause problems especially if multiple services need to communicate with one another, resulting in ping-pong-like affect in terms of network requests. You will also have to account for a service going down altogether.

### Data Persistence

Most applications have some sort of stateful layer, like databases or task queues. Microservice stacks also need to keep track of where services are deployed and the total number of deployed instances, so that when a new instance of a particular service is stood up, traffic can be re-routed appropriately. This is often referred to as service discovery.

Since we'll be dealing with containers, we need to take special care in how we handle stateful containers since they should not come down.

Isolating a particular service's state so that it is not shared or duplicated is incredible difficult. You'll often have to deal with various sources of truth, which will have to be reconciled frequently. Again, this comes down to design.

### Integration Tests

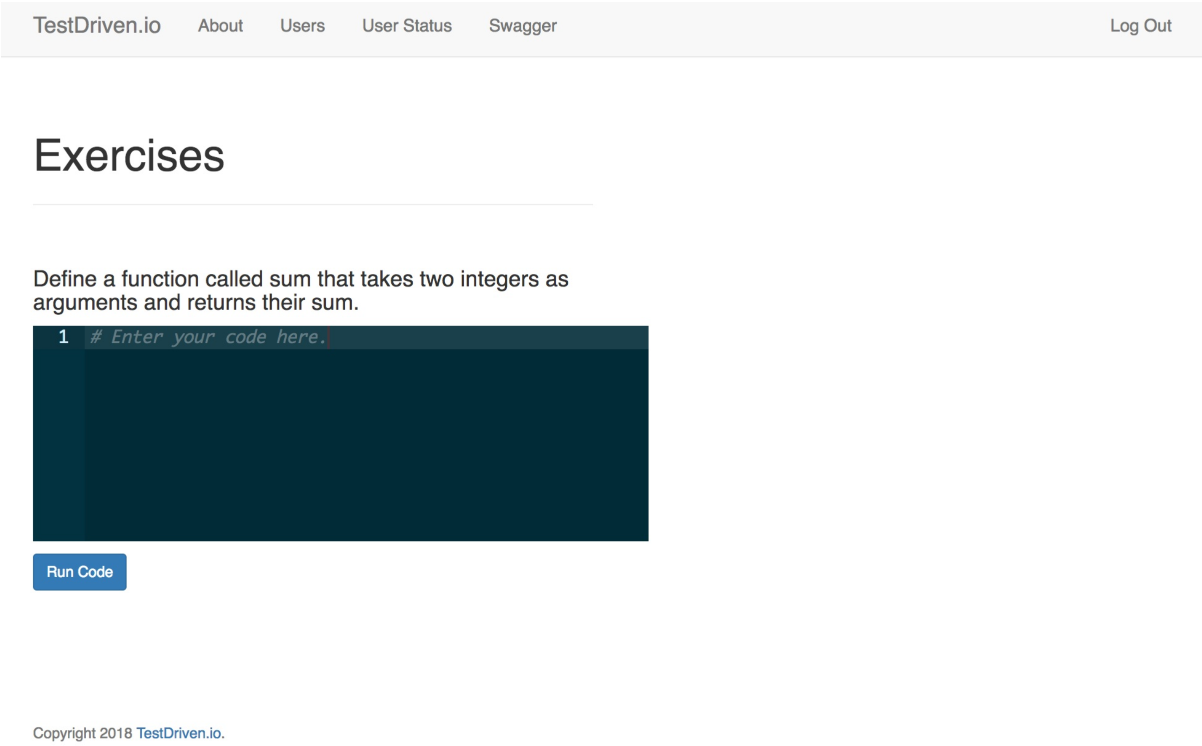
Often, when developing applications with a microservice architecture, you cannot fully test out all services until you deploy to a staging or production server. This takes much too long to get feedback. Fortunately, Docker helps to speed up this process by making it easier to link together small, independent services locally.

# App Overview

What are we building?

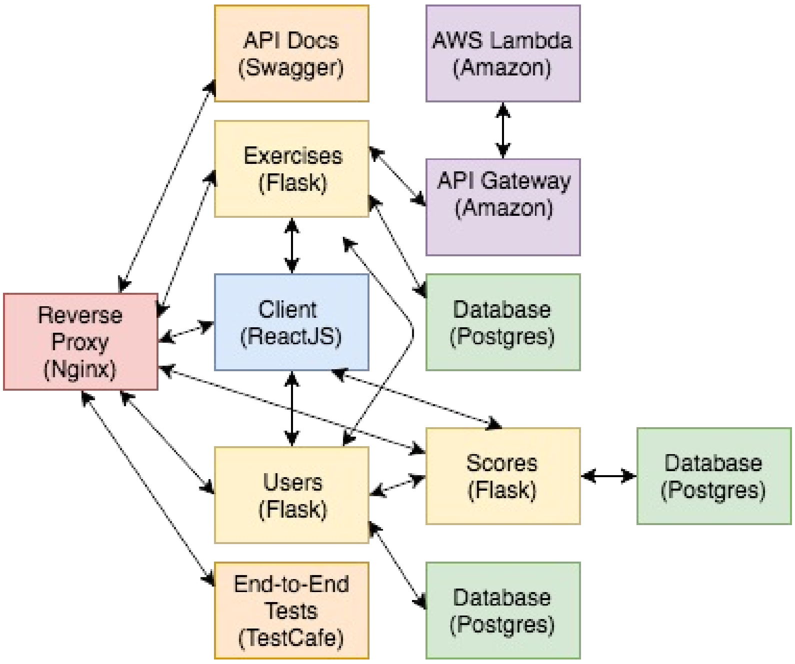
---

By the end of this course, you will have built a code evaluation tool for grading code exercises, similar to Codecademy, with Python, Flask, JavaScript, and ReactJS. The app, itself, will allow a user to log in and submit solutions to a coding problem. They will also be able to get feedback on whether a particular solution is correct or not.




We'll use the twelve-factor app pattern as we develop and design each microservice.

Along with twelve-factor, we'll also practice test-driven development (TDD), writing tests first when it makes sense to do so. The focus will be on server-side unit, functional, and integration tests, client-side unit tests, and end-to-end tests to ensure the entire system works as expected.




Finally, we'll dive into Docker and container orchestration to help manage, scale, and deploy our fleet of microservices.

# Getting Started

In this lesson, we'll set up the base project structure and define the first service...

---

Create a new project and install Flask:

```
$ mkdir testdriven-app && cd testdriven-app
$ mkdir services && cd services
$ mkdir users && cd users
$ mkdir project
$ python3.6 -m venv env
$ source env/bin/activate
(env)$ pip install flask==0.12.2
```

Add an *__init__.py* file to the "project" directory and configure the first route:

```
# services/users/project/__init__.py


from flask import Flask, jsonify


# instantiate the app
app = Flask(__name__)


@app.route('/users/ping', methods=['GET'])
def ping_pong():
    return jsonify({
        'status': 'success',
        'message': 'pong!'
    })
```

Next, let's configure the Flask CLI tool to run and manage the app from the command line.

> Feel free to replace the Flask CLI tool with Flask Script if you're used to it. Just keep in mind that it is deprecated.

First, add a *manage.py* file to the "users" directory:

```
# services/users/manage.py


from flask.cli import FlaskGroup

from project import app
```

```
cli = FlaskGroup(app)


if __name__ == '__main__':
    cli()
```

Here, we created a new `FlaskGroup` instance to extend the normal CLI with commands related to the Flask app.

Run the server:

```
(env)$ export FLASK_APP=project/__init__.py
(env)$ python manage.py run
```

Navigate to http://localhost:5000/users/ping in your browser. You should see:

```
{
  "message": "pong!",
  "status": "success"
}
```

Kill the server and add a new file called *config.py* to the "project" directory:

```
# services/users/project/config.py


class BaseConfig:
    """Base configuration"""
    TESTING = False


class DevelopmentConfig(BaseConfig):
    """Development configuration"""
    pass


class TestingConfig(BaseConfig):
    """Testing configuration"""
    TESTING = True


class ProductionConfig(BaseConfig):
    """Production configuration"""
    pass
```

Update *__init__.py* to pull in the dev config on init:

```
# services/users/project/__init__.py


from flask import Flask, jsonify


# instantiate the app
app = Flask(__name__)

# set config
app.config.from_object('project.config.DevelopmentConfig')


@app.route('/users/ping', methods=['GET'])
def ping_pong():
    return jsonify({
        'status': 'success',
        'message': 'pong!'
    })
```

Run the app again. This time, let's enable debug mode:

```
$ export FLASK_DEBUG=1
$ python manage.py run
* Serving Flask app "project"
* Forcing debug mode on
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
* Restarting with stat
* Debugger is active!
* Debugger PIN: 128-868-573
```

Now when you make changes to the code, the app will automatically reload. Once done, kill the server and deactivate from the virtual environment. Then, add a *requirements.txt* file to the "users" directory:

```
Flask==0.12.2
```

Finally, add a *.gitignore*, to the project root:

```
__pycache__
env
```

Init a git repo and commit your code to GitHub.

# Docker Config

Let's containerize the Flask app...

---

Start by ensuring that you have Docker, Docker Compose, and Docker Machine installed:

```
$ docker -v
Docker version 17.12.0-ce, build c97c6d6
$ docker-compose -v
docker-compose version 1.18.0, build 8dd22a9
$ docker-machine -v
docker-machine version 0.13.0, build 9ba6da9
```

Next, we need to create a new Docker host with Docker Machine and point the Docker client at it:

```
$ docker-machine create -d virtualbox testdriven-dev
$ docker-machine env testdriven-dev
$ eval "$(docker-machine env testdriven-dev)"
```

> Learn more about the `eval` command here.

Add a *Dockerfile-dev* to the "users" directory, making sure to review the code comments:

```
FROM python:3.6.4

# set working directory
RUN mkdir -p /usr/src/app
WORKDIR /usr/src/app

# add requirements
COPY ./requirements.txt /usr/src/app/requirements.txt

# install requirements
RUN pip install -r requirements.txt

# add app
COPY . /usr/src/app

# run server
CMD python manage.py run -h 0.0.0.0
```

Add *.dockerignore* to the "users" directory as well:

```
env
.dockerignore
```

```
Dockerfile-dev
Dockerfile-prod
```

Like the *.gitignore* file, the .dockerignore file lets you exclude certain files and folders from being copied over to the image.

Then add a *docker-compose-dev.yml* file to the root:

```
version: '3.4'

services:

  users:
    container_name: users
    build:
      context: ./services/users
      dockerfile: Dockerfile-dev
    volumes:
      - './services/users:/usr/src/app'
    ports:
      - 5001:5000
    environment:
      - FLASK_APP=project/__init__.py
      - FLASK_DEBUG=1
```

This config will create a container called `users`, from the Dockerfile.

> Directories are relative to the *docker-compose-dev.yml* file.

The `volume` is used to mount the code into the container. This is a must for a development environment in order to update the container whenever a change to the source code is made. Without this, you would have to re-build the image after each code change.

Take note of the Docker compose file version used - `3.4`. Keep in mind that this does *not* relate directly to the version of Docker Compose installed - it simply specifies the file format that you want to use.

Build the image:

```
$ docker-compose -f docker-compose-dev.yml build
```

This will take a few minutes the first time. Subsequent builds will be much faster since Docker caches the results of the first build. Once the build is done, fire up the container:

```
$ docker-compose -f docker-compose-dev.yml up -d
```

> The `-d` flag is used to run the containers in the background.

Grab the IP associated with the machine:

```
$ docker-machine ip testdriven-dev
```

Navigate to http://DOCKER_MACHINE_IP:5001/users/ping. Make sure you see the same JSON response as before. Next, add an environment variable to the *docker-compose-dev.yml* file to load the app config for the dev environment:

```
version: '3.4'

services:

  users:
    container_name: users
    build:
      context: ./services/users
      dockerfile: Dockerfile-dev
    volumes:
      - './services/users:/usr/src/app'
    ports:
      - 5001:5000
    environment:
      - FLASK_APP=project/__init__.py
      - FLASK_DEBUG=1
      - APP_SETTINGS=project.config.DevelopmentConfig
```

Then update *project/__init__.py*, to pull in the environment variables:

```
# services/users/project/__init__.py


import os
from flask import Flask, jsonify


# instantiate the app
app = Flask(__name__)

# set config
app_settings = os.getenv('APP_SETTINGS')
app.config.from_object(app_settings)


@app.route('/users/ping', methods=['GET'])
def ping_pong():
    return jsonify({
        'status': 'success',
        'message': 'pong!'
```

```
    })
```

Update the container:

```
$ docker-compose -f docker-compose-dev.yml up -d
```

Want to test, to ensure the proper config was loaded? Add a `print` statement to the *__init__.py*, right before the route handler, to view the app config to ensure that it is working:

```
import sys
print(app.config, file=sys.stderr)
```

Then just view the logs:

```
$ docker-compose -f docker-compose-dev.yml logs
```

You should see something like:

```
<Config {
  'DEBUG': True, 'TESTING': False, 'PROPAGATE_EXCEPTIONS': None,
  'PRESERVE_CONTEXT_ON_EXCEPTION': None, 'SECRET_KEY': None,
  'PERMANENT_SESSION_LIFETIME': datetime.timedelta(31), 'USE_X_SENDFILE':
  False, 'LOGGER_NAME': 'project', 'LOGGER_HANDLER_POLICY': 'always',
  'SERVER_NAME': None, 'APPLICATION_ROOT': None, 'SESSION_COOKIE_NAME':
  'session', 'SESSION_COOKIE_DOMAIN': None, 'SESSION_COOKIE_PATH': None,
  'SESSION_COOKIE_HTTPONLY': True, 'SESSION_COOKIE_SECURE': False,
  'SESSION_REFRESH_EACH_REQUEST': True, 'MAX_CONTENT_LENGTH': None,
  'SEND_FILE_MAX_AGE_DEFAULT': datetime.timedelta(0, 43200),
  'TRAP_BAD_REQUEST_ERRORS': False, 'TRAP_HTTP_EXCEPTIONS': False,
  'EXPLAIN_TEMPLATE_LOADING': False, 'PREFERRED_URL_SCHEME': 'http',
  'JSON_AS_ASCII': True, 'JSON_SORT_KEYS': True,
  'JSONIFY_PRETTYPRINT_REGULAR': True, 'JSONIFY_MIMETYPE':
  'application/json', 'TEMPLATES_AUTO_RELOAD': None}
>
```

Make sure to remove the `print` statement before moving on.

# Postgres Setup

In this lesson, we'll configure Postgres, get it up and running in another container, and link it to the `users` container...

---

Add Flask-SQLAlchemy and psycopg2 to the *requirements.txt* file:

```
Flask-SQLAlchemy==2.3.2
psycopg2==2.7.3.2
```

Update *config.py*:

```
# services/users/project/config.py


import os


class BaseConfig:
    """Base configuration"""
    TESTING = False
    SQLALCHEMY_TRACK_MODIFICATIONS = False


class DevelopmentConfig(BaseConfig):
    """Development configuration"""
    SQLALCHEMY_DATABASE_URI = os.environ.get('DATABASE_URL')


class TestingConfig(BaseConfig):
    """Testing configuration"""
    TESTING = True
    SQLALCHEMY_DATABASE_URI = os.environ.get('DATABASE_TEST_URL')


class ProductionConfig(BaseConfig):
    """Production configuration"""
    SQLALCHEMY_DATABASE_URI = os.environ.get('DATABASE_URL')
```

Update *__init__.py*, to create a new instance of SQLAlchemy and define the database model:

```
# services/users/project/__init__.py


import os
```

```
import datetime
from flask import Flask, jsonify
from flask_sqlalchemy import SQLAlchemy


# instantiate the app
app = Flask(__name__)

# set config
app_settings = os.getenv('APP_SETTINGS')
app.config.from_object(app_settings)

# instantiate the db
db = SQLAlchemy(app)


# model
class User(db.Model):
    __tablename__ = "users"
    id = db.Column(db.Integer, primary_key=True, autoincrement=True)
    username = db.Column(db.String(128), nullable=False)
    email = db.Column(db.String(128), nullable=False)
    active = db.Column(db.Boolean(), default=True, nullable=False)

    def __init__(self, username, email):
        self.username = username
        self.email = email


# routes
@app.route('/users/ping', methods=['GET'])
def ping_pong():
    return jsonify({
        'status': 'success',
        'message': 'pong!'
    })
```

Add a "db" directory to "project", and add a *create.sql* file in that new directory:

```
CREATE DATABASE users_prod;
CREATE DATABASE users_dev;
CREATE DATABASE users_test;
```

Next, add a *Dockerfile* to the same directory:

```
FROM postgres

# run create.sql on init
ADD create.sql /docker-entrypoint-initdb.d
```

Here, we extend the official Postgres image by adding a SQL file to the "docker-entrypoint-initdb.d" directory in the container, which will execute on init.

Update *docker-compose.yml-dev*:

```
version: '3.4'

services:

  users:
    container_name: users
    build:
      context: ./services/users
      dockerfile: Dockerfile-dev
    volumes:
      - './services/users:/usr/src/app'
    ports:
      - 5001:5000
    environment:
      - FLASK_APP=project/__init__.py
      - FLASK_DEBUG=1
      - APP_SETTINGS=project.config.DevelopmentConfig
      - DATABASE_URL=postgres://postgres:postgres@users-db:5432/users_dev
      - DATABASE_TEST_URL=postgres://postgres:postgres@users-db:5432/users_test
    depends_on:
      - users-db
    links:
      - users-db

  users-db:
    container_name: users-db
    build:
      context: ./services/users/project/db
      dockerfile: Dockerfile
    ports:
      - 5435:5432
    environment:
      - POSTGRES_USER=postgres
      - POSTGRES_PASSWORD=postgres
```

Once spun up, Postgres will be available on port `5435` on the host machine and on port `5432` for services running in other containers. Since the `users` service is dependent not only on the container being up and running but also the actual Postgres instance also being up and healthy, let's add an *entrypoint.sh* file to "users":

```
#!/bin/sh

echo "Waiting for postgres..."
```

```
while ! nc -z users-db 5432; do
  sleep 0.1
done

echo "PostgreSQL started"

python manage.py run -h 0.0.0.0
```

Update *Dockerfile-dev*:

```
FROM python:3.6.4

# install environment dependencies
RUN apt-get update -yqq \
  && apt-get install -yqq --no-install-recommends \
    netcat \
  && apt-get -q clean

# set working directory
RUN mkdir -p /usr/src/app
WORKDIR /usr/src/app

# add requirements
COPY ./requirements.txt /usr/src/app/requirements.txt

# install requirements
RUN pip install -r requirements.txt

# add entrypoint.sh
COPY ./entrypoint.sh /usr/src/app/entrypoint.sh

# add app
COPY . /usr/src/app

# run server
CMD ["./entrypoint.sh"]
```

Update the file permissions:

```
$ chmod +x services/users/entrypoint.sh
```

Sanity check:

```
$ docker-compose -f docker-compose-dev.yml up -d --build
```

Ensure http://DOCKER_MACHINE_IP:5001/users/ping still works:

```
{
  "message": "pong!",
  "status": "success"
}
```

Update *manage.py*:

```python
# services/users/manage.py


from flask.cli import FlaskGroup

from project import app, db


cli = FlaskGroup(app)


@cli.command()
def recreate_db():
    db.drop_all()
    db.create_all()
    db.session.commit()


if __name__ == '__main__':
    cli()
```

This registers a new command, `recreate_db`, to the CLI so that we can run it from the command line. Apply the model to the dev database:

```
$ docker-compose -f docker-compose-dev.yml \
  run users python manage.py recreate_db
```

Did this work? Let's hop into psql...

```
$ docker exec -ti $(docker ps -aqf "name=users-db") psql -U postgres

# \c users_dev
You are now connected to database "users_dev" as user "postgres".

# \dt
         List of relations
 Schema | Name  | Type  |  Owner
--------+-------+-------+----------
 public | users | table | postgres
(1 row)
```

```
# \q
```

# Test Setup

Let's get our tests up and running for this endpoint...

---

Add a "tests" directory to the "project" directory, and then create the following files inside the newly created directory:

1. *__init__.py*
2. *base.py*
3. *test_config.py*
4. *test_users.py*

***__init__.py***

```
# services/users/project/tests/__init__.py
```

***base.py***

```
# services/users/project/tests/base.py


from flask_testing import TestCase

from project import app, db


class BaseTestCase(TestCase):
    def create_app(self):
        app.config.from_object('project.config.TestingConfig')
        return app

    def setUp(self):
        db.create_all()
        db.session.commit()

    def tearDown(self):
        db.session.remove()
        db.drop_all()
```

***test_config.py***:

```
# services/users/project/tests/test_config.py


import os
```

```python
import unittest

from flask import current_app
from flask_testing import TestCase

from project import app


class TestDevelopmentConfig(TestCase):
    def create_app(self):
        app.config.from_object('project.config.DevelopmentConfig')
        return app

    def test_app_is_development(self):
        self.assertTrue(app.config['SECRET_KEY'] == 'my_precious')
        self.assertFalse(current_app is None)
        self.assertTrue(
            app.config['SQLALCHEMY_DATABASE_URI'] ==
            os.environ.get('DATABASE_URL')
        )


class TestTestingConfig(TestCase):
    def create_app(self):
        app.config.from_object('project.config.TestingConfig')
        return app

    def test_app_is_testing(self):
        self.assertTrue(app.config['SECRET_KEY'] == 'my_precious')
        self.assertTrue(app.config['TESTING'])
        self.assertFalse(app.config['PRESERVE_CONTEXT_ON_EXCEPTION'])
        self.assertTrue(
            app.config['SQLALCHEMY_DATABASE_URI'] ==
            os.environ.get('DATABASE_TEST_URL')
        )


class TestProductionConfig(TestCase):
    def create_app(self):
        app.config.from_object('project.config.ProductionConfig')
        return app

    def test_app_is_production(self):
        self.assertTrue(app.config['SECRET_KEY'] == 'my_precious')
        self.assertFalse(app.config['TESTING'])


if __name__ == '__main__':
    unittest.main()
```

***test_users.py***

```
# services/users/project/tests/test_users.py


import json
import unittest

from project.tests.base import BaseTestCase


class TestUserService(BaseTestCase):
    """Tests for the Users Service."""

    def test_users(self):
        """Ensure the /ping route behaves correctly."""
        response = self.client.get('/users/ping')
        data = json.loads(response.data.decode())
        self.assertEqual(response.status_code, 200)
        self.assertIn('pong!', data['message'])
        self.assertIn('success', data['status'])


if __name__ == '__main__':
    unittest.main()
```

Add Flask-Testing to the requirements file:

```
Flask-Testing==0.6.2
```

Add a new command to *manage.py*, to discover and run the tests:

```
@cli.command()
def test():
    """ Runs the tests without code coverage"""
    tests = unittest.TestLoader().discover('project/tests', pattern='test*.py')
    result = unittest.TextTestRunner(verbosity=2).run(tests)
    if result.wasSuccessful():
        return 0
    return 1
```

Don't forget to import `unittest`:

```
import unittest
```

We need to re-build the images since requirements are installed at build time rather than run time:

```
$ docker-compose -f docker-compose-dev.yml up -d --build
```

With the containers up and running, run the tests:

```
$ docker-compose -f docker-compose-dev.yml \
  run users python manage.py test
```

You should see the following error:

```
self.assertTrue(app.config['SECRET_KEY'] == 'my_precious')
```

Update the base config:

```
class BaseConfig:
    """Base configuration"""
    TESTING = False
    SQLALCHEMY_TRACK_MODIFICATIONS = False
    SECRET_KEY = 'my_precious'
```

Then re-test!

```
----------------------------------------------------------------------
Ran 4 tests in 0.071s

OK
```

# Flask Blueprints

With tests in place, let's refactor the app, adding in Blueprints...

---

> Unfamiliar with Blueprints? Check out the official Flask documentation. Essentially, they are self-contained components, used for encapsulating code, templates, and static files.

Create a new directory in "project" called "api", and add an *__init__.py* file along with *users.py* and *models.py*. Then within *users.py* add the following:

```
# services/users/project/api/users.py


from flask import Blueprint, jsonify


users_blueprint = Blueprint('users', __name__)


@users_blueprint.route('/users/ping', methods=['GET'])
def ping_pong():
    return jsonify({
        'status': 'success',
        'message': 'pong!'
    })
```

Here, we created a new instance of the `Blueprint` class and bound the `ping_pong()` function to it.

Then, add the following code to *models.py*:

```
# services/users/project/api/models.py


from project import db


class User(db.Model):
    __tablename__ = "users"
    id = db.Column(db.Integer, primary_key=True, autoincrement=True)
    username = db.Column(db.String(128), nullable=False)
    email = db.Column(db.String(128), nullable=False)
    active = db.Column(db.Boolean(), default=True, nullable=False)

    def __init__(self, username, email):
        self.username = username
```

```
        self.email = email
```

Update *project/__init__.py*, removing the route and model and adding the Application Factory pattern:

```
# services/users/project/__init__.py


import os

from flask import Flask, jsonify
from flask_sqlalchemy import SQLAlchemy


# instantiate the db
db = SQLAlchemy()


def create_app(script_info=None):

    # instantiate the app
    app = Flask(__name__)

    # set config
    app_settings = os.getenv('APP_SETTINGS')
    app.config.from_object(app_settings)

    # set up extensions
    db.init_app(app)

    # register blueprints
    from project.api.users import users_blueprint
    app.register_blueprint(users_blueprint)

    # shell context for flask cli
    app.shell_context_processor({'app': app, 'db': db})
    return app
```

Take note of the `shell_context_processor`. This is used to register the `app` and `db` to the shell. Now we can work with the application context and the database without having to import them directly into the shell.

Update *manage.py*:

```
# services/users/manage.py


import unittest

from flask.cli import FlaskGroup
```

```
from project import create_app, db
from project.api.models import User

app = create_app()
cli = FlaskGroup(create_app=create_app)


@cli.command()
def recreate_db():
    db.drop_all()
    db.create_all()
    db.session.commit()


@cli.command()
def test():
    """ Runs the tests without code coverage"""
    tests = unittest.TestLoader().discover('project/tests', pattern='test*.py')
    result = unittest.TextTestRunner(verbosity=2).run(tests)
    if result.wasSuccessful():
        return 0
    return 1


if __name__ == '__main__':
    cli()
```

Update the imports at the top of *project/tests/base.py* and *project/tests/test_config.py*:

```
from project import create_app

app = create_app()
```

(import `db` as well in *base.py*)

Finally, remove the `FLASK_APP` environment variable from *docker-compose-dev.yml*:

```
environment:
  - FLASK_DEBUG=1
  - APP_SETTINGS=project.config.DevelopmentConfig
  - DATABASE_URL=postgres://postgres:postgres@users-db:5432/users_dev
  - DATABASE_TEST_URL=postgres://postgres:postgres@users-db:5432/users_test
```

Test!

```
$ docker-compose -f docker-compose-dev.yml up -d
```

```
$ docker-compose -f docker-compose-dev.yml \
  run users python manage.py recreate_db

$ docker-compose -f docker-compose-dev.yml \
  run users python manage.py test
```

Correct any errors and move on...

# RESTful Routes

Next, let's set up three new routes, following RESTful best practices, with TDD:

| Endpoint | HTTP Method | CRUD Method | Result |
|---|---|---|---|
| /users | GET | READ | get all users |
| /users/:id | GET | READ | get single user |
| /users | POST | CREATE | add a user |

For each, we'll-

1. write a test
2. run the test, watching it fail (**red**)
3. write just enough code to get the test to pass (**green**)
4. **refactor** (if necessary)

Let' start with the POST route...

## POST

Add the test to the `TestUserService()` class in *project/tests/test_users.py*:

```
def test_add_user(self):
    """Ensure a new user can be added to the database."""
    with self.client:
        response = self.client.post(
            '/users',
            data=json.dumps({
                'username': 'michael',
                'email': 'michael@mherman.org'
            }),
            content_type='application/json',
        )
        data = json.loads(response.data.decode())
        self.assertEqual(response.status_code, 201)
        self.assertIn('michael@mherman.org was added!', data['message'])
        self.assertIn('success', data['status'])
```

Run the test to ensure it fails:

```
$ docker-compose -f docker-compose-dev.yml \
  run users python manage.py test
```

Then add the route handler to *project/api/users.py*

```
@users_blueprint.route('/users', methods=['POST'])
def add_user():
    post_data = request.get_json()
    username = post_data.get('username')
    email = post_data.get('email')
    db.session.add(User(username=username, email=email))
    db.session.commit()
    response_object = {
        'status': 'success',
        'message': f'{email} was added!'
    }
    return jsonify(response_object), 201
```

Update the imports as well:

```
from flask import Blueprint, jsonify, request

from project.api.models import User
from project import db
```

Run the tests. They all should pass:

```
Ran 5 tests in 0.092s

OK
```

What about errors and exceptions? Like:

1. A payload is not sent
2. The payload is invalid - i.e., the JSON object is empty or it contains the wrong keys
3. The user already exists in the database

Add some tests:

```
def test_add_user_invalid_json(self):
    """Ensure error is thrown if the JSON object is empty."""
    with self.client:
        response = self.client.post(
            '/users',
            data=json.dumps({}),
            content_type='application/json',
        )
        data = json.loads(response.data.decode())
        self.assertEqual(response.status_code, 400)
        self.assertIn('Invalid payload.', data['message'])
        self.assertIn('fail', data['status'])

def test_add_user_invalid_json_keys(self):
```

```
    """
    Ensure error is thrown if the JSON object does not have a username key.
    """
    with self.client:
        response = self.client.post(
            '/users',
            data=json.dumps({'email': 'michael@mherman.org'}),
            content_type='application/json',
        )
        data = json.loads(response.data.decode())
        self.assertEqual(response.status_code, 400)
        self.assertIn('Invalid payload.', data['message'])
        self.assertIn('fail', data['status'])

def test_add_user_duplicate_email(self):
    """Ensure error is thrown if the email already exists."""
    with self.client:
        self.client.post(
            '/users',
            data=json.dumps({
                'username': 'michael',
                'email': 'michael@mherman.org'
            }),
            content_type='application/json',
        )
        response = self.client.post(
            '/users',
            data=json.dumps({
                'username': 'michael',
                'email': 'michael@mherman.org'
            }),
            content_type='application/json',
        )
        data = json.loads(response.data.decode())
        self.assertEqual(response.status_code, 400)
        self.assertIn(
            'Sorry. That email already exists.', data['message'])
        self.assertIn('fail', data['status'])
```

Ensure the tests fail, and then update the route handler:

```
@users_blueprint.route('/users', methods=['POST'])
def add_user():
    post_data = request.get_json()
    response_object = {
        'status': 'fail',
        'message': 'Invalid payload.'
    }
    if not post_data:
        return jsonify(response_object), 400
```

```
    username = post_data.get('username')
    email = post_data.get('email')
    try:
        user = User.query.filter_by(email=email).first()
        if not user:
            db.session.add(User(username=username, email=email))
            db.session.commit()
            response_object['status'] = 'success'
            response_object['message'] = f'{email} was added!'
            return jsonify(response_object), 201
        else:
            response_object['message'] = 'Sorry. That email already exists.'
            return jsonify(response_object), 400
    except exc.IntegrityError as e:
        db.session.rollback()
        return jsonify(response_object), 400
```

Add the import:

```
from sqlalchemy import exc
```

Ensure the tests pass, and then move on to the next route...

## GET single user

Start with a test:

```
def test_single_user(self):
    """Ensure get single user behaves correctly."""
    user = User(username='michael', email='michael@mherman.org')
    db.session.add(user)
    db.session.commit()
    with self.client:
        response = self.client.get(f'/users/{user.id}')
        data = json.loads(response.data.decode())
        self.assertEqual(response.status_code, 200)
        self.assertIn('michael', data['data']['username'])
        self.assertIn('michael@mherman.org', data['data']['email'])
        self.assertIn('success', data['status'])
```

Add the following imports:

```
from project import db
from project.api.models import User
```

Ensure the test breaks before writing the view:

```
@users_blueprint.route('/users/<user_id>', methods=['GET'])
def get_single_user(user_id):
    """Get single user details"""
    user = User.query.filter_by(id=user_id).first()
    response_object = {
        'status': 'success',
        'data': {
            'id': user.id,
            'username': user.username,
            'email': user.email,
            'active': user.active
        }
    }
    return jsonify(response_object), 200
```

The tests should pass. Now, what about error handling?

1. An `id` is not provided
2. The `id` does not exist

Tests:

```
def test_single_user_no_id(self):
    """Ensure error is thrown if an id is not provided."""
    with self.client:
        response = self.client.get('/users/blah')
        data = json.loads(response.data.decode())
        self.assertEqual(response.status_code, 404)
        self.assertIn('User does not exist', data['message'])
        self.assertIn('fail', data['status'])

def test_single_user_incorrect_id(self):
    """Ensure error is thrown if the id does not exist."""
    with self.client:
        response = self.client.get('/users/999')
        data = json.loads(response.data.decode())
        self.assertEqual(response.status_code, 404)
        self.assertIn('User does not exist', data['message'])
        self.assertIn('fail', data['status'])
```

Updated view:

```
@users_blueprint.route('/users/<user_id>', methods=['GET'])
def get_single_user(user_id):
    """Get single user details"""
    response_object = {
        'status': 'fail',
        'message': 'User does not exist'
    }
```

```
    try:
        user = User.query.filter_by(id=int(user_id)).first()
        if not user:
            return jsonify(response_object), 404
        else:
            response_object = {
                'status': 'success',
                'data': {
                    'id': user.id,
                    'username': user.username,
                    'email': user.email,
                    'active': user.active
                }
            }
            return jsonify(response_object), 200
    except ValueError:
        return jsonify(response_object), 404
```

## GET all users

Again, let's start with a test. Since we'll have to add a few users first, let's add a quick helper function to the top of the *project/tests/test_users.py* file, just above the `TestUserService()` class.

```
def add_user(username, email):
    user = User(username=username, email=email)
    db.session.add(user)
    db.session.commit()
    return user
```

Now, refactor the *test_single_user()* test, like so:

```
def test_single_user(self):
    """Ensure get single user behaves correctly."""
    user = add_user('michael', 'michael@mherman.org')
    with self.client:
        response = self.client.get(f'/users/{user.id}')
        data = json.loads(response.data.decode())
        self.assertEqual(response.status_code, 200)
        self.assertIn('michael', data['data']['username'])
        self.assertIn('michael@mherman.org', data['data']['email'])
        self.assertIn('success', data['status'])
```

With that, let's add the new test:

```
def test_all_users(self):
    """Ensure get all users behaves correctly."""
    add_user('michael', 'michael@mherman.org')
    add_user('fletcher', 'fletcher@notreal.com')
```

```
    with self.client:
        response = self.client.get('/users')
        data = json.loads(response.data.decode())
        self.assertEqual(response.status_code, 200)
        self.assertEqual(len(data['data']['users']), 2)
        self.assertIn('michael', data['data']['users'][0]['username'])
        self.assertIn(
            'michael@mherman.org', data['data']['users'][0]['email'])
        self.assertIn('fletcher', data['data']['users'][1]['username'])
        self.assertIn(
            'fletcher@notreal.com', data['data']['users'][1]['email'])
        self.assertIn('success', data['status'])
```

Make sure it fails. Then add the view:

```
@users_blueprint.route('/users', methods=['GET'])
def get_all_users():
    """Get all users"""
    response_object = {
        'status': 'success',
        'data': {
            'users': [user.to_json() for user in User.query.all()]
        }
    }
    return jsonify(response_object), 200
```

Add the `to_json` method to the models:

```
def to_json(self):
    return {
        'id': self.id,
        'username': self.username,
        'email': self.email,
        'active': self.active
    }
```

Does the test past?

Before moving on, let's test the route in the browser - http://DOCKER_MACHINE_IP:5001/users. You should see:

```
{
  "data": {
    "users": [ ]
  },
  "status": "success"
}
```

Add a seed command to the *manage.py* file to populate the database with some initial data:

```
@cli.command()
def seed_db():
    """Seeds the database."""
    db.session.add(User(username='michael', email="hermanmu@gmail.com"))
    db.session.add(User(username='michaelherman', email="michael@mherman.org"))
    db.session.commit()
```

Try it out:

```
$ docker-compose -f docker-compose-dev.yml \
  run users python manage.py seed_db
```

Make sure you can view the users in the JSON response http://DOCKER_MACHINE_IP:5001/users.

> Think about how you could trim down some of the tests with shared setup code. Try not to sacrifice readability if you do decide to refactor.

# Deployment

With the routes up and tested, let's get this app deployed!

---

Follow the instructions here to sign up for AWS (if necessary) and create an IAM user (again, if necessary), making sure to add the credentials to an *~/.aws/credentials* file.

> Need help with IAM? Review the Controlling Access to Amazon EC2 Resources article.

Then, create the new host:

```
$ docker-machine create --driver amazonec2 testdriven-prod
```

> For more, review the Amazon Web Services (AWS) EC2 example from Docker.

Once done, set it as the active host and point the Docker client at it:

```
$ docker-machine env testdriven-prod
$ eval $(docker-machine env testdriven-prod)
```

Run the following command to view the currently running Machines:

```
$ docker-machine ls
```

Create a new compose file called *docker-compose-prod.yml* and add the contents of the other compose file minus the `FLASK_DEBUG` environment variable (alternatively, you could set it to `FLASK_DEBUG=1`) and the `volumes`.

> What would happen if you left the volumes in?

Spin up the containers, create the database, seed, and run the tests:

```
$ docker-compose -f docker-compose-prod.yml up -d --build

$ docker-compose -f docker-compose-prod.yml \
  run users python manage.py recreate_db

$ docker-compose -f docker-compose-prod.yml \
  run users python manage.py seed_db

$ docker-compose -f docker-compose-prod.yml \
  run users python manage.py test
```

Add port 5001 to the AWS Security Group. Grab the IP and make sure to test in the browser.

## Config

What about the app config and environment variables? Are these set up right? Are we using the production config? To check, run:

```
$ docker-compose -f docker-compose-prod.yml run users env
```

You should see the `APP_SETTINGS` variable assigned to `project.config.DevelopmentConfig`.

To update this, change the environment variables within *docker-compose-prod.yml*:

```
environment:
  - APP_SETTINGS=project.config.ProductionConfig
  - DATABASE_URL=postgres://postgres:postgres@users-db:5432/users_prod
  - DATABASE_TEST_URL=postgres://postgres:postgres@users-db:5432/users_test
```

Update:

```
$ docker-compose -f docker-compose-prod.yml up -d
```

Re-create the db and apply the seed again:

```
$ docker-compose -f docker-compose-prod.yml \
  run users python manage.py recreate_db

$ docker-compose -f docker-compose-prod.yml \
  run users python manage.py seed_db
```

Ensure the app is still running and check the environment variables again.

## Gunicorn

To use Gunicorn, first add the dependency to the *requirements.txt* file:

```
gunicorn==19.7.1
```

Create a new file in "users" called *entrypoint-prod.sh*:

```
#!/bin/sh

echo "Waiting for postgres..."

while ! nc -z users-db 5432; do
  sleep 0.1
done
```

```
echo "PostgreSQL started"

gunicorn -b 0.0.0.0:5000 manage:app
```

Update the permissions:

```
$ chmod +x services/users/entrypoint-prod.sh
```

Add a new Dockerfile called *Dockerfile-prod*:

```
FROM python:3.6.4

# install environment dependencies
RUN apt-get update -yqq \
  && apt-get install -yqq --no-install-recommends \
    netcat \
  && apt-get -q clean

# set working directory
RUN mkdir -p /usr/src/app
WORKDIR /usr/src/app

# add requirements
COPY ./requirements.txt /usr/src/app/requirements.txt

# install requirements
RUN pip install -r requirements.txt

# add entrypoint.sh
COPY ./entrypoint-prod.sh /usr/src/app/entrypoint-prod.sh

# add app
COPY . /usr/src/app

# run server
CMD ["./entrypoint-prod.sh"]
```

Then, change the `build` context for the `users` in *docker-compose-prod.yml* to reference the new Dockerfile:

```
build:
  context: ./services/users
  dockerfile: Dockerfile-prod
```

Update:

```
$ docker-compose -f docker-compose-prod.yml up -d --build
```

> The `--build` flag is necessary since we need to install the new dependency.

## Nginx

Next, let's get Nginx up and running as a reverse proxy to the web server. Create a new folder called "nginx" in the "services" folder, and then add a *Dockerfile*:

```
FROM nginx:1.13.8

RUN rm /etc/nginx/conf.d/default.conf
COPY /flask.conf /etc/nginx/conf.d
```

Add a new config file called *flask.conf* to the "nginx" folder as well:

```
server {

  listen 80;

  location / {
    proxy_pass         http://users:5000;
    proxy_redirect     default;
    proxy_set_header   Host $host;
    proxy_set_header   X-Real-IP $remote_addr;
    proxy_set_header   X-Forwarded-For $proxy_add_x_forwarded_for;
    proxy_set_header    X-Forwarded-Host $server_name;
  }

}
```

Add an `nginx` service to the *docker-compose-prod.yml*:

```
nginx:
  container_name: nginx
  build: ./services/nginx
  restart: always
  ports:
    - 80:80
  depends_on:
    - users
```

Then, remove the exposed `ports` from the `users` service and only expose port `5000` to other containers:

```
expose:
  - '5000'
```

Build the image and run the container:

```
$ docker-compose -f docker-compose-prod.yml up -d --build nginx
```

Add port `80` to the Security Group on AWS. Test the site in the browser again, this time at http://DOCKER_MACHINE_PROD_IP/users.

Let's update this locally as well. First, add `nginx` to the *docker-compose-dev.yml* file:

```
nginx:
  container_name: nginx
  build: ./services/nginx
  restart: always
  ports:
    - 80:80
  depends_on:
    - users
```

Next, we need to update the active host. To check which host is currently active, run:

```
$ docker-machine active
testdriven-prod
```

Change the active machine to `testdriven-dev`:

```
$ eval "$(docker-machine env testdriven-dev)"
```

Run the nginx container:

```
$ docker-compose -f docker-compose-dev.yml up -d --build nginx
```

Grab the IP and test it out!

> Did you notice that you can access the site locally with or without the ports - http://DOCKER_MACHINE_DEV_IP/users or http://DOCKER_MACHINE_DEV_IP:5001/users. Why? On prod, you can only access the site at http://DOCKER_MACHINE_PROD_IP/users, though. Why?

# Jinja Templates

Instead of just serving up a JSON API, let's spice it up with server-side templates...

---

Add a new route handler to *services/users/project/api/users.py*:

```
@users_blueprint.route('/', methods=['GET'])
def index():
    return render_template('index.html')
```

Update the Blueprint config as well:

```
users_blueprint = Blueprint('users', __name__, template_folder='./templates')
```

Be sure to update the imports:

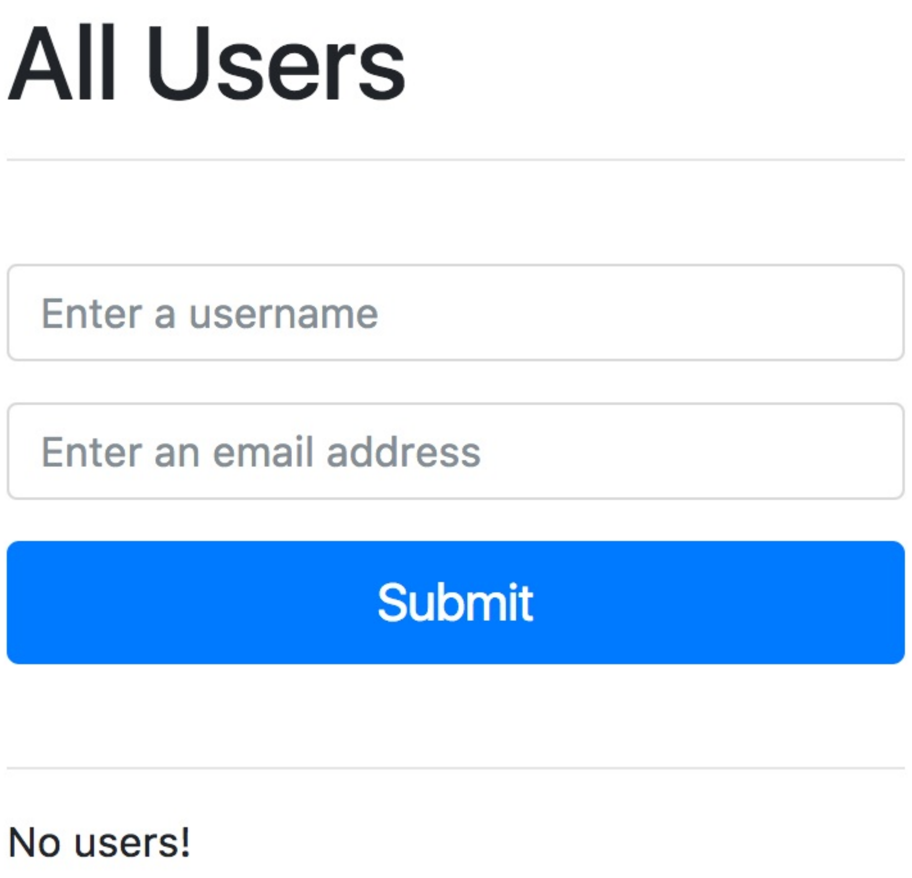
```
from flask import Blueprint, jsonify, request, render_template
```

Then add a "templates" folder to "project/api", and add an *index.html* file to that folder:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Flask on Docker</title>
    <!-- meta -->
    <meta name="description" content="">
    <meta name="author" content="">
    <meta name="viewport" content="width=device-width,initial-scale=1">
    <!-- styles -->
    <link
      href="//maxcdn.bootstrapcdn.com/bootstrap/4.0.0/css/bootstrap.min.css"
      rel="stylesheet"
    >
    {% block css %}{% endblock %}
  </head>
  <body>
    <div class="container">
      <div class="row">
        <div class="col-md-4">
          <br>
          <h1>All Users</h1>
          <hr><br>
          <form action="/" method="POST">
            <div class="form-group">
```

```
            <input
              name="username" class="form-control input-lg"
              type="text" placeholder="Enter a username" required>
          </div>
          <div class="form-group">
            <input
              name="email" class="form-control input-lg"
              type="email" placeholder="Enter an email address" required>
          </div>
          <input
            type="submit" class="btn btn-primary btn-lg btn-block"
            value="Submit">
        </form>
        <br>
        <hr>
          {% if users %}
            <ol>
              {% for user in users %}
                <li>{{user.username}}</li>
              {% endfor %}
            </ol>
          {% else %}
            <p>No users!</p>
          {% endif %}
        </div>
      </div>
    </div>
  </div>
  <!-- scripts -->
  <script
    type="text/javascript"
    src="//code.jquery.com/jquery-2.2.4.min.js"
  ></script>
  <script
    type="text/javascript"
    src="//maxcdn.bootstrapcdn.com/bootstrap/4.0.0/js/bootstrap.min.js"
  >
  </script>
  {% block js %}{% endblock %}
</body>
</html>
```

Ready to test? Simply open your browser and navigate to the IP associated with the `testdriven-dev` machine.




How about a test?

```
def test_main_no_users(self):
    """Ensure the main route behaves correctly when no users have been
    added to the database."""
    response = self.client.get('/')
    self.assertEqual(response.status_code, 200)
    self.assertIn(b'<h1>All Users</h1>', response.data)
    self.assertIn(b'<p>No users!</p>', response.data)
```

Do they pass?

```
$ docker-compose -f docker-compose-dev.yml \
  run users python manage.py test
```

Let's update the route handler to grab all users from the database and send them to the template, starting with a test:

```
def test_main_with_users(self):
```

```
    """Ensure the main route behaves correctly when users have been
    added to the database."""
    add_user('michael', 'michael@mherman.org')
    add_user('fletcher', 'fletcher@notreal.com')
    with self.client:
        response = self.client.get('/')
        self.assertEqual(response.status_code, 200)
        self.assertIn(b'<h1>All Users</h1>', response.data)
        self.assertNotIn(b'<p>No users!</p>', response.data)
        self.assertIn(b'michael', response.data)
        self.assertIn(b'fletcher', response.data)
```

Make sure it fails, and then update the view:

```
@users_blueprint.route('/', methods=['GET'])
def index():
    users = User.query.all()
    return render_template('index.html', users=users)
```

The test should now pass!

How about a form? Users should be able to add a new user and submit the form, which will then add the user to the database. Again, start with a test:

```
def test_main_add_user(self):
    """Ensure a new user can be added to the database."""
    with self.client:
        response = self.client.post(
            '/',
            data=dict(username='michael', email='michael@sonotreal.com'),
            follow_redirects=True
        )
        self.assertEqual(response.status_code, 200)
        self.assertIn(b'<h1>All Users</h1>', response.data)
        self.assertNotIn(b'<p>No users!</p>', response.data)
        self.assertIn(b'michael', response.data)
```

Then update the view:

```
@users_blueprint.route('/', methods=['GET', 'POST'])
def index():
    if request.method == 'POST':
        username = request.form['username']
        email = request.form['email']
        db.session.add(User(username=username, email=email))
        db.session.commit()
    users = User.query.all()
    return render_template('index.html', users=users)
```

Finally, let's update the code on AWS.

1. `eval $(docker-machine env testdriven-prod)`
2. `docker-compose -f docker-compose-prod.yml up -d --build`
3. Test:
   - http://DOCKER_MACHINE_PROD_IP
   - http://DOCKER_MACHINE_PROD_IP/users

# Workflow

Reference guide...

## Aliases

To save some precious keystrokes, let's create aliases for both the `docker-compose` and `docker-machine` commands - `dc` and `dm`, respectively.

Simply add the following lines to your *.bashrc* file:

```
alias dc='docker-compose'
alias dm='docker-machine'
```

Save the file, then execute it:

```
$ source ~/.bashrc
```

Test them out!

> On Windows? You will first need to create a PowerShell Profile (if you don't already have one), and then you can add the aliases to it using Set-Alias - i.e., `Set-Alias dc docker-compose`.

## "Saved" State

Is the VM stuck in a "Saved" state?

```
$ docker-machine ls

NAME                   ACTIVE    DRIVER       STATE      URL                            SWAR
M    DOCKER          ERRORS
testdriven-prod         *        amazonec2    Running    tcp://34.207.173.181:2376
    v17.09.0-ce
testdriven-dev          -        virtualbox   Saved
    Unknown
```

First, try:

```
$ docker-machine start testdriven-dev
```

If that doesn't work, to break out of this, you'll need to power off the VM:

1. Start virtualbox - `virtualbox`
2. Select the VM and click "start"
3. Exit the VM and select "Power off the machine"

4. Exit virtualbox

The VM should now have a "Stopped" state:

```
$ docker-machine ls

NAME              ACTIVE   DRIVER       STATE     URL                          SWAR
M   DOCKER          ERRORS
testdriven-prod    *        amazonec2    Running   tcp://34.207.173.181:2376
    v17.09.0-ce
testdriven-dev     -        virtualbox   Stopped
```

Now you can start the machine:

```
$ docker-machine start dev
```

It should be "Running":

```
$ docker-machine ls

NAME              ACTIVE   DRIVER       STATE     URL                          SWAR
M   DOCKER          ERRORS
testdriven-prod    *        amazonec2    Running   tcp://34.207.173.181:2376
    v17.09.0-ce
testdriven-dev     -        virtualbox   Running   tcp://192.168.99.100:2376
    v17.09.0-ce
```

## Can't Download Python Packages?

Are you running into this error when trying to `pip install` inside a Docker Machine?

```
Retrying (Retry(total=4, connect=None, read=None, redirect=None))
after connection broken by 'NewConnectionError(
  '<pip._vendor.requests.packages.urllib3.connection.VerifiedHTTPSConnection object
 at 0x7f0f88deec18>:
Failed to establish a new connection: [Errno -2] Name or service not known',)':
/simple/flask/
```

Restart the Machine and then start over:

```
$ docker-machine restart testdriven-dev
$ docker-machine env testdriven-dev
$ eval $(docker-machine env testdriven-dev)
$ docker-compose -f docker-compose-dev.yml up -d --build
```

## Common Commands

Build the images:

```
$ docker-compose -f docker-compose-dev.yml build
```

Run the containers:

```
$ docker-compose -f docker-compose-dev.yml up -d
```

Create the database:

```
$ docker-compose -f docker-compose-dev.yml \
  run users python manage.py recreate_db
```

Seed the database:

```
$ docker-compose -f docker-compose-dev.yml \
  run users python manage.py seed_db
```

Run the tests:

```
$ docker-compose -f docker-compose-dev.yml \
  run users python manage.py test
```

### Other commands

To stop the containers:

```
$ docker-compose -f docker-compose-dev.yml stop
```

To bring down the containers:

```
$ docker-compose -f docker-compose-dev.yml down
```

Want to force a build?

```
$ docker-compose -f docker-compose-dev.yml build --no-cache
```

Remove images:

```
$ docker rmi $(docker images -q)
```

### Postgres

Want to access the database via psql?

```
$ docker exec -ti users-db psql -U postgres -W
```

Then, you can connect to the database and run SQL queries. For example:

```
# \c users_dev
# select * from users;
```

# Structure

At the end of part 1, your project structure should look like this:

```
├── docker-compose-dev.yml
├── docker-compose-prod.yml
└── services
    ├── nginx
    │   ├── Dockerfile
    │   └── flask.conf
    └── users
        ├── Dockerfile-dev
        ├── Dockerfile-prod
        ├── entrypoint-prod.sh
        ├── entrypoint.sh
        ├── manage.py
        ├── project
        │   ├── __init__.py
        │   ├── api
        │   │   ├── __init__.py
        │   │   ├── models.py
        │   │   ├── templates
        │   │   │   └── index.html
        │   │   └── users.py
        │   ├── config.py
        │   ├── db
        │   │   ├── Dockerfile
        │   │   └── create.sql
        │   └── tests
        │       ├── __init__.py
        │       ├── base.py
        │       ├── test_config.py
        │       └── test_users.py
        └── requirements.txt
```

Code for part 1: https://github.com/testdrivenio/testdriven-app-2.2/releases/tag/part1

# Part 2

In part 2, we'll add *code coverage* and *continuous integration* testing to ensure that each service can be run and tested independently from the whole. Finally, we'll add *ReactJS* along with *Jest*, a JavaScript test runner, and *Enzyme*, a testing library made specifically for React, to the client-side.

## Structure

Before diving in, take a quick look at the current project structure:

```
├── docker-compose-dev.yml
├── docker-compose-prod.yml
└── services
    ├── nginx
    │   ├── Dockerfile
    │   └── flask.conf
    └── users
        ├── Dockerfile-dev
        ├── Dockerfile-prod
        ├── entrypoint-prod.sh
        ├── entrypoint.sh
        ├── manage.py
        ├── project
        │   ├── __init__.py
        │   ├── api
        │   │   ├── __init__.py
        │   │   ├── models.py
        │   │   ├── templates
        │   │   │   └── index.html
        │   │   └── users.py
        │   ├── config.py
        │   ├── db
        │   │   ├── Dockerfile
        │   │   └── create.sql
        │   └── tests
        │       ├── __init__.py
        │       ├── base.py
        │       ├── test_config.py
        │       └── test_users.py
        └── requirements.txt
```

Notice how we are managing each microservice in a single project, with a single git repo. It's important to note that you can also break each service into a separate project, each with its own git repo. There are pros and cons to each approach - mono repo vs multiple repo. Do your research.

> Interested in the mono approach? Review the code from version 1 of this course:

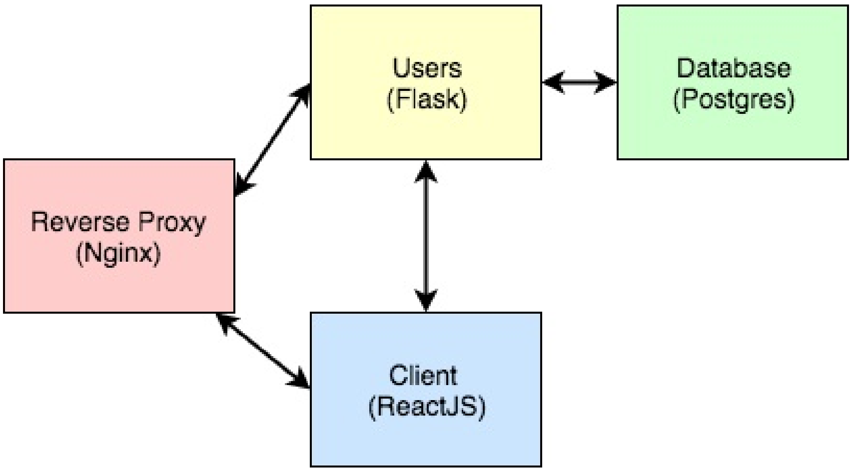
> 1. flask-microservices-main - Docker Compose files, Nginx, admin scripts
> 2. flask-microservices-users - Flask app for managing users and auth
> 3. flask-microservices-client - client-side, React app
> 4. flask-microservices-swagger - Swagger API docs
> 5. flask-microservices-eval - Flask app for managing user scores and exercises

## Objectives

By the end of this part, you will be able to...

1. Manage services housed in multiple git repos from a single Docker Compose file
2. Utilize a git repo as the "build context" for Docker Compose
3. Run unit and integration tests with code coverage inside a Docker Container
4. Check your code for any code quality issues via a linter
5. Work with each service independently without Docker
6. Configure Travis CI for continuous integration testing
7. Explain what React is
8. Work with React running inside a Docker Container
9. Unit test React components with Jest and Enzyme
10. Create a Single Page Application with React components
11. Use React props and state appropriately
12. Manage the state of a React component via component lifecycle methods
13. Pass environment variables to a Docker image at build time
14. Use React controlled components to handle form submissions
15. Create a production Dockerfile that uses multistage Docker builds

### App




Check out the live apps, running on EC2 -

1. Production
2. Staging

You can also test out the following endpoints...

| Endpoint | HTTP Method | CRUD Method | Result |
|---|---|---|---|
| / | GET | READ | Load React app |
| /users | GET | READ | get all users |
| /users/:id | GET | READ | get single user |
| /users | POST | CREATE | add a user |
| /users/ping | GET | READ | sanity check |

> The `/users` POST endpoint is restricted as of part 3.

Finished code for part 2: https://github.com/testdrivenio/testdriven-app-2.2/releases/tag/part2

## Dependencies

You will use the following dependencies in part 2:

1. Coverage.py v4.4.2
2. flake8 v3.5.0
3. Flask Debug Toolbar v0.10.1
4. Node v9.4.0
5. npm v5.6.0
6. Create React App v1.5.1
7. React v16.2.0
8. React Scripts v1.1.0
9. React Dom 16.2.0
10. Axios v0.17.1
11. Flask-CORS v3.0.3
12. Enzyme v3.3.0
13. enzyme-adapter-react-16 v1.1.1

# Code Coverage and Quality

In this lesson, we'll add code coverage via Coverage.py to the project...

---

Start by setting `testdriven-dev` as the active Docker Machine:

```
$ docker-machine env testdriven-dev
$ eval $(docker-machine env testdriven-dev)
```

Update the containers:

```
$ docker-compose -f docker-compose-dev.yml up -d
```

Ensure the app is working in the browser, and then run the tests:

```
$ docker-compose -f docker-compose-dev.yml \
  run users python manage.py test
```

## Code Coverage

Code coverage is the process of finding areas of your code not exercised by tests. Keep in mind that this does not measure the overall effectiveness of the test suite.

Add Coverage.py to the *requirements.txt* file in the "users" directory:

```
coverage==4.4.2
```

Next, we need to configure the coverage reports in *manage.py*. Start by adding the configuration right after the imports:

```
COV = coverage.coverage(
    branch=True,
    include='project/*',
    omit=[
        'project/tests/*',
        'project/config.py',
    ]
)
COV.start()
```

Then add the new CLI command:

```
@cli.command()
def cov():
    """Runs the unit tests with coverage."""
    tests = unittest.TestLoader().discover('project/tests')
    result = unittest.TextTestRunner(verbosity=2).run(tests)
    if result.wasSuccessful():
        COV.stop()
        COV.save()
        print('Coverage Summary:')
        COV.report()
        COV.html_report()
        COV.erase()
        return 0
    return 1
```

Don't forget the import!

```
import coverage
```

Update the containers:

```
$ docker-compose -f docker-compose-dev.yml up -d --build
```

Run the tests with coverage:

```
$ docker-compose -f docker-compose-dev.yml \
  run users python manage.py cov
```

You should see something like:

```
Coverage Summary:
Name                    Stmts   Miss Branch BrPart  Cover
---------------------------------------------------------
project/__init__.py        13      5      0      0    62%
project/api/models.py      12      9      0      0    25%
project/api/users.py       48     10     10      0    83%
---------------------------------------------------------
TOTAL                      73     24     10      0    71%
```

The HTML version can be viewed within the newly created "htmlcov" directory. Now you can quickly see which parts of the code are, and are not, covered by a test.

Add this directory to the *.gitignore* and *.dockerignore* files.

## Code Quality

Linting is the process of checking your code for stylistic or programming errors. Although there are a number of commonly used linters for Python, we'll use Flake8 since it combines two other popular linters - pep8 and pyflakes.

Add flake8 to the *requirements.txt* file in the "users" directory:

```
flake8===3.5.0
```

Update the containers:

```
$ docker-compose -f docker-compose-dev.yml up -d --build
```

Run flake8:

```
$ docker-compose -f docker-compose-dev.yml \
  run users flake8 project
```

Were any errors found?

```
project/__init__.py:6:1: F401 'flask.jsonify' imported but unused
```

Correct any issues before moving on. Commit your code, and push it to GitHub.

# Continuous Integration

Next, we'll add continuous integration (CI), via Travis CI, to our projects...

---

Follow steps 1 and 2 of the Getting Started guide to enable Travis in the project.

To trigger a build, add a *.travis.yml* to the project root:

```
sudo: required

services:
  - docker

env:
  DOCKER_COMPOSE_VERSION: 1.18.0

before_install:
  - sudo rm /usr/local/bin/docker-compose
  - curl -L https://github.com/docker/compose/releases/download/${DOCKER_COMPOSE_VE
RSION}/docker-compose-`uname -s`-`uname -m` > docker-compose
  - chmod +x docker-compose
  - sudo mv docker-compose /usr/local/bin

before_script:
  - docker-compose -f docker-compose-dev.yml up --build -d

script:
  - docker-compose -f docker-compose-dev.yml run users python manage.py test
  - docker-compose -f docker-compose-dev.yml run users flake8 project

after_script:
  - docker-compose -f docker-compose-dev.yml down
```

Commit your changes, and then push to GitHub. This *should* trigger a new build, which *should* pass. Once done, add a *README.md* file to the project root, adding the Travis status badge:

```
# Microservices with Docker, Flask, and React

[![Build Status](https://travis-ci.org/YOUR_GITHUB_USERNAME/testdriven-app.svg?bran
ch=master)](https://travis-ci.org/YOUR_GITHUB_USERNAME/testdriven-app)
```

> Be sure to replace `YOUR_GITHUB_USERNAME` with your actual GitHub username.

In terms of workflow, for now, while the project structure is still somewhat simple, we'll:

1. Code a new feature locally

2. Commit and push code
3. Ensure tests pass on Travis

# Flask Debug Toolbar

Let's wire up the Flask Debug Toolbar before diving into React...

---

Flask Debug Toolbar is a Flask extension that helps you debug your applications. It adds a debugging toolbar into the view which provides info on HTTP headers, request variables, configuration settings, and the number of SQLAlchemy queries it took to render the view. You can use this information to find bottlenecks in the rendering of the view.

Add the package to the *requirements.txt* file:

```
flask-debugtoolbar==0.10.1
```

To enable, create an instance of the toolbar and then add it to the app in `create_app()` in *services/users/project/__init__.py*:

```
# services/users/project/__init__.py


import os

from flask import Flask
from flask_sqlalchemy import SQLAlchemy
from flask_debugtoolbar import DebugToolbarExtension


# instantiate the extensions
db = SQLAlchemy()
toolbar = DebugToolbarExtension()


def create_app(script_info=None):

    # instantiate the app
    app = Flask(__name__)

    # set config
    app_settings = os.getenv('APP_SETTINGS')
    app.config.from_object(app_settings)

    # set up extensions
    db.init_app(app)
    toolbar.init_app(app)

    # register blueprints
    from project.api.users import users_blueprint
```

```
    app.register_blueprint(users_blueprint)

    # shell context for flask cli
    app.shell_context_processor({'app': app, 'db': db})
    return app
```

Next, update the config:

```
# services/users/project/config.py


import os


class BaseConfig:
    """Base configuration"""
    TESTING = False
    SQLALCHEMY_TRACK_MODIFICATIONS = False
    SECRET_KEY = 'my_precious'
    DEBUG_TB_ENABLED = False
    DEBUG_TB_INTERCEPT_REDIRECTS = False


class DevelopmentConfig(BaseConfig):
    """Development configuration"""
    SQLALCHEMY_DATABASE_URI = os.environ.get('DATABASE_URL')
    DEBUG_TB_ENABLED = True


class TestingConfig(BaseConfig):
    """Testing configuration"""
    TESTING = True
    SQLALCHEMY_DATABASE_URI = os.environ.get('DATABASE_TEST_URL')


class ProductionConfig(BaseConfig):
    """Production configuration"""
    SQLALCHEMY_DATABASE_URI = os.environ.get('DATABASE_URL')
```

> Review the docs for more info on the configuration options.

Add the new configuration to the tests in *services/users/project/tests/test_config.py*:

```
# services/users/project/tests/test_config.py


import os
import unittest
```

```
from flask import current_app
from flask_testing import TestCase

from project import create_app

app = create_app()


class TestDevelopmentConfig(TestCase):
    def create_app(self):
        app.config.from_object('project.config.DevelopmentConfig')
        return app

    def test_app_is_development(self):
        self.assertTrue(app.config['SECRET_KEY'] == 'my_precious')
        self.assertFalse(current_app is None)
        self.assertTrue(
            app.config['SQLALCHEMY_DATABASE_URI'] ==
            os.environ.get('DATABASE_URL')
        )
        self.assertTrue(app.config['DEBUG_TB_ENABLED'])


class TestTestingConfig(TestCase):
    def create_app(self):
        app.config.from_object('project.config.TestingConfig')
        return app

    def test_app_is_testing(self):
        self.assertTrue(app.config['SECRET_KEY'] == 'my_precious')
        self.assertTrue(app.config['TESTING'])
        self.assertFalse(app.config['PRESERVE_CONTEXT_ON_EXCEPTION'])
        self.assertTrue(
            app.config['SQLALCHEMY_DATABASE_URI'] ==
            os.environ.get('DATABASE_TEST_URL')
        )
        self.assertFalse(app.config['DEBUG_TB_ENABLED'])


class TestProductionConfig(TestCase):
    def create_app(self):
        app.config.from_object('project.config.ProductionConfig')
        return app

    def test_app_is_production(self):
        self.assertTrue(app.config['SECRET_KEY'] == 'my_precious')
        self.assertFalse(app.config['TESTING'])
        self.assertFalse(app.config['DEBUG_TB_ENABLED'])


if __name__ == '__main__':
```

```
    unittest.main()
```

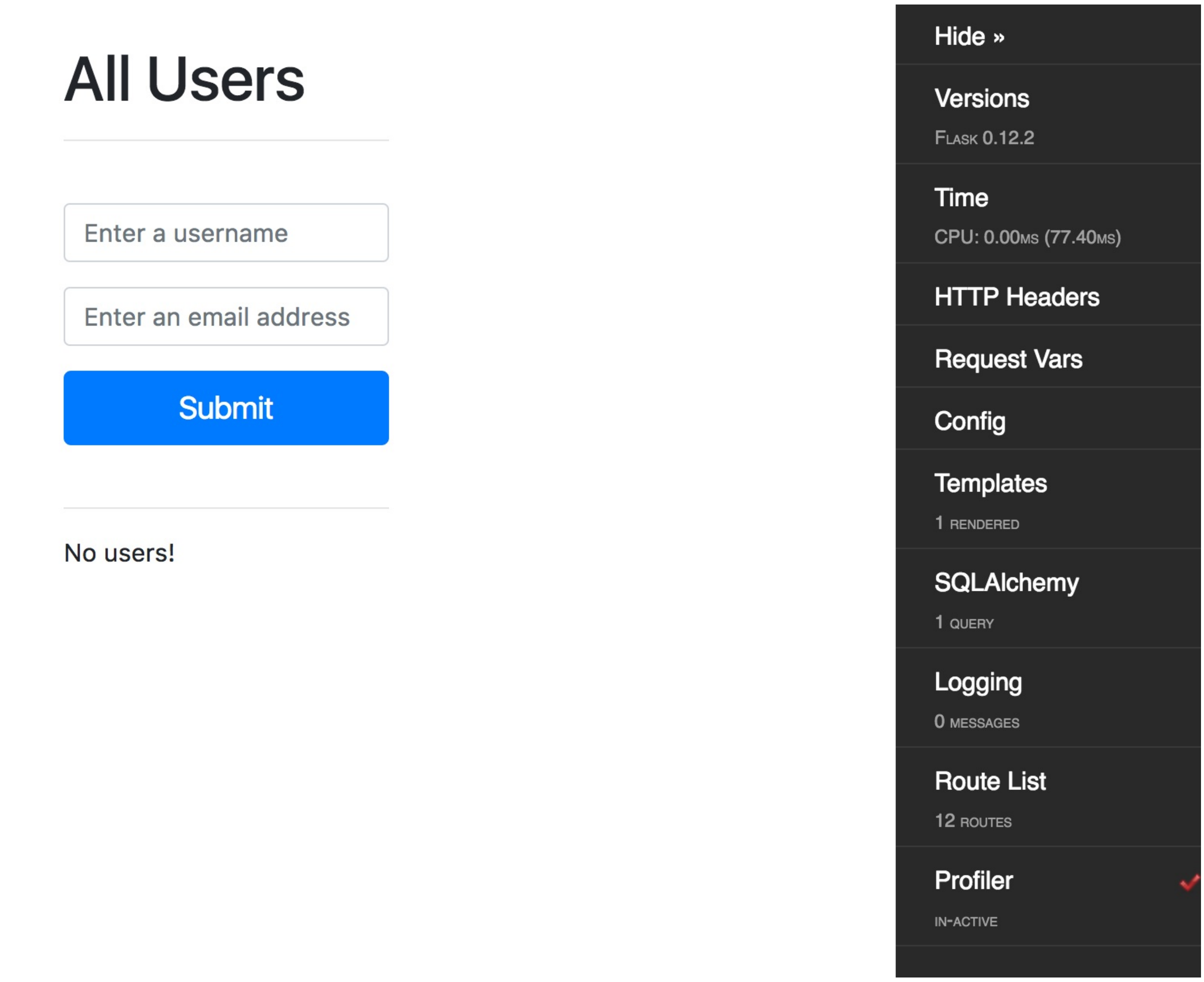
Update the containers and run the tests:

```
$ docker-compose -f docker-compose-dev.yml up -d
$ docker-compose -f docker-compose-dev.yml \
  run users python manage.py test
```

Finally, grab the IP associated with the `testdriven-dev` machine:

```
$ docker-machine ip testdriven-dev
```

Navigate to http://DOCKER_MACHINE_IP in your browser to view the toolbar in action:




> You may be wondering why we installed the toolbar in the first place since we won't be using server-rendered views all that much in this course. Well, it can still come in handy from time to time and it's a nice to have if you ever do serve up some Jinja templates.

# React Setup

Let's turn our attention to the client-side and add React...

---

React is a declarative, component-based, JavaScript library for building user interfaces.

If you're new to React, review the Quick Start and the excellent Why did we build React? blog post. You may also want to step through the Intro to React tutorial to learn more about Babel and Webpack - and how they work beneath the scenes.

Make sure you have Node and NPM installed before continuing:

```
$ node -v
v9.4.0
$ npm -v
5.6.0
```

## Project Setup

We'll use the amazing Create React App CLI to generate a boilerplate that's all set up and ready to go.

> Again, it's important to understand what's happening under the hood with Webpack and Babel. For more, check out the Intro to React tutorial.

Start by installing Create React App globally:

```
$ npm install create-react-app@1.5.1 --global
```

Add a new directory to "services" called "client", and then `cd` into the newly created directory and create the boilerplate:

```
$ create-react-app .
```

Along with creating the basic project structure, this will also install all dependencies. Once done, start the server:

```
$ npm start
```

After staring the server, Create React App automatically launches the app in your default browser on http://localhost:3000.

Ensure all is well, and then kill the server.

Next, to simplify the development process, remove the package-lock.json file. Let's also tell npm not to create one, during future module installations, for this project:

```
$ echo 'package-lock=false' >> .npmrc
```

> Review the npm docs for more info on the *.npmrc* config file.

Now we're ready build our first component!

## First Component

First, to simplify the structure, remove the *App.css*, *App.js*, *App.test.js*, and *index.css* from the "src" folder, and then update *index.js*:

```
import React from 'react';
import ReactDOM from 'react-dom';

const App = () => {
  return (
    <div className="container">
      <div className="row">
        <div className="col-md-4">
          <br/>
          <h1>All Users</h1>
          <hr/><br/>
        </div>
      </div>
    </div>
  )
};

ReactDOM.render(
  <App />,
  document.getElementById('root')
);
```

What's happening?

1. After importing the `React` and `ReactDom` classes, we created a functional component called `App`, which returns JSX.
2. We then used the `render` method from `ReactDOM` to mount the App to the DOM into the HTML element with an ID of `root`.

   > Take note of `<div id="root"></div>` within the *index.html* file in the "public" folder.

Add Bootstrap to *index.html* (found in the "public" folder) in the `head`:

```
<link
```

```
  href="//maxcdn.bootstrapcdn.com/bootstrap/4.0.0/css/bootstrap.min.css"
  rel="stylesheet"
>
```

Start the server again to see the changes in the browser:

```
$ npm start
```

## Class-based Component

Update *index.js*:

```
import React, { Component } from 'react';
import ReactDOM from 'react-dom';

class App extends Component {
  constructor() {
    super();
  }
  render() {
    return (
      <div className="container">
        <div className="row">
          <div className="col-md-4">
            <br/>
            <h1>All Users</h1>
            <hr/><br/>
          </div>
        </div>
      </div>
    )
  }
};

ReactDOM.render(
  <App />,
  document.getElementById('root')
);
```

What's happening?

1. We created a class-based component, which runs automatically when an instance is created (behind the scenes).
2. When ran, `super()` calls the constructor of `Component` , which `App` extends from.

You may have already noticed, but the output, in the browser, is the exact same as before, despite using a class-based component. We'll look at the differences between the two shortly!

## AJAX

To connect the client to the server, add a `getUsers()` method to the `App` class, which uses Axios to manage the AJAX call:

```
getUsers() {
  axios.get(`${process.env.REACT_APP_USERS_SERVICE_URL}/users`)
  .then((res) => { console.log(res); })
  .catch((err) => { console.log(err); });
}
```

Install Axios:

```
$ npm install axios@0.17.1 --save
```

Add the import:

```
import axios from 'axios';
```

You should now have:

```
import React, { Component } from 'react';
import ReactDOM from 'react-dom';
import axios from 'axios';

class App extends Component {
  constructor() {
    super();
  }
  getUsers() {
    axios.get(`${process.env.REACT_APP_USERS_SERVICE_URL}/users`)
    .then((res) => { console.log(res); })
    .catch((err) => { console.log(err); });
  }
  render() {
    return (
      <div className="container">
        <div className="row">
          <div className="col-md-4">
            <br/>
            <h1>All Users</h1>
            <hr/><br/>
          </div>
        </div>
      </div>
    )
  }
};
```

```
ReactDOM.render(
  <App />,
  document.getElementById('root')
);
```

To connect this up to the users service, open a new terminal window, navigate to the project root, set `testdriven-dev` as the active Docker Machine:

```
$ docker-machine env testdriven-dev
$ eval $(docker-machine env testdriven-dev)
```

Update the containers:

```
$ docker-compose -f docker-compose-dev.yml up -d
```

Ensure the app is working in the browser, and then run the tests:

```
$ docker-compose -f docker-compose-dev.yml \
  run users python manage.py test
```

Now, turning back to React, we need to add the environment variable, `process.env.REACT_APP_USERS_SERVICE_URL`. Kill the Create React App server, and then run:

```
$ export REACT_APP_USERS_SERVICE_URL=http://DOCKER_MACHINE_DEV_IP
```

> All custom environment variables must begin with `REACT_APP_`. For more, check out the official docs.

We still need to call the `getUsers()` method, which we can do, for now, in the `constructor()`:

```
constructor() {
  super();
  this.getUsers();
};
```

Run the server - via `npm start` - and then within Chrome DevTools, open the JavaScript Console. You should see the following error:

```
Failed to load http://192.168.99.100/users:
No 'Access-Control-Allow-Origin' header is present on the requested resource.
Origin 'http://localhost:3000' is therefore not allowed access.
```

In short, we're making a cross-origin AJAX request (from `http://localhost:3000` to `http://192.168.99.100`), which is a violation of the browser's "same origin policy". Fortunately, we can use the Flask-CORS extension to handle this.

Within the "users" directory, add Flask-CORS to the *requirements.txt* file:

```
flask-cors==3.0.3
```

To keep things simple, let's allow cross origin requests on all routes, from any domain. Simply update `create_app()` in *services/users/project/__init__.py* like so:

```
def create_app(script_info=None):

    # instantiate the app
    app = Flask(__name__)

    # enable CORS
    CORS(app)

    # set config
    app_settings = os.getenv('APP_SETTINGS')
    app.config.from_object(app_settings)

    # set up extensions
    db.init_app(app)
    toolbar.init_app(app)

    # register blueprints
    from project.api.users import users_blueprint
    app.register_blueprint(users_blueprint)

    # shell context for flask cli
    app.shell_context_processor({'app': app, 'db': db})
    return app
```

Add the import at the top:

```
from flask_cors import CORS
```

To test, start by updating the containers:

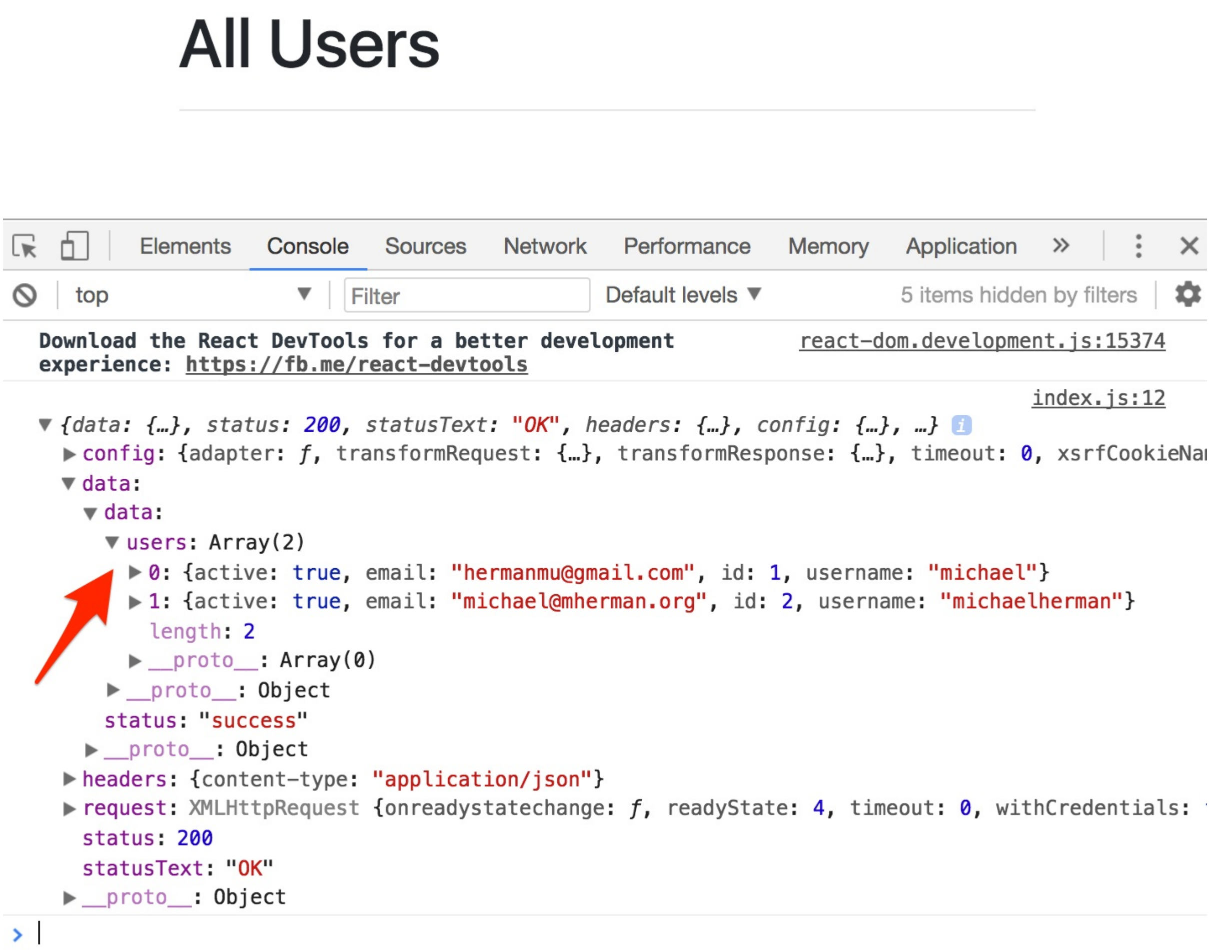
```
$ docker-compose -f docker-compose-dev.yml up -d --build
```

Then, update and seed the database:

```
$ docker-compose -f docker-compose-dev.yml \
  run users python manage.py recreate_db
```

```
$ docker-compose -f docker-compose-dev.yml \
  run users python manage.py seed_db
```

Fire back up both servers, open the JavaScript Console again, and this time you should see the results of `console.log(res);` :




Let's parse the JSON object:

```
getUsers() {
  axios.get(`${process.env.REACT_APP_USERS_SERVICE_URL}/users`)
  .then((res) => { console.log(res.data.data); })
  .catch((err) => { console.log(err); })
}
```

Now you should have an array with two objects in the JavaScript Console:

```
[
  {
    "active": true,
    "email": "hermanmu@gmail.com",
```

```
    "id": 1,
    "username": "michael"
  },
  {
    "active": true,
    "email": "michael@mherman.org",
    "id": 2,
    "username": "michaelherman"
  }
]
```

Before we move on, we need to do a quick refactor. Remember how we called the `getUsers()` method in the constructor?

```
constructor() {
  super();
  this.getUsers();
};
```

Well, the `constructor()` fires *before* the component is mounted to the DOM. What would happen if the AJAX request took longer than expected and the component mounted before the request completed? This introduces a race condition. Fortunately, React makes it fairly simple to correct this via Lifecycle Methods.

## Component Lifecycle Methods

Class-based components have several functions available to them that execute at certain times during the life of the component. These are called Lifecycle Methods. Take a quick look at the official documentation to learn about each method and when each is called.

The AJAX call should be made in the `componentDidMount()` method:

```
componentDidMount() {
  this.getUsers();
};
```

Update the component:

```
class App extends Component {
  constructor() {
    super();
  };
  componentDidMount() {
    this.getUsers();
  };
  getUsers() {
    axios.get(`${process.env.REACT_APP_USERS_SERVICE_URL}/users`)
```

```
    .then((res) => { console.log(res.data.data.users); })
    .catch((err) => { console.log(err); })
  };
  render() {
    return (
      <div className="container">
        <div className="row">
          <div className="col-md-4">
            <br/>
            <h1>All Users</h1>
            <hr/><br/>
          </div>
        </div>
      </div>
    )
  }
};
```

Make sure everything still works as it did before.

## State

To add the state - i.e., the users - to the component we need to use `setState()`, which is an asynchronous function used to update state.

Update `getUsers()`:

```
getUsers() {
  axios.get(`${process.env.REACT_APP_USERS_SERVICE_URL}/users`)
  .then((res) => { this.setState({ users: res.data.data.users }); })
  .catch((err) => { console.log(err); });
};
```

Add state to the constructor:

```
constructor() {
  super();
  this.state = {
    users: []
  };
};
```

So, `this.state` adds the state `property` to the class and sets `users` to an empty array.

> Review Using State Correctly from the official docs.

Finally, update the `render()` method to display the data returned from the AJAX call to the end user:

```
render() {
  return (
    <div className="container">
      <div className="row">
        <div className="col-md-6">
          <br/>
          <h1>All Users</h1>
          <hr/><br/>
          {
            this.state.users.map((user) => {
              return (
                <h4
                  key={user.id}
                  className="card card-body bg-light"
                >{ user.username }
                </h4>
              )
            })
          }
        </div>
      </div>
    </div>
  )
};
```

What's happening?

1. We iterated over the users (from the AJAX request) and created a new H4 element. This is why we needed to set an initial state of an empty array - it prevents `map` from exploding.
2. `key` ? - used by React to keep track of each element. Review the official docs for more.

## Functional Component

Let's create a new component for the users list. Add a new folder called "components" to "src". Add a new file to that folder called *UsersList.jsx*:

```
import React from 'react';

const UsersList = (props) => {
  return (
    <div>
      {
        props.users.map((user) => {
          return (
            <h4
              key={user.id}
              className="card card-body bg-light"
            >{user.username}
            </h4>
```

```
          )
        })
      }
    </div>
  )
};

export default UsersList;
```

Why did we use a functional component here rather than a class-based component?

Notice how we used `props` instead of `state` in this component. Essentially, you can pass state to a component with either `props` or `state`:

1. Props - data flows down via `props` (from `state` to `props`), read only
2. State - data is tied to a component, read and write

> For more, check out ReactJS: Props vs. State.

It's a good practice to limit the number of class-based (stateful) components since they can manipulate state and are, thus, less predictable. If you just need to render data (like in the above case), then use a functional (state-less) component.

Now we need to pass state from the parent to the child component via `props`. First, add the import to *index.js*:

```
import UsersList from './components/UsersList';
```

Then, update the `render()` method:

```
render() {
  return (
    <div className="container">
      <div className="row">
        <div className="col-md-6">
          <br/>
          <h1>All Users</h1>
          <hr/><br/>
            <UsersList users={this.state.users}/>
        </div>
      </div>
    </div>
  )
}
```

Review the code in each component and add comments as necessary. Commit your code.

# Testing React

Let's look at testing React components...

---

Create React App uses Jest, a JavaScript test runner, by default, so we can start writing test specs without having to install a runner. Along with Jest, we'll use Enzyme, a fantastic utility library made specifically for testing React components.

Install it as well enzyme-adapter-react-16:

```
$ npm install --save-dev enzyme@3.3.0 enzyme-adapter-react-16@1.1.1
```

To configure Enzyme to use the React 16 adapter, add a new file to "src" called *setupTests.js*:

```
import { configure } from 'enzyme';
import Adapter from 'enzyme-adapter-react-16';

configure({ adapter: new Adapter() });
```

> For more on setting up Enzyme, review the official docs.

With that, run the tests:

```
$ npm test
```

You should see:

```
No tests found related to files changed since last commit.
```

By default, the tests run in watch mode, so the tests will re-run every time you save a file.

## Testing Components

Add a new directory called "__tests__" within the "components" directory. Then, create a new file called *UsersList.test.jsx* in "__tests__":

```
import React from 'react';
import { shallow } from 'enzyme';

import UsersList from '../UsersList';

const users = [
  {
```

```
    'active': true,
    'email': 'hermanmu@gmail.com',
    'id': 1,
    'username': 'michael'
  },
  {
    'active': true,
    'email': 'michael@mherman.org',
    'id': 2,
    'username': 'michaelherman'
  }
];

test('UsersList renders properly', () => {
  const wrapper = shallow(<UsersList users={users}/>);
  const element = wrapper.find('h4');
  expect(element.length).toBe(2);
  expect(element.get(0).props.className).toBe('well');
  expect(element.get(0).props.children).toBe('michael');
});
```

In this test, we used the shallow helper method to create the `UsersList` component and then we retrieved the output and made assertions against it. It's important to note that with "shallow rendering", we can test the component in complete isolation, which helps to ensure child components do not indirectly affect assertions.

> For more on shallow rendering, along with the other methods of rendering components for testing, `mount` and `render`, see this Stack Overflow article.

Run the test to ensure it passes.

```
PASS  src/components/__tests__/UsersList.test.jsx
 ✓ UsersList renders properly (4ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        0.118s, estimated 1s
Ran all test suites related to changed files
```

## Snapshot Testing

Next, add a Snapshot test to ensure the UI does not change:

```
test('UsersList renders a snapshot properly', () => {
  const tree = renderer.create(<UsersList users={users}/>).toJSON();
  expect(tree).toMatchSnapshot();
});
```

Add the import to the top:

```
import renderer from 'react-test-renderer';
```

Run the tests:

```
PASS  src/components/__tests__/UsersList.test.jsx
 ✓ UsersList renders properly (3ms)
 ✓ UsersList renders a snapshot properly (9ms)

Snapshot Summary
› 1 snapshot written in 1 test suite.

Test Suites: 1 passed, 1 total
Tests:       2 passed, 2 total
Snapshots:   1 added, 1 total
Time:        0.468s, estimated 2s
Ran all test suites related to changed files.
```

On the first test run, a snapshot of the component output is saved to the "__snapshots__" folder:

```
// Jest Snapshot v1, https://goo.gl/fbAQLP

exports[`UsersList renders a snapshot properly 1`] = `
<div>
  <h4
    className="card card-body bg-light"
  >
    michael
  </h4>
  <h4
    className="card card-body bg-light"
  >
    michaelherman
  </h4>
</div>
`;
```

During subsequent test runs, the new output will be compared to the saved output. The test will fail if they differ.

Let's run a quick sanity check!

With the tests in watch mode, change `{user.username}` to `{user.email}` in the `UsersList` component. Save the changes to trigger a new test run. You should see both tests failing, which is exactly what we want:

```
FAIL  src/components/__tests__/UsersList.test.jsx
```

```
  ● UsersList renders properly

    expect(received).toBe(expected)

    Expected value to be (using ===):
      "michael"
    Received:
      "hermanmu@gmail.com"

      at Object.<anonymous>.test (src/components/__tests__/UsersList.test.jsx:27:41)
          at new Promise (<anonymous>)
      at Promise.resolve.then.el (node_modules/p-map/index.js:46:16)
          at <anonymous>

  ● UsersList renders a snapshot properly

    expect(value).toMatchSnapshot()

    Received value does not match stored snapshot 1.

    - Snapshot
    + Received

     <div>
       <h4
         className="card card-body bg-light"
       >
    -    michael
    +    hermanmu@gmail.com
       </h4>
       <h4
         className="card card-body bg-light"
       >
    -    michaelherman
    +    michael@mherman.org
       </h4>
     </div>

      at Object.<anonymous>.test (src/components/__tests__/UsersList.test.jsx:32:16)
          at new Promise (<anonymous>)
      at Promise.resolve.then.el (node_modules/p-map/index.js:46:16)
          at <anonymous>

  × UsersList renders properly (5ms)
  × UsersList renders a snapshot properly (4ms)

Snapshot Summary
› 1 snapshot test failed in 1 test suite.
Inspect your code changes or press `u` to update them.

Test Suites: 1 failed, 1 total
```

```
Tests:       2 failed, 2 total
Snapshots:   1 failed, 1 total
Time:        0.544s, estimated 1s
Ran all test suites related to changed files.
```

Now, if this change is intentional, you need to update the snapshot. To do so, just need to press the `u` key:

```
Watch Usage
 › Press a to run all tests.
 › Press u to update failing snapshots.
 › Press p to filter by a filename regex pattern.
 › Press t to filter by a test name regex pattern.
 › Press q to quit watch mode.
 › Press Enter to trigger a test run.
```

Try it out - press `u`. The tests will run again and the snapshot test should pass:

```
FAIL  src/components/__tests__/UsersList.test.jsx
 ● UsersList renders properly

   expect(received).toBe(expected)

   Expected value to be (using ===):
     "michael"
   Received:
     "hermanmu@gmail.com"

     at Object.<anonymous>.test (src/components/__tests__/UsersList.test.jsx:27:41)
         at new Promise (<anonymous>)
     at Promise.resolve.then.el (node_modules/p-map/index.js:46:16)
         at <anonymous>

 × UsersList renders properly (7ms)
 ✓ UsersList renders a snapshot properly (3ms)

Snapshot Summary
› 1 snapshot updated in 1 test suite.

Test Suites: 1 failed, 1 total
Tests:       1 failed, 1 passed, 2 total
Snapshots:   1 updated, 1 total
Time:        0.126s, estimated 1s
Ran all test suites related to changed files.
```

Once done, revert the changes we just made in the component, update the tests, ensure they pass, add the `__snapshots__` folder to the *.gitignore* file, and then commit your code.

## Test Coverage

Curious about test coverage?

```
$ react-scripts test --coverage
```

> You may need to globally install React Scripts - `npm install react-scripts@1.1.0 --global`.

```
PASS  src/components/__tests__/UsersList.test.jsx
 ✓ UsersList renders properly (10ms)
 ✓ UsersList renders a snapshot properly (8ms)

Test Suites: 1 passed, 1 total
Tests:       2 passed, 2 total
Snapshots:   1 passed, 1 total
Time:        1.071s
Ran all test suites.
---------------------------|----------|----------|----------|----------|-----------
-----|
File                       |  % Stmts | % Branch |  % Funcs |  % Lines |Uncovered L
ines |
---------------------------|----------|----------|----------|----------|-----------
-----|
All files                  |     7.14 |        0 |     8.33 |     12.9 |
     |
src                        |     1.89 |        0 |        0 |     3.57 |
    |
 index.js                  |        0 |        0 |        0 |        0 |... 19,20,23
,38 |
 registerServiceWorker.js |        0 |        0 |        0 |        0 |... 137,138,
139 |
 setupTests.js             |      100 |      100 |      100 |      100 |
    |
src/components             |      100 |      100 |      100 |      100 |
    |
 UsersList.jsx             |      100 |      100 |      100 |      100 |
    |
---------------------------|----------|----------|----------|----------|-----------
-----|
```

## Testing Interactions

Enzyme can also be used to test user interactions. We can simulate actions and events and then test that the actual results are the same as the expected results. We'll look at this in a future lesson.

> It's worth noting that we'll focus much of our React testing on unit testing the individual components. We'll let the end-to-end tests handle testing user interaction as well as the interaction between the client and server.

## `requestAnimationFrame` polyfill error

Do you get this error when your tests run?

```
console.error node_modules/fbjs/lib/warning.js:33
    Warning: React depends on requestAnimationFrame.
    Make sure that you load a polyfill in older browsers.
    http://fb.me/react-polyfills
```

If so, add a new folder to "services/client/src/components" called "__mocks__", and then add a file to that folder called *react.js*:

```
const react = require('react');
// Resolution for requestAnimationFrame not supported in jest error :
// https://github.com/facebook/react/issues/9102#issuecomment-283873039
global.window = global;
window.addEventListener = () => {};
window.requestAnimationFrame = () => {
  throw new Error('requestAnimationFrame is not supported in Node');
};

module.exports = react;
```

Review this comment on GitHub for more info.

# React Forms

In this lesson, we'll create a functional component for adding a new user....

---

Add two new files:

1. *services/client/src/components/AddUser.jsx*
2. *services/client/src/components/__tests__/AddUser.test.jsx*

Start with the test:

```
import React from 'react';
import { shallow } from 'enzyme';
import renderer from 'react-test-renderer';

import AddUser from './AddUser';

test('AddUser renders properly', () => {
  const wrapper = shallow(<AddUser/>);
  const element = wrapper.find('form');
  expect(element.find('input').length).toBe(3);
  expect(element.find('input').get(0).props.name).toBe('username');
  expect(element.find('input').get(1).props.name).toBe('email');
  expect(element.find('input').get(2).props.type).toBe('submit');
});
```

Here, we're asserting that a form, with three inputs, is present. Run the tests to ensure they fail, and then add the component:

```
import React from 'react';

const AddUser = (props) => {
  return (
    <form>
      <div className="form-group">
        <input
          name="username"
          className="form-control input-lg"
          type="text"
          placeholder="Enter a username"
          required
        />
      </div>
      <div className="form-group">
        <input
          name="email"
```

```
            className="form-control input-lg"
            type="email"
            placeholder="Enter an email address"
            required
          />
        </div>
        <input
          type="submit"
          className="btn btn-primary btn-lg btn-block"
          value="Submit"
        />
      </form>
    )
};

export default AddUser;
```

Import the component in *index.js*:

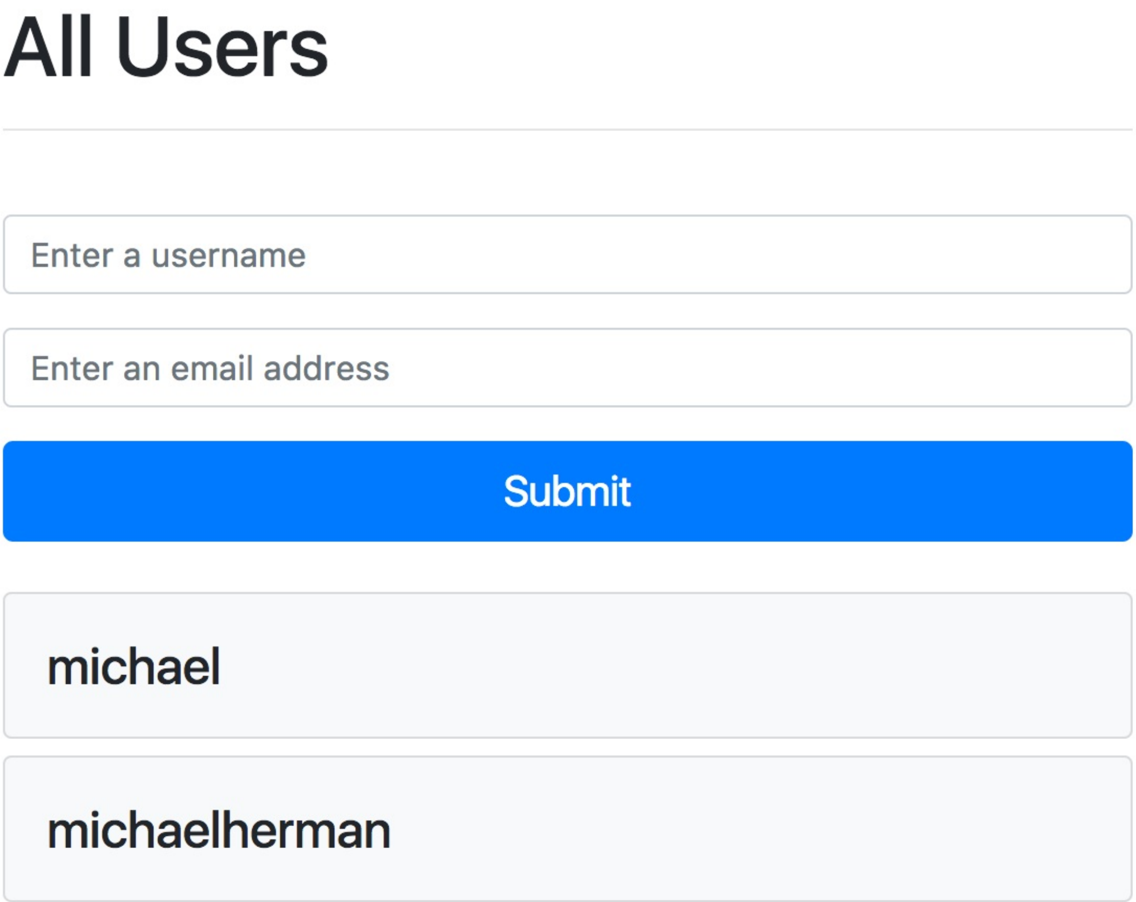
```
import AddUser from '../components/AddUser';
```

Then update the `render` method:

```
render() {
  return (
    <div className="container">
      <div className="row">
        <div className="col-md-6">
          <br/>
          <h1>All Users</h1>
          <hr/><br/>
          <AddUser/>
          <br/>
          <UsersList users={this.state.users}/>
        </div>
      </div>
    </div>
  )
};
```

Ensure the `testdriven-dev` machine is up and running and the `REACT_APP_USERS_SERVICE_URL` environment variable is properly assigned to the IP associated with the `testdriven-dev` machine. Run `npm start` to test.

If all went well, you should see the form along with the users.




Make sure the tests past as well:

```
PASS  src/components/__tests__/UsersList.test.jsx
PASS  src/components/__tests__/AddUser.test.jsx

Test Suites: 2 passed, 2 total
Tests:       3 passed, 3 total
Snapshots:   1 passed, 1 total
Time:        0.593s, estimated 1s
Ran all test suites related to changed files.
```

With that, let's add a snapshot test to *AddUser.test.jsx*:

```
test('AddUser renders a snapshot properly', () => {
  const tree = renderer.create(<AddUser/>).toJSON();
  expect(tree).toMatchSnapshot();
});
```

Ensure it passes before moving on.

Now, since this is a single page application, we want to prevent the normal browser behavior when a form is submitted to avoid a page refresh.

*Steps*:

1. Handle form submit event
2. Obtain user input
3. Send AJAX request
4. Update the page

## Handle form submit event

To handle the submit event, simply update the `form` element in *AddUser.jsx*:

```
<form onSubmit={(event) => event.preventDefault()}>
```

Enter a dummy username and email address, and then try submitting the form. Nothing should happen, which is exactly what we want - we prevented the normal browser behavior.

Next, add the following method to the `App` component:

```
addUser(event) {
  event.preventDefault();
  console.log('sanity check!');
};
```

Since `AddUser` is a functional component, we need to pass this method down to it via props. Update the `AddUser` element in the `render` method like so:

```
<AddUser addUser={this.addUser}/>
```

Update the `form` element again:

```
<form onSubmit={(event) => props.addUser(event)}>
```

Then, update the constructor as well:

```
constructor() {
  super();
  this.state = {
    users: []
  };
  this.addUser = this.addUser.bind(this);
};
```

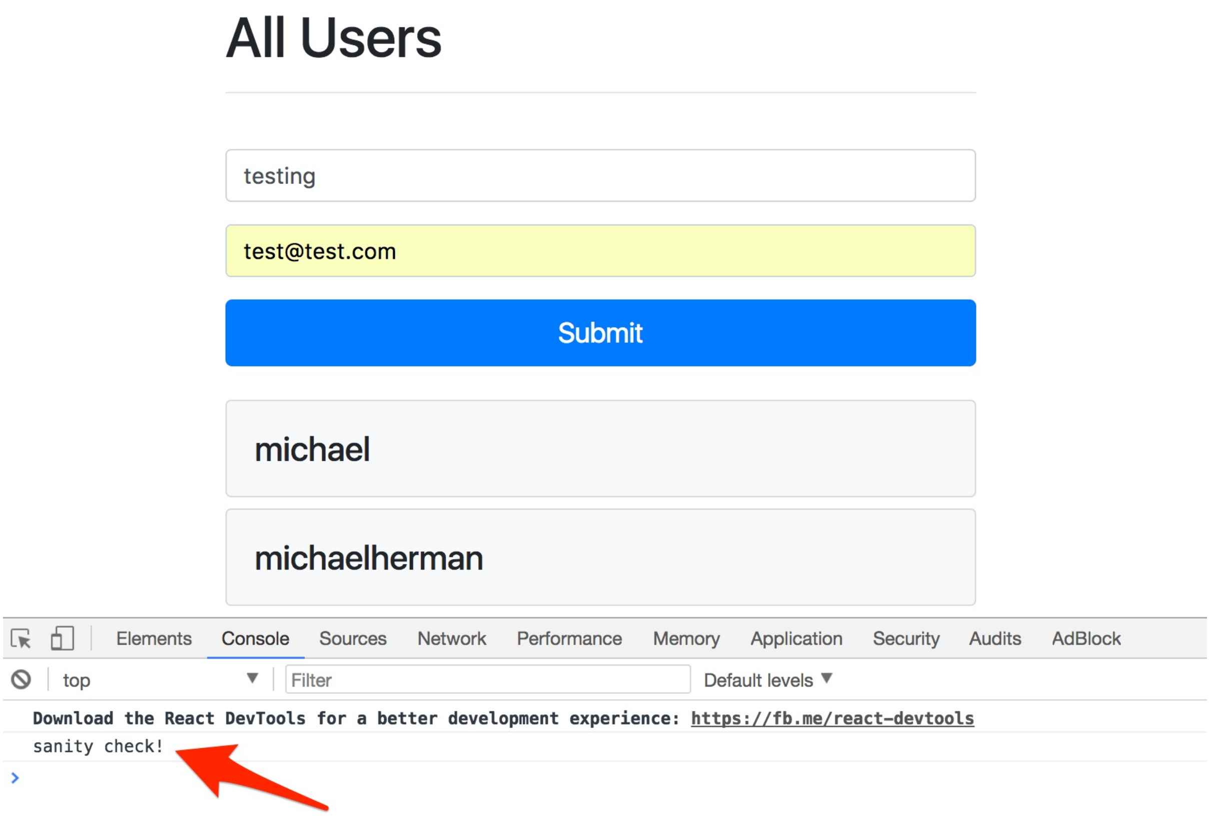
Here, we bound the context of `this` manually via `bind()` :

```
this.addUser = this.addUser.bind(this);
```

Without it, the context of `this` inside the method will not be correct. Want to test this out? Simply add `console.log(this)` to `addUser()` and then submit the form. What's the context? Remove the `bind` and test it again. What's the context now?

> For more on this, review Handling Events from the official React docs.

Test it out in the browser. You should see `sanity check!` in the JavaScript console on form submit.




## Obtain user input

We'll use controlled components to obtain the user submitted input. Start by adding two new properties to the state object in the `App` component:

```
this.state = {
  users: [],
  username: '',
  email: '',
};
```

Then, pass them through to the component:

```
<AddUser
  username={this.state.username}
  email={this.state.email}
```

```
  addUser={this.addUser}
/>
```

These are accessible now via the `props` object, which can be used as the current value of the input like so:

```
<div className="form-group">
  <input
    name="username"
    className="form-control input-lg"
    type="text"
    placeholder="Enter a username"
    required
    value={props.username}
  />
</div>
<div className="form-group">
  <input
    name="email"
    className="form-control input-lg"
    type="email"
    placeholder="Enter an email address"
    required
    value={props.email}
  />
</div>
```

So, this defines the value of the inputs from the parent component. Test out the form now. What happens if you try to add a username? You shouldn't see anything being typed since the value is being "pushed" down from the parent - and that value is `''`.

> What do you think will happen if the initial state of those values was set as `test` rather than an empty string? Try it.

How do we update the state in the parent component so that it updates when the user enters text into the input boxes?

First, add a `handleChange` method to the `App` component:

```
handleChange(event) {
  const obj = {};
  obj[event.target.name] = event.target.value;
  this.setState(obj);
};
```

Add the bind to the constructor:

```
this.handleChange = this.handleChange.bind(this);
```

Then, pass the method down to the component:

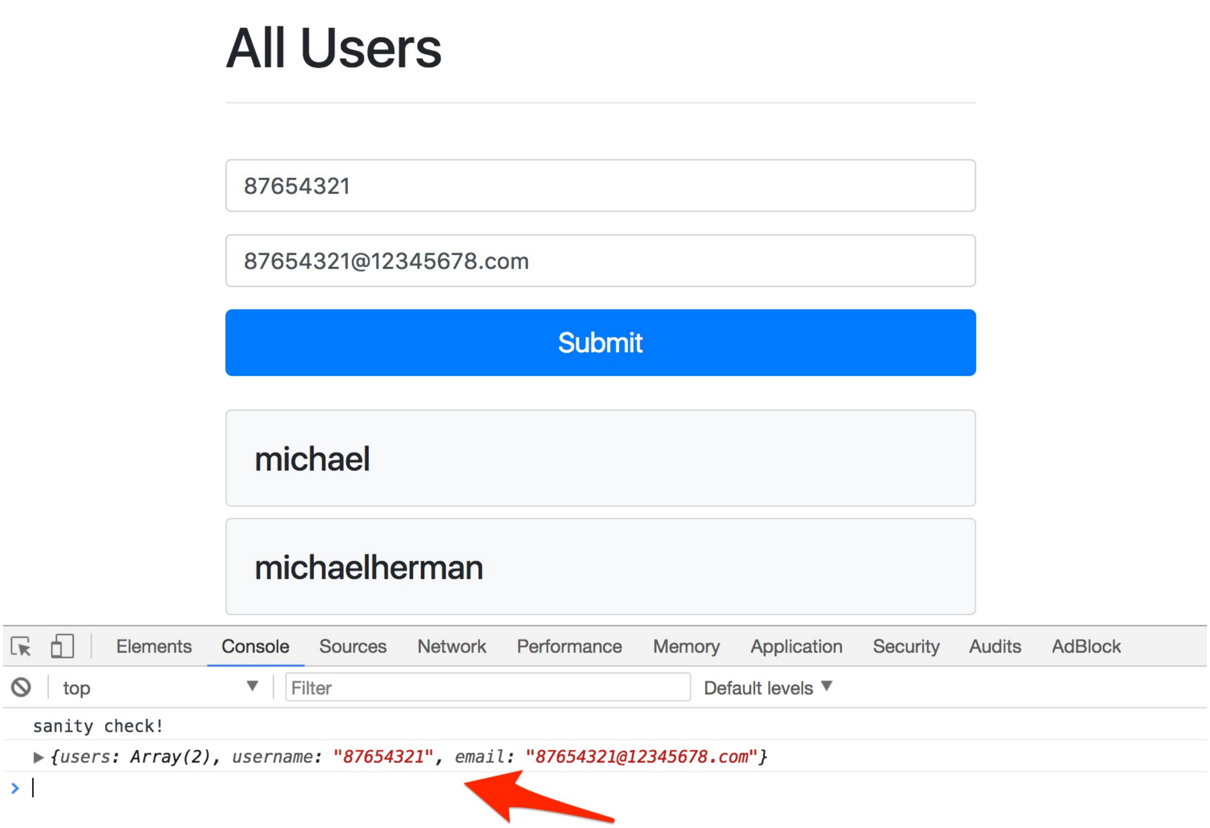
```
<AddUser
  username={this.state.username}
  email={this.state.email}
  handleChange={this.handleChange}
  addUser={this.addUser}
/>
```

Add it to the form inputs:

```
<div className="form-group">
  <input
    name="username"
    className="form-control input-lg"
    type="text"
    placeholder="Enter a username"
    required
    value={props.username}
    onChange={props.handleChange}
  />
</div>
<div className="form-group">
  <input
    name="email"
    className="form-control input-lg"
    type="email"
    placeholder="Enter an email address"
    required
    value={props.email}
    onChange={props.handleChange}
  />
</div>
```

Test the form out now. It should be working. If curious, you can see the value of the state by logging it to the console in the `addUser` method:

```
addUser(event) {
  event.preventDefault();
  console.log('sanity check!');
  console.log(this.state);
};
```




Now that we have the values, let's fire off the AJAX request to add the data to the database and then update the DOM...

## Send AJAX request

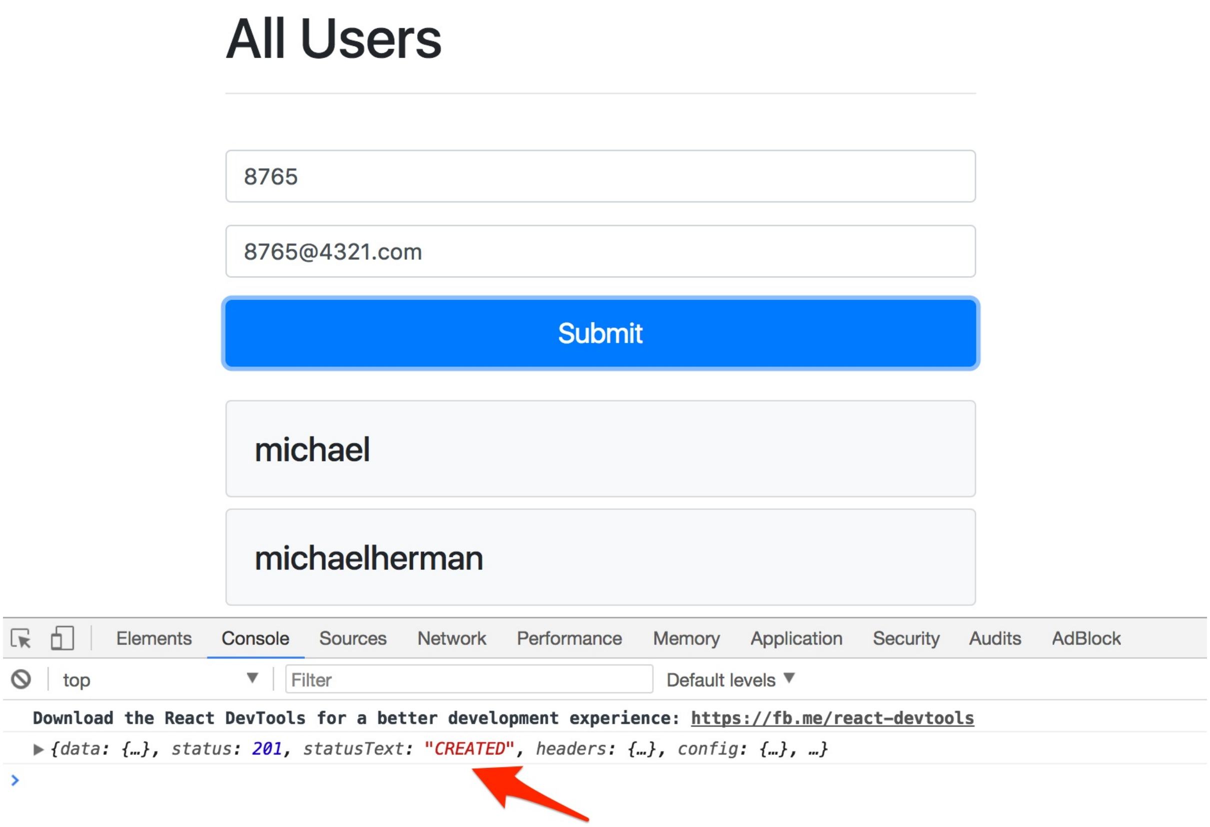
Turn your attention back to `users` service. What do we need to send in the JSON payload to add a user - username and email, right?

```
db.session.add(User(username=username, email=email))
```

Use Axios to send the POST request:

```
addUser(event) {
  event.preventDefault();
  const data = {
    username: this.state.username,
    email: this.state.email
  };
  axios.post(`${process.env.REACT_APP_USERS_SERVICE_URL}/users`, data)
  .then((res) => { console.log(res); })
  .catch((err) => { console.log(err); });
}
```

Test it out. Be sure to use a unique username and email. Although this does not matter so much now, it will present problems in the future since unique constraints will be added to the database table.




> If you have problems, analyze the response object from the "Network" tab in Chrome Developer Tools. You can also fire up the `users` service outside of Docker and debug using the Flask debugger or with `print` statements.

## Update the page

Finally, let's update the list of users on a successful form submit and then clear the form:

```
addUser(event) {
  event.preventDefault();
  const data = {
    username: this.state.username,
    email: this.state.email
  };
  axios.post(`${process.env.REACT_APP_USERS_SERVICE_URL}/users`, data)
  .then((res) => {
    this.getUsers();
    this.setState({ username: '', email: '' });
  })
  .catch((err) => { console.log(err); });
}
```

That's it. Manually test it out. Then, run the test suite. Update the snapshot test (by pressing `u` on the keyboard).

```
PASS  src/components/__tests__/AddUser.test.jsx
PASS  src/components/__tests__/UsersList.test.jsx

Test Suites: 2 passed, 2 total
Tests:       4 passed, 4 total
Snapshots:   2 passed, 2 total
Time:        0.159s, estimated 1s
Ran all test suites related to changed files.
```

Review and then commit your code.

# React and Docker

Let's containerize the React app...

---

## Local Development

Add *Dockerfile-dev* to the root of the "client" directory, making sure to review the code comments:

```
# base image
FROM node:9.4

# set working directory
RUN mkdir /usr/src/app
WORKDIR /usr/src/app

# add `/usr/src/app/node_modules/.bin` to $PATH
ENV PATH /usr/src/app/node_modules/.bin:$PATH

# install and cache app dependencies
COPY package.json /usr/src/app/package.json
RUN npm install --silent
RUN npm install react-scripts@1.1.0 -g --silent

# start app
CMD ["npm", "start"]
```

> Silencing the NPM output via `--silent` is a personal choice. It's often frowned upon, though, since it can swallow errors. Keep this in mind so you don't waste time debugging.

Add a *.dockerignore*:

```
node_modules
coverage
build
env
htmlcov
.dockerignore
Dockerfile-dev
Dockerfile-prod
```

Then, add the new service to the *docker-compose-dev.yml* file like so:

```
client:
  container_name: client
  build:
```

```
      context: ./services/client
      dockerfile: Dockerfile-dev
    volumes:
      - './services/client:/usr/src/app'
      - '/usr/src/app/node_modules'
    ports:
      - '3007:3000'
    environment:
      - NODE_ENV=development
      - REACT_APP_USERS_SERVICE_URL=${REACT_APP_USERS_SERVICE_URL}
    depends_on:
      - users
    links:
      - users
```

In the terminal, make sure `testdriven-dev` is the active machine and then add the valid environment variable:

```
$ export REACT_APP_USERS_SERVICE_URL=http://DOCKER_MACHINE_DEV_IP
```

Build the image and fire up the new container:

```
$ docker-compose -f docker-compose-dev.yml up --build -d client
```

Run the client-side tests:

```
$ docker-compose -f docker-compose-dev.yml \
  run client npm test
```

Navigate to http://DOCKER_MACHINE_DEV_IP:3007/ in your browser to test the app.

What happens if you navigate to the main route? Since we're still routing traffic to the Flask app (via Nginx), you will see the old app, served up with server-side templating. We need to update the Nginx configuration to route traffic to that main route to the React app.

Update *services/nginx/flask.conf*:

```
server {

  listen 80;

  location / {
    proxy_pass http://client:3000;
    proxy_redirect    default;
    proxy_set_header  Host $host;
    proxy_set_header  X-Real-IP $remote_addr;
    proxy_set_header  X-Forwarded-For $proxy_add_x_forwarded_for;
```

```
    proxy_set_header  X-Forwarded-Host $server_name;
  }

  location /users {
    proxy_pass        http://users:5000;
    proxy_redirect    default;
    proxy_set_header  Host $host;
    proxy_set_header  X-Real-IP $remote_addr;
    proxy_set_header  X-Forwarded-For $proxy_add_x_forwarded_for;
    proxy_set_header  X-Forwarded-Host $server_name;
  }

}
```

What's happening?

1. The `location` blocks define the reverse proxies.
2. When a requested URI matches the URI in a location block, Nginx passes the request either to the Create React App development server (serving the React app) or to the Flask development server (serving up the Flask app).

Also, `client` needs to spin up before `nginx`, so update *docker-compose-dev.yml*:

```
nginx:
  container_name: nginx
  build: ./services/nginx
  restart: always
  ports:
    - 80:80
  depends_on:
    - users
    - client
```

Update the containers (via `docker-compose -f docker-compose-dev.yml up -d --build`) and then test the app out in the browser:

1. http://DOCKER_MACHINE_DEV_IP/
2. http://DOCKER_MACHINE_DEV_IP/users

We can also take advantage of auto-reload since we set up a volume. To test, fire up the logs:

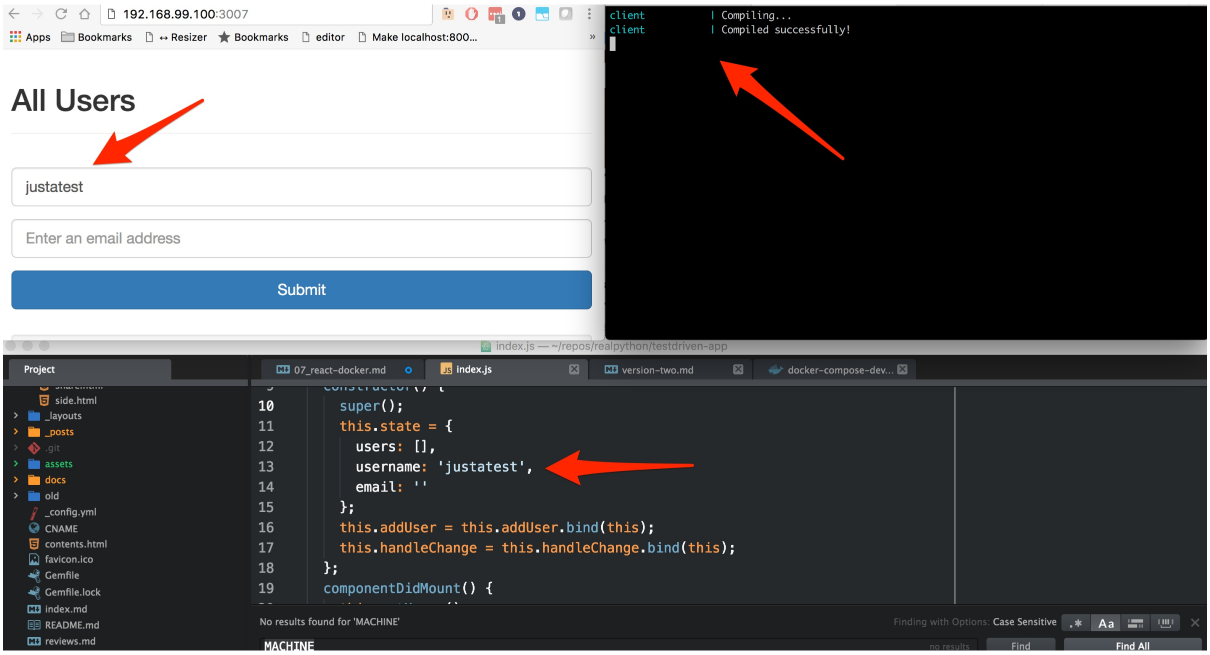
```
$ docker-compose -f docker-compose-dev.yml logs -f
```

Clear the terminal screen, and then change the state object in the `App` component:

```
this.state = {
  users: [],
  username: 'justatest',
```

```
  email: ''
};
```

As soon as you save, you should see the app re-compile and the browser should refresh on its own:




Make sure to change the state back before moving on.

> Having problems getting auto-reload to work properly with Docker Machine and VirtualBox?
>
> 1. Try enabling a polling mechanism via chokidar by adding the following environment variable key/pair to the *docker-compose-dev.yml* file - `CHOKIDAR_USEPOLLING=true`. Review Dockerizing a React App for more info.
> 2. Try reseting the Docker environment back to localhost. Rebuild the images, spin up the containers, and test auto-reload again.

## Create React App Build

Before updating the production environment, let's create a build with Create React App locally, outside of Docker (e.g., in a new terminal window), which will generate static files.

Make sure the `REACT_APP_USERS_SERVICE_URL` environment variable is set:

```
$ export REACT_APP_USERS_SERVICE_URL=http://DOCKER_MACHINE_DEV_IP
```

> All environment variables are embedded into the app at build time. Keep this in mind.

Then run the `build` command from the "services/client" directory:

```
$ npm run build
```

You should see a "build" directory, within "services/client", with the static files. We need to serve this up with a basic web server. Let's use the HTTP server from the Python standard library. Navigate to the "build" directory, and then run the server:

```
$ python3 -m http.server
```

This will serve up the app on http://localhost:8000/. Test it out in the browser to make sure it works. Once done, kill the server and navigate back to the project root.

## Production

Add *Dockerfile-prod* to the root of the "client" directory:

```
# build environment
FROM node:9.4 as builder
RUN mkdir /usr/src/app
WORKDIR /usr/src/app
ENV PATH /usr/src/app/node_modules/.bin:$PATH
ARG REACT_APP_USERS_SERVICE_URL
ARG NODE_ENV
ENV NODE_ENV $NODE_ENV
ENV REACT_APP_USERS_SERVICE_URL $REACT_APP_USERS_SERVICE_URL
COPY package.json /usr/src/app/package.json
RUN npm install --silent
RUN npm install react-scripts@1.1.0 -g --silent
COPY . /usr/src/app
RUN npm run build

# production environment
FROM nginx:1.13.5-alpine
COPY --from=builder /usr/src/app/build /usr/share/nginx/html
EXPOSE 80
CMD ["nginx", "-g", "daemon off;"]
```

Here, we used multistage builds to create a temporary image (build environment) used for building the artifact that is then copied over to the production image (production environment). The temporary build image is discarded along with the original files and folders associated with the image. This produces a lean, production-ready image.

When the image is built, arguments will be passed to the *Dockerfile*, via the ARG instruction, which can then be used as environment variables. `npm run build` will generate static files that are served up on port 80 via Nginx.

Let's test it without Docker Compose.

First, from "services/client", build the image, making sure to use the `--build-arg` flag to pass in the appropriate arguments:

```
$ docker build -f Dockerfile-prod -t "test" ./ \
  --build-arg NODE_ENV=development \
  --build-arg REACT_APP_USERS_SERVICE_URL=http://DOCKER_MACHINE_PROD_IP
```

> Make sure to replace `DOCKER_MACHINE_PROD_IP` with your actual IP.

This uses the *Dockerfile-prod* file found in "services/client", `./`, to build a new image called `test` with the required build arguments.

> You can view all images by running `docker image`.

Spin up the container from the `test` image, mapping port 80 in the container to port 9000 outside the container:

```
$ docker run -d -p 9000:80 test
```

Navigate to http://localhost:9000/ in your browser to test.

Stop and remove the container once done:

```
$ docker stop CONTAINER_ID
$ docker rm CONTAINER_ID
```

Finally, remove the image:

```
$ docker rmi test
```

With the *Dockerfile-prod* file set up and tested, add the service to *docker-compose-prod.yml*:

```
client:
  container_name: client
  build:
    context: ./services/client
    dockerfile: Dockerfile-prod
    args:
      - NODE_ENV=production
      - REACT_APP_USERS_SERVICE_URL=${REACT_APP_USERS_SERVICE_URL}
  ports:
    - '3007:80'
  depends_on:
    - users
  links:
    - users
```

So, instead of passing `NODE_ENV` and `REACT_APP_USERS_SERVICE_URL` as environment variables, which happens at runtime, we defined them as build arguments.

Again, the `client` service needs to spin up before `nginx`, so update *docker-compose-prod.yml*:

```
nginx:
  container_name: nginx
  build: ./services/nginx
  restart: always
  ports:
    - 80:80
  depends_on:
    - users
    - client
```

Did you notice that we exposed a different internal port for the `client` service in production?

- `3000` for dev
- `80` for prod

Because of this, we need to use a different Nginx config file in production:

1. Rename *services/nginx/Dockerfile* to *services/nginx/Dockerfile-dev*
2. Rename *services/nginx/flask.conf* to *services/nginx/dev.conf*
3. Update *services/nginx/Dockerfile-dev*:

```
FROM nginx:1.13.8

RUN rm /etc/nginx/conf.d/default.conf
COPY /dev.conf /etc/nginx/conf.d
```

4. Update `nginx` in *docker-compose-dev.yml*:

```
nginx:
  container_name: nginx
  build:
    context: ./services/nginx
    dockerfile: Dockerfile-dev
  restart: always
  ports:
    - 80:80
  depends_on:
    - users
    - client
```

5. Create a new file called *services/nginx/Dockerfile-prod*:

```
FROM nginx:1.13.8

RUN rm /etc/nginx/conf.d/default.conf
COPY /prod.conf /etc/nginx/conf.d
```

6. Add *services/nginx/prod.conf* as well:

```
server {

  listen 80;

  location / {
    proxy_pass http://client:80;
    proxy_redirect    default;
    proxy_set_header  Host $host;
    proxy_set_header  X-Real-IP $remote_addr;
    proxy_set_header  X-Forwarded-For $proxy_add_x_forwarded_for;
    proxy_set_header  X-Forwarded-Host $server_name;
  }

  location /users {
    proxy_pass        http://users:5000;
    proxy_redirect    default;
    proxy_set_header  Host $host;
    proxy_set_header  X-Real-IP $remote_addr;
    proxy_set_header  X-Forwarded-For $proxy_add_x_forwarded_for;
    proxy_set_header  X-Forwarded-Host $server_name;
  }

}
```

7. Update `nginx` in *docker-compose-prod.yml*:

```
nginx:
  container_name: nginx
  build:
    context: ./services/nginx
    dockerfile: Dockerfile-prod
  restart: always
  ports:
    - 80:80
  depends_on:
    - users
    - client
```

To update production, set the `testdriven-prod` machine as the active machine, change the `REACT_APP_USERS_SERVICE_URL` environment variable to the IP associated with the `testdriven-prod` machine, and update the containers:

```
$ docker-machine env testdriven-prod
$ eval $(docker-machine env testdriven-prod)
$ export REACT_APP_USERS_SERVICE_URL=http://DOCKER_MACHINE_AWS_IP
```

```
$ docker-compose -f docker-compose-prod.yml up -d --build
```

> Remember: Since the environment variables are added at the build time, if you change the variables, you *will* have to re-build the Docker image.

Make sure all is well in the browser.

## Travis

One more thing: Add the `REACT_APP_USERS_SERVICE_URL` environment variable to the *.travis.yml* file, within the `before_script`:

```
before_script:
  - export REACT_APP_USERS_SERVICE_URL=http://127.0.0.1
  - docker-compose -f docker-compose-dev.yml up --build -d
```

Commit and push your code to GitHub. Ensure the Travis build passes before moving on.

## Next Steps

Now is a great time to pause, review the code, and write more unit and integration tests. Do this on your own to check your understanding.

> Want feedback on your code? Shoot an email to `michael@mherman.org` with a link to the GitHub repo. Cheers!

# Structure

At the end of part 2, your project structure should look like this:

```
├── README.md
├── docker-compose-dev.yml
├── docker-compose-prod.yml
└── services
    ├── client
    │   ├── Dockerfile-dev
    │   ├── Dockerfile-prod
    │   ├── README.md
    │   ├── build
    │   ├── coverage
    │   ├── package.json
    │   ├── public
    │   │   ├── favicon.ico
    │   │   ├── index.html
    │   │   └── manifest.json
    │   └── src
    │       ├── components
    │       │   ├── AddUser.jsx
    │       │   ├── UsersList.jsx
    │       │   └── __tests__
    │       │       ├── AddUser.test.jsx
    │       │       ├── UsersList.test.jsx
    │       │       └── __snapshots__
    │       │           ├── AddUser.test.jsx.snap
    │       │           └── UsersList.test.jsx.snap
    │       ├── index.js
    │       ├── logo.svg
    │       ├── registerServiceWorker.js
    │       └── setupTests.js
    ├── nginx
    │   ├── Dockerfile-dev
    │   ├── Dockerfile-prod
    │   ├── dev.conf
    │   └── prod.conf
    └── users
        ├── Dockerfile-dev
        ├── Dockerfile-prod
        ├── entrypoint-prod.sh
        ├── entrypoint.sh
        ├── htmlcov
        ├── manage.py
        ├── project
        │   ├── __init__.py
        │   ├── api
        │   │   ├── __init__.py
```

```
    │   │   ├── models.py
    │   │   ├── templates
    │   │   │   └── index.html
    │   │   └── users.py
    │   ├── config.py
    │   ├── db
    │   │   ├── Dockerfile
    │   │   └── create.sql
    │   └── tests
    │       ├── __init__.py
    │       ├── base.py
    │       ├── test_config.py
    │       └── test_users.py
    └── requirements.txt
```

Code for part 2: https://github.com/testdrivenio/testdriven-app-2.2/releases/tag/part2

# Part 3

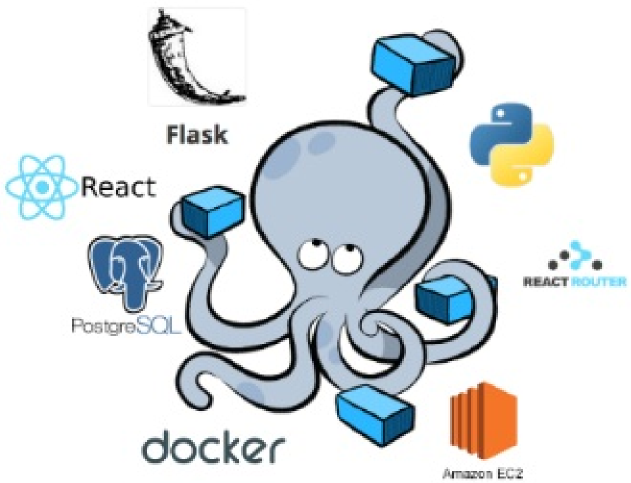
In part 3, we'll add *database migrations* along with *password hashing* in order to implement *token-based authentication* to the `users` service with JSON Web Tokens (JWTs). We'll then turn our attention to the client-side and add *React Router* to the React app to enable client-side routing along with client-side authentication.

## Objectives

By the end of part 3, you will be able to...

1. Use Flask Migrate to handle database migrations
2. Configure Flask Bcrypt for password hashing
3. Implement user authentication with JWTs
4. Write tests to create and verify JWTs and user authentication
5. Use React Router to define client-side routes in React
6. Build UI components with React Bootstrap
7. Explain the difference between user authentication and authorization
8. Test user interactions with Jest and Enzyme
9. Implement user authorization

## App




Check out the live apps, running on EC2 -

1. Production
2. Staging

You can also test out the following endpoints...

| Endpoint | HTTP Method | Authenticated? | Result |
|---|---|---|---|
| /auth/register | POST | No | register user |

| | | | |
|---|---|---|---|
| /auth/login | POST | No | log in user |
| /auth/logout | GET | Yes | log out user |
| /auth/status | GET | Yes | check user status |
| /users | GET | No | get all users |
| /users/:id | GET | No | get single user |
| /users | POST | Yes (admin) | add a user |
| /users/ping | GET | No | sanity check |

Finished code for part 3: https://github.com/testdrivenio/testdriven-app-2.2/releases/tag/part3

## Dependencies

You will use the following dependencies in part 3:

1. Flask-Migrate v2.1.1
2. Flask-Bcrypt v0.7.1
3. PyJWT v1.5.3
4. react-router-dom v4.2.2
5. React Bootstrap v0.32.1
6. React Router Bootstrap v0.24.4

# Flask Migrate

In this lesson, we'll utilize Flask Migrate to handle database migrations...

---

Set `testdriven-dev` as the active Docker Machine:

```
$ docker-machine env testdriven-dev
$ eval $(docker-machine env testdriven-dev)
```

Update the containers:

```
$ docker-compose -f docker-compose-dev.yml up -d
```

Ensure the app is working in the browser, and then run the tests:

```
$ docker-compose -f docker-compose-dev.yml \
  run users python manage.py test

$ docker-compose -f docker-compose-dev.yml \
    run client npm test
```

## Model

Let's make a few changes to the schema in *services/users/project/api/models.py*:

1. `username` must be unique
2. `email` must be unique

We'll also add a password field (in an upcoming lesson), which will be hashed before it's added to the database:

```
password = db.Column(db.String(255), nullable=False)
```

Don't make any changes just yet. Let's start with some tests. Add a new file to "services/users/project/tests" called *test_user_model.py*. This file will hold tests related to our database model:

```
# services/users/project/tests/test_user_model.py

import unittest

from project import db
from project.api.models import User
```

```
from project.tests.base import BaseTestCase


class TestUserModel(BaseTestCase):

    def test_add_user(self):
        user = User(
            username='justatest',
            email='test@test.com',
        )
        db.session.add(user)
        db.session.commit()
        self.assertTrue(user.id)
        self.assertEqual(user.username, 'justatest')
        self.assertEqual(user.email, 'test@test.com')
        self.assertTrue(user.active)

    def test_add_user_duplicate_username(self):
        user = User(
            username='justatest',
            email='test@test.com',
        )
        db.session.add(user)
        db.session.commit()
        duplicate_user = User(
            username='justatest',
            email='test@test2.com',
        )
        db.session.add(duplicate_user)
        self.assertRaises(IntegrityError, db.session.commit)

    def test_add_user_duplicate_email(self):
        user = User(
            username='justatest',
            email='test@test.com',
        )
        db.session.add(user)
        db.session.commit()
        duplicate_user = User(
            username='justanothertest',
            email='test@test.com',
        )
        db.session.add(duplicate_user)
        self.assertRaises(IntegrityError, db.session.commit)

    def test_to_json(self):
      user = User(
          username='justatest',
          email='test@test.com',
      )
      db.session.add(user)
```

```
        db.session.commit()
        self.assertTrue(isinstance(user.to_json(), dict))

if __name__ == '__main__':
    unittest.main()
```

Notice how we didn't invoke `db.session.commit` the second time, when adding a user. Instead, we passed `db.session.commit` to `assertRaises()` and let `assertRaises()` invoke it and assert that the exception was raised.

It's worth nothing that you could use `assertRaises` as a context manager instead:

```
with self.assertRaises(IntegrityError):
    db.session.commit()
```

Add the import:

```
from sqlalchemy.exc import IntegrityError
```

Run the tests. You should see two failures:

```
test_add_user_duplicate_email (test_user_model.TestUserModel) ... FAIL
test_add_user_duplicate_username (test_user_model.TestUserModel) ... FAIL
```

Error:

```
NameError: name 'add_user' is not defined
AssertionError: IntegrityError not raised by do
AssertionError: IntegrityError not raised by do
```

## Flask Migrate Setup

Since we need to make a schema change, add Flask-Migrate to the *requirements.txt* file:

```
flask-migrate==2.1.1
```

In *services/users/project/__init__.py*, add the import, create a new instance, and update `create_app()`:

```
# services/users/project/__init__.py


import os

from flask import Flask
```

```python
from flask_sqlalchemy import SQLAlchemy
from flask_debugtoolbar import DebugToolbarExtension
from flask_cors import CORS
from flask_migrate import Migrate


# instantiate the extensions
db = SQLAlchemy()
toolbar = DebugToolbarExtension()
migrate = Migrate()


def create_app(script_info=None):

    # instantiate the app
    app = Flask(__name__)

    # enable CORS
    CORS(app)

    # set config
    app_settings = os.getenv('APP_SETTINGS')
    app.config.from_object(app_settings)

    # set up extensions
    db.init_app(app)
    toolbar.init_app(app)
    migrate.init_app(app, db)

    # register blueprints
    from project.api.users import users_blueprint
    app.register_blueprint(users_blueprint)

    # shell context for flask cli
    app.shell_context_processor({'app': app, 'db': db})
    return app
```

Before we create the migrations, update the *.dockerignore*:

```
env
htmlcov
.dockerignore
Dockerfile-dev
Dockerfile-prod
migrations
```

Then, update the containers:

```
$ docker-compose -f docker-compose-dev.yml up -d --build
```

Generate the migrations folder, add the initial migration, and then apply it to the database:

```
$ docker-compose -f docker-compose-dev.yml \
  run users python manage.py db init

$ docker-compose -f docker-compose-dev.yml \
  run users python manage.py db migrate

$ docker-compose -f docker-compose-dev.yml \
  run users python manage.py db upgrade
```

> Review the Flask-Migrate documentation for more info on the above commands.

Now, we can make the changes to the schema:

```
class User(db.Model):
    __tablename__ = "users"
    id = db.Column(db.Integer, primary_key=True, autoincrement=True)
    username = db.Column(db.String(128), unique=True, nullable=False)
    email = db.Column(db.String(128), unique=True, nullable=False)
    active = db.Column(db.Boolean, default=True, nullable=False)
```

Again, run:

```
$ docker-compose -f docker-compose-dev.yml \
  run users python manage.py db migrate

$ docker-compose -f docker-compose-dev.yml \
  run users python manage.py db upgrade
```

> Keep in mind that if you have any duplicate usernames and/or emails already in your database, you will get an error when trying to apply the migration to the database. You can either update the data or drop the database and start over.

Hop into psql to ensure the table was updated:

```
$ docker exec -ti users-db psql -U postgres -W
```

Then:

```
# \c users_dev
# \d+ users
```

You should see the unique constraints:

```
Indexes:
```

```
    "users_pkey" PRIMARY KEY, btree (id)
    "users_email_key" UNIQUE CONSTRAINT, btree (email)
    "users_username_key" UNIQUE CONSTRAINT, btree (username)
```

Run the tests again. You should now just have a single error:

```
NameError: name 'add_user' is not defined
```

## Refactor

Now is a good time to do some refactoring...

Did you notice that we added a new user a number of times in the *test_user_model.py* tests? Let's abstract out the `add_user` helper function from *test_users.py* to a utility file so we can use it in both test files.

Add a new file called *utils.py* to "tests":

```
# services/users/project/tests/utils.py


from project import db
from project.api.models import User


def add_user(username, email):
    user = User(username=username, email=email)
    db.session.add(user)
    db.session.commit()
    return user
```

Then remove the helper from *test_users.py* and add the import to the same file:

```
from project.tests.utils import add_user
```

Refactor *test_user_model.py* like so:

```
# services/users/project/tests/test_user_model.py

import unittest

from sqlalchemy.exc import IntegrityError

from project import db
from project.api.models import User
from project.tests.base import BaseTestCase
from project.tests.utils import add_user
```

```
class TestUserModel(BaseTestCase):

    def test_add_user(self):
        user = add_user('justatest', 'test@test.com')
        self.assertTrue(user.id)
        self.assertEqual(user.username, 'justatest')
        self.assertEqual(user.email, 'test@test.com')
        self.assertTrue(user.active)

    def test_add_user_duplicate_username(self):
        add_user('justatest', 'test@test.com')
        duplicate_user = User(
            username='justatest',
            email='test@test2.com',
        )
        db.session.add(duplicate_user)
        self.assertRaises(IntegrityError, db.session.commit)

    def test_add_user_duplicate_email(self):
        add_user('justatest', 'test@test.com')
        duplicate_user = User(
            username='justatest2',
            email='test@test.com',
        )
        db.session.add(duplicate_user)
        self.assertRaises(IntegrityError, db.session.commit)

    def test_to_json(self):
        user = add_user('justatest', 'test@test.com')
        self.assertTrue(isinstance(user.to_json(), dict))


if __name__ == '__main__':
    unittest.main()
```

Run the tests again to ensure nothing broke from the refactor:

```
----------------------------------------------------------------------
Ran 19 tests in 0.512s
```

What about flake8?

```
$ docker-compose -f docker-compose-dev.yml \
  run users flake8 project
```

Correct any issues, and then commit and push your code to GitHub. Make sure the Travis build passes.

# Flask Bcrypt

In this lesson, we'll add support for password hashing...

---

## Flask Bcrypt Setup

To manage password hashing, we'll use the Flask-Bcrypt extension. Add it to the *requirements.txt* file like so:

```
flask-bcrypt==0.7.1
```

> Feel free to swap out Flask-Bcrypt for Werkzeug's password hashing helpers if that's what your prefer. Review this thread for more info on the differences.

Next, wire it up to the app in *services/users/project/__init__.py*:

```python
# services/users/project/__init__.py


import os

from flask import Flask
from flask_sqlalchemy import SQLAlchemy
from flask_debugtoolbar import DebugToolbarExtension
from flask_cors import CORS
from flask_migrate import Migrate
from flask_bcrypt import Bcrypt


# instantiate the extensions
db = SQLAlchemy()
toolbar = DebugToolbarExtension()
migrate = Migrate()
bcrypt = Bcrypt()


def create_app(script_info=None):

    # instantiate the app
    app = Flask(__name__)

    # enable CORS
    CORS(app)

    # set config
    app_settings = os.getenv('APP_SETTINGS')
```

```
    app.config.from_object(app_settings)

    # set up extensions
    db.init_app(app)
    toolbar.init_app(app)
    migrate.init_app(app, db)
    bcrypt.init_app(app)

    # register blueprints
    from project.api.users import users_blueprint
    app.register_blueprint(users_blueprint)

    # shell context for flask cli
    app.shell_context_processor({'app': app, 'db': db})
    return app
```

Before we update the model, add the following test to *test_user_model.py*:

```
def test_passwords_are_random(self):
    user_one = add_user('justatest', 'test@test.com', 'greaterthaneight')
    user_two = add_user('justatest2', 'test@test2.com', 'greaterthaneight')
    self.assertNotEqual(user_one.password, user_two.password)
```

Update the helper to take a password:

```
def add_user(username, email, password):
    user = User(username=username, email=email, password=password)
    db.session.add(user)
    db.session.commit()
    return user
```

Make sure to pass in an argument for all instances of `add_user()` and `User()` as well as in the payload for POST requests to `/users` and `/` in both *test_user_model.py* and *test_users.py*. Do this now.

Examples:

```
response = self.client.post(
    '/users',
    data=json.dumps({
        'username': 'michael',
        'email': 'michael@mherman.org',
        'password': 'greaterthaneight'
    }),
    content_type='application/json',
)

...
```

```
user = add_user('justatest', 'test@test.com', 'greaterthaneight')

...

duplicate_user = User(
    username='justatest',
    email='test@test2.com',
    password='greaterthaneight'
)
```

Update the containers:

```
$ docker-compose -f docker-compose-dev.yml up -d --build
```

Run the tests:

```
$ docker-compose -f docker-compose-dev.yml \
  run users python manage.py test
```

You should see a number of failures:

```
TypeError: __init__() got an unexpected keyword argument 'password'
```

To get them green, first add the password field to the model in *services/users/project/api/models.py*:

```
class User(db.Model):
    __tablename__ = "users"
    id = db.Column(db.Integer, primary_key=True, autoincrement=True)
    username = db.Column(db.String(128), unique=True, nullable=False)
    email = db.Column(db.String(128), unique=True, nullable=False)
    password = db.Column(db.String(255), nullable=False)
    active = db.Column(db.Boolean, default=True, nullable=False)

    def __init__(self, username, email, password):
        self.username = username
        self.email = email
        self.password = bcrypt.generate_password_hash(password).decode()
```

Then, add the `bcrypt` import:

```
from project import db, bcrypt
```

Run the tests again. More failures, right?

```
TypeError: __init__() missing 1 required positional argument: 'password'
```

Apply the migrations:

```
$ docker-compose -f docker-compose-dev.yml \
  run users python manage.py db migrate

$ docker-compose -f docker-compose-dev.yml \
  run users python manage.py db upgrade
```

Did you get this error?

```
sqlalchemy.exc.IntegrityError: (psycopg2.IntegrityError) column "password"
contains null values
[SQL: 'ALTER TABLE users ADD COLUMN password VARCHAR(255) NOT NULL']
(Background on this error at: http://sqlalche.me/e/gkpj)
```

If so, you can either-

1. Remove all users in psql - `delete from users;`
2. Update the actual migration file to create the new field without adding the non-nullable constraint, modify the data, and then add the constraint:

   ```
   def upgrade():
       # ### commands auto generated by Alembic - please adjust! ###
       op.add_column('users', sa.Column('password', sa.String(length=255)))
       op.execute('UPDATE users SET password=email')
       op.alter_column('users', 'password', nullable=False)
       # ### end Alembic commands ###
   ```

> Since we're operating in development mode at this point, it really does not matter which one you go with. Take a minute to think through how you'd handle this on a live database, though.

Moving on, update `add_user()` in *services/users/project/api/users.py*:

```
@users_blueprint.route('/users', methods=['POST'])
def add_user():
    post_data = request.get_json()
    response_object = {
        'status': 'fail',
        'message': 'Invalid payload.'
    }
    if not post_data:
        return jsonify(response_object), 400
    username = post_data.get('username')
    email = post_data.get('email')
    password = post_data.get('password')
    try:
        user = User.query.filter_by(email=email).first()
```

```
        if not user:
            db.session.add(User(
                username=username, email=email, password=password))
            db.session.commit()
            response_object['status'] = 'success'
            response_object['message'] = f'{email} was added!'
            return jsonify(response_object), 201
        else:
            response_object['message'] = 'Sorry. That email already exists.'
            return jsonify(response_object), 400
    except exc.IntegrityError as e:
        db.session.rollback()
        return jsonify(response_object), 400
```

Also, update `index()` :

```
@users_blueprint.route('/', methods=['GET', 'POST'])
def index():
    if request.method == 'POST':
        username = request.form['username']
        email = request.form['email']
        password = request.form['password']
        db.session.add(User(username=username, email=email, password=password))
        db.session.commit()
    users = User.query.all()
    return render_template('index.html', users=users)
```

The tests should now pass:

```
----------------------------------------------------------------------
Ran 20 tests in 6.121s
```

Turning back to the API, what if we don't pass a password into the payload? Write a test!

*test_users.py*:

```
def test_add_user_invalid_json_keys_no_password(self):
    """
    Ensure error is thrown if the JSON object
    does not have a password key.
    """
    with self.client:
        response = self.client.post(
            '/users',
            data=json.dumps(dict(
                username='michael',
                email='michael@reallynotreal.com')),
            content_type='application/json',
```

```
        )
        data = json.loads(response.data.decode())
        self.assertEqual(response.status_code, 400)
        self.assertIn('Invalid payload.', data['message'])
        self.assertIn('fail', data['status'])
```

You should see the following error when the tests are ran:

```
raise ValueError('Password must be non-empty.')
ValueError: Password must be non-empty.
```

To fix, add a another exception handler to the try/except block in the `add_user` view handler:

```
except (exc.IntegrityError, ValueError) as e:
    db.session.rollback()
    return jsonify(response_object), 400
```

Test again. Then, update the following test in *test_user_model.py*, asserting the user object has a password field:

```
def test_add_user(self):
    user = add_user('justatest', 'test@test.com', 'test')
    self.assertTrue(user.id)
    self.assertEqual(user.username, 'justatest')
    self.assertEqual(user.email, 'test@test.com')
    self.assertTrue(user.active)
    self.assertTrue(user.password)
```

## Log Rounds

Finally, did you notice that the tests are running *much* slower than before? This is due to the `BCRYPT_LOG_ROUNDS` setting for Flask Bcrypt. Since we have not defined a value yet in the app config, Flask Bcrypt uses the default value of 12, which is unnecessarily high for a test environment.

Update the test specs in *services/users/project/tests/test_config.py*:

```
class TestDevelopmentConfig(TestCase):
    def create_app(self):
        app.config.from_object('project.config.DevelopmentConfig')
        return app

    def test_app_is_development(self):
        self.assertTrue(app.config['SECRET_KEY'] == 'my_precious')
        self.assertFalse(current_app is None)
        self.assertTrue(
            app.config['SQLALCHEMY_DATABASE_URI'] ==
            os.environ.get('DATABASE_URL')
```

```
        )
        self.assertTrue(app.config['DEBUG_TB_ENABLED'])
        self.assertTrue(app.config['BCRYPT_LOG_ROUNDS'] == 4)


class TestTestingConfig(TestCase):
    def create_app(self):
        app.config.from_object('project.config.TestingConfig')
        return app

    def test_app_is_testing(self):
        self.assertTrue(app.config['SECRET_KEY'] == 'my_precious')
        self.assertTrue(app.config['TESTING'])
        self.assertFalse(app.config['PRESERVE_CONTEXT_ON_EXCEPTION'])
        self.assertTrue(
            app.config['SQLALCHEMY_DATABASE_URI'] ==
            os.environ.get('DATABASE_TEST_URL')
        )
        self.assertFalse(app.config['DEBUG_TB_ENABLED'])
        self.assertTrue(app.config['BCRYPT_LOG_ROUNDS'] == 4)


class TestProductionConfig(TestCase):
    def create_app(self):
        app.config.from_object('project.config.ProductionConfig')
        return app

    def test_app_is_production(self):
        self.assertTrue(app.config['SECRET_KEY'] == 'my_precious')
        self.assertFalse(app.config['TESTING'])
        self.assertFalse(app.config['DEBUG_TB_ENABLED'])
        self.assertTrue(app.config['BCRYPT_LOG_ROUNDS'] == 13)
```

Make sure the tests fail, then update *services/users/project/config.py*:

```
# services/users/project/config.py


import os


class BaseConfig:
    """Base configuration"""
    DEBUG = False
    TESTING = False
    SQLALCHEMY_TRACK_MODIFICATIONS = False
    SECRET_KEY = 'my_precious'
    BCRYPT_LOG_ROUNDS = 13
```

```python
class DevelopmentConfig(BaseConfig):
    """Development configuration"""
    DEBUG = True
    SQLALCHEMY_DATABASE_URI = os.environ.get('DATABASE_URL')
    BCRYPT_LOG_ROUNDS = 4


class TestingConfig(BaseConfig):
    """Testing configuration"""
    DEBUG = True
    TESTING = True
    SQLALCHEMY_DATABASE_URI = os.environ.get('DATABASE_TEST_URL')
    BCRYPT_LOG_ROUNDS = 4


class ProductionConfig(BaseConfig):
    """Production configuration"""
    DEBUG = False
    SQLALCHEMY_DATABASE_URI = os.environ.get('DATABASE_URL')
```

Then, update `__init__` from the `User` model:

```python
class User(db.Model):
    __tablename__ = "users"
    id = db.Column(db.Integer, primary_key=True, autoincrement=True)
    username = db.Column(db.String(128), unique=True, nullable=False)
    email = db.Column(db.String(128), unique=True, nullable=False)
    password = db.Column(db.String(255), nullable=False)
    active = db.Column(db.Boolean, default=True, nullable=False)

    def __init__(self, username, email, password):
        self.username = username
        self.email = email
        self.password = bcrypt.generate_password_hash(
            password, current_app.config.get('BCRYPT_LOG_ROUNDS')
        ).decode()
```

Don't forget the import:

```python
from flask import current_app
```

Run the tests again!

1. Do they pass?
2. Are they faster? (0.371s vs 4.322s on my end)

> Need help deciding how many rounds to use in production? Check out this Stack Exchange article.

Commit, then push your code to GitHub. Make sure the Travis build passes. With that, let's get JWTs up and running...

# JWT Setup

In this lesson, we'll add JWT to the `users` service...

---

If you're new to JWTs and/or token-based authentication, review the Introduction of the Token-Based Authentication With Flask post. How We Solved Authentication and Authorization in Our Microservice Architecture is an excellent read as well.

The auth workflow works as follows:

1. The end user submits login credentials from the `client` to the `users` service via AJAX
2. The `users` service then verifies that the credentials are valid and responds with an auth token
3. The token is stored on the client and is sent with all subsequent requests, where the `users` service decodes the token and validates it

Tokens have three main parts:

1. Header
2. Payload
3. Signature

> If you're curious, you can read more about each part from Introduction to JSON Web Tokens.

## PyJWT

To work with JSON Web Tokens in our app, add the PyJWT package to the *requirements.txt* file:

```
pyjwt==1.5.3
```

### Encode Token

Add the following test to `TestUserModel()` in *services/users/project/tests/test_user_model.py*:

```
def test_encode_auth_token(self):
    user = add_user('justatest', 'test@test.com', 'test')
    auth_token = user.encode_auth_token(user.id)
    self.assertTrue(isinstance(auth_token, bytes))
```

As always, make sure the tests fail. Next, add the `encode_auth_token` method to the `User()` class in *models.py*:

```
def encode_auth_token(self, user_id):
    """Generates the auth token"""
    try:
        payload = {
```

```
                'exp': datetime.datetime.utcnow() + datetime.timedelta(
                    days=0, seconds=5),
                'iat': datetime.datetime.utcnow(),
                'sub': user_id
            }
            return jwt.encode(
                payload,
                current_app.config.get('SECRET_KEY'),
                algorithm='HS256'
            )
        except Exception as e:
            return e
```

Given a user id, `encode_auth_token` encodes and returns a token. Take note of the payload. This is where we add metadata about the token and information about the user. This info is often referred to as JWT Claims. We utilized the following "claims":

1. `exp` : token expiration date
2. `iat` (issued at): token generation date
3. `sub` : the subject of the token e.g., - the user whom it identifies

Add the following imports:

1. `import datetime`
2. `import jwt`

Update the containers:

```
$ docker-compose -f docker-compose-dev.yml up -d --build
```

Run the tests. They should pass, right?

Turn to the app config. The secret key needs to be updated for production. Let's configure it with an environment variable.

First, within *test_config.py*, change all instances of:

```
self.assertTrue(app.config['SECRET_KEY'] == 'my_precious')
```

To:

```
self.assertTrue(
    app.config['SECRET_KEY'] ==
    os.environ.get('SECRET_KEY'))
```

Then update `BaseConfig` in *services/users/project/config.py*:

```
class BaseConfig:
```

```
    """Base configuration"""
    TESTING = False
    SQLALCHEMY_TRACK_MODIFICATIONS = False
    SECRET_KEY = os.environ.get('SECRET_KEY')
    DEBUG_TB_ENABLED = False
    DEBUG_TB_INTERCEPT_REDIRECTS = False
    BCRYPT_LOG_ROUNDS = 13
```

Add the `SECRET_KEY` environment variable to `environment` within *docker-compose-dev.yml*:

```
environment:
  - FLASK_DEBUG=1
  - APP_SETTINGS=project.config.DevelopmentConfig
  - DATABASE_URL=postgres://postgres:postgres@users-db:5432/users_dev
  - DATABASE_TEST_URL=postgres://postgres:postgres@users-db:5432/users_test
  - SECRET_KEY=my_precious
```

Let's also add the token expiration to the config:

```
class BaseConfig:
    """Base configuration"""
    TESTING = False
    SQLALCHEMY_TRACK_MODIFICATIONS = False
    SECRET_KEY = os.environ.get('SECRET_KEY')
    DEBUG_TB_ENABLED = False
    DEBUG_TB_INTERCEPT_REDIRECTS = False
    BCRYPT_LOG_ROUNDS = 13
    TOKEN_EXPIRATION_DAYS = 30
    TOKEN_EXPIRATION_SECONDS = 0

class DevelopmentConfig(BaseConfig):
    """Development configuration"""
    SQLALCHEMY_DATABASE_URI = os.environ.get('DATABASE_URL')
    DEBUG_TB_ENABLED = True
    BCRYPT_LOG_ROUNDS = 4


class TestingConfig(BaseConfig):
    """Testing configuration"""
    TESTING = True
    SQLALCHEMY_DATABASE_URI = os.environ.get('DATABASE_TEST_URL')
    BCRYPT_LOG_ROUNDS = 4
    TOKEN_EXPIRATION_DAYS = 0
    TOKEN_EXPIRATION_SECONDS = 3


class ProductionConfig(BaseConfig):
    """Production configuration"""
    SQLALCHEMY_DATABASE_URI = os.environ.get('DATABASE_URL')
```

Update the tests:

```python
class TestDevelopmentConfig(TestCase):
    def create_app(self):
        app.config.from_object('project.config.DevelopmentConfig')
        return app

    def test_app_is_development(self):
        self.assertTrue(
            app.config['SECRET_KEY'] ==
            os.environ.get('SECRET_KEY'))
        self.assertFalse(current_app is None)
        self.assertTrue(
            app.config['SQLALCHEMY_DATABASE_URI'] ==
            os.environ.get('DATABASE_URL')
        )
        self.assertTrue(app.config['DEBUG_TB_ENABLED'])
        self.assertTrue(app.config['BCRYPT_LOG_ROUNDS'] == 4)
        self.assertTrue(app.config['TOKEN_EXPIRATION_DAYS'] == 30)
        self.assertTrue(app.config['TOKEN_EXPIRATION_SECONDS'] == 0)


class TestTestingConfig(TestCase):
    def create_app(self):
        app.config.from_object('project.config.TestingConfig')
        return app

    def test_app_is_testing(self):
        self.assertTrue(
            app.config['SECRET_KEY'] ==
            os.environ.get('SECRET_KEY'))
        self.assertTrue(app.config['TESTING'])
        self.assertFalse(app.config['PRESERVE_CONTEXT_ON_EXCEPTION'])
        self.assertTrue(
            app.config['SQLALCHEMY_DATABASE_URI'] ==
            os.environ.get('DATABASE_TEST_URL')
        )
        self.assertFalse(app.config['DEBUG_TB_ENABLED'])
        self.assertTrue(app.config['BCRYPT_LOG_ROUNDS'] == 4)
        self.assertTrue(app.config['TOKEN_EXPIRATION_DAYS'] == 0)
        self.assertTrue(app.config['TOKEN_EXPIRATION_SECONDS'] == 3)


class TestProductionConfig(TestCase):
    def create_app(self):
        app.config.from_object('project.config.ProductionConfig')
        return app

    def test_app_is_production(self):
```

```
        self.assertTrue(
            app.config['SECRET_KEY'] ==
            os.environ.get('SECRET_KEY'))
        self.assertFalse(app.config['TESTING'])
        self.assertFalse(app.config['DEBUG_TB_ENABLED'])
        self.assertTrue(app.config['BCRYPT_LOG_ROUNDS'] == 13)
        self.assertTrue(app.config['TOKEN_EXPIRATION_DAYS'] == 30)
        self.assertTrue(app.config['TOKEN_EXPIRATION_SECONDS'] == 0)
```

Then update the `encode_auth_token` in the model:

```
def encode_auth_token(self, user_id):
    """Generates the auth token"""
    try:
        payload = {
            'exp': datetime.datetime.utcnow() + datetime.timedelta(
                days=current_app.config.get('TOKEN_EXPIRATION_DAYS'),
                seconds=current_app.config.get('TOKEN_EXPIRATION_SECONDS')
            ),
            'iat': datetime.datetime.utcnow(),
            'sub': user_id
        }
        return jwt.encode(
            payload,
            current_app.config.get('SECRET_KEY'),
            algorithm='HS256'
        )
    except Exception as e:
        return e
```

> Now is a great time to check your understanding: See if you can write the test as well as the code for decoding a token on your own.

### Decode Token

Moving on, add the following test to *test_user_model.py* for decoding a token:

```
def test_decode_auth_token(self):
    user = add_user('justatest', 'test@test.com', 'test')
    auth_token = user.encode_auth_token(user.id)
    self.assertTrue(isinstance(auth_token, bytes))
    self.assertEqual(User.decode_auth_token(auth_token), user.id)
```

Add the following method to the `User()` class:

```
@staticmethod
def decode_auth_token(auth_token):
    """
```

```
    Decodes the auth token - :param auth_token: - :return: integer|string
    """
    try:
        payload = jwt.decode(
            auth_token, current_app.config.get('SECRET_KEY'))
        return payload['sub']
    except jwt.ExpiredSignatureError:
        return 'Signature expired. Please log in again.'
    except jwt.InvalidTokenError:
        return 'Invalid token. Please log in again.'
```

Again, every authenticated request *must* include the auth token to verity the user's authenticity. Make sure the tests pass before moving on.

# Auth Routes

Now we can configure the authentication routes...

---

Before writing any code, let's ensure that the test coverage does not decrease as we add the new routes. Where are we at right now?

```
Coverage Summary:
Name                        Stmts   Miss Branch BrPart  Cover
----------------------------------------------------------
project/__init__.py            24     12      0      0    50%
project/api/models.py          31     21      2      0    30%
project/api/users.py           50     10     10      0    83%
----------------------------------------------------------
TOTAL                         105     43     12      0    62%
```

## Routes Setup

We'll set up the following routes...

| Endpoint | HTTP Method | Authenticated? | Result |
|---|---|---|---|
| /auth/register | POST | No | register a user |
| /auth/login | POST | No | log in a user |
| /auth/logout | GET | Yes | log out a user |
| /auth/status | GET | Yes | get user status |

Add a new file to the "services/users/project/api" directory called *auth.py*:

```
# services/users/project/api/auth.py


from flask import Blueprint, jsonify, request
from sqlalchemy import exc, or_

from project.api.models import User
from project import db, bcrypt


auth_blueprint = Blueprint('auth', __name__)
```

Then, register the new Blueprint with the app in *services/users/project/__init__.py*:

```
...
```

```
# register blueprints
from project.api.users import users_blueprint
app.register_blueprint(users_blueprint)
from project.api.auth import auth_blueprint
app.register_blueprint(auth_blueprint)
...
```

Add a new file called *test_auth.py* to the "tests" folder to hold all tests associated with the Blueprint:

```
# services/users/project/tests/test_auth.py


import json

from project import db
from project.api.models import User
from project.tests.base import BaseTestCase
from project.tests.utils import add_user


class TestAuthBlueprint(BaseTestCase):
    pass
```

## Register Route

Start with a test:

```
def test_user_registration(self):
    with self.client:
        response = self.client.post(
            '/auth/register',
            data=json.dumps({
                'username': 'justatest',
                'email': 'test@test.com',
                'password': '123456',
            }),
            content_type='application/json'
        )
        data = json.loads(response.data.decode())
        self.assertTrue(data['status'] == 'success')
        self.assertTrue(data['message'] == 'Successfully registered.')
        self.assertTrue(data['auth_token'])
        self.assertTrue(response.content_type == 'application/json')
        self.assertEqual(response.status_code, 201)
```

This only tests the happy path. What about failures?

1. email already exists

2. username already exists
3. invalid payload (empty, no username, no email, no password)

```
def test_user_registration_duplicate_email(self):
    add_user('test', 'test@test.com', 'test')
    with self.client:
        response = self.client.post(
            '/auth/register',
            data=json.dumps({
                'username': 'michael',
                'email': 'test@test.com',
                'password': 'test'
            }),
            content_type='application/json',
        )
        data = json.loads(response.data.decode())
        self.assertEqual(response.status_code, 400)
        self.assertIn(
            'Sorry. That user already exists.', data['message'])
        self.assertIn('fail', data['status'])

def test_user_registration_duplicate_username(self):
    add_user('test', 'test@test.com', 'test')
    with self.client:
        response = self.client.post(
            '/auth/register',
            data=json.dumps({
                'username': 'test',
                'email': 'test@test.com2',
                'password': 'test'
            }),
            content_type='application/json',
        )
        data = json.loads(response.data.decode())
        self.assertEqual(response.status_code, 400)
        self.assertIn(
            'Sorry. That user already exists.', data['message'])
        self.assertIn('fail', data['status'])

def test_user_registration_invalid_json(self):
    with self.client:
        response = self.client.post(
            '/auth/register',
            data=json.dumps({}),
            content_type='application/json'
        )
        data = json.loads(response.data.decode())
        self.assertEqual(response.status_code, 400)
        self.assertIn('Invalid payload.', data['message'])
        self.assertIn('fail', data['status'])
```

```
def test_user_registration_invalid_json_keys_no_username(self):
    with self.client:
        response = self.client.post(
            '/auth/register',
            data=json.dumps({
                'email': 'test@test.com',
                'password': 'test'
            }),
            content_type='application/json',
        )
        data = json.loads(response.data.decode())
        self.assertEqual(response.status_code, 400)
        self.assertIn('Invalid payload.', data['message'])
        self.assertIn('fail', data['status'])

def test_user_registration_invalid_json_keys_no_email(self):
    with self.client:
        response = self.client.post(
            '/auth/register',
            data=json.dumps({
                'username': 'justatest',
                'password': 'test'
            }),
            content_type='application/json',
        )
        data = json.loads(response.data.decode())
        self.assertEqual(response.status_code, 400)
        self.assertIn('Invalid payload.', data['message'])
        self.assertIn('fail', data['status'])

def test_user_registration_invalid_json_keys_no_password(self):
    with self.client:
        response = self.client.post(
            '/auth/register',
            data=json.dumps({
                'username': 'justatest',
                'email': 'test@test.com'
            }),
            content_type='application/json',
        )
        data = json.loads(response.data.decode())
        self.assertEqual(response.status_code, 400)
        self.assertIn('Invalid payload.', data['message'])
        self.assertIn('fail', data['status'])
```

Ensure the tests fail, and then add the view:

```
@auth_blueprint.route('/auth/register', methods=['POST'])
def register_user():
```

```
    # get post data
    post_data = request.get_json()
    response_object = {
        'status': 'fail',
        'message': 'Invalid payload.'
    }
    if not post_data:
        return jsonify(response_object), 400
    username = post_data.get('username')
    email = post_data.get('email')
    password = post_data.get('password')
    try:
        # check for existing user
        user = User.query.filter(
            or_(User.username == username, User.email == email)).first()
        if not user:
            # add new user to db
            new_user = User(
                username=username,
                email=email,
                password=password
            )
            db.session.add(new_user)
            db.session.commit()
            # generate auth token
            auth_token = new_user.encode_auth_token(new_user.id)
            response_object['status'] = 'success'
            response_object['message'] = 'Successfully registered.'
            response_object['auth_token'] = auth_token.decode()
            return jsonify(response_object), 201
        else:
            response_object['message'] = 'Sorry. That user already exists.'
            return jsonify(response_object), 400
    # handler errors
    except (exc.IntegrityError, ValueError) as e:
        db.session.rollback()
        return jsonify(response_object), 400
```

Be sure the tests pass!

## Login Route

Again, start with a few tests:

```
def test_registered_user_login(self):
    with self.client:
        add_user('test', 'test@test.com', 'test')
        response = self.client.post(
            '/auth/login',
```

```
            data=json.dumps({
                'email': 'test@test.com',
                'password': 'test'
            }),
            content_type='application/json'
        )
        data = json.loads(response.data.decode())
        self.assertTrue(data['status'] == 'success')
        self.assertTrue(data['message'] == 'Successfully logged in.')
        self.assertTrue(data['auth_token'])
        self.assertTrue(response.content_type == 'application/json')
        self.assertEqual(response.status_code, 200)

def test_not_registered_user_login(self):
    with self.client:
        response = self.client.post(
            '/auth/login',
            data=json.dumps({
                'email': 'test@test.com',
                'password': 'test'
            }),
            content_type='application/json'
        )
        data = json.loads(response.data.decode())
        self.assertTrue(data['status'] == 'fail')
        self.assertTrue(data['message'] == 'User does not exist.')
        self.assertTrue(response.content_type == 'application/json')
        self.assertEqual(response.status_code, 404)
```

Run the tests. They should fail. Now, add the view:

```
@auth_blueprint.route('/auth/login', methods=['POST'])
def login_user():
    # get post data
    post_data = request.get_json()
    response_object = {
        'status': 'fail',
        'message': 'Invalid payload.'
    }
    if not post_data:
        return jsonify(response_object), 400
    email = post_data.get('email')
    password = post_data.get('password')
    try:
        # fetch the user data
        user = User.query.filter_by(email=email).first()
        if user and bcrypt.check_password_hash(user.password, password):
            auth_token = user.encode_auth_token(user.id)
            if auth_token:
                response_object['status'] = 'success'
```

```
                response_object['message'] = 'Successfully logged in.'
                response_object['auth_token'] = auth_token.decode()
                return jsonify(response_object), 200
        else:
            response_object['message'] = 'User does not exist.'
            return jsonify(response_object), 404
    except Exception as e:
        response_object['message'] = 'Try again.'
        return jsonify(response_object), 500
```

Do the tests pass?

## Logout Route

Test valid logout:

```
def test_valid_logout(self):
    add_user('test', 'test@test.com', 'test')
    with self.client:
        # user login
        resp_login = self.client.post(
            '/auth/login',
            data=json.dumps({
                'email': 'test@test.com',
                'password': 'test'
            }),
            content_type='application/json'
        )
        # valid token logout
        token = json.loads(resp_login.data.decode())['auth_token']
        response = self.client.get(
            '/auth/logout',
            headers={'Authorization': f'Bearer {token}'}
        )
        data = json.loads(response.data.decode())
        self.assertTrue(data['status'] == 'success')
        self.assertTrue(data['message'] == 'Successfully logged out.')
        self.assertEqual(response.status_code, 200)
```

Test invalid logout:

```
def test_invalid_logout_expired_token(self):
    add_user('test', 'test@test.com', 'test')
    with self.client:
        resp_login = self.client.post(
            '/auth/login',
            data=json.dumps({
                'email': 'test@test.com',
                'password': 'test'
```

```
            }),
            content_type='application/json'
        )
        # invalid token logout
        time.sleep(4)
        token = json.loads(resp_login.data.decode())['auth_token']
        response = self.client.get(
            '/auth/logout',
            headers={'Authorization': f'Bearer {token}'}
        )
        data = json.loads(response.data.decode())
        self.assertTrue(data['status'] == 'fail')
        self.assertTrue(
            data['message'] == 'Signature expired. Please log in again.')
        self.assertEqual(response.status_code, 401)

def test_invalid_logout(self):
    with self.client:
        response = self.client.get(
            '/auth/logout',
            headers={'Authorization': 'Bearer invalid'})
        data = json.loads(response.data.decode())
        self.assertTrue(data['status'] == 'fail')
        self.assertTrue(
            data['message'] == 'Invalid token. Please log in again.')
        self.assertEqual(response.status_code, 401)
```

Add the import:

```
import time
```

Update the views:

```
@auth_blueprint.route('/auth/logout', methods=['GET'])
def logout_user():
    # get auth token
    auth_header = request.headers.get('Authorization')
    response_object = {
        'status': 'fail',
        'message': 'Provide a valid auth token.'
    }
    if auth_header:
        auth_token = auth_header.split(' ')[1]
        resp = User.decode_auth_token(auth_token)
        if not isinstance(resp, str):
            response_object['status'] = 'success'
            response_object['message'] = 'Successfully logged out.'
            return jsonify(response_object), 200
        else:
```

```
            response_object['message'] = resp
            return jsonify(response_object), 401
    else:
        return jsonify(response_object), 403
```

Run the tests:

```
----------------------------------------------------------------------
Ran 35 tests in 5.040s

OK
```

Did you notice the `time.sleep(4)` in the `test_invalid_logout_expired_token` test? This adds an additional 4 seconds to our test suite. To speed things up, let's update the `TOKEN_EXPIRATION_SECONDS` for this specific test:

```
def test_invalid_logout_expired_token(self):
    add_user('test', 'test@test.com', 'test')
    current_app.config['TOKEN_EXPIRATION_SECONDS'] = -1
    with self.client:
        resp_login = self.client.post(
            '/auth/login',
            data=json.dumps({
                'email': 'test@test.com',
                'password': 'test'
            }),
            content_type='application/json'
        )
        # invalid token logout
        token = json.loads(resp_login.data.decode())['auth_token']
        response = self.client.get(
            '/auth/logout',
            headers={'Authorization': f'Bearer {token}'}
        )
        data = json.loads(response.data.decode())
        self.assertTrue(data['status'] == 'fail')
        self.assertTrue(
            data['message'] == 'Signature expired. Please log in again.')
        self.assertEqual(response.status_code, 401)
```

Add the import:

```
from flask import current_app
```

You can also remove the `time` import:

```
import time
```

Make sure the tests still pass:

```
----------------------------------------------------------------------
Ran 35 tests in 1.015s

OK
```

## Status Route

Remember: In order to get the user details of the currently logged in user, the auth token *must* be sent with the request.

Start with some tests:

```python
def test_user_status(self):
    add_user('test', 'test@test.com', 'test')
    with self.client:
        resp_login = self.client.post(
            '/auth/login',
            data=json.dumps({
                'email': 'test@test.com',
                'password': 'test'
            }),
            content_type='application/json'
        )
        token = json.loads(resp_login.data.decode())['auth_token']
        response = self.client.get(
            '/auth/status',
            headers={'Authorization': f'Bearer {token}'}
        )
        data = json.loads(response.data.decode())
        self.assertTrue(data['status'] == 'success')
        self.assertTrue(data['data'] is not None)
        self.assertTrue(data['data']['username'] == 'test')
        self.assertTrue(data['data']['email'] == 'test@test.com')
        self.assertTrue(data['data']['active'] is True)
        self.assertEqual(response.status_code, 200)

def test_invalid_status(self):
    with self.client:
        response = self.client.get(
            '/auth/status',
            headers={'Authorization': 'Bearer invalid'})
        data = json.loads(response.data.decode())
        self.assertTrue(data['status'] == 'fail')
        self.assertTrue(
            data['message'] == 'Invalid token. Please log in again.')
        self.assertEqual(response.status_code, 401)
```

The tests should fail. Now, in the route handler, we should:

1. extract the auth token and check its validity
2. grab the user id from the payload and get the user details (if the token is valid, of course)

```
@auth_blueprint.route('/auth/status', methods=['GET'])
def get_user_status():
    # get auth token
    auth_header = request.headers.get('Authorization')
    response_object = {
        'status': 'fail',
        'message': 'Provide a valid auth token.'
    }
    if auth_header:
        auth_token = auth_header.split(' ')[1]
        resp = User.decode_auth_token(auth_token)
        if not isinstance(resp, str):
            user = User.query.filter_by(id=resp).first()
            response_object['status'] = 'success'
            response_object['message'] = 'Success.'
            response_object['data'] = user.to_json()
            return jsonify(response_object), 200
        response_object['message'] = resp
        return jsonify(response_object), 401
    else:
        return jsonify(response_object), 401
```

Test one final time.

```
----------------------------------------------------------------------
Ran 37 tests in 1.144s

OK
```

Then, check coverage:

```
Coverage Summary:
Name                        Stmts   Miss Branch BrPart  Cover
-------------------------------------------------------------
project/__init__.py            26     12      0      0    54%
project/api/auth.py            78     15     18      4    80%
project/api/models.py          31     17      2      0    48%
project/api/users.py           50     10     10      0    83%
-------------------------------------------------------------
TOTAL                         185     54     30      4    73%
```

Finally, update `seed_db()` in *manage.py*:

```
@cli.command()
def seed_db():
    """Seeds the database."""
    db.session.add(User(
        username='michael',
        email='michael@reallynotreal.com',
        password='greaterthaneight'
    ))
    db.session.add(User(
        username='michaelherman',
        email='michael@mherman.org',
        password='greaterthaneight'
    ))
    db.session.commit()
```

Commit and push your code. Do the tests pass on Travis CI?

# React Router

In this lesson, we'll wire up routing in the React app to manage navigation between different components so the end user has unique pages to interact with...

---

Let's add an `/about` route!

At this point, you should already be quite familiar with the concept of routing on the server-side. Well, as the name suggests, client-side routing is the really the same - it's just happening in the browser.

> For more on this, review the excellent Deep dive into client-side routing article.

## Check Your Understanding

> This part is optional but highly recommended.

Put your skills to test!

1. Start a new React App on your own with Create React App in a new project directory.
2. Add two components - `Home` and `Contact`. These should be functional components that just display an `<h2>` element with the name of the component.
3. Follow the official Quick Start guide to add react-router-dom to your app.

## Quick Refactor

Before adding the router, let's move the `App` component out of *index.js* to clean things up. Add an *App.jsx* file to the "src" directory, and then update both files...

*App.jsx*:

```
import React, { Component } from 'react';
import axios from 'axios';

import UsersList from './components/UsersList';
import AddUser from './components/AddUser';

class App extends Component {
  constructor() {
    super();
    this.state = {
      users: [],
      username: '',
      email: ''
    };
    this.addUser = this.addUser.bind(this);
    this.handleChange = this.handleChange.bind(this);
```

```
  };
  componentDidMount() {
    this.getUsers();
  };
  getUsers() {
    axios.get(`${process.env.REACT_APP_USERS_SERVICE_URL}/users`)
    .then((res) => { this.setState({ users: res.data.data.users }); })
    .catch((err) => { console.log(err); });
  };
  addUser(event) {
    event.preventDefault();
    const data = {
      username: this.state.username,
      email: this.state.email
    };
    axios.post(`${process.env.REACT_APP_USERS_SERVICE_URL}/users`, data)
    .then((res) => {
      this.getUsers();
      this.setState({ username: '', email: '' });
    })
    .catch((err) => { console.log(err); });
  }
  handleChange(event) {
    const obj = {};
    obj[event.target.name] = event.target.value;
    this.setState(obj);
  };
  render() {
    return (
      <div className="container">
        <div className="row">
          <div className="col-md-6">
            <br/>
            <h1>All Users</h1>
            <hr/><br/>
            <AddUser
              username={this.state.username}
              email={this.state.email}
              handleChange={this.handleChange}
              addUser={this.addUser}
            />
            <br/>
            <UsersList users={this.state.users}/>
          </div>
        </div>
      </div>
    )
  };

};
```

```
export default App;
```

*index.js*:

```
import React from 'react';
import ReactDOM from 'react-dom';

import App from './App.jsx';

ReactDOM.render(
  <App/>,
  document.getElementById('root')
);
```

Update the containers:

```
$ docker-compose -f docker-compose-dev.yml up -d --build
```

Manually test in the browser, making sure all is well. Then, run the tests:

```
$ docker-compose -f docker-compose-dev.yml \
  run client npm test
```

Finally, let's add a new test to ensure the overall app renders. Create a new file called *App.test.jsx* within the "services/client/src/components/__tests__/" directory:

```
import React from 'react';
import { shallow } from 'enzyme';

import App from '../../App';

test('App renders without crashing', () => {
  const wrapper = shallow(<App/>);
});
```

Make sure the tests still pass!

```
PASS  src/components/__tests__/UsersList.test.jsx
PASS  src/components/__tests__/App.test.jsx
PASS  src/components/__tests__/AddUser.test.jsx

Test Suites: 3 passed, 3 total
Tests:       5 passed, 5 total
Snapshots:   2 passed, 2 total
Time:        1.415s, estimated 8s
Ran all test suites.
```

## Router Setup

Add react-router-dom to the `dependencies` within *services/client/package.json* file:

```
"dependencies": {
  "axios": "^0.17.1",
  "react": "^16.2.0",
  "react-dom": "^16.2.0",
  "react-router-dom": "^4.2.2",
  "react-scripts": "1.1.0"
},
```

Update the containers to install the new dependency:

```
$ docker-compose -f docker-compose-dev.yml up -d --build
```

React Router has two main components:

1. `Router` : keeps your UI and URL in sync
2. `Route` : maps a route to a component

> We'll be using the `BrowserRouter` for routing, which uses the HTML 5 History API. Review the docs for more info.

Add the router to *index.js*:

```
import React from 'react';
import ReactDOM from 'react-dom';
import { BrowserRouter as Router } from 'react-router-dom';

import App from './App.jsx';

ReactDOM.render((
  <Router>
    <App />
  </Router>
), document.getElementById('root'))
```

Now, let's add a basic `/about route` ...

# New Component

We'll start by adding a new `About` component, starting with a test of course:

```
import React from 'react';
import { shallow } from 'enzyme';
import renderer from 'react-test-renderer';
```

```
import About from '../About';

test('About renders properly', () => {
  const wrapper = shallow(<About/>);
  const element = wrapper.find('p');
  expect(element.length).toBe(1);
  expect(element.text()).toBe('Add something relevant here.');
});
```

Add the test to a new file in "services/client/src/components/__tests__" called *About.test.jsx*. And then run the tests to ensure they fail:

```
FAIL  src/components/__tests__/About.test.jsx
 ● Test suite failed to run

   Cannot find module '../About' from 'About.test.jsx'
```

Add a new component to use for the route to new file called *About.jsx* within "components":

```
import React from 'react';

const About = () => (
  <div>
    <h1>About</h1>
    <hr/><br/>
    <p>Add something relevant here.</p>
  </div>
)

export default About;
```

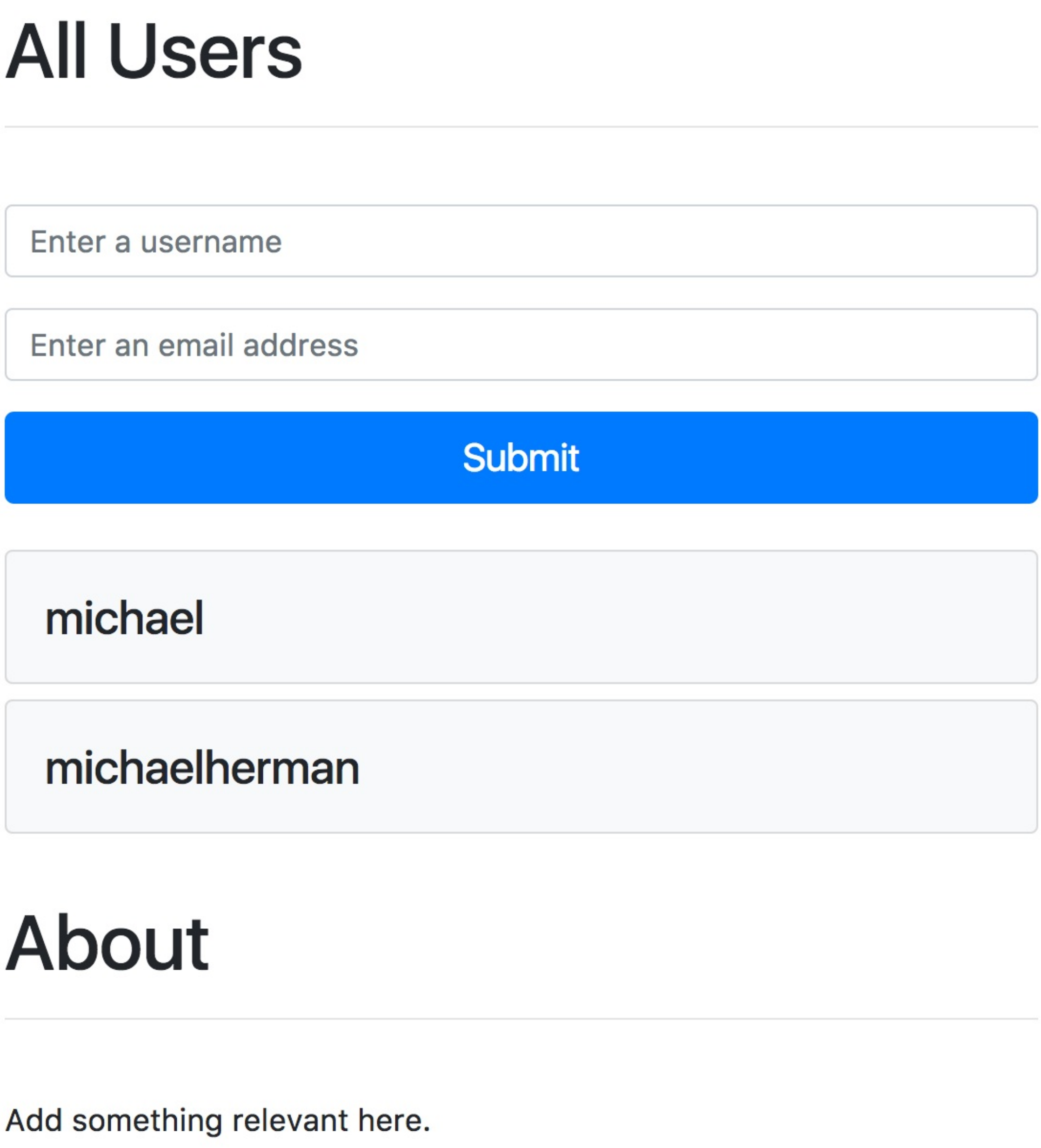
To get a quick sanity check, import the component into *App.jsx*:

```
import About from './components/About';
```

Then add the component to the `render` method, just below the `UsersList` component:

```
...
<UsersList users={this.state.users}/>
<br/>
<About/>
...
```

Make sure you can view the new component in the browser:




Now, to render the `About` component in a different route, update the `render` method again:

```
render() {
  return (
    <div className="container">
      <div className="row">
        <div className="col-md-6">
          <br/>
          <Switch>
            <Route exact path='/' render={() => (
              <div>
                <h1>All Users</h1>
                <hr/><br/>
                <AddUser
                  username={this.state.username}
```

```
                    email={this.state.email}
                    handleChange={this.handleChange}
                    addUser={this.addUser}
                  />
                  <br/>
                  <UsersList users={this.state.users}/>
                </div>
              )} />
              <Route exact path='/about' component={About}/>
            </Switch>
          </div>
        </div>
      </div>
    )
  };
```

Here, we used the `<Switch>` component to group `<Route>` s and then defined two routes - `/` and `/about` .

> Make sure to review the official documentation on the Switch component.

Don't forget the import:

```
import { Route, Switch } from 'react-router-dom';
```

Save, and then test each route in the browser. Once done, return to the terminal and make sure the tests pass.

Now, let's add a quick snapshot test to *About.test.jsx*:

```
test('About renders a snapshot properly', () => {
  const tree = renderer.create(<About/>).toJSON();
  expect(tree).toMatchSnapshot();
});
```

Take the snapshot. Commit and push your code.

# React Bootstrap

In this lesson, we'll add a Navbar and a form component with React Bootstrap to set the stage for adding in full auth...

---

## Setup

Add React Bootstrap and React Router Boostrap to the *package.json* file:

```
"dependencies": {
  "axios": "^0.17.1",
  "react": "^16.2.0",
  "react-bootstrap": "^0.32.1",
  "react-dom": "^16.2.0",
  "react-router-bootstrap": "^0.24.4",
  "react-router-dom": "^4.2.2",
  "react-scripts": "1.1.0"
},
```

For each component, we'll roughly follow these steps:

1. Write a unit test
2. Run the test to ensure it fails
3. Create the component file
4. Add the component
5. Wire up the component to *App.jsx*, passing down any necessary `props`
6. Manually test it in the browser
7. Ensure the unit tests pass
8. Write a snapshot test

## Navbar

Create two new files:

1. *services/client/src/components/__tests__/NavBar.test.jsx*
2. *services/client/src/components/NavBar.jsx*

Start with some tests:

```
import React from 'react';
import { shallow } from 'enzyme';
import renderer from 'react-test-renderer';

import NavBar from '../NavBar';
```

```
const title = 'Hello, World!';

test('NavBar renders properly', () => {
  const wrapper = shallow(<NavBar title={title}/>);
  const element = wrapper.find('span');
  expect(element.length).toBe(1);
  expect(element.get(0).props.children).toBe(title);
});
```

Ensure it fails, and then add the component:

```
import React from 'react';
import { Navbar, Nav, NavItem } from 'react-bootstrap';
import { LinkContainer } from 'react-router-bootstrap';

const NavBar = (props) => (
  <Navbar inverse collapseOnSelect>
    <Navbar.Header>
      <Navbar.Brand>
        <span>{props.title}</span>
      </Navbar.Brand>
      <Navbar.Toggle />
    </Navbar.Header>
    <Navbar.Collapse>
      <Nav>
        <LinkContainer to="/">
          <NavItem eventKey={1}>Home</NavItem>
        </LinkContainer>
        <LinkContainer to="/about">
          <NavItem eventKey={2}>About</NavItem>
        </LinkContainer>
        <LinkContainer to="/status">
          <NavItem eventKey={3}>User Status</NavItem>
        </LinkContainer>
      </Nav>
      <Nav pullRight>
        <LinkContainer to="/register">
          <NavItem eventKey={1}>Register</NavItem>
        </LinkContainer>
        <LinkContainer to="/login">
          <NavItem eventKey={2}>Log In</NavItem>
        </LinkContainer>
        <LinkContainer to="/logout">
          <NavItem eventKey={3}>Log Out</NavItem>
        </LinkContainer>
      </Nav>
    </Navbar.Collapse>
  </Navbar>
)
```

```
export default NavBar;
```

Add the import to *App.jsx*:

```
import NavBar from './components/NavBar';
```

Add a title to `state`:

```
this.state = {
  users: [],
  username: '',
  email: '',
  title: 'TestDriven.io'
};
```

And update `render()`:

```
render() {
  return (
    <div>
      <NavBar
        title={this.state.title}
      />
      <div className="container">
        <div className="row">
          <div className="col-md-6">
            <br/>
            <Switch>
              <Route exact path='/' render={() => (
                <div>
                  <h1>All Users</h1>
                  <hr/><br/>
                  <AddUser
                    username={this.state.username}
                    email={this.state.email}
                    handleChange={this.handleChange}
                    addUser={this.addUser}
                  />
                  <br/>
                  <UsersList users={this.state.users}/>
                </div>
              )} />
              <Route exact path='/about' component={About}/>
            </Switch>
          </div>
        </div>
      </div>
    </div>
```

```
  )
};
```

Update the Bootstrap stylesheet in *services/client/public/index.html*:

```
<link
  rel="stylesheet" href="//maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.mi
n.css"
>
```

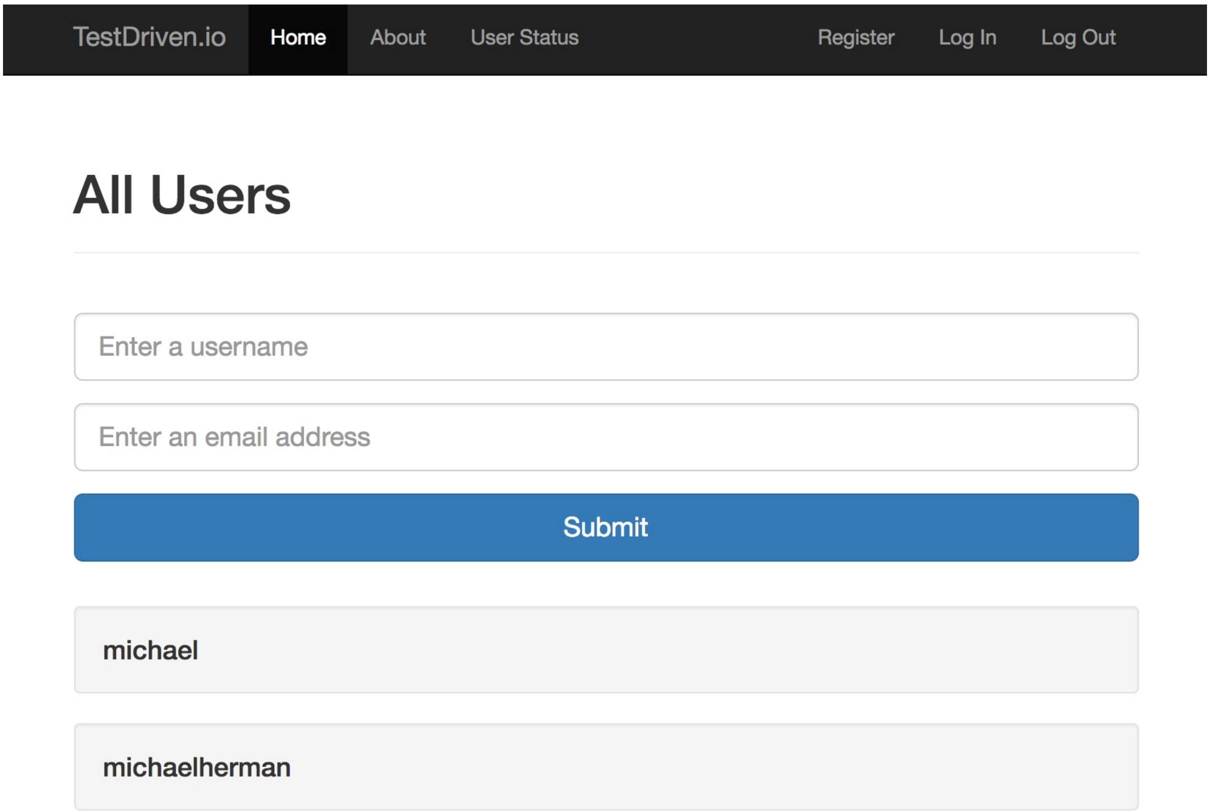
Then, update the `className` in the `UsersList` component:

```
import React from 'react';

const UsersList = (props) => {
  return (
    <div>
      {
        props.users.map((user) => {
          return (
            <h4
              key={user.id}
              className="well"
            >{user.username}
            </h4>
          )
        })
      }
    </div>
  )
};

export default UsersList;
```

Be sure to make the change in the test as well, and then update the container:

```
$ docker-compose -f docker-compose-dev.yml up -d --build
```

Test it out in the browser.




> Run into this issue?
>
> ```
> Cannot find module 'react-bootstrap' from 'NavBar.jsx'
> ```
>
> Enter the bash shell in the container and manually install the dependencies:
>
> ```
> $ docker-compose -f docker-compose-dev.yml run client bash
> # npm install
> added 11 packages in 21.307s
> ```

Ensure the tests pass. Then, add a snapshot test:

```
test('NavBar renders a snapshot properly', () => {
  const tree = renderer.create(
    <Router location="/"><NavBar title={title}/></Router>
  ).toJSON();
  expect(tree).toMatchSnapshot();
});
```

Add the import:

```
import { MemoryRouter as Router } from 'react-router-dom';
```

Here, we used the MemoryRouter to provide context to the Router for the test.

> Review the official Testing guide for more info.

## Form

Instead of using two different components to handle user registration and login, let's create a generic form component and customize it based on the state.

Add the files:

1. *Form.test.jsx*
2. *Form.jsx*

Test:

```
import React from 'react';
import { shallow } from 'enzyme';
import renderer from 'react-test-renderer';

import Form from '../Form';

const formData = {
  username: '',
  email: '',
  password: ''
};

test('Register Form renders properly', () => {
  const component = <Form formType={'Register'} formData={formData} />;
  const wrapper = shallow(component);
  const h1 = wrapper.find('h1');
  expect(h1.length).toBe(1);
  expect(h1.get(0).props.children).toBe('Register');
  const formGroup = wrapper.find('.form-group');
  expect(formGroup.length).toBe(3);
  expect(formGroup.get(0).props.children.props.name).toBe('username');
  expect(formGroup.get(0).props.children.props.value).toBe('');
});

test('Login Form renders properly', () => {
  const component = <Form formType={'Login'} formData={formData} />;
  const wrapper = shallow(component);
  const h1 = wrapper.find('h1');
  expect(h1.length).toBe(1);
  expect(h1.get(0).props.children).toBe('Login');
  const formGroup = wrapper.find('.form-group');
  expect(formGroup.length).toBe(2);
  expect(formGroup.get(0).props.children.props.name).toBe('email');
```

```
  expect(formGroup.get(0).props.children.props.value).toBe('');
});
```

Component:

```
import React from 'react';

const Form = (props) => {
  return (
    <div>
      <h1>{props.formType}</h1>
      <hr/><br/>
      <form onSubmit={(event) => props.handleUserFormSubmit(event)}>
        {props.formType === 'Register' &&
          <div className="form-group">
            <input
              name="username"
              className="form-control input-lg"
              type="text"
              placeholder="Enter a username"
              required
              value={props.formData.username}
              onChange={props.handleFormChange}
            />
          </div>
        }
        <div className="form-group">
          <input
            name="email"
            className="form-control input-lg"
            type="email"
            placeholder="Enter an email address"
            required
            value={props.formData.email}
            onChange={props.handleFormChange}
          />
        </div>
        <div className="form-group">
          <input
            name="password"
            className="form-control input-lg"
            type="password"
            placeholder="Enter a password"
            required
            value={props.formData.password}
            onChange={props.handleFormChange}
          />
        </div>
        <input
          type="submit"
```

```
          className="btn btn-primary btn-lg btn-block"
          value="Submit"
        />
      </form>
    </div>
  )
};

export default Form;
```

Did you notice the inline if statement - `props.formType === 'Register' &&` ? Review the code above, adding in code comments as needed.

Import the component into *App.jsx*, and then update the state in the constructor:

```
this.state = {
  users: [],
  username: '',
  email: '',
  title: 'TestDriven.io',
  formData: {
    username: '',
    email: '',
    password: ''
  }
};
```

Add the component to the `<Switch>` , within the `render` :

```
<Route exact path='/register' render={() => (
  <Form
    formType={'Register'}
    formData={this.state.formData}
  />
)} />
<Route exact path='/login' render={() => (
  <Form
    formType={'Login'}
    formData={this.state.formData}
  />
)} />
```

Make sure the routes work in the browser, but don't try to submit the forms just yet - we still need to wire them up!

Add the snapshot tests:

```
test('Register Form renders a snapshot properly', () => {
```

```
  const component = <Form formType={'Register'} formData={formData} />;
  const tree = renderer.create(component).toJSON();
  expect(tree).toMatchSnapshot();
});

test('Login Form renders a snapshot properly', () => {
  const component = <Form formType={'Login'} formData={formData} />;
  const tree = renderer.create(component).toJSON();
  expect(tree).toMatchSnapshot();
});
```

Make sure the tests pass!

```
PASS  src/components/__tests__/Form.test.jsx
PASS  src/components/__tests__/NavBar.test.jsx
PASS  src/components/__tests__/App.test.jsx
PASS  src/components/__tests__/UsersList.test.jsx
PASS  src/components/__tests__/AddUser.test.jsx
PASS  src/components/__tests__/About.test.jsx

Test Suites: 6 passed, 6 total
Tests:       13 passed, 13 total
Snapshots:   6 passed, 6 total
Time:        2.761s, estimated 7s
Ran all test suites.
```

## Refactor

Before moving on, let's do two quick refactors...

### Form tests

This code is not DRY. It may be fine for the two forms we have now, but what if we had 20? Re-write this on your own before reviewing the solution.

```
import React from 'react';
import { shallow } from 'enzyme';
import renderer from 'react-test-renderer';

import Form from '../Form';

const testData = [
  {
    formType: 'Register',
    formData: {
      username: '',
      email: '',
      password: ''
```

```
      },
    },
    {
      formType: 'Login',
      formData: {
        email: '',
        password: ''
      },
    }
]

testData.forEach((el) => {
  test(`${el.formType} Form renders properly`, () => {
    const component = <Form formType={el.formType} formData={el.formData} />;
    const wrapper = shallow(component);
    const h1 = wrapper.find('h1');
    expect(h1.length).toBe(1);
    expect(h1.get(0).props.children).toBe(el.formType);
    const formGroup = wrapper.find('.form-group');
    expect(formGroup.length).toBe(Object.keys(el.formData).length);
    expect(formGroup.get(0).props.children.props.name).toBe(Object.keys(el.formData
)[0]);
    expect(formGroup.get(0).props.children.props.value).toBe('');
  });
  test(`${el.formType} Form renders a snapshot properly`, () => {
    const component = <Form formType={el.formType} formData={el.formData} />;
    const tree = renderer.create(component).toJSON();
    expect(tree).toMatchSnapshot();
  });
});
```

Run the tests again.

```
PASS  src/components/__tests__/NavBar.test.jsx
PASS  src/components/__tests__/App.test.jsx
PASS  src/components/__tests__/Form.test.jsx
PASS  src/components/__tests__/AddUser.test.jsx
PASS  src/components/__tests__/UsersList.test.jsx
PASS  src/components/__tests__/About.test.jsx

Test Suites: 6 passed, 6 total
Tests:       13 passed, 13 total
Snapshots:   6 passed, 6 total
Time:        4.426s, estimated 5s
Ran all test suites.
```

Also, it's important to ensure that the rendered components have not actually changed. Start by visually inspecting them in the browser, and then, as long as the two snapshots pass without having to create new snapshots, then we know all is well. How do you know if the snapshot has changed?

You'll see the following message in the terminal when the tests run:

```
Snapshot Summary
 › 1 snapshot written in 1 test suite.
 › 1 obsolete snapshot found, press `u` to remove them.
```

If you see that message, immediately revert your code back, take a new snapshot, and then start the refactor over again.

Commit your code. Push to GitHub.

# React Authentication - part 1

Moving right along, let's add some methods to handle a user signing up, logging in, and logging out...

---

With the `Form` component set up, we can now configure the methods to:

1. Handle form submit event
2. Obtain user input
3. Send AJAX request
4. Update the page

> These steps should look familiar since we already went through this process in the React Forms lesson. Put your skills to the test and implement the code on your own before going through this lesson.

## Handle form submit event

Turn to *Form.jsx*. Which method gets fired on the form submit?

```
<form onSubmit={(event) => props.handleUserFormSubmit(event)}>
```

Add the method to the `App` component:

```
handleUserFormSubmit(event) {
  event.preventDefault();
  console.log('sanity check!');
};
```

Bind the method in the constructor:

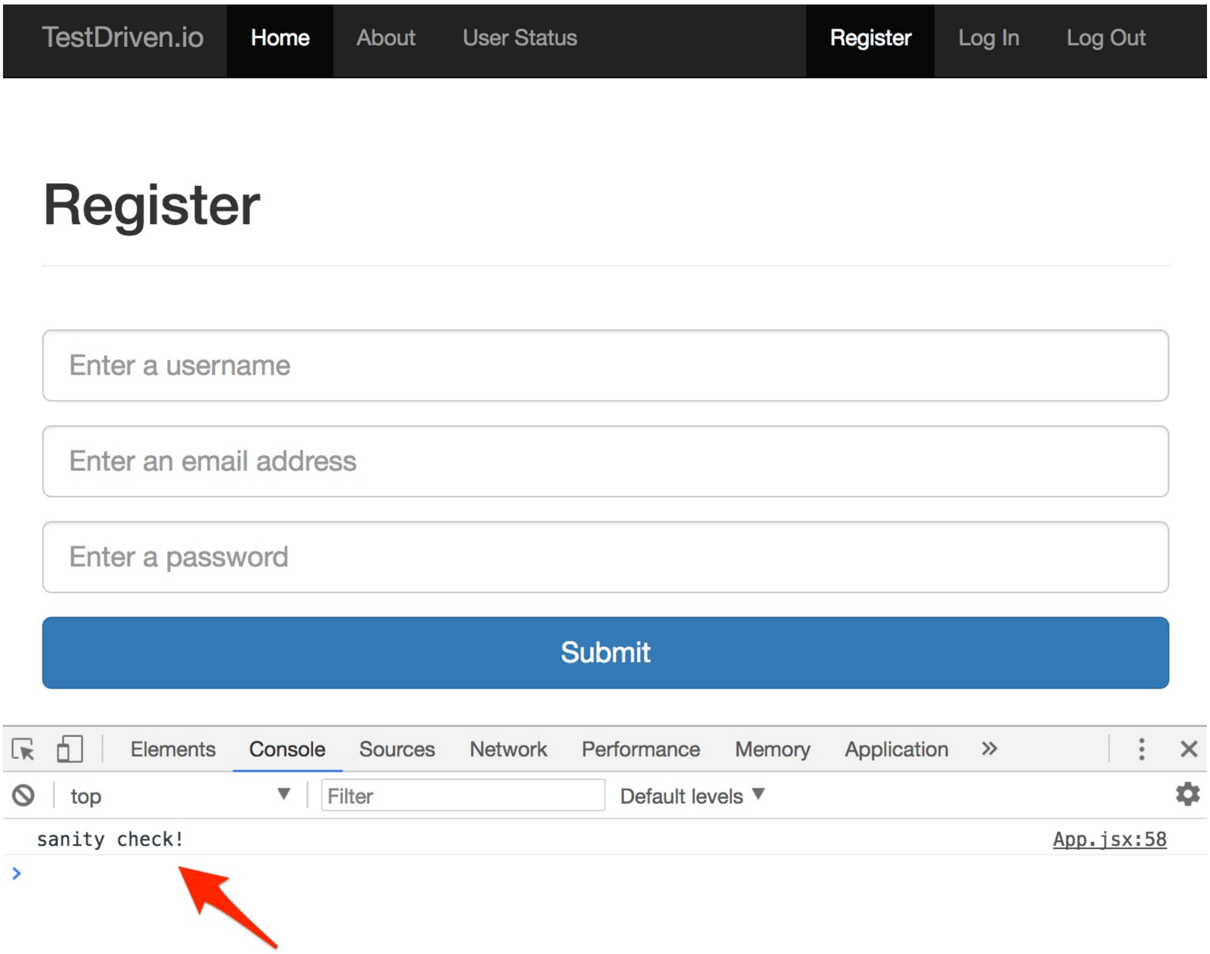
```
this.handleUserFormSubmit = this.handleUserFormSubmit.bind(this);
```

And then pass it down via the `props` :

```
<Route exact path='/register' render={() => (
  <Form
    formType={'Register'}
    formData={this.state.formData}
    handleUserFormSubmit={this.handleUserFormSubmit}
  />
)} />
<Route exact path='/login' render={() => (
  <Form
    formType={'Login'}
```

```
    formData={this.state.formData}
    handleUserFormSubmit={this.handleUserFormSubmit}
  />
)} />
```

To test, remove the `required` attribute on each of the form `input` s in *services/client/src/components/Form.jsx*. Then, you should see `sanity check!` in the JavaScript console on form submit for both forms in the browser.




Remove `console.log('sanity check!')` and add the `required` attributes back when done.

## Obtain user input

Next, to get the user inputs, add the following method to the `App` component:

```
handleFormChange(event) {
  const obj = this.state.formData;
  obj[event.target.name] = event.target.value;
  this.setState(obj);
};
```

Again, bind it in the constructor, and then pass it down to the components via the `props` :

```
handleFormChange={this.handleFormChange}
```

Add a `console.log()` to the method - `console.log(this.state.formData);` - to ensure it works when you test it in the browser. Remove it once done.

What's next? AJAX!

## Send AJAX request

Update the `handleUserFormSubmit` method to send the data to the `user` service on a successful form submit:

```
handleUserFormSubmit(event) {
  event.preventDefault();
  const formType = window.location.href.split('/').reverse()[0];
  let data = {
    email: this.state.formData.email,
    password: this.state.formData.password,
  };
  if (formType === 'register') {
    data.username = this.state.formData.username;
  }
  const url = `${process.env.REACT_APP_USERS_SERVICE_URL}/auth/${formType}`
  axios.post(url, data)
  .then((res) => {
    console.log(res.data);
  })
  .catch((err) => { console.log(err); });
};
```

Add a new `location` block to both Nginx config files to handle requests to `/auth` :

```
location /auth {
  proxy_pass          http://users:5000;
  proxy_redirect      default;
  proxy_set_header    Host $host;
  proxy_set_header    X-Real-IP $remote_addr;
  proxy_set_header    X-Forwarded-For $proxy_add_x_forwarded_for;
  proxy_set_header    X-Forwarded-Host $server_name;
}
```

Set the `REACT_APP_USERS_SERVICE_URL` environment variable:

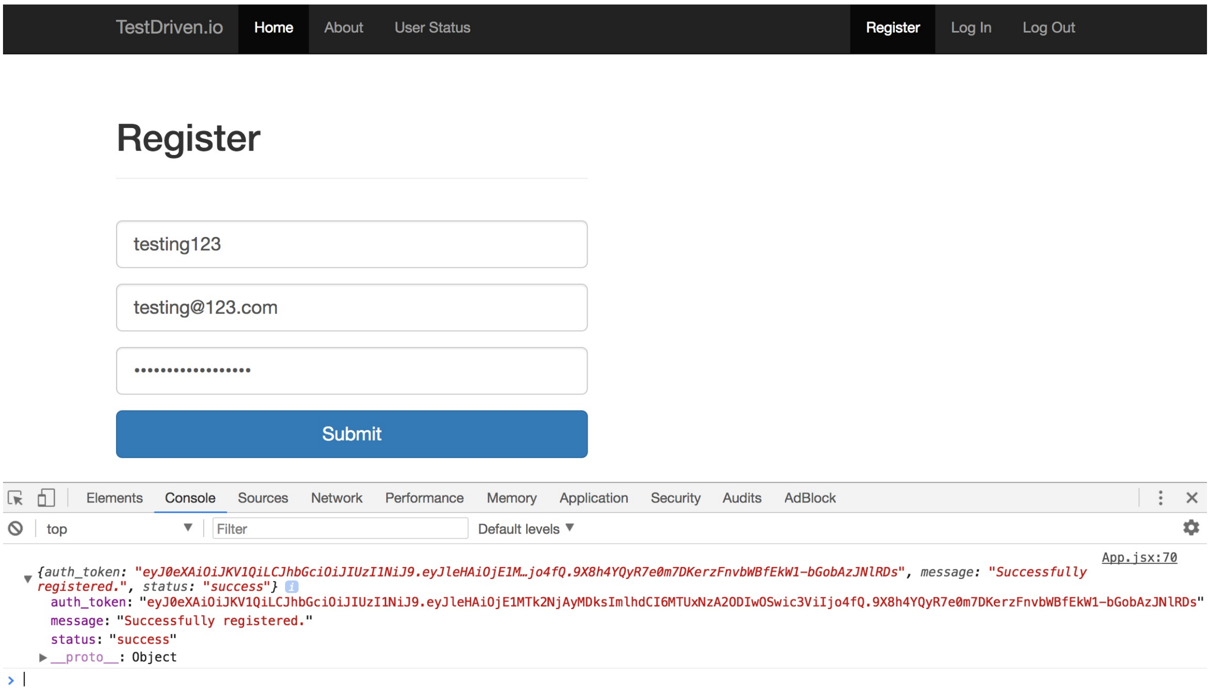
```
$ export REACT_APP_USERS_SERVICE_URL=http://DOCKER_MACHINE_DEV_IP
```

Update the containers:

```
$ docker-compose -f docker-compose-dev.yml up -d --build
```

Test the user registration out. If you have everything set up correctly, you should see an object in the JavaScript console with an auth token:

```
{
  "auth_token": "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJleHAiOjE0OTc3NTM2ODMsImlhd
CI6MTQ5Nzc1MzY3OCwic3ViIjo0fQ.vcRFb5v3znHkz8An12QUxrgXsLqoKv93kIsMf-pdfVw",
  "message": "Successfully registered.",
  "status": "success"
}
```




Test logging in as well. Again, you should see the very same object in the console.

## Update the page

After a user register or logs in, we need to:

1. Clear the `formData` object
2. Save the auth token in the browser's LocalStorage, a client-side data store
3. Update the state to indicate that the user is authenticated
4. Redirect the user to `/`

First, to clear the form, update the `.then` block within `handleUserFormSubmit()` :

```
.then((res) => {
  this.setState({
    formData: {username: '', email: '', password: '' },
    username: '',
    email: ''
  });
})
```

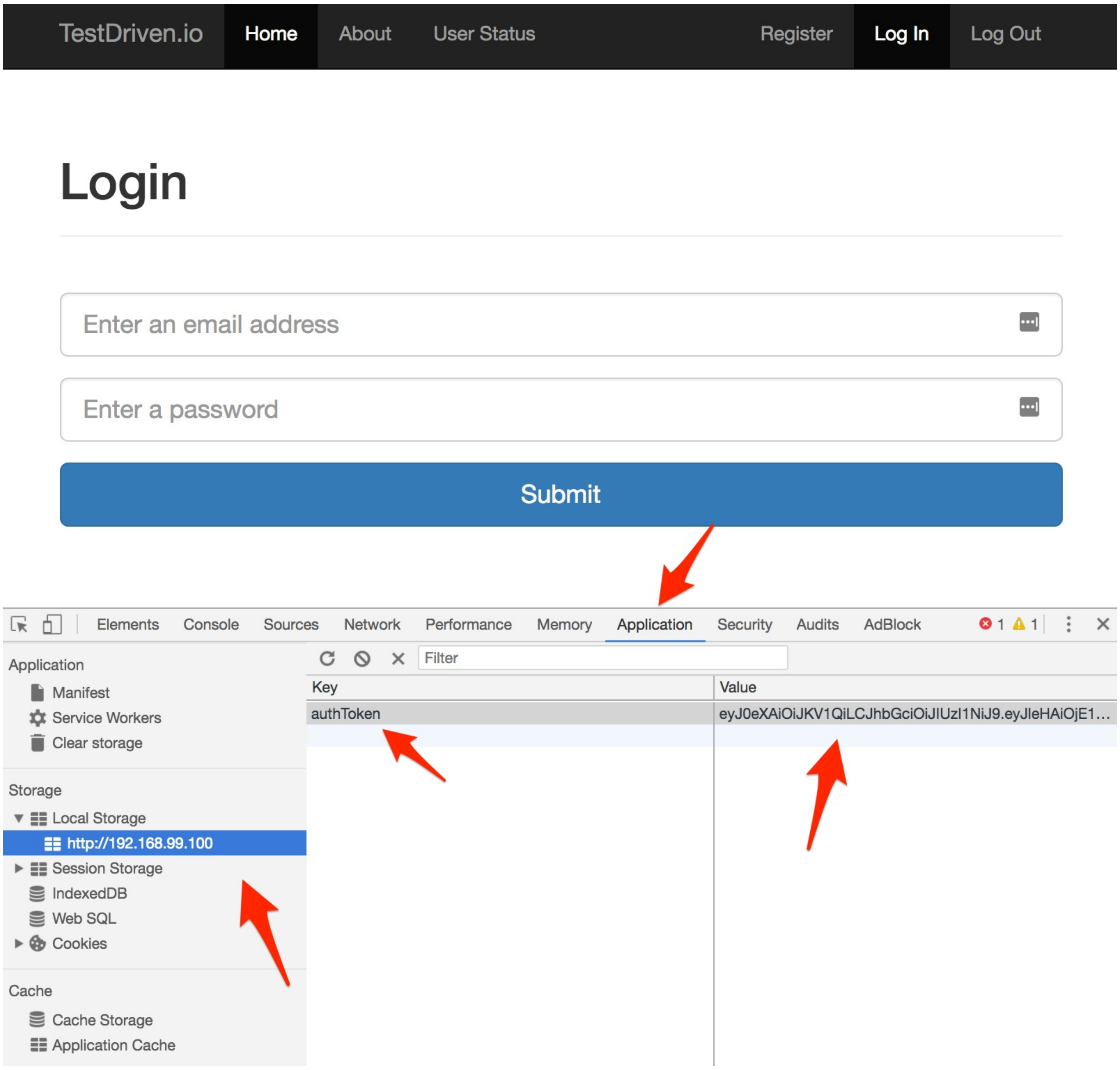
Try this out. After you register or log in, the field inputs should be cleared since we set the properties in the `formData` object to empty strings.

> What happens if you enter data for the registration form but *don't* submit it and then navigate to the login form? The fields should remain. Is this okay? Should we clear the state on page load? Your call. You could simply update the state within the `componentWillMount` lifecycle method.

Next, let's save the auth token in LocalStorage so that we can use it for subsequent API calls that require a user to be authenticated. To do this, add the following code to the `.then`, just below the `setState`:

```
window.localStorage.setItem('authToken', res.data.auth_token);
```

Try logging in again. After a successful login, open the "Application" tab in Chrome DevTools. Click the arrow pointing toward LocalStorage and select the IP (which should be the IP associated with the Docker Machine). You should see a key of `authToken` with a value of the actual token in the pane.




Instead of always checking LocalStorage for the auth token, let's add a boolean to the state so we can quickly tell if a user is authenticated.

Add an `isAuthenticated` property to the state:

```
this.state = {
  users: [],
  username: '',
  email: '',
  title: 'TestDriven.io',
  formData: {
    username: '',
    email: '',
    password: ''
  },
  isAuthenticated: false,
};
```

Now, we can update the state in the `.then` within `handleUserFormSubmit()`:

```
this.setState({
  formData: {username: '', email: '', password: '' },
  username: '',
  email: '',
  isAuthenticated: true,
});
```

Finally, to redirect the user after a successful log in or registration, pass `isAuthenticated` through to the `Form` component:

```
<Route exact path='/register' render={() => (
  <Form
    formType={'Register'}
    formData={this.state.formData}
    handleUserFormSubmit={this.handleUserFormSubmit}
    handleFormChange={this.handleFormChange}
    isAuthenticated={this.state.isAuthenticated}
  />
)} />
<Route exact path='/login' render={() => (
  <Form
    formType={'Login'}
    formData={this.state.formData}
    handleUserFormSubmit={this.handleUserFormSubmit}
    handleFormChange={this.handleFormChange}
    isAuthenticated={this.state.isAuthenticated}
  />
)} />
```

Then, within *Form.jsx* add the following conditional right before the `return`:

```
if (props.isAuthenticated) {
  return <Redirect to='/' />;
}
```

Add the import:

```
import { Redirect } from 'react-router-dom';
```

To test, log in and then make sure that you are redirected to `/`. Also, once logged in, you should be redirected if you try to go to the `/register` or `/login` links. Before moving on, try registering a new user. Did you notice that even though the redirect works, the user list is not updating?

To update that, fire `this.getUsers()` in the `.then` within `handleUserFormSubmit()`:

```
.then((res) => {
  this.setState({
    formData: {username: '', email: '', password: ''},
    username: '',
    email: '',
    isAuthenticated: true
  });
  window.localStorage.setItem('authToken', res.data.auth_token);
  this.getUsers();
})
```

Test it out again.

## Logout

How about logging out? Add a new file called *Logout.test.jsx* to the "services/client/src/components/__tests__" directory:

```
import React from 'react';
import { shallow } from 'enzyme';

import Logout from '../Logout';

const logoutUser = jest.fn();

test('Logout renders properly', () => {
  const wrapper = shallow(<Logout logoutUser={logoutUser}/>);
  const element = wrapper.find('p');
  expect(element.length).toBe(1);
  expect(element.get(0).props.children[0]).toContain('You are now logged out.');
});
```

Here, we're using `jest.fn()` to mock the `logoutUser` function. Ensure the tests fail, and then add a new component to the "components" folder called *Logout.jsx*:

```
import React, { Component } from 'react';
import { Link } from 'react-router-dom';

class Logout extends Component {
  componentDidMount() {
    this.props.logoutUser();
  };
  render() {
    return (
      <div>
        <p>You are now logged out. Click <Link to="/login">here</Link> to log back
in.</p>
      </div>
```

```
    )
  };
};

export default Logout;
```

Then, add a `logoutUser` method to the `App` component to remove the token from LocalStorage and update the state:

```
logoutUser() {
  window.localStorage.clear();
  this.setState({ isAuthenticated: false });
};
```

Bind the method:

```
this.logoutUser = this.logoutUser.bind(this);
```

Import the component into *App.jsx*, and then add the new route:

```
<Route exact path='/logout' render={() => (
  <Logout
    logoutUser={this.logoutUser}
    isAuthenticated={this.state.isAuthenticated}
  />
)} />
```

To test:

1. Log in
2. Verify that the token was added to LocalStorage
3. Log out
4. Verify that the token was removed from LocalStorage

Once you're done manually testing in the browser, ensure the unit tests pass. Then, add a snapshot test:

```
test('Logout renders a snapshot properly', () => {
  const tree = renderer.create(
    <Router><Logout logoutUser={logoutUser}/></Router>
  ).toJSON();
  expect(tree).toMatchSnapshot();
});
```

We need to provide the `<Router>` context (via the MemoryRouter) since it's required in the component (by the `Link` ).

Don't forget the imports:

```
import renderer from 'react-test-renderer';
import { MemoryRouter as Router } from 'react-router-dom';
```

Ensure the tests pass:

```
PASS  src/components/__tests__/NavBar.test.jsx
PASS  src/components/__tests__/App.test.jsx
PASS  src/components/__tests__/Logout.test.jsx
PASS  src/components/__tests__/UsersList.test.jsx
PASS  src/components/__tests__/About.test.jsx
PASS  src/components/__tests__/Form.test.jsx
PASS  src/components/__tests__/AddUser.test.jsx

Test Suites: 7 passed, 7 total
Tests:       15 passed, 15 total
Snapshots:   7 passed, 7 total
Time:        3.847s, estimated 5s
Ran all test suites.
```

Commit your code.

# Mocking User Interaction

Let's look at how to test user interactions with Enzyme...

---

When testing components, especially user interactions, pay close attention to both the inputs and outputs:

1. Inputs - props, state, user interactions
2. Output - what the component renders

So, given the `Form` component, for the register route, what are the inputs:

1. `formType={'Register'}`
2. `formData={this.state.formData}`
3. `handleUserFormSubmit={this.handleUserFormSubmit}`
4. `handleFormChange={this.handleFormChange}`
5. `isAuthenticated={this.state.isAuthenticated}`

What happens when a user submits the registration form correctly? What does the component render? Does the component behave differently based on the provided inputs? What would change if the value of `formType` was `Login` ?

## Refactor

Let's start by refactoring the current tests in *services/client/src/components/__tests__/Form.test.jsx*:

```
describe('When not authenticated', () => {
  testData.forEach((el) => {
    const component = <Form
      formType={el.formType}
      formData={el.formData}
      isAuthenticated={false}
    />;
    it(`${el.formType} Form renders properly`, () => {
      const wrapper = shallow(component);
      const h1 = wrapper.find('h1');
      expect(h1.length).toBe(1);
      expect(h1.get(0).props.children).toBe(el.formType);
      const formGroup = wrapper.find('.form-group');
      expect(formGroup.length).toBe(Object.keys(el.formData).length);
      expect(formGroup.get(0).props.children.props.name).toBe(Object.keys(el.formDa
ta)[0]);
      expect(formGroup.get(0).props.children.props.value).toBe('');
    });
    it(`${el.formType} Form renders a snapshot properly`, () => {
      const tree = renderer.create(component).toJSON();
```

```
      expect(tree).toMatchSnapshot();
    });
  })
});
```

Run the tests with the `--verbose` flag so we can see the full output:

```
docker-compose -f docker-compose-dev.yml \
  run client npm test -- --verbose
```

You should see something similar to:

```
PASS  src/components/__tests__/NavBar.test.jsx
 ✓ NavBar renders properly (4ms)
 ✓ NavBar renders a snapshot properly (9ms)

PASS  src/components/__tests__/App.test.jsx
 ✓ App renders without crashing (5ms)

PASS  src/components/__tests__/AddUser.test.jsx
 ✓ AddUser renders properly (13ms)
 ✓ AddUser renders a snapshot properly (2ms)

PASS  src/components/__tests__/Logout.test.jsx
 ✓ Logout renders properly (3ms)
 ✓ Logout renders a snapshot properly (2ms)

PASS  src/components/__tests__/Form.test.jsx
 When not authenticated
   ✓ Register Form renders properly (7ms)
   ✓ Register Form renders a snapshot properly (7ms)
   ✓ Login Form renders properly (2ms)
   ✓ Login Form renders a snapshot properly (5ms)

PASS  src/components/__tests__/UsersList.test.jsx
 ✓ UsersList renders properly (3ms)
 ✓ UsersList renders a snapshot properly (1ms)

PASS  src/components/__tests__/About.test.jsx
 ✓ About renders properly (2ms)
 ✓ About renders a snapshot properly (2ms)

Test Suites: 7 passed, 7 total
Tests:       15 passed, 15 total
Snapshots:   7 passed, 7 total
Time:        3.415s, estimated 4s
Ran all test suites.
```

Now, turn back to the component. What will happen if `isAuthenticated` is `true`? Will this cause the component to behave differently?

Need a hint?

```
if (props.isAuthenticated) {
  return <Redirect to='/' />;
}
```

Add a another set of test cases:

```
describe('When authenticated', () => {
  testData.forEach((el) => {
    const component = <Form
      formType={el.formType}
      formData={el.formData}
      isAuthenticated={true}
    />;
    it(`${el.formType} redirects properly`, () => {
      const wrapper = shallow(component);
      expect(wrapper.find('Redirect')).toHaveLength(1);
    });
  })
});
```

For this test case, we're just asserting that the `render` component is rendered. Ensure the tests pass.

```
Test Suites: 7 passed, 7 total
Tests:       17 passed, 17 total
Snapshots:   7 passed, 7 total
Time:        3.483s, estimated 4s
Ran all test suites.
```

Next, let's look at how to test a user interaction...

## Testing Interactions

Before we start, brainstorm on your own for a bit on what happens during a form submit, paying particular attention to the component's inputs and outputs...

### Form Submit

Add a new `describe` block to *services/client/src/components/__tests__/Form.test.jsx*:

```
describe('When not authenticated', () => {
  const testValues = {
```

```
    formType: 'Register',
    formData: {
      username: '',
      email: '',
      password: ''
    },
    handleUserFormSubmit: jest.fn(),
    handleFormChange: jest.fn(),
    isAuthenticated: false,
  };
  const component = <Form {...testValues} />;
  it(`${testValues.formType} Form submits the form properly`, () => {
    const wrapper = shallow(component);
    expect(testValues.handleUserFormSubmit).toHaveBeenCalledTimes(0);
    wrapper.find('form').simulate('submit')
    expect(testValues.handleUserFormSubmit).toHaveBeenCalledTimes(1);
  });
});
```

Here, we used `jest.fn()` to mock the `handleUserFormSubmit` method and then asserted that the function was called on the simulated form submit.

## Form Values

Let's take it one step further and assert that the form values are being handled correctly. Update the `it` block like so:

```
it(`${testValues.formType} Form submits the form properly`, () => {
  const wrapper = shallow(component);
  expect(testValues.handleUserFormSubmit).toHaveBeenCalledTimes(0);
  wrapper.find('form').simulate('submit', testValues.formData)
  expect(testValues.handleUserFormSubmit).toHaveBeenCalledWith(
    testValues.formData);
  expect(testValues.handleUserFormSubmit).toHaveBeenCalledTimes(1);
});
```

## OnChange

How about the `onChange` ?

```
it(`${testValues.formType} Form submits the form properly`, () => {
  const wrapper = shallow(component);
  const input = wrapper.find('input[type="text"]');
  expect(testValues.handleUserFormSubmit).toHaveBeenCalledTimes(0);
  expect(testValues.handleFormChange).toHaveBeenCalledTimes(0);
  input.simulate('change')
  expect(testValues.handleFormChange).toHaveBeenCalledTimes(1);
  wrapper.find('form').simulate('submit', testValues.formData)
  expect(testValues.handleUserFormSubmit).toHaveBeenCalledWith(
```

```
    testValues.formData);
  expect(testValues.handleUserFormSubmit).toHaveBeenCalledTimes(1);
});
```

### Refactor

Finally, update the tests to incorporate the previous `it` block into the original describe `block` :

```
import React from 'react';
import { shallow, simulate } from 'enzyme';
import renderer from 'react-test-renderer';
import { MemoryRouter, Switch, Redirect } from 'react-router-dom';

import Form from '../Form';

const testData = [
  {
    formType: 'Register',
    formData: {
      username: '',
      email: '',
      password: ''
    },
    handleUserFormSubmit: jest.fn(),
    handleFormChange: jest.fn(),
    isAuthenticated: false,
  },
  {
    formType: 'Login',
    formData: {
      email: '',
      password: ''
    },
    handleUserFormSubmit: jest.fn(),
    handleFormChange: jest.fn(),
    isAuthenticated: false,
  }
]

describe('When not authenticated', () => {
  testData.forEach((el) => {
    const component = <Form {...el} />;
    it(`${el.formType} Form renders properly`, () => {
      const wrapper = shallow(component);
      const h1 = wrapper.find('h1');
      expect(h1.length).toBe(1);
      expect(h1.get(0).props.children).toBe(el.formType);
      const formGroup = wrapper.find('.form-group');
      expect(formGroup.length).toBe(Object.keys(el.formData).length);
      expect(formGroup.get(0).props.children.props.name).toBe(
```

```
        Object.keys(el.formData)[0]);
      expect(formGroup.get(0).props.children.props.value).toBe('');
    });
    it(`${el.formType} Form submits the form properly`, () => {
      const wrapper = shallow(component);
      const input = wrapper.find('input[type="email"]');
      expect(el.handleUserFormSubmit).toHaveBeenCalledTimes(0);
      expect(el.handleFormChange).toHaveBeenCalledTimes(0);
      input.simulate('change')
      expect(el.handleFormChange).toHaveBeenCalledTimes(1);
      wrapper.find('form').simulate('submit', el.formData)
      expect(el.handleUserFormSubmit).toHaveBeenCalledWith(el.formData);
      expect(el.handleUserFormSubmit).toHaveBeenCalledTimes(1);
    });
    it(`${el.formType} Form renders a snapshot properly`, () => {
      const tree = renderer.create(component).toJSON();
      expect(tree).toMatchSnapshot();
    });
  })
});

describe('When authenticated', () => {
  testData.forEach((el) => {
    const component = <Form
      formType={el.formType}
      formData={el.formData}
      isAuthenticated={true}
    />;
    it(`${el.formType} redirects properly`, () => {
      const wrapper = shallow(component);
      expect(wrapper.find('Redirect')).toHaveLength(1);
    });
  })
});
```

Insure the tests pass before moving on:

```
PASS  src/components/__tests__/NavBar.test.jsx
 ✓ NavBar renders properly (4ms)
 ✓ NavBar renders a snapshot properly (9ms)

PASS  src/components/__tests__/App.test.jsx
 ✓ App renders without crashing (5ms)

PASS  src/components/__tests__/AddUser.test.jsx
 ✓ AddUser renders properly (5ms)
 ✓ AddUser renders a snapshot properly (2ms)

PASS  src/components/__tests__/UsersList.test.jsx
 ✓ UsersList renders properly (4ms)
```

```
  ✓ UsersList renders a snapshot properly (2ms)

PASS  src/components/__tests__/About.test.jsx
  ✓ About renders properly (2ms)
  ✓ About renders a snapshot properly (2ms)

PASS  src/components/__tests__/Form.test.jsx
  When not authenticated
    ✓ Register Form renders properly (3ms)
    ✓ Register Form submits the form properly (2ms)
    ✓ Register Form renders a snapshot properly (1ms)
    ✓ Login Form renders properly (4ms)
    ✓ Login Form submits the form properly (4ms)
    ✓ Login Form renders a snapshot properly (2ms)
  When authenticated
    ✓ Register redirects properly (1ms)
    ✓ Login redirects properly (1ms)

PASS  src/components/__tests__/Logout.test.jsx
  ✓ Logout renders properly (3ms)
  ✓ Logout renders a snapshot properly (5ms)

Test Suites: 7 passed, 7 total
Tests:       19 passed, 19 total
Snapshots:   7 passed, 7 total
Time:        3.553s, estimated 4s
Ran all test suites.
```

Commit and push your code.

# React Authentication - part 2

Moving on, let's finish up user auth...

---

## User Status

For the `/status` link, we need to add a new component that displays the response from a call to `/auth/status` from the users service. *Remember*: You need to be authenticated to hit this end-point successfully. So, we will need to add the token to the header prior to sending the AJAX request.

First, add a new component called *UserStatus.jsx*:

```
import React, { Component } from 'react';
import axios from 'axios';

class UserStatus extends Component {
  constructor (props) {
    super(props);
    this.state = {
      email: '',
      id: '',
      username: ''
    };
  };
  componentDidMount() {
    this.getUserStatus();
  };
  getUserStatus(event) {
    const options = {
      url: `${process.env.REACT_APP_USERS_SERVICE_URL}/auth/status`,
      method: 'get',
      headers: {
        'Content-Type': 'application/json',
        Authorization: `Bearer ${window.localStorage.authToken}`
      }
    };
    return axios(options)
    .then((res) => { console.log(res.data.data) })
    .catch((error) => { console.log(error); });
  };
  render() {
    return (
      <div>
        <p>test</p>
      </div>
    )
  };
```

```
};

export default UserStatus;
```

Here, we used a stateful, class-based component to give the component its own internal state. Notice how we also included the header with the AJAX request.

Import the component into *App.jsx*, and then add a new route:

```
<Route exact path='/status' component={UserStatus}/>
```

Test this out first when you're not logged in. You should see a 401 error in the JavaScript console. Try again when you are logged in. You should see an object with the keys `active` , `email` , `id` , and `username` in the console.

To add the values to the component, update the `.then` :

```
.then((res) => {
  this.setState({
    email: res.data.data.email,
    id: res.data.data.id,
    username: res.data.data.username
  })
})
```

Also, update the `render()` :

```
render() {
  return (
    <div>
      <ul>
        <li><strong>User ID:</strong> {this.state.id}</li>
        <li><strong>Email:</strong> {this.state.email}</li>
        <li><strong>Username:</strong> {this.state.username}</li>
      </ul>
    </div>
  )
};
```

Test it out.

## Update Navbar

Finally, let's make the following changes to the `Navbar` :

1. When the user is logged in, the register and log in links should be hidden
2. When the user is logged out, the log out and user status links should be hidden

Update the `NavBar` component like so to show/hide based on the value of `isAuthenticated` :

```
const NavBar = (props) => (
  <Navbar inverse collapseOnSelect>
    <Navbar.Header>
      <Navbar.Brand>
        <span>{props.title}</span>
      </Navbar.Brand>
      <Navbar.Toggle />
    </Navbar.Header>
    <Navbar.Collapse>
      <Nav>
        <LinkContainer to="/">
          <NavItem eventKey={1}>Home</NavItem>
        </LinkContainer>
        <LinkContainer to="/about">
          <NavItem eventKey={2}>About</NavItem>
        </LinkContainer>
        {props.isAuthenticated &&
          <LinkContainer to="/status">
            <NavItem eventKey={3}>User Status</NavItem>
          </LinkContainer>
        }
      </Nav>
      <Nav pullRight>
        {!props.isAuthenticated &&
          <LinkContainer to="/register">
            <NavItem eventKey={1}>Register</NavItem>
          </LinkContainer>
        }
        {!props.isAuthenticated &&
          <LinkContainer to="/login">
            <NavItem eventKey={2}>Log In</NavItem>
          </LinkContainer>
        }
        {props.isAuthenticated &&
          <LinkContainer to="/logout">
            <NavItem eventKey={3}>Log Out</NavItem>
          </LinkContainer>
        }
      </Nav>
    </Navbar.Collapse>
  </Navbar>
)
```

Make sure to pass `isAuthenticated` down on the `props` :

```
<NavBar
  title={this.state.title}
```

```
  isAuthenticated={this.state.isAuthenticated}
/>
```

This merely hides the links. An unauthenticated user could still access the route via entering the URL into the URL bar. To restrict access, update the `render()` in *UserStatus.jsx*:

```
render() {
  if (!this.props.isAuthenticated) {
    return (
      <p>You must be logged in to view this. Click <Link to="/login">here</Link> to
 log back in.</p>
    )
  };
  return (
    <div>
      <ul>
        <li><strong>User ID:</strong> {this.state.id}</li>
        <li><strong>Email:</strong> {this.state.email}</li>
        <li><strong>Username:</strong> {this.state.username}</li>
      </ul>
    </div>
  )
};
```

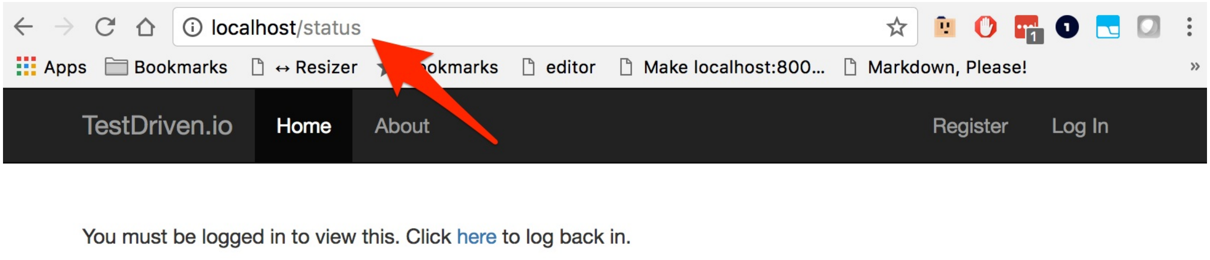
Add the import:

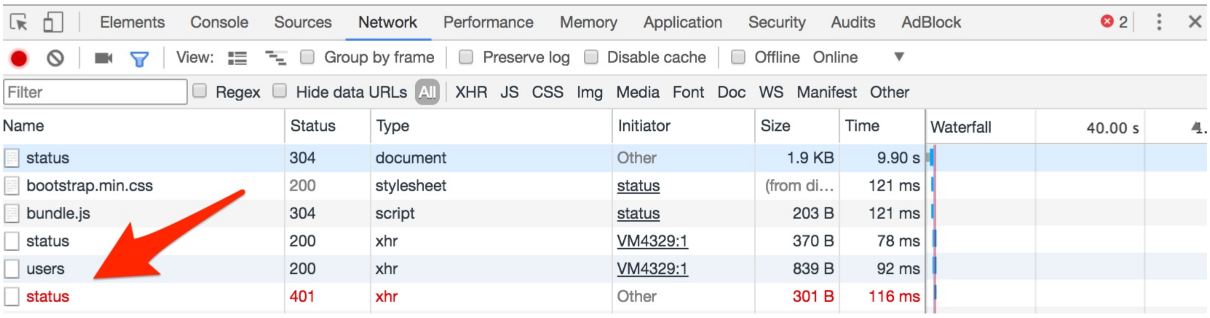
```
import { Link } from 'react-router-dom';
```

Then update the route in the `App` component:

```
<Route exact path='/status' render={() => (
  <UserStatus
    isAuthenticated={this.state.isAuthenticated}
  />
)} />
```

Open the JavaScript console, and then try this out. Did you notice that the AJAX request still fires when you were unauthenticated?




To fix, add a conditional to the `componentDidMount()` in the `UserStatus` component:

```
componentDidMount() {
  if (this.props.isAuthenticated) {
    this.getUserStatus();
  }
};
```

Commit your code.

# Authorization

With authentication done we can now turn our attention to authorization...

---

First, some definitions:

1. *Authentication* - verifying (via user credentials) that the user is who they say they are
2. *Authorization* - ensuring (via permissions) that a user is allowed to do something

> Review Authentication vs. Authorization on Wikipedia for more info.

## Docker Machine

Set `testdriven-dev` as the active Docker Machine:

```
$ docker-machine env testdriven-dev
$ eval $(docker-machine env testdriven-dev)
```

Ensure the app is working in the browser, and then run the tests:

```
$ docker-compose -f docker-compose-dev.yml \
  run users python manage.py test

$ docker-compose -f docker-compose-dev.yml \
  run users flake8 project

$ docker-compose -f docker-compose-dev.yml \
  run client npm test -- --verbose
```

## Routes

| Endpoint | HTTP Method | Authenticated? | Active? | Admin? |
|---|---|---|---|---|
| /auth/register | POST | No | N/A | N/A |
| /auth/login | POST | No | N/A | N/A |
| /auth/logout | GET | Yes | Yes | No |
| /auth/status | GET | Yes | Yes | No |
| /users | GET | No | N/A | N/A |
| /users/:id | GET | No | N/A | N/A |
| /users | POST | Yes | Yes | Yes |
| /users/ping | GET | No | N/A | N/A |

Users must be active to view authenticated routes, and users must be an admin to POST to the `/users` endpoint.

## Active

Start with a test. Add the following to *services/users/project/tests/test_auth.py*:

```
def test_invalid_logout_inactive(self):
    add_user('test', 'test@test.com', 'test')
    # update user
    user = User.query.filter_by(email='test@test.com').first()
    user.active = False
    db.session.commit()
    with self.client:
        resp_login = self.client.post(
            '/auth/login',
            data=json.dumps({
                'email': 'test@test.com',
                'password': 'test'
            }),
            content_type='application/json'
        )
        token = json.loads(resp_login.data.decode())['auth_token']
        response = self.client.get(
            '/auth/logout',
            headers={'Authorization': f'Bearer {token}'}
        )
        data = json.loads(response.data.decode())
        self.assertTrue(data['status'] == 'fail')
        self.assertTrue(data['message'] == 'Provide a valid auth token.')
        self.assertEqual(response.status_code, 401)
```

Add the imports:

```
from project import db
from project.api.models import User
```

Ensure the tests fail, and then update `logout_user()` in *services/users/project/api/auth.py*:

```
@auth_blueprint.route('/auth/logout', methods=['GET'])
def logout_user():
    # get auth token
    auth_header = request.headers.get('Authorization')
    response_object = {
        'status': 'fail',
        'message': 'Provide a valid auth token.'
    }
    if auth_header:
```

```
        auth_token = auth_header.split(' ')[1]
        resp = User.decode_auth_token(auth_token)
        if not isinstance(resp, str):
            user = User.query.filter_by(id=resp).first()
            if not user or not user.active:
                return jsonify(response_object), 401
            else:
                response_object['status'] = 'success'
                response_object['message'] = 'Successfully logged out.'
                return jsonify(response_object), 200
        else:
            response_object['message'] = resp
            return jsonify(response_object), 401
    else:
        return jsonify(response_object), 403
```

Before moving on, let's do a quick refactor to keep our code DRY. We can move the auth logic out of the route handler and into a decorator.

Create a new file in "project/api" called *utils.py*:

```
# services/users/project/api/utils.py


from functools import wraps

from flask import request, jsonify

from project.api.models import User


def authenticate(f):
    @wraps(f)
    def decorated_function(*args, **kwargs):
        response_object = {
            'status': 'fail',
            'message': 'Provide a valid auth token.'
        }
        auth_header = request.headers.get('Authorization')
        if not auth_header:
            return jsonify(response_object), 403
        auth_token = auth_header.split(" ")[1]
        resp = User.decode_auth_token(auth_token)
        if isinstance(resp, str):
            response_object['message'] = resp
            return jsonify(response_object), 401
        user = User.query.filter_by(id=resp).first()
        if not user or not user.active:
            return jsonify(response_object), 401
        return f(resp, *args, **kwargs)
```

```
    return decorated_function
```

Here, we abstracted out all the logic for ensuring a token is present and valid and that the associated user is active.

Import the decorator into *services/users/project/api/auth.py*:

```
from project.api.utils import authenticate
```

Update the view:

```
@auth_blueprint.route('/auth/logout', methods=['GET'])
@authenticate
def logout_user(resp):
    response_object = {
        'status': 'success',
        'message': 'Successfully logged out.'
    }
    return jsonify(response_object), 200
```

The code is DRY and now we can test the auth logic separate from the view in a unit test! Win-win. Let's do the same thing for the `/auth/status` endpoint.

Add the test:

```
def test_invalid_status_inactive(self):
    add_user('test', 'test@test.com', 'test')
    # update user
    user = User.query.filter_by(email='test@test.com').first()
    user.active = False
    db.session.commit()
    with self.client:
        resp_login = self.client.post(
            '/auth/login',
            data=json.dumps({
                'email': 'test@test.com',
                'password': 'test'
            }),
            content_type='application/json'
        )
        token = json.loads(resp_login.data.decode())['auth_token']
        response = self.client.get(
            '/auth/status',
            headers={'Authorization': f'Bearer {token}'}
        )
        data = json.loads(response.data.decode())
        self.assertTrue(data['status'] == 'fail')
        self.assertTrue(data['message'] == 'Provide a valid auth token.')
```

```
        self.assertEqual(response.status_code, 401)
```

Now, update `get_user_status()` :

```
@auth_blueprint.route('/auth/status', methods=['GET'])
@authenticate
def get_user_status(resp):
    user = User.query.filter_by(id=resp).first()
    response_object = {
        'status': 'success',
        'message': 'success',
        'data': user.to_json()
    }
    return jsonify(response_object), 200
```

Make sure the tests pass:

```
----------------------------------------------------------------------
Ran 39 tests in 1.349s

OK
```

Moving on, for the `/users` POST endpoint, add a new test: to *services/users/project/tests/test_users.py*

```
def test_add_user_inactive(self):
    add_user('test', 'test@test.com', 'test')
    # update user
    user = User.query.filter_by(email='test@test.com').first()
    user.active = False
    db.session.commit()
    with self.client:
        resp_login = self.client.post(
            '/auth/login',
            data=json.dumps({
                'email': 'test@test.com',
                'password': 'test'
            }),
            content_type='application/json'
        )
        token = json.loads(resp_login.data.decode())['auth_token']
        response = self.client.post(
            '/users',
            data=json.dumps({
                'username': 'michael',
                'email': 'michael@sonotreal.com',
                'password': 'test'
            }),
```

```
            content_type='application/json',
            headers={'Authorization': f'Bearer {token}'}
        )
        data = json.loads(response.data.decode())
        self.assertTrue(data['status'] == 'fail')
        self.assertTrue(data['message'] == 'Provide a valid auth token.')
        self.assertEqual(response.status_code, 401)
```

Add the imports:

```
from project import db
from project.api.models import User
```

Make sure it fails, and then add the decorator to `add_user()` in *services/users/project/api/users.py*:

```
@users_blueprint.route('/users', methods=['POST'])
@authenticate
def add_user(resp):
    ...
```

Don't forget the import:

```
from project.api.utils import authenticate
```

Run the tests. You should see a number of failures since we are not passing a valid token within the requests in the remaining tests for that endpoint:

```
FAIL: test_add_user (test_users.TestUserService)
FAIL: test_add_user_duplicate_email (test_users.TestUserService)
FAIL: test_add_user_invalid_json (test_users.TestUserService)
FAIL: test_add_user_invalid_json_keys (test_users.TestUserService)
FAIL: test_add_user_invalid_json_keys_no_password (test_users.TestUserService)
```

To fix, in each of the failing tests, you need to-

1. Add a user:

   ```
   add_user('test', 'test@test.com', 'test')
   ```

2. Log the user in:

   ```
   resp_login = self.client.post(
       '/auth/login',
       data=json.dumps({
           'email': 'test@test.com',
           'password': 'test'
   ```

```
    }),
    content_type='application/json'
)
```

3. Add the token to the request:

```
token = json.loads(resp_login.data.decode())['auth_token']
response = self.client.post(
    '/users',
    data=json.dumps({
        'username': 'michael',
        'email': 'michael@sonotreal.com',
        'password': 'test'
    }),
    content_type='application/json',
    headers={'Authorization': f'Bearer {token}'}
)
```

Refactor as necessary. Test again to make sure all tests pass:

```
----------------------------------------------------------------------
Ran 40 tests in 1.490s

OK
```

## Admin

Finally, in order to POST to the `/users` endpoint, you must be an admin. Turn to the models. Do we have an admin property? No. Let's add one. Start by adding an additional assert to the `test_add_user` test in *services/users/project/tests/test_user_model.py*:

```
def test_add_user(self):
    user = add_user('justatest', 'test@test.com', 'test')
    self.assertTrue(user.id)
    self.assertEqual(user.username, 'justatest')
    self.assertEqual(user.email, 'test@test.com')
    self.assertTrue(user.password)
    self.assertTrue(user.active)
    self.assertFalse(user.admin)
```

After the tests fail - `AttributeError: 'User' object has no attribute 'admin'` - add the property to the model:

```
admin = db.Column(db.Boolean, default=False, nullable=False)
```

Create the migration:

```
$ docker-compose -f docker-compose-dev.yml \
  run users python manage.py db migrate
```

Do not apply it to the actual database just yet, though. Instead, find the newly created migration file and change the `upgrade()`:

```
def upgrade():
    # ### commands auto generated by Alembic - please adjust! ###
    op.add_column('users', sa.Column('admin', sa.Boolean(), nullable=True))
    op.execute('UPDATE users SET admin=False')
    op.alter_column('users', 'admin', nullable=False)
    # ### end Alembic commands ###
```

Now, when we apply the migration, `nullable` is first set to `true`, the users are updated, and then `nullable` is changed to `false`.

```
$ docker-compose -f docker-compose-dev.yml \
  run users python manage.py db upgrade
```

The tests should pass. Next, let's add a new test to *services/users/project/tests/test_users.py*:

```
def test_add_user_not_admin(self):
    add_user('test', 'test@test.com', 'test')
    with self.client:
        # user login
        resp_login = self.client.post(
            '/auth/login',
            data=json.dumps({
                'email': 'test@test.com',
                'password': 'test'
            }),
            content_type='application/json'
        )
        token = json.loads(resp_login.data.decode())['auth_token']
        response = self.client.post(
            '/users',
            data=json.dumps({
                'username': 'michael',
                'email': 'michael@sonotreal.com',
                'password': 'test'
            }),
            content_type='application/json',
            headers={'Authorization': f'Bearer {token}'}
        )
        data = json.loads(response.data.decode())
        self.assertTrue(data['status'] == 'fail')
        self.assertTrue(
```

```
            data['message'] == 'You do not have permission to do that.')
        self.assertEqual(response.status_code, 401)
```

Add a helper to *services/users/project/api/utils.py*:

```
def is_admin(user_id):
    user = User.query.filter_by(id=user_id).first()
    return user.admin
```

Import it in to *services/users/project/api/users.py*, and then add the check to the top of the function:

```
@users_blueprint.route('/users', methods=['POST'])
@authenticate
def add_user(resp):
    post_data = request.get_json()
    response_object = {
        'status': 'fail',
        'message': 'Invalid payload.'
    }
    if not is_admin(resp):
        response_object['message'] = 'You do not have permission to do that.'
        return jsonify(response_object), 401
    ...
```

The full view should now look like:

```
@users_blueprint.route('/users', methods=['POST'])
@authenticate
def add_user(resp):
    post_data = request.get_json()
    response_object = {
        'status': 'fail',
        'message': 'Invalid payload.'
    }
    if not is_admin(resp):
        response_object['message'] = 'You do not have permission to do that.'
        return jsonify(response_object), 401
    if not post_data:
        return jsonify(response_object), 400
    username = post_data.get('username')
    email = post_data.get('email')
    password = post_data.get('password')
    try:
        user = User.query.filter_by(email=email).first()
        if not user:
            db.session.add(User(
                username=username, email=email, password=password))
            db.session.commit()
```

```
            response_object['status'] = 'success'
            response_object['message'] = f'{email} was added!'
            return jsonify(response_object), 201
        else:
            response_object['message'] = 'Sorry. That email already exists.'
            return jsonify(response_object), 400
    except exc.IntegrityError as e:
        db.session.rollback()
        return jsonify(response_object), 400
    except (exc.IntegrityError, ValueError) as e:
        db.session.rollback()
        return jsonify(response_object), 400
```

Run the tests. Even though `test_add_user_not_admin` should now pass, you should see a number of failures:

```
test_add_user (test_users.TestUserService)
test_add_user_duplicate_email (test_users.TestUserService)
test_add_user_invalid_json (test_users.TestUserService)
test_add_user_invalid_json_keys (test_users.TestUserService)
test_add_user_invalid_json_keys_no_password (test_users.TestUserService)
```

Add the following to the top of the failing tests, right after `add_user('test', 'test@test.com', 'test')`:

```
# update user
user = User.query.filter_by(email='test@test.com').first()
user.admin = True
db.session.commit()
```

Test it again:

```
----------------------------------------------------------------------
Ran 41 tests in 1.476s

OK
```

> You may want to encapsulate the logic of adding a new admin user into a helper function:
>
> ```
> def add_admin(username, email, password):
>     user = User(
>       username=username, email=email,
>       password=password, admin=True
>     )
>     db.session.add(user)
>     db.session.commit()
>     return user
> ```

## `to_json`

Before moving on, we should update the `to_json` method in *services/users/project/api/models.py* since we updated the model. This will affect the data sent back in these routes:

1. /auth/status
2. /users

So, let's update the tests.

1. `test_user_status` :

```
def test_user_status(self):
    add_user('test', 'test@test.com', 'test')
    with self.client:
        resp_login = self.client.post(
            '/auth/login',
            data=json.dumps({
                'email': 'test@test.com',
                'password': 'test'
            }),
            content_type='application/json'
        )
        token = json.loads(resp_login.data.decode())['auth_token']
        response = self.client.get(
            '/auth/status',
            headers={'Authorization': f'Bearer {token}'}
        )
        data = json.loads(response.data.decode())
        self.assertTrue(data['status'] == 'success')
        self.assertTrue(data['data'] is not None)
        self.assertTrue(data['data']['username'] == 'test')
        self.assertTrue(data['data']['email'] == 'test@test.com')
        self.assertTrue(data['data']['active'])
        self.assertFalse(data['data']['admin'])
        self.assertEqual(response.status_code, 200)
```

2. `test_all_users` :

```
def test_all_users(self):
    """Ensure get all users behaves correctly."""
    add_user('michael', 'michael@mherman.org', 'test')
    add_user('fletcher', 'fletcher@noteal.com', 'test')
    with self.client:
        response = self.client.get('/users')
        data = json.loads(response.data.decode())
        self.assertEqual(response.status_code, 200)
        self.assertEqual(len(data['data']['users']), 2)
        self.assertIn('michael', data['data']['users'][0]['username'])
```

```
            self.assertIn(
                'michael@mherman.org', data['data']['users'][0]['email'])
            self.assertTrue(data['data']['users'][0]['active'])
            self.assertFalse(data['data']['users'][0]['admin'])
            self.assertIn('fletcher', data['data']['users'][1]['username'])
            self.assertIn(
                'fletcher@noteal.com', data['data']['users'][1]['email'])
            self.assertTrue(data['data']['users'][1]['active'])
            self.assertFalse(data['data']['users'][1]['admin'])
            self.assertIn('success', data['status'])
```

Make sure the tests fail:

```
self.assertFalse(data['data']['admin'])
KeyError: 'admin'
```

Update the method:

```
def to_json(self):
    return {
        'id': self.id,
        'username': self.username,
        'email': self.email,
        'active': self.active,
        'admin': self.admin
    }
```

Ensure the tests pass:

```
----------------------------------------------------------------------
Ran 41 tests in 1.474s

OK
```

How about coverage?

```
Coverage Summary:
Name                        Stmts   Miss Branch BrPart  Cover
----------------------------------------------------------
project/__init__.py            26     12      0      0    54%
project/api/auth.py            60     16     10      2    74%
project/api/models.py          32     18      2      0    47%
project/api/users.py           58     12     14      0    83%
project/api/utils.py           22      8      6      1    68%
----------------------------------------------------------
TOTAL                         198     66     32      3    70%
```

Commit your code and move on.

> It's probably a good time to refactor some of the tests to keep them DRY. Do this on your own.

# Update Component

In this lesson, we'll refactor the `UsersList` and `UserStatus` components...

---

## Docker Machine

Set `testdriven-dev` as the active Docker Machine:

```
$ docker-machine env testdriven-dev
$ eval $(docker-machine env testdriven-dev)
```

Ensure the app is working in the browser, and then run the tests:

```
$ docker-compose -f docker-compose-dev.yml \
  run users python manage.py test

$ docker-compose -f docker-compose-dev.yml \
  run users flake8 project

$ docker-compose -f docker-compose-dev.yml \
  run client npm test -- --verbose
```

## UsersList

Let's remove the add user form and display a Bootstrap-styled table of users...

### Remove the form

To remove the form, update *UsersList.jsx*:

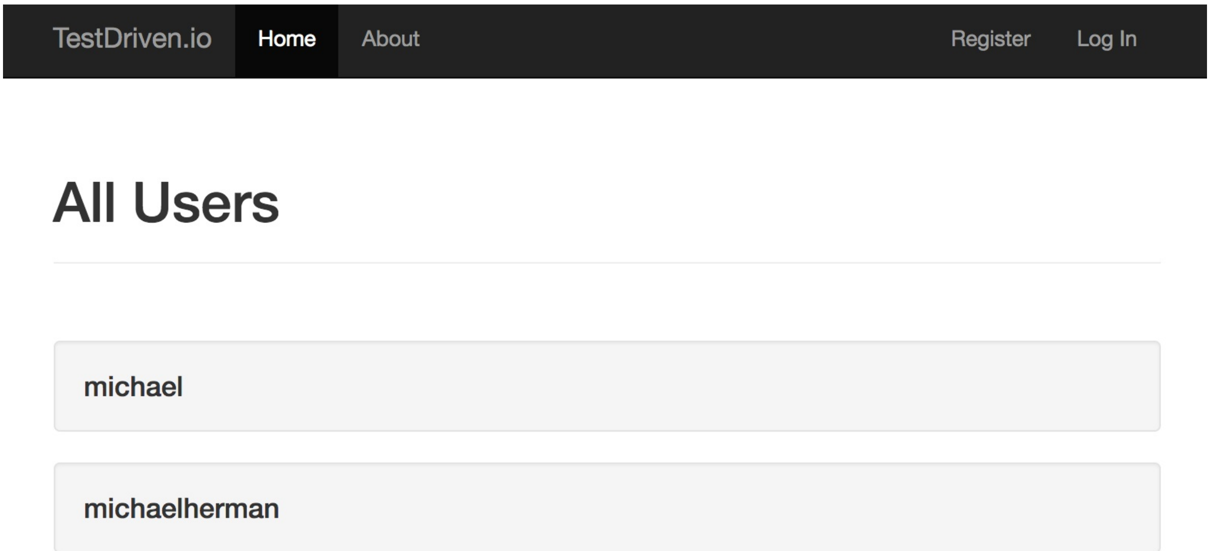
```
import React from 'react';

const UsersList = (props) => {
  return (
    <div>
      <h1>All Users</h1>
      <hr/><br/>
      {
        props.users.map((user) => {
          return (
            <h4
              key={user.id}
              className="well"
            >{user.username}
            </h4>
```

```
            )
          })
        }
      </div>
    )
};

export default UsersList;
```

Update the route in the `App` component:

```
<Route exact path='/' render={() => (
  <UsersList
    users={this.state.users}
  />
)} />
```

Make sure to remove the `AddUser` import at the top of the file, and then test it out in the browser:




What about the tests?

```
$ docker-compose -f docker-compose-dev.yml \
  run client npm test -- --verbose
```

You should see the snapshot test fail:

```
× UsersList renders a snapshot properly
```

It should also spit out the diff to the terminal:

```
FAIL  src/components/__tests__/UsersList.test.js
 ● UsersList renders a snapshot properly
```

```
  expect(value).toMatchSnapshot()

  Received value does not match stored snapshot 1.

  - Snapshot
  + Received

  @@ -1,6 +1,11 @@
   <div>
  +   <h1>
  +     All Users
  +   </h1>
  +   <hr />
  +   <br />
      <h4
        className="well"
      >
        michael
      </h4>
```

Since this is expected, update the snapshot by pressing the `u` key:

```
Watch Usage
 › Press a to run all tests.
 › Press u to update failing snapshots.
 › Press p to filter by a filename regex pattern.
 › Press t to filter by a test name regex pattern.
 › Press q to quit watch mode.
 › Press Enter to trigger a test run.
```

All tests should now pass. Before moving on, since `<h1>All Users</h1>` is now part of the `UsersList` component, let's update the unit test:

```
test('UsersList renders properly', () => {
  const wrapper = shallow(<UsersList users={users}/>);
  const element = wrapper.find('h4');
  expect(wrapper.find('h1').get(0).props.children).toBe('All Users');
  expect(element.length).toBe(2);
  expect(element.get(0).props.className).toBe('well');
  expect(element.get(0).props.children).toBe('michael');
});
```

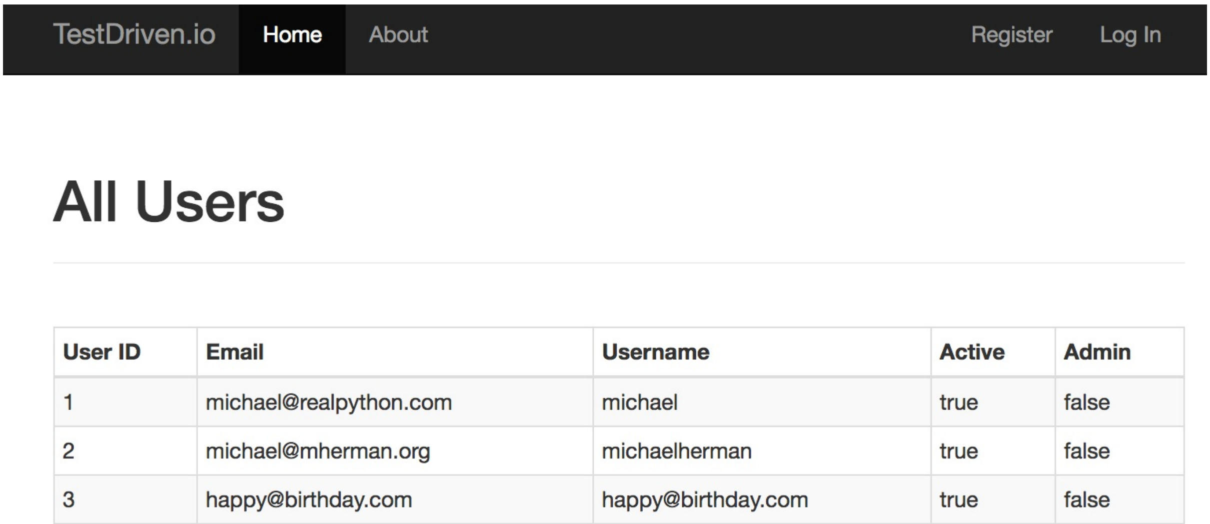
## Add a table

Next, let's use React-Bootstrap to add a table to the `UsersList` component:

```
import React from 'react';
```

```
import { Table } from 'react-bootstrap';

const UsersList = (props) => {
  return (
    <div>
      <h1>All Users</h1>
      <hr/><br/>
      <Table striped bordered condensed hover>
        <thead>
          <tr>
            <th>User ID</th>
            <th>Email</th>
            <th>Username</th>
            <th>Active</th>
            <th>Admin</th>
          </tr>
        </thead>
        <tbody>
          {
            props.users.map((user) => {
              return (
                <tr key={user.id}>
                  <td>{user.id}</td>
                  <td>{user.email}</td>
                  <td>{user.username}</td>
                  <td>{String(user.active)}</td>
                  <td>{String(user.admin)}</td>
                </tr>
              )
            })
          }
        </tbody>
      </Table>
    </div>
  )
};

export default UsersList;
```

The rendered component should now look like:




Update the test:

```
test('UsersList renders properly', () => {
  const wrapper = shallow(<UsersList users={users}/>);
  expect(wrapper.find('h1').get(0).props.children).toBe('All Users');
  // table
  const table = wrapper.find('Table');
  expect(table.length).toBe(1);
  expect(table.get(0).props.striped).toBe(true);
  expect(table.get(0).props.bordered).toBe(true);
  expect(table.get(0).props.condensed).toBe(true);
  expect(table.get(0).props.hover).toBe(true);
  // table head
  expect(wrapper.find('thead').length).toBe(1);
  const th = wrapper.find('th');
  expect(th.length).toBe(5);
  expect(th.get(0).props.children).toBe('User ID');
  expect(th.get(1).props.children).toBe('Email');
  expect(th.get(2).props.children).toBe('Username');
  expect(th.get(3).props.children).toBe('Active');
  expect(th.get(4).props.children).toBe('Admin');
  // table body
  expect(wrapper.find('tbody').length).toBe(1);
  expect(wrapper.find('tbody > tr').length).toBe(2);
  const td = wrapper.find('tbody > tr > td');
  expect(td.length).toBe(10);
  expect(td.get(0).props.children).toBe(1);
  expect(td.get(1).props.children).toBe('hermanmu@gmail.com');
  expect(td.get(2).props.children).toBe('michael');
  expect(td.get(3).props.children).toBe('true');
  expect(td.get(4).props.children).toBe('false');
});
```

Make sure to update the fixture as well:

```
const users = [
  {
    'active': true,
    'admin': false,
    'email': 'hermanmu@gmail.com',
    'id': 1,
    'username': 'michael'
  },
  {
    'active': true,
    'admin': false,
    'email': 'michael@mherman.org',
    'id': 2,
    'username': 'michaelherman'
  }
]
```

Run the tests, making sure to update the snapshot test again:

```
Snapshot Summary
 › 1 snapshot updated in 1 test suite.

Test Suites: 7 passed, 7 total
Tests:       19 passed, 19 total
Snapshots:   1 updated, 6 passed, 7 total
Time:        5.178s, estimated 10s
Ran all test suites.
```

## UserStatus

Next, let's add the `active` and `admin` properties to the `UserStatus` component:

```
class UserStatus extends Component {
  constructor (props) {
    super(props);
    this.state = {
      email: '',
      id: '',
      username: '',
      active: '',
      admin: ''
    };
  };
  componentDidMount() {
    if (this.props.isAuthenticated) {
      this.getUserStatus();
```

```
    };
  }
  getUserStatus(event) {
    const options = {
      url: `${process.env.REACT_APP_USERS_SERVICE_URL}/auth/status`,
      method: 'get',
      headers: {
        'Content-Type': 'application/json',
        Authorization: `Bearer ${window.localStorage.authToken}`
      }
    };
    return axios(options)
    .then((res) => {
      this.setState({
        email: res.data.data.email,
        id: res.data.data.id,
        username: res.data.data.username,
        active: String(res.data.data.active),
        admin: String(res.data.data.admin),
      })
    })
    .catch((error) => { console.log(error); });
  };
  render() {
    if (!this.props.isAuthenticated) {
      return <p>You must be logged in to view this. Click <Link to="/login">here</Link> to log back in.</p>
    };
    return (
      <div>
        <ul>
          <li><strong>User ID:</strong> {this.state.id}</li>
          <li><strong>Email:</strong> {this.state.email}</li>
          <li><strong>Username:</strong> {this.state.username}</li>
          <li><strong>Active:</strong> {this.state.active}</li>
          <li><strong>Admin:</strong> {this.state.admin}</li>
        </ul>
      </div>
    )
  };
};
```

> We'll look at how to test this one in a future lesson.

That's it. Short lesson. Make sure the tests still pass.

```
PASS  src/components/__tests__/NavBar.test.jsx
 ✓ NavBar renders properly (5ms)
 ✓ NavBar renders a snapshot properly (13ms)
```

```
PASS  src/components/__tests__/App.test.jsx
 ✓ App renders without crashing (6ms)

PASS  src/components/__tests__/UsersList.test.jsx
 ✓ UsersList renders properly (12ms)
 ✓ UsersList renders a snapshot properly (3ms)

PASS  src/components/__tests__/Form.test.jsx
 When not authenticated
   ✓ Register Form renders properly (4ms)
   ✓ Register Form submits the form properly (2ms)
   ✓ Register Form renders a snapshot properly (2ms)
   ✓ Login Form renders properly (2ms)
   ✓ Login Form submits the form properly (1ms)
   ✓ Login Form renders a snapshot properly (1ms)
 When authenticated
   ✓ Register redirects properly
   ✓ Login redirects properly (1ms)

PASS  src/components/__tests__/AddUser.test.jsx
 ✓ AddUser renders properly (5ms)
 ✓ AddUser renders a snapshot properly (2ms)

PASS  src/components/__tests__/Logout.test.jsx
 ✓ Logout renders properly (4ms)
 ✓ Logout renders a snapshot properly (11ms)

PASS  src/components/__tests__/About.test.jsx
 ✓ About renders properly (2ms)
 ✓ About renders a snapshot properly (1ms)

Test Suites: 7 passed, 7 total
Tests:       19 passed, 19 total
Snapshots:   7 passed, 7 total
Time:        4.163s
Ran all test suites.
```

Commit you code.

> You may have noticed that we are not handling errors on the client. We'll tackle that in an upcoming lesson!

# Update Docker

In this last lesson, we'll update Docker on AWS...

---

Change the machine to `testdriven-prod`:

```
$ docker-machine env testdriven-prod
$ eval $(docker-machine env testdriven-prod)
```

We need to add the `SECRET_KEY` environment variable for the `users` in *docker-compose-prod.yml*:

```
users:
  container_name: users
  build:
    context: ./services/users
    dockerfile: Dockerfile-prod
  expose:
    - '5000'
  environment:
    - APP_SETTINGS=project.config.ProductionConfig
    - DATABASE_URL=postgres://postgres:postgres@users-db:5432/users_dev
    - DATABASE_TEST_URL=postgres://postgres:postgres@users-db:5432/users_test
    - SECRET_KEY=${SECRET_KEY}
  depends_on:
    - users-db
  links:
    - users-db
```

Since this key should truly be random, we'll set the key locally and pull it into the container at the build time.

To create a key, open the Python shell and run:

```
>>> import binascii
>>> import os
>>> binascii.hexlify(os.urandom(24))
b'0ccd512f8c3493797a23557c32db38e7d51ed74f14fa7580'
```

Exit the shell. Set it as an environment variable:

```
$ export SECRET_KEY=0ccd512f8c3493797a23557c32db38e7d51ed74f14fa7580
```

Grab the IP for the `testdriven-prod` machine and use it for the `REACT_APP_USERS_SERVICE_URL` environment variable:

```
$ export REACT_APP_USERS_SERVICE_URL=http://DOCKER_MACHINE_AWS_IP
```

Then, update the containers:

```
$ docker-compose -f docker-compose-prod.yml up -d --build
```

Re-create and seed the database:

```
$ docker-compose -f docker-compose-prod.yml \
  run users python manage.py recreate_db

$ docker-compose -f docker-compose-prod.yml \
  run users python manage.py seed_db
```

Manually test it in the browser. Try navigating to an individual route from the URL bar:

1. http://DOCKER_MACHINE_AWS_IP/login
2. http://DOCKER_MACHINE_AWS_IP/about

You should see a 404. Why? Essentially, the Docker Nginx image is overriding the behavior of React Router.

To fix, update *services/client/Dockerfile-prod*:

```
# build environment
FROM node:9.4 as builder
RUN mkdir /usr/src/app
WORKDIR /usr/src/app
ENV PATH /usr/src/app/node_modules/.bin:$PATH
ARG REACT_APP_USERS_SERVICE_URL
ARG NODE_ENV
ENV NODE_ENV $NODE_ENV
ENV REACT_APP_USERS_SERVICE_URL $REACT_APP_USERS_SERVICE_URL
COPY package.json /usr/src/app/package.json
RUN npm install --silent
RUN npm install react-scripts@1.1.0 -g --silent
COPY . /usr/src/app
RUN npm run build

# production environment
FROM nginx:1.13.5-alpine
RUN rm -rf /etc/nginx/conf.d
COPY conf /etc/nginx
COPY --from=builder /usr/src/app/build /usr/share/nginx/html
EXPOSE 80
```

```
CMD ["nginx", "-g", "daemon off;"]
```

Take note of the two new lines:

```
RUN rm -rf /etc/nginx/conf.d
COPY conf /etc/nginx
```

Here, we removed the default Nginx configuration and replaced it with our own. Add a new folder to "services/client" called "conf", add a new folder in "conf" called "conf.d", and then add a new file to "conf.d" called *default.conf*:

```
└── conf
    └── conf.d
        └── default.conf
```

Finally, update *default.conf*:

```
server {
  listen 80;
  location / {
    root   /usr/share/nginx/html;
    index  index.html index.htm;
    try_files $uri $uri/ /index.html;
  }
  error_page   500 502 503 504  /50x.html;
  location = /50x.html {
    root   /usr/share/nginx/html;
  }
}
```

Update the containers:

```
$ docker-compose -f docker-compose-prod.yml up -d --build
```

Manually test in the browser again. Commit and push your code once done.

# Structure

At the end of part 3, your project structure should look like this:

```
├── README.md
├── docker-compose-dev.yml
├── docker-compose-prod.yml
└── services
    ├── client
    │   ├── Dockerfile-dev
    │   ├── Dockerfile-prod
    │   ├── README.md
    │   ├── build
    │   ├──── conf
    │   │      └── conf.d
    │   │          └── default.conf
    │   ├── coverage
    │   ├── package.json
    │   ├── public
    │   │   ├── favicon.ico
    │   │   ├── index.html
    │   │   └── manifest.json
    │   └── src
    │       ├── App.jsx
    │       ├── components
    │       │   ├── About.jsx
    │       │   ├── AddUser.jsx
    │       │   ├── Form.jsx
    │       │   ├── Logout.jsx
    │       │   ├── NavBar.jsx
    │       │   ├── UserStatus.jsx
    │       │   ├── UsersList.jsx
    │       │   └── __tests__
    │       │       ├── About.test.jsx
    │       │       ├── AddUser.test.jsx
    │       │       ├── App.test.jsx
    │       │       ├── Form.test.jsx
    │       │       ├── Logout.test.jsx
    │       │       ├── NavBar.test.jsx
    │       │       ├── UsersList.test.jsx
    │       │       └── __snapshots__
    │       │           ├── About.test.jsx.snap
    │       │           ├── AddUser.test.jsx.snap
    │       │           ├── Form.test.jsx.snap
    │       │           ├── Logout.test.jsx.snap
    │       │           ├── NavBar.test.jsx.snap
    │       │           └── UsersList.test.jsx.snap
    │       ├── index.js
    │       ├── logo.svg
```

```
    │       ├── registerServiceWorker.js
    │       └── setupTests.js
    ├── nginx
    │   ├── Dockerfile-dev
    │   ├── Dockerfile-prod
    │   ├── dev.conf
    │   └── prod.conf
    └── users
        ├── Dockerfile-dev
        ├── Dockerfile-prod
        ├── entrypoint-prod.sh
        ├── entrypoint.sh
        ├── htmlcov
        ├── manage.py
        ├── migrations
        ├── project
        │   ├── __init__.py
        │   ├── api
        │   │   ├── __init__.py
        │   │   ├── auth.py
        │   │   ├── models.py
        │   │   ├── templates
        │   │   │   └── index.html
        │   │   ├── users.py
        │   │   └── utils.py
        │   ├── config.py
        │   ├── db
        │   │   ├── Dockerfile
        │   │   └── create.sql
        │   └── tests
        │       ├── __init__.py
        │       ├── base.py
        │       ├── test_auth.py
        │       ├── test_config.py
        │       ├── test_user_model.py
        │       ├── test_users.py
        │       └── utils.py
        └── requirements.txt
```

Code for part 3: https://github.com/testdrivenio/testdriven-app-2.2/releases/tag/part3

# Part 4

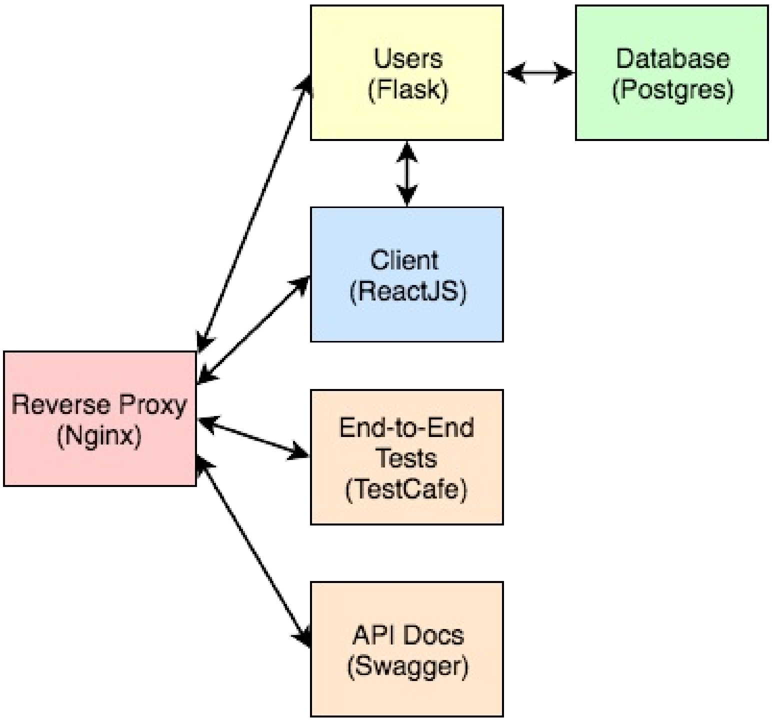
In part 4, we'll add an *end-to-end* (e2e) testing solution, *form validation* to the React app, a *Swagger* service to document the API, and deal with some *tech debt*. We'll also set up a *staging* environment to test on before the app goes into production.

## Objectives

By the end of part 4, you will be able to...

1. Test the entire set of services with functional, end-to-end tests via TestCafe
2. Integrate TestCafe into the continuous integration process
3. Handle form validation within React
4. Add a flash messaging system to the React app
5. Describe the purpose of Swagger
6. Generate a Swagger Spec based on an existing RESTful API
7. Configure Swagger to interact with a service running inside a Docker Container
8. Set up a staging environment on AWS

## App




Check out the live apps, running on EC2 -

1. Production
2. Staging

You can also test out the following endpoints...

| Endpoint | HTTP Method | Authenticated? | Result |
|---|---|---|---|
| /auth/register | POST | No | register user |
| /auth/login | POST | No | log in user |
| /auth/logout | GET | Yes | log out user |
| /auth/status | GET | Yes | check user status |
| /users | GET | No | get all users |
| /users/:id | GET | No | get single user |
| /users | POST | Yes (admin) | add a user |
| /users/ping | GET | No | sanity check |

Finished code for part 4: https://github.com/testdrivenio/testdriven-app-2.2/releases/tag/part4

## Dependencies

You will use the following dependencies in part 4:

1. TestCafe v0.18.6
2. node-randomstring v1.1.5
3. Swagger UI v3.9.3

# End-to-End Test Setup

In this lesson, we'll set up e2e testing with TestCafe...

## Test Script

Before adding TestCafe into the mix, let's simplify the running of tests. Start by setting `testdriven-dev` as the active Docker Machine:

```
$ docker-machine env testdriven-dev
$ eval $(docker-machine env testdriven-dev)
```

Ensure the app is working in the browser, and then run the tests:

```
$ docker-compose -f docker-compose-dev.yml \
  run users python manage.py test

$ docker-compose -f docker-compose-dev.yml \
  run users flake8 project

$ docker-compose -f docker-compose-dev.yml \
  run client npm test -- --verbose
```

Once done, add a new file to the project root called *test.sh*:

```
#!/bin/bash

env=$1
file=""
fails=""

if [[ "${env}" == "stage" ]]; then
  file="docker-compose-dev.yml"
elif [[ "${env}" == "dev" ]]; then
  file="docker-compose-dev.yml"
elif [[ "${env}" == "prod" ]]; then
  file="docker-compose-prod.yml"
else
  echo "USAGE: sh test.sh environment_name"
  echo "* environment_name: must either be 'dev', 'stage', or 'prod'"
  exit 1
fi

inspect() {
  if [ $1 -ne 0 ]; then
```

```
    fails="${fails} $2"
  fi
}

docker-compose -f $file run users python manage.py test
inspect $? users
docker-compose -f $file run users flake8 project
inspect $? users-lint
if [[ "${env}" == "dev" ]]; then
  docker-compose -f $file run client npm test -- --coverage
  inspect $? client
fi

if [ -n "${fails}" ]; then
  echo "Tests failed: ${fails}"
  exit 1
else
  echo "Tests passed!"
  exit 0
fi
```

Here, we run the tests, calculate the number of failures (via the `inspect` function), and then exit with the proper code.

Run the tests to ensure all is well:

```
$ sh test.sh dev
```

Update the `script` in *.travis.yml*:

```
script:
  - bash test.sh stage
```

Commit and push your code to ensure the tests still pass on Travis.

## TestCafe

Unlike the majority of other end-to-end (e2e) testing tools, TestCafe is not dependent on Selenium or WebDriver. Instead, it injects scripts into the browser to communicate directly with the DOM and handle events. It works on any modern browser that supports HTML5 without any plugins.

> Please review the Getting Started guide before beginning.

To install, first add a *package.json* to the project root:

```
{
  "name": "test-driven"
}
```

Next, to simplify the development process, let's tell npm not to create a package-lock.json file for this project:

```
$ echo 'package-lock=false' >> .npmrc
```

> Review the npm docs for more info on the *.npmrc* config file.

Then install the dependency:

```
$ npm install testcafe@0.18.6 --save-dev
```

The `--save-dev` flag adds the dependency info to the *package.json* as a development dependency:

```
{
  "name": "test-driven",
  "devDependencies": {
    "testcafe": "^0.18.6"
  }
}
```

The dependency (and sub dependencies) were installed to a newly created "node_modules" directory. Add this directory to the *.gitignore* file.

Let's write our first test spec!

## First Test

First, add a new folder to the project root called "e2e". Then add a new file to that folder called *index.test.js*:

```
import { Selector } from 'testcafe';

const TEST_URL = process.env.TEST_URL;


fixture('/').page(`${TEST_URL}/`);

test(`users should be able to view the '/' page`, async (t) => {

  await t
    .navigateTo(TEST_URL)
    .expect(Selector('H1').withText('All Users').exists).ok()

});
```

This test simply navigates to the main URL, `/`, and then asserts that an `H1` element exists with the text `All Users`.

Set the environment variable:

```
$ export TEST_URL=http://DOCKER_MACHINE_DEV_IP
```

Run the tests:

```
$ testcafe chrome e2e
```

You should see a new Chrome window open, which navigates to the main page and then TestCafe runs the assertion.

In the terminal, you should see something like:

```
Using locally installed version of TestCafe.
 Running tests in:
 - Chrome 63.0.3239 / Mac OS X 10.12.0

 /
 ✓ users should be able to view the '/' page


 1 passed (1s)
```

Experiment with this. Try navigating to a different page. Add a click action. Set up additional selectors and run some more assertions.

## CI

Add the test to the *test.sh* file:

```
#!/bin/bash

env=$1
file=""
fails=""

if [[ "${env}" == "stage" ]]; then
  file="docker-compose-dev.yml"
elif [[ "${env}" == "dev" ]]; then
  file="docker-compose-dev.yml"
elif [[ "${env}" == "prod" ]]; then
  file="docker-compose-prod.yml"
else
  echo "USAGE: sh test.sh environment_name"
```

```
  echo "* environment_name: must either be 'dev', 'stage', or 'prod'"
  exit 1
fi

inspect() {
  if [ $1 -ne 0 ]; then
    fails="${fails} $2"
  fi
}

docker-compose -f $file run users python manage.py test
inspect $? users
docker-compose -f $file run users flake8 project
inspect $? users-lint
if [[ "${env}" == "dev" ]]; then
  docker-compose -f $file run client npm test -- --coverage
  inspect $? client
  testcafe chrome e2e
  inspect $? e2e
else
  testcafe chrome e2e/index.test.js
  inspect $? e2e
fi

if [ -n "${fails}" ]; then
  echo "Tests failed: ${fails}"
  exit 1
else
  echo "Tests passed!"
  exit 0
fi
```

Make sure the tests pass again, and then update *.travis.yml* like so:

```
language: node_js
node_js: '9'

before_install:
  - stty cols 80

dist: trusty
sudo: required

addons:
  apt:
    sources:
     - google-chrome
    packages:
     - google-chrome-stable
```

```
services:
  - docker

env:
  DOCKER_COMPOSE_VERSION: 1.18.0

before_install:
  - sudo rm /usr/local/bin/docker-compose
  - curl -L https://github.com/docker/compose/releases/download/${DOCKER_COMPOSE_VE
RSION}/docker-compose-`uname -s`-`uname -m` > docker-compose
  - chmod +x docker-compose
  - sudo mv docker-compose /usr/local/bin

before_script:
  - export TEST_URL=http://127.0.0.1
  - export REACT_APP_USERS_SERVICE_URL=http://127.0.0.1
  - export DISPLAY=:99.0
  - sh -e /etc/init.d/xvfb start
  - sleep 3
  - docker-compose -f docker-compose-prod.yml up --build -d

script:
  - bash test.sh stage

after_script:
  - docker-compose -f docker-compose-prod.yml down
```

Here, we added the Node version along with some basic Chrome settings. Also, we used xvfb to fake a GUI so that Chrome thinks it's running in a graphical environment.

> Review the Running Tests in Firefox and Chrome Using Travis CI for more info.

Commit your code and push it up to GitHub. Make sure the tests pass on Travis.

# End-to-End Test Specs

With TestCafe in place, we can now write some test cases...

---

What should we test?

Turn to your app. Navigate through it as an end user. What are some common user interactions? How about frequent error cases that you expect *most* users to encounter?

Turn your answers into test cases...

## Test Cases

`/register` :

1. should display the registration form
2. should allow a user to register
3. should throw an error if the username is taken
4. should throw an error if the email address is taken

`/login` :

1. should display the sign in form
2. should allow a user to sign in
3. should throw an error if the credentials are incorrect

`/logout` :

1. should log a user out

`/status` :

1. should display user info if a user is logged in
2. should not display user info if a user is not logged in

`/` :

1. should display the page correctly if a user is not logged in

## Register

Add a new file called *register.test.js* to the "e2e" directory:

```
import { Selector } from 'testcafe';

const TEST_URL = process.env.TEST_URL;
```

```
fixture('/register').page(`${TEST_URL}/register`);
```

Now add the following test specs:

1. *should display the registration form*

```
test(`should display the registration form`, async (t) => {
  await t
    .navigateTo(`${TEST_URL}/register`)
    .expect(Selector('H1').withText('Register').exists).ok()
    .expect(Selector('form').exists).ok()
});
```

2. *should allow a user to register*

```
test(`should allow a user to register`, async (t) => {

  // register user
  await t
    .navigateTo(`${TEST_URL}/register`)
    .typeText('input[name="username"]', username)
    .typeText('input[name="email"]', email)
    .typeText('input[name="password"]', 'test')
    .click(Selector('input[type="submit"]'))

  // assert user is redirected to '/'
  // assert '/' is displayed properly
  const tableRow = Selector('td').withText(username).parent();
  await t
    .expect(Selector('H1').withText('All Users').exists).ok()
    .expect(tableRow.child().withText(username).exists).ok()
    .expect(tableRow.child().withText(email).exists).ok()
    .expect(Selector('a').withText('User Status').exists).ok()
    .expect(Selector('a').withText('Log Out').exists).ok()
    .expect(Selector('a').withText('Register').exists).notOk()
    .expect(Selector('a').withText('Log In').exists).notOk()

});
```

Add the import and global variables at the top:

```
const randomstring = require('randomstring');

const username = randomstring.generate();
const email = `${username}@test.com`;
```

Make sure to install the dependency as well:

```
$ npm install randomstring@1.1.5 --save-dev
```

Since we're not handling errors yet, let's hold off on these two test cases:

1. *should throw an error if the username is taken*
2. *should throw an error if the email address is taken*

## Login

> Try writing the next few test cases on your own!

Add a new file called *login.test.js* to the "e2e" directory:

```
import { Selector } from 'testcafe';

const randomstring = require('randomstring');

const username = randomstring.generate();
const email = `${username}@test.com`;

const TEST_URL = process.env.TEST_URL;


fixture('/login').page(`${TEST_URL}/login`);
```

Now add the following test specs:

1. *should display the sign in form*

```
test(`should display the sign in form`, async (t) => {
  await t
    .navigateTo(`${TEST_URL}/login`)
    .expect(Selector('H1').withText('Login').exists).ok()
    .expect(Selector('form').exists).ok()
});
```

2. *should allow a user to sign in*

```
test(`should allow a user to sign in`, async (t) => {

  // register user
  await t
    .navigateTo(`${TEST_URL}/register`)
    .typeText('input[name="username"]', username)
    .typeText('input[name="email"]', email)
    .typeText('input[name="password"]', 'test')
    .click(Selector('input[type="submit"]'))
```

```
    // log a user out
    await t
      .click(Selector('a').withText('Log Out'))

    // log a user in
    await t
      .navigateTo(`${TEST_URL}/login`)
      .typeText('input[name="email"]', email)
      .typeText('input[name="password"]', 'test')
      .click(Selector('input[type="submit"]'))

    // assert user is redirected to '/'
    // assert '/' is displayed properly
    const tableRow = Selector('td').withText(username).parent();
    await t
      .expect(Selector('H1').withText('All Users').exists).ok()
      .expect(tableRow.child().withText(username).exists).ok()
      .expect(tableRow.child().withText(email).exists).ok()
      .expect(Selector('a').withText('User Status').exists).ok()
      .expect(Selector('a').withText('Log Out').exists).ok()
      .expect(Selector('a').withText('Register').exists).notOk()
      .expect(Selector('a').withText('Log In').exists).notOk()

  });
```

Again, since we're not handling errors yet, let's hold off on the following test case: *should throw an error if the credentials are incorrect*.

## Logout

Let's just add *should log a user out* to the previous test case, *should allow a user to sign in*, in *login.test.js*:

```
test(`should allow a user to sign in`, async (t) => {

  // register user
  await t
    .navigateTo(`${TEST_URL}/register`)
    .typeText('input[name="username"]', username)
    .typeText('input[name="email"]', email)
    .typeText('input[name="password"]', 'test')
    .click(Selector('input[type="submit"]'))

  // log a user out
  await t
    .click(Selector('a').withText('Log Out'))

  // log a user in
  await t
```

```
    .navigateTo(`${TEST_URL}/login`)
    .typeText('input[name="email"]', email)
    .typeText('input[name="password"]', 'test')
    .click(Selector('input[type="submit"]'))

  // assert user is redirected to '/'
  // assert '/' is displayed properly
  const tableRow = Selector('td').withText(username).parent();
  await t
    .expect(Selector('H1').withText('All Users').exists).ok()
    .expect(tableRow.child().withText(username).exists).ok()
    .expect(tableRow.child().withText(email).exists).ok()
    .expect(Selector('a').withText('User Status').exists).ok()
    .expect(Selector('a').withText('Log Out').exists).ok()
    .expect(Selector('a').withText('Register').exists).notOk()
    .expect(Selector('a').withText('Log In').exists).notOk()

  // log a user out
  await t
    .click(Selector('a').withText('Log Out'))

  // assert '/logout' is displayed properly
  await t
    .expect(Selector('p').withText('You are now logged out').exists).ok()
    .expect(Selector('a').withText('User Status').exists).notOk()
    .expect(Selector('a').withText('Log Out').exists).notOk()
    .expect(Selector('a').withText('Register').exists).ok()
    .expect(Selector('a').withText('Log In').exists).ok()

});
```

## Status

Add a new file called *status.test.js* to the "e2e" directory:

```
import { Selector } from 'testcafe';

const randomstring = require('randomstring');

const username = randomstring.generate();
const email = `${username}@test.com`;

const TEST_URL = process.env.TEST_URL;


fixture('/status').page(`${TEST_URL}/status`);
```

Add the following test specs:

1. *should not display user info if a user is not logged in*

```
test(`should not display user info if a user is not logged in`, async (t) => {
  await t
    .navigateTo(`${TEST_URL}/status`)
    .expect(Selector('p').withText(
      'You must be logged in to view this.').exists).ok()
    .expect(Selector('a').withText('User Status').exists).notOk()
    .expect(Selector('a').withText('Log Out').exists).notOk()
    .expect(Selector('a').withText('Register').exists).ok()
    .expect(Selector('a').withText('Log In').exists).ok()
});
```

2. *should display user info if a user is logged in*

```
test(`should display user info if a user is logged in`, async (t) => {

  // register user
  await t
    .navigateTo(`${TEST_URL}/register`)
    .typeText('input[name="username"]', username)
    .typeText('input[name="email"]', email)
    .typeText('input[name="password"]', 'test')
    .click(Selector('input[type="submit"]'))

  // assert '/status' is displayed properly
  await t
    .navigateTo(`${TEST_URL}/status`)
    .expect(Selector('li > strong').withText('User ID:').exists).ok()
    .expect(Selector('li > strong').withText('Email:').exists).ok()
    .expect(Selector('li').withText(email).exists).ok()
    .expect(Selector('li > strong').withText('Username:').exists).ok()
    .expect(Selector('li').withText(username).exists).ok()
    .expect(Selector('a').withText('User Status').exists).ok()
    .expect(Selector('a').withText('Log Out').exists).ok()
    .expect(Selector('a').withText('Register').exists).notOk()
    .expect(Selector('a').withText('Log In').exists).notOk()

});
```

## Main Page

Within *index.test.js*, remove *users should be able to view the page* and, in its place, add *should display the page correctly if a user is not logged in*:

```
test(`should display the page correctly if a user is not logged in`, async (t) => {
  await t
    .navigateTo(TEST_URL)
```

```
    .expect(Selector('H1').withText('All Users').exists).ok()
    .expect(Selector('a').withText('User Status').exists).notOk()
    .expect(Selector('a').withText('Log Out').exists).notOk()
    .expect(Selector('a').withText('Register').exists).ok()
    .expect(Selector('a').withText('Log In').exists).ok()
});
```

## Test!

Set the environment variable:

```
$ export REACT_APP_USERS_SERVICE_URL=http://DOCKER_MACHINE_DEV_IP
```

Update the containers:

```
$ docker-compose -f docker-compose-dev.yml up --build -d
```

Set the `TEST_URL` variable:

```
$ export TEST_URL=http://DOCKER_MACHINE_DEV_IP
```

Run the tests. You should see *should display user info if a user is logged in* fail.

```
Using locally installed version of TestCafe.
 Running tests in:
 - Chrome 63.0.3239 / Mac OS X 10.12.0

 /
 ✓ should display the page correctly if a user is not logged in

 /login
 ✓ should display the sign in form
 ✓ should allow a user to sign in

 /register
 ✓ should display the registration form
 ✓ should allow a user to register

 /status
 ✓ should not display user info if a user is not logged in
 ✖ should display user info if a user is logged in

   1) AssertionError: expected false to be truthy

      Browser: Chrome 63.0.3239 / Mac OS X 10.12.0

         31 |     .click(Selector('input[type="submit"]'))
```

```
         32 |
         33 |   // assert '/status' is displayed properly
         34 |   await t
         35 |     .navigateTo(`${TEST_URL}/status`)
       > 36 |     .expect(Selector('li > strong').withText('User ID:').exists).ok()
         37 |     .expect(Selector('li > strong').withText('Email:').exists).ok()
         38 |     .expect(Selector('li').withText(email).exists).ok()
         39 |     .expect(Selector('li > strong').withText('Username:').exists).ok(
)
         40 |     .expect(Selector('li').withText(username).exists).ok()
         41 |     .expect(Selector('a').withText('User Status').exists).ok()

         at <anonymous>
      (/testdriven-app/e2e/status.test.js:36:67)



 1/7 failed (19s)
```

Why? Well, in that test we logged a user in and then instead of clicking the link for user status, we navigated to it in the browser. Try manually testing both scenarios - clicking the `/status` link and navigating to the route in the browser. Essentially, when we navigate to the route in the browser, `isAuthenticated` is reset to its initial value of `false` .

> Take a moment to find `isAuthenticated` in the code. When does the value change from `false` to `true` ? What happened in the `UserStatus` component?

To fix this, we can set the state of `isAuthenticated` to `true` if there is a token in LocalStorage by adding the following Lifecycle Method to the `App` component:

```
componentWillMount() {
  if (window.localStorage.getItem('authToken')) {
    this.setState({ isAuthenticated: true });
  };
};
```

> What would happen at this point if an unauthorized user simply added an object to LocalStorage with a key of `authToken` and a dummy value? What would be displayed? Would they have access to any sensitive data from the server-side? Why or why not?

Update the containers and run the tests again:

```
$ testcafe chrome e2e
```

They should pass:

```
 Running tests in:
 - Chrome 63.0.3239 / Mac OS X 10.12.0
```

```
/
✓ should display the page correctly if a user is not logged in

/login
✓ should display the sign in form
✓ should allow a user to sign in

/register
✓ should display the registration form
✓ should allow a user to register

/status
✓ should not display user info if a user is not logged in
✓ should display user info if a user is logged in

7 passed (20s)
```

A few tips...

1. Want to run a single test or fixture to debug? Use the only method.
2. Notice your tests running slower than mine? Try running TestCafe against the production build with a headless version of Chrome:

   ```
   $ docker-compose -f docker-compose-prod.yml up -d --build
   $ testcafe "chrome:headless" e2e
   ```

Since we made a change in a React component, let's run the unit tests as well:

```
$ docker-compose -f docker-compose-dev.yml \
  run client npm test -- --verbose
```

You should see a single failure:

```
× App renders without crashing

TypeError: Cannot read property 'getItem' of undefined
```

This is failing because LocalStorage is part of the browser, which is not available during a unit test. So, we need to mock it:

```
beforeAll(() => {
  global.localStorage = {
    getItem: () => 'someToken'
   };
});
```

```
test('App renders without crashing', () => {
  const wrapper = shallow(<App/>);
});
```

The tests should now pass.

Finally, "shallow rendering" does not capture the full Lifecycle of the component, so `componentWillMount` will not be fired in the test. Instead, we can use mount to test the full rendering along with the Lifecycle methods:

```
test('App will call componentWillMount when mounted', () => {
  const onWillMount = jest.fn();
  App.prototype.componentWillMount = onWillMount;
  const wrapper = mount(<Router><App/></Router>);
  expect(onWillMount).toHaveBeenCalledTimes(1)
});
```

Add the imports:

```
import { shallow, mount } from 'enzyme';
import { MemoryRouter as Router } from 'react-router-dom';
```

Make sure the tests pass:

```
PASS  src/components/__tests__/App.test.jsx
 ✓ App renders without crashing (6ms)
 ✓ App will call componentWillMount when mounted (20ms)

PASS  src/components/__tests__/UsersList.test.jsx
 ✓ UsersList renders properly (29ms)
 ✓ UsersList renders a snapshot properly (4ms)

PASS  src/components/__tests__/NavBar.test.jsx
 ✓ NavBar renders properly (4ms)
 ✓ NavBar renders a snapshot properly (15ms)

PASS  src/components/__tests__/Logout.test.jsx
 ✓ Logout renders properly (3ms)
 ✓ Logout renders a snapshot properly (2ms)

PASS  src/components/__tests__/Form.test.jsx
 When not authenticated
   ✓ Register Form renders properly (4ms)
   ✓ Register Form submits the form properly (2ms)
   ✓ Register Form renders a snapshot properly (2ms)
   ✓ Login Form renders properly (2ms)
   ✓ Login Form submits the form properly (2ms)
   ✓ Login Form renders a snapshot properly (1ms)
```

```
  When authenticated
    ✓ Register redirects properly
    ✓ Login redirects properly (1ms)

PASS  src/components/__tests__/AddUser.test.jsx
 ✓ AddUser renders properly (4ms)
 ✓ AddUser renders a snapshot properly (2ms)

PASS  src/components/__tests__/About.test.jsx
 ✓ About renders properly (3ms)
 ✓ About renders a snapshot properly (2ms)

Test Suites: 7 passed, 7 total
Tests:       20 passed, 20 total
Snapshots:   7 passed, 7 total
Time:        3.958s, estimated 5s
Ran all test suites.
```

Keep in mind that the end-to-end tests are nowhere near being DRY. Plus, multiple tests are testing the same thing. Although this is fine on the first go around, you generally want to avoid this, especially with end-to-end tests since they are so expensive. Now is a great time to refactor! Do this on your own.

Commit your code once done.

# React Component Refactor

In this lesson, we'll convert a stateless, functional component to a stateful, class-based component...

---

Before jumping into validation, let's refactor the `Form` component into a class-based component, so state can be managed in the component itself.

Update *src/components/Form.jsx* like so:

```
import React, { Component } from 'react';
import { Redirect } from 'react-router-dom';

class Form extends Component {
  constructor (props) {
    super(props);
  };
  render() {
    if (this.props.isAuthenticated) {
      return <Redirect to='/' />;
    };
    return (
      <div>
        <h1>{this.props.formType}</h1>
        <hr/><br/>
        <form onSubmit={(event) => this.props.handleUserFormSubmit(event)}>
          {this.props.formType === 'Register' &&
            <div className="form-group">
              <input
                name="username"
                className="form-control input-lg"
                type="text"
                placeholder="Enter a username"
                required
                value={this.props.formData.username}
                onChange={this.props.handleFormChange}
              />
            </div>
          }
          <div className="form-group">
            <input
              name="email"
              className="form-control input-lg"
              type="email"
              placeholder="Enter an email address"
              required
              value={this.props.formData.email}
              onChange={this.props.handleFormChange}
```

```
              />
            </div>
            <div className="form-group">
              <input
                name="password"
                className="form-control input-lg"
                type="password"
                placeholder="Enter a password"
                required
                value={this.props.formData.password}
                onChange={this.props.handleFormChange}
              />
            </div>
            <input
              type="submit"
              className="btn btn-primary btn-lg btn-block"
              value="Submit"
            />
          </form>
        </div>
      )
    };
  };

export default Form;
```

Update the containers:

```
$ docker-compose -f docker-compose-dev.yml up -d --build
```

And then run the tests to ensure we didn't break anything:

```
$ sh test.sh dev
```

Now, instead of passing everything down via the props, we can manage the state of the component within the component itself.

Again, update *src/components/Form.jsx*:

```
import React, { Component } from 'react';
import axios from 'axios';
import { Redirect } from 'react-router-dom';

class Form extends Component {
  constructor (props) {
    super(props);
    this.state = {
      formData: {
```

```
        username: '',
        email: '',
        password: ''
      }
    };
    this.handleUserFormSubmit = this.handleUserFormSubmit.bind(this);
    this.handleFormChange = this.handleFormChange.bind(this);
  };
  componentDidMount() {
    this.clearForm();
  };
  componentWillReceiveProps(nextProps) {
    if (this.props.formType !== nextProps.formType) {
      this.clearForm();
    };
  };
  clearForm() {
    this.setState({
      formData: {username: '', email: '', password: ''}
    });
  };
  handleFormChange(event) {
    const obj = this.state.formData;
    obj[event.target.name] = event.target.value;
    this.setState(obj);
  };
  handleUserFormSubmit(event) {
    event.preventDefault();
    const formType = this.props.formType
    let data;
    if (formType === 'login') {
      data = {
        email: this.state.formData.email,
        password: this.state.formData.password
      };
    };
    if (formType === 'register') {
      data = {
        username: this.state.formData.username,
        email: this.state.formData.email,
        password: this.state.formData.password
      };
    };
    const url = `${process.env.REACT_APP_USERS_SERVICE_URL}/auth/${formType}`;
    axios.post(url, data)
    .then((res) => {
      this.clearForm();
      this.props.loginUser(res.data.auth_token);
    })
    .catch((err) => { console.log(err); });
  };
```

```
  render() {
    if (this.props.isAuthenticated) {
      return <Redirect to='/' />;
    };
    return (
      <div>
        <h1 style={{'textTransform':'capitalize'}}>{this.props.formType}</h1>
        <hr/><br/>
        <form onSubmit={(event) => this.handleUserFormSubmit(event)}>
          {this.props.formType === 'register' &&
            <div className="form-group">
              <input
                name="username"
                className="form-control input-lg"
                type="text"
                placeholder="Enter a username"
                required
                value={this.state.formData.username}
                onChange={this.handleFormChange}
              />
            </div>
          }
          <div className="form-group">
            <input
              name="email"
              className="form-control input-lg"
              type="email"
              placeholder="Enter an email address"
              required
              value={this.state.formData.email}
              onChange={this.handleFormChange}
            />
          </div>
          <div className="form-group">
            <input
              name="password"
              className="form-control input-lg"
              type="password"
              placeholder="Enter a password"
              required
              value={this.state.formData.password}
              onChange={this.handleFormChange}
            />
          </div>
          <input
            type="submit"
            className="btn btn-primary btn-lg btn-block"
            value="Submit"
          />
        </form>
      </div>
```

```
    )
  };
};

export default Form;
```

> If you have the client tests running in watch mode, you should see the tests failing:
>
> ```
> Test Suites: 1 failed, 6 passed, 7 total
> Tests:       5 failed, 15 passed, 20 total
> Snapshots:   2 failed, 5 passed, 7 total
> Time:        3.216s, estimated 4s
> ```

Then update *src/App.jsx*:

```
import React, { Component } from 'react';
import { Route, Switch } from 'react-router-dom';
import axios from 'axios';

import UsersList from './components/UsersList';
import About from './components/About';
import NavBar from './components/NavBar';
import Form from './components/Form';
import Logout from './components/Logout';
import UserStatus from './components/UserStatus';

class App extends Component {
  constructor() {
    super();
    this.state = {
      users: [],
      title: 'TestDriven.io',
      isAuthenticated: false
    };
    this.logoutUser = this.logoutUser.bind(this);
    this.loginUser = this.loginUser.bind(this)
  }
  componentWillMount() {
    if (window.localStorage.getItem('authToken')) {
      this.setState({ isAuthenticated: true });
    };
  };
  componentDidMount() {
    this.getUsers();
  };
  getUsers() {
    axios.get(`${process.env.REACT_APP_USERS_SERVICE_URL}/users`)
    .then((res) => { this.setState({ users: res.data.data.users }); })
    .catch((err) => {  });
```

```
  };
  logoutUser() {
    window.localStorage.clear();
    this.setState({ isAuthenticated: false });
  };
  loginUser(token) {
    window.localStorage.setItem('authToken', token);
    this.setState({ isAuthenticated: true });
    this.getUsers();
  };
  render() {
    return (
      <div>
        <NavBar
          title={this.state.title}
          isAuthenticated={this.state.isAuthenticated}
        />
        <div className="container">
          <div className="row">
            <div className="col-md-6">
              <br/>
              <Switch>
                <Route exact path='/' render={() => (
                  <UsersList
                    users={this.state.users}
                  />
                )} />
                <Route exact path='/about' component={About}/>
                <Route exact path='/register' render={() => (
                  <Form
                    formType={'register'}
                    isAuthenticated={this.state.isAuthenticated}
                    loginUser={this.loginUser}
                  />
                )} />
                <Route exact path='/login' render={() => (
                  <Form
                    formType={'login'}
                    isAuthenticated={this.state.isAuthenticated}
                    loginUser={this.loginUser.bind(this)}
                  />
                )} />
                <Route exact path='/logout' render={() => (
                  <Logout
                    logoutUser={this.logoutUser}
                    isAuthenticated={this.state.isAuthenticated}
                  />
                )} />
                <Route exact path='/status' render={() => (
                  <UserStatus
                    isAuthenticated={this.state.isAuthenticated}
```

```
                    />
                  )} />
                </Switch>
              </div>
            </div>
          </div>
        </div>
      )
    };
  };

  export default App;
```

Review the changes. Notice anything new? There's a number of changes, but really the only thing that you have not seen before is the use of the `componentWillReceiveProps` Lifecycle Method:

```
componentWillReceiveProps(nextProps) {
  if (this.props.formType !== nextProps.formType) {
    this.clearForm();
  };
};
```

This method is called *after* the initial rendering and *before* a component receives new props. So, if you have a change in props, not on the initial render, then this method will fire.

Remember: We are sharing state for both signing up and logging in. This can cause problems with form validation on a route change - i.e., `/login` to `/register` - if the state is not cleared out. In other words, if an end user fills out the login form, and it validates correctly, and for whatever reason does not submit the form but instead navigates to `/register`, the registration form will automatically be valid. To prevent that from happening, `componentWillReceiveProps()` fires on the route change, clearing the state of the form.

It's important to note that this method can be called by React for strange reasons, at odd times. For that reason, you should *always* compare the current ( `this.props.formType` ) and next prop values ( `nextProps.formType` ) if you only want to do something based on a prop change.

With that, update the containers and run the tests:

```
$ docker-compose -f docker-compose-dev.yml up -d --build

$ docker-compose -f docker-compose-dev.yml \
  run client npm test -- --verbose
```

You should see a number of failures:

```
FAIL  src/components/__tests__/Form.test.js
 When not authenticated
```

```
    × Register Form renders properly (7ms)
    × Register Form submits the form properly (2ms)
    × Register Form renders a snapshot properly (5ms)
    ✓ Login Form renders properly (1ms)
    × Login Form submits the form properly (2ms)
    × Login Form renders a snapshot properly (3ms)
  When authenticated
    ✓ Register redirects properly (1ms)
    ✓ Login redirects properly (1ms)

PASS  src/components/__tests__/UsersList.test.js
  ✓ UsersList renders properly (14ms)
  ✓ UsersList renders a snapshot properly (7ms)

PASS  src/components/__tests__/App.test.js
  ✓ App renders without crashing (8ms)
  ✓ App will call componentWillMount when mounted (34ms)

PASS  src/components/__tests__/NavBar.test.js
  ✓ NavBar renders properly (5ms)
  ✓ NavBar renders a snapshot properly (15ms)

PASS  src/components/__tests__/About.test.js
  ✓ About renders properly (4ms)
  ✓ About renders a snapshot properly (5ms)

PASS  src/components/__tests__/Logout.test.js
  ✓ Logout renders properly (3ms)
  ✓ Logout renders a snapshot properly (5ms)

PASS  src/components/__tests__/AddUser.test.js
  ✓ AddUser renders properly (6ms)
  ✓ AddUser renders a snapshot properly (6ms)

Snapshot Summary
› 2 snapshot tests failed in 1 test suite. Inspect your code changes or press `u` t
o update them.

Test Suites: 1 failed, 6 passed, 7 total
Tests:       5 failed, 15 passed, 20 total
Snapshots:   2 failed, 5 passed, 7 total
Time:        3.17s, estimated 4s
Ran all test suites.
```

Fortunately, we can fix this by making two small changes to the form tests in *services/client/src/components/__tests__/Form.test.jsx*.

1. First, update the `testData` array, to pass in the correct props:

   ```
   const testData = [
   ```

```
    {
      formType: 'register',
      formData: {
        username: '',
        email: '',
        password: ''
      },
      isAuthenticated: false,
      loginUser: jest.fn(),
    },
    {
      formType: 'login',
      formData: {
        email: '',
        password: ''
      },
      isAuthenticated: false,
      loginUser: jest.fn(),
    }
  ]
```

2. Then, update the following `it` block:

```
  it(`${el.formType} Form submits the form properly`, () => {
    const wrapper = shallow(component);
    wrapper.instance().handleUserFormSubmit = jest.fn();
    wrapper.update();
    const input = wrapper.find('input[type="email"]');
    expect(wrapper.instance().handleUserFormSubmit).toHaveBeenCalledTimes(0);
    input.simulate(
      'change', { target: { name: 'email', value: 'test@test.com'} })
    wrapper.find('form').simulate('submit', el.formData)
    expect(wrapper.instance().handleUserFormSubmit).toHaveBeenCalledWith(el.form
 Data);
    expect(wrapper.instance().handleUserFormSubmit).toHaveBeenCalledTimes(1);
  });
```

Take note of `wrapper.update()`. This is shorthand for `wrapper.instance().forceUpdate()`, and it is used to re-render the component, adding the mocked method ( `handleUserFormSubmit` ) to the form instance. Without it, the actual, non-mocked method would have been called.

Press the `u` key to updated the snapshot tests, and then make sure they all pass:

```
PASS  src/components/__tests__/UsersList.test.js
 ✓ UsersList renders properly (13ms)
 ✓ UsersList renders a snapshot properly (3ms)

PASS  src/components/__tests__/App.test.js
 ✓ App renders without crashing (10ms)
```

```
 ✓ App will call componentWillMount when mounted (48ms)

PASS  src/components/__tests__/Form.test.js
 When not authenticated
   ✓ register Form renders properly (7ms)
   ✓ register Form submits the form properly (3ms)
   ✓ register Form renders a snapshot properly (4ms)
   ✓ login Form renders properly (3ms)
   ✓ login Form submits the form properly (3ms)
   ✓ login Form renders a snapshot properly (4ms)
 When authenticated
   ✓ register redirects properly (2ms)
   ✓ login redirects properly (1ms)

PASS  src/components/__tests__/NavBar.test.js
 ✓ NavBar renders properly (9ms)
 ✓ NavBar renders a snapshot properly (8ms)

PASS  src/components/__tests__/Logout.test.js
 ✓ Logout renders properly (4ms)
 ✓ Logout renders a snapshot properly (3ms)

PASS  src/components/__tests__/AddUser.test.js
 ✓ AddUser renders properly (5ms)
 ✓ AddUser renders a snapshot properly (5ms)

PASS  src/components/__tests__/About.test.js
 ✓ About renders properly (3ms)
 ✓ About renders a snapshot properly (2ms)

Test Suites: 7 passed, 7 total
Tests:       20 passed, 20 total
Snapshots:   7 passed, 7 total
Time:        3.53s, estimated 4s
Ran all test suites.
```

Finally, run the end-to-end tests:

```
$ testcafe chrome e2e
```

# React Form Validation

In this lesson, we'll add form validation to the register and sign in forms (example)...

---

Since we are using controlled inputs to obtain the user submitted input, we can evaluate whether the form is valid on every value change as the input values are on the state.

Let's test-drive the updates!

## Rules

Register:

1. Username and email are greater than 5 characters
2. Password must be greater than 10 characters
3. Email is a valid email address ( `something@something.something` )

Login:

1. Username and email must not be empty

## Disable Button

Let's add a `disabled` attribute to the button and set the initial value to `true` so the form cannot be submitted. Then, when the form validates properly, `disabled` will be set to `false` .

### Test

Add the following assert to *should display the registration form* in *e2e/register.test.js* and *should display the sign in form* in *e2e/login.test.js*:

```
.expect(Selector('input[disabled]').exists).ok()
```

Re-build the components, and then run the end-to-end tests. Ensure they fail. Then, run the unit tests. They should pass. Let's add a test to the `forEach` in *services/client/src/components/__tests__/Form.test.jsx*:

```
it(`${el.formType} Form should be disabled by default`, () => {
  const wrapper = shallow(component);
  const input = wrapper.find('input[type="submit"]');
  expect(input.get(0).props.disabled).toEqual(true);
});
```

The test should fail:

```
Expected value to equal:
  true
Received:
  undefined
```

### Component

Update the `input` button in *src/components/Form.jsx*:

```
<input
  type="submit"
  className="btn btn-primary btn-lg btn-block"
  value="Submit"
  disabled={true}
/>
```

Re-build and run the tests again. The unit tests should pass along with the updated end-to-end test, but a number of new end-to-end test cases should fail since the form can no longer be submitted. To update, let's validate the form on submit

Add a new property called `valid` to the state in the `Form()` component:

```
this.state = {
  formData: {
    username: '',
    email: '',
    password: ''
  },
  valid: false,
};
```

As the name suggests, when `valid` is `true` , the form input values are valid and the form can be properly submitted.

Next, update the `input` button again, changing how the `disabled` attribute is set:

```
<input
  type="submit"
  className="btn btn-primary btn-lg btn-block"
  value="Submit"
  disabled={!this.state.valid}
/>
```

So, when the form is valid, `disabled` is `false` . Next, Add a method to update the state of `valid` :

```
validateForm() {
```

```
  this.setState({valid: true});
};
```

When should we call this method?

```
handleFormChange(event) {
  const obj = this.state.formData;
  obj[event.target.name] = event.target.value;
  this.setState(obj);
  this.validateForm();
};
```

Re-build the components. Run both sets of tests. They should pass!

### Test

Then, update the following test, asserting that the `validateForm` method gets called when the form is submitted:

```
it(`${el.formType} Form submits the form properly`, () => {
  const wrapper = shallow(component);
  wrapper.instance().handleUserFormSubmit = jest.fn();
  wrapper.instance().validateForm = jest.fn();
  wrapper.update();
  const input = wrapper.find('input[type="email"]');
  expect(wrapper.instance().handleUserFormSubmit).toHaveBeenCalledTimes(0);
  input.simulate(
    'change', { target: { name: 'email', value: 'test@test.com'} })
  wrapper.find('form').simulate('submit', el.formData)
  expect(wrapper.instance().handleUserFormSubmit).toHaveBeenCalledWith(el.formData)
;
  expect(wrapper.instance().handleUserFormSubmit).toHaveBeenCalledTimes(1);
  expect(wrapper.instance().validateForm).toHaveBeenCalledTimes(1);
});
```

Make sure the unit tests still pass:

```
Test Suites: 7 passed, 7 total
Tests:       22 passed, 22 total
Snapshots:   7 passed, 7 total
Time:        2.994s, estimated 4s
```

We still need to add validation logic to `validateForm()`, but before that we need to define the rules...

## Validation Rules

Next, let's add the validation rules below each input field, starting with some tests...

## Test

Update *should display the sign in form*

```
test(`should display the sign in form`, async (t) => {
  await t
    .navigateTo(`${TEST_URL}/login`)
    .expect(Selector('H1').withText('Login').exists).ok()
    .expect(Selector('form').exists).ok()
    .expect(Selector('input[disabled]').exists).ok()
    .expect(Selector('.validation-list').exists).ok()
    .expect(Selector('.validation-list > .error').nth(0).withText(
      'Email is required.').exists).ok()
});
```

Update *should display the registration form*

```
test(`should display the registration form`, async (t) => {
  await t
    .navigateTo(`${TEST_URL}/register`)
    .expect(Selector('H1').withText('Register').exists).ok()
    .expect(Selector('form').exists).ok()
    .expect(Selector('input[disabled]').exists).ok()
    .expect(Selector('.validation-list').exists).ok()
    .expect(Selector('.validation-list > .error').nth(0).withText(
      'Username must be greater than 5 characters.').exists).ok()
});
```

Make sure the tests fail.

## Component

To fix, we first need to define the rules.

First, add a new folder in "components" called "forms", and move the *Form.jsx* file to that new folder. Be sure to update the imports in *App.jsx* and *Form.test.jsx*.

Then add a new file called *form-rules.js* to "forms":

```
export const registerFormRules = [
  {
    id: 1,
    field: 'username',
    name: 'Username must be greater than 5 characters.',
    valid: false
  },
  {
```

```
    id: 2,
    field: 'email',
    name: 'Email must be greater than 5 characters.',
    valid: false
  },
  {
    id: 3,
    field: 'email',
    name: 'Email must be a valid email address.',
    valid: false
  },
  {
    id: 4,
    field: 'password',
    name: 'Password must be greater than 10 characters.',
    valid: false
  }
];

export const loginFormRules = [
  {
    id: 1,
    field: 'email',
    name: 'Email is required.',
    valid: false
  },
  {
    id: 2,
    field: 'password',
    name: 'Password is required.',
    valid: false
  }
];
```

Update the `state` object in the component:

```
this.state = {
  formData: {
    username: '',
    email: '',
    password: ''
  },
  registerFormRules: registerFormRules,
  loginFormRules: loginFormRules,
  valid: false,
};
```

Don't forget the import:

```
import { registerFormRules, loginFormRules } from './form-rules.js';
```

You could render these within the `Form` component, but since there is a bit of logic separate from the form, let's create a new functional component.

### Test

Add a new test file called *FormErrors.test.jsx*:

```
import React from 'react';
import { shallow, mount } from 'enzyme';
import renderer from 'react-test-renderer';
import { MemoryRouter as Router } from 'react-router-dom';

import FormErrors from '../forms/FormErrors';
import { registerFormRules, loginFormRules } from '../forms/form-rules.js';

const registerFormProps = {
  formType: 'Register',
  formRules: registerFormRules,
}

const loginFormProps = {
  formType: 'Login',
  formRules: loginFormRules,
}

test('FormErrors (with register form) renders properly', () => {
  const wrapper = shallow(<FormErrors {...registerFormProps} />);
  const ul = wrapper.find('ul');
  expect(ul.length).toBe(1);
  const li = wrapper.find('li');
  expect(li.length).toBe(4);
  expect(li.get(0).props.children).toContain(
    'Username must be greater than 5 characters.');
  expect(li.get(1).props.children).toContain(
    'Email must be greater than 5 characters.');
  expect(li.get(2).props.children).toContain(
    'Email must be a valid email address.');
  expect(li.get(3).props.children).toContain(
    'Password must be greater than 10 characters.');
});

test('FormErrors (with register form) renders a snapshot properly', () => {
  const tree = renderer.create(
    <Router><FormErrors {...registerFormProps} /></Router>
  ).toJSON();
  expect(tree).toMatchSnapshot();
});
```

```
test('FormErrors (with login form) renders properly', () => {
  const wrapper = shallow(<FormErrors {...loginFormProps} />);
  const ul = wrapper.find('ul');
  expect(ul.length).toBe(1);
  const li = wrapper.find('li');
  expect(li.length).toBe(2);
  expect(li.get(0).props.children).toContain(
    'Email is required.');
  expect(li.get(1).props.children).toContain(
    'Password is required.');
});

test('FormErrors (with login form) renders a snapshot properly', () => {
  const tree = renderer.create(
    <Router><FormErrors {...loginFormProps} /></Router>
  ).toJSON();
  expect(tree).toMatchSnapshot();
});
```

> Try re-factoring this into a `for` loop, like we did with the form component tests.

### Component

*FormErrors.jsx*:

```
import React from 'react';

import './FormErrors.css';

const FormErrors = (props) => {
  return (
    <div>
      <ul className="validation-list">
        {
          props.formRules.map((rule) => {
            return <li
              className={rule.valid ? "success" : "error"} key={rule.id}>{rule.name
}
            </li>
          })
        }
      </ul>
    </div>
  )
};

export default FormErrors;
```

Add this file to the "forms" directory, and then add the associated styles to a new file called *FormErrors.css*:

```
.validation-list {
  padding-left: 25px;
}

.validation-list > li {
  display: block;
}

li:before {
  font-family: 'Glyphicons Halflings';
  font-size: 9px;
  float: left;
  margin-top: 4px;
  margin-left: -17px;
}

.error {
  color: red;
}

.error:before {
  content: "\e014";
  color: red;
}

.success {
  color: green;
}

.success:before {
  content: "\e013";
  color: green;
}
```

Finally, render the component just above the form, back within the `Form` component:

```
render() {
  if (this.props.isAuthenticated) {
    return <Redirect to='/' />;
  };
  let formRules = this.state.loginFormRules;
  if (this.props.formType === 'register') {
    formRules = this.state.registerFormRules;
  }
  return (
    <div>
```

```
      <h1 style={{'textTransform':'capitalize'}}>{this.props.formType}</h1>
      <hr/><br/>
      <FormErrors
        formType={this.props.formType}
        formRules={formRules}
      />
      <form onSubmit={(event) => this.handleUserFormSubmit(event)}>
        ...
      </form>
    </div>
  )
};
```

Add the import as well:

```
import FormErrors from './FormErrors.jsx';
```

Run the unit tests, making sure to update the snapshot tests.

```
PASS  src/components/__tests__/UsersList.test.jsx
 ✓ UsersList renders properly (19ms)
 ✓ UsersList renders a snapshot properly (2ms)

PASS  src/components/__tests__/NavBar.test.jsx
 ✓ NavBar renders properly (3ms)
 ✓ NavBar renders a snapshot properly (10ms)

PASS  src/components/__tests__/Form.test.jsx
 When not authenticated
   ✓ register Form renders properly (4ms)
   ✓ register Form submits the form properly (3ms)
   ✓ register Form renders a snapshot properly (3ms)
   ✓ register Form should be disabled by default (1ms)
   ✓ login Form renders properly (2ms)
   ✓ login Form submits the form properly (2ms)
   ✓ login Form renders a snapshot properly (2ms)
   ✓ login Form should be disabled by default (1ms)
 When authenticated
   ✓ register redirects properly (1ms)
   ✓ login redirects properly (1ms)

PASS  src/components/__tests__/About.test.jsx
 ✓ About renders properly (2ms)
 ✓ About renders a snapshot properly (1ms)

PASS  src/components/__tests__/FormErrors.test.jsx
 ✓ FormErrors (with register form) renders properly (3ms)
 ✓ FormErrors (with register form) renders a snapshot properly (2ms)
 ✓ FormErrors (with login form) renders properly (1ms)
```

```
  ✓ FormErrors (with login form) renders a snapshot properly (1ms)

PASS  src/components/__tests__/Logout.test.jsx
  ✓ Logout renders properly (3ms)
  ✓ Logout renders a snapshot properly (2ms)

PASS  src/components/__tests__/AddUser.test.jsx
  ✓ AddUser renders properly (4ms)
  ✓ AddUser renders a snapshot properly (2ms)

Test Suites: 8 passed, 8 total
Tests:       26 passed, 26 total
Snapshots:   9 passed, 9 total
Time:        3.5s, estimated 4s
Ran all test suites.
```

Ensure the end-to-end tests pass as well.

## Validate Password Input

To keep things simple, we can start with validating a single input.

### Test

To test, add the following spec to the register tests:

```
test(`should validate the password field`, async (t) => {
  await t
    .navigateTo(`${TEST_URL}/register`)
    .expect(Selector('H1').withText('Register').exists).ok()
    .expect(Selector('form').exists).ok()
    .expect(Selector('input[disabled]').exists).ok()
    .expect(Selector('.validation-list > .error').nth(3).withText(
      'Password must be greater than 10 characters.').exists).ok()
    .typeText('input[name="password"]', 'greaterthanten')
    .expect(Selector('.validation-list').exists).ok()
    .expect(Selector('.validation-list > .error').nth(3).withText(
      'Password must be greater than 10 characters.').exists).notOk()
    .expect(Selector('.validation-list > .success').nth(0).withText(
      'Password must be greater than 10 characters.').exists).ok()
});
```

### Component

Update `validateForm()` to check whether the password has a length greater than 10:

```
validateForm() {
  // define self as this
```

```
  const self = this;
  // get form data
  const formData = this.state.formData;
  // reset all rules
  self.resetRules()
  // validate register form
  if (self.props.formType === 'register') {
    const formRules = self.state.registerFormRules;
    if (formData.password.length > 10) formRules[3].valid = true;
    self.setState({registerFormRules: formRules})
    if (self.allTrue()) self.setState({valid: true});
  }
};
```

Then add the helpers:

```
allTrue() {
  let formRules = registerFormRules;
  if (this.props.formType === 'login') {
    formRules = loginFormRules;
  }
  for (const rule of formRules) {
    if (!rule.valid) return false;
  }
  return true;
};
resetRules() {
  if (this.props.formType === 'login') {
    const formRules = this.state.loginFormRules;
    for (const rule of formRules) {
      rule.valid = false;
    }
    this.setState({loginFormRules: formRules})
  }
  if (this.props.formType === 'register') {
    const formRules = this.state.registerFormRules;
    for (const rule of formRules) {
      rule.valid = false;
    }
    this.setState({registerFormRules: formRules})
  }
  this.setState({valid: false});
};
```

`allTrue()` simply iterates through all the rules and returns `true` only if they are all valid. Meanwhile, `resetRules()` simply resets all instances of `valid` back to `false`.

Update the containers, and then run the tests.

### Test

Before moving on, we need to update the `componentWillReceiveProps` method, to reset the `rules` on a route change. Why is this necessary? Let's look. Update the test:

```
test(`should validate the password field`, async (t) => {
  await t
    .navigateTo(`${TEST_URL}/register`)
    .expect(Selector('H1').withText('Register').exists).ok()
    .expect(Selector('form').exists).ok()
    .expect(Selector('input[disabled]').exists).ok()
    .expect(Selector('.validation-list > .error').nth(3).withText(
      'Password must be greater than 10 characters.').exists).ok()
    .typeText('input[name="password"]', 'greaterthanten')
    .expect(Selector('.validation-list').exists).ok()
    .expect(Selector('.validation-list > .error').nth(3).withText(
      'Password must be greater than 10 characters.').exists).notOk()
    .expect(Selector('.validation-list > .success').nth(0).withText(
      'Password must be greater than 10 characters.').exists).ok()
    .click(Selector('a').withText('Log In'))
    .click(Selector('a').withText('Register'))
    .expect(Selector('.validation-list > .error').nth(3).withText(
      'Password must be greater than 10 characters.').exists).ok()
});
```

### Component

We just need to call `resetRules()` in `componentWillReceiveProps()`

```
componentWillReceiveProps(nextProps) {
  if (this.props.formType !== nextProps.formType) {
    this.clearForm();
    this.resetRules();
  };
};
```

Re-build the containers. Test again.

## Validate Inputs

Now, we need to apply that same logic to the remaining fields...

### Test

First, add a password to the top of *login.test.js*, *register.test.js*, and *status.test.js*:

```
const password = 'greaterthanten';
```

Change `.typeText('input[name="password"]', 'test')` to `.typeText('input[name="password"]', password)` in those same files, and then run the tests to ensure they still properly fail.

### Component

Update `validateForm()`:

```
validateForm() {
  // define self as this
  const self = this;
  // get form data
  const formData = this.state.formData;
  // reset all rules
  self.resetRules()
  // validate register form
  if (self.props.formType === 'register') {
    const formRules = self.state.registerFormRules;
    if (formData.username.length > 5) formRules[0].valid = true;
    if (formData.email.length > 5) formRules[1].valid = true;
    if (this.validateEmail(formData.email)) formRules[2].valid = true;
    if (formData.password.length > 10) formRules[3].valid = true;
    self.setState({registerFormRules: formRules})
    if (self.allTrue()) self.setState({valid: true});
  }
  // validate login form
  if (self.props.formType === 'login') {
    const formRules = self.state.loginFormRules;
    if (formData.email.length > 0) formRules[0].valid = true;
    if (formData.password.length > 0) formRules[1].valid = true;
    self.setState({registerFormRules: formRules})
    if (self.allTrue()) self.setState({valid: true});
  }
};
```

Add the following regular expression to validate the email address:

```
validateEmail(email) {
  // eslint-disable-next-line
  var re = /^(([^<>()\[\]\\.,;:\s@"]+(\.[^<>()\[\]\\.,;:\s@"]+)*)|(".+"))@((\[[0-9]
{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}])|(([a-zA-Z\-0-9]+\.)+[a-zA-Z]{2,}))$/;
  return re.test(email);
};
```

Re-build. Test. Commit your code.

> We didn't test any of the validation logic in our unit tests. Do this on your own. Also, what about special characters? Try restricting these on your own as well.

# React Flash Messaging

Let's add flash messaging to send quick alerts to the end user...

---

## Create Message

### End-to-End Test

Start by adding `.expect(Selector('.alert-success').withText('Welcome!').exists).ok()` to the *should allow a user to sign in* test:

```
...
// assert user is redirected to '/'
// assert '/' is displayed properly
const tableRow = Selector('td').withText(username).parent();
await t
  .expect(Selector('H1').withText('All Users').exists).ok()
  .expect(tableRow.child().withText(username).exists).ok()
  .expect(tableRow.child().withText(email).exists).ok()
  .expect(Selector('a').withText('User Status').exists).ok()
  .expect(Selector('a').withText('Log Out').exists).ok()
  .expect(Selector('a').withText('Register').exists).notOk()
  .expect(Selector('a').withText('Log In').exists).notOk()
  .expect(Selector('.alert-success').withText('Welcome!').exists).ok()
...
```

With `testdriven-dev` as the active Docker Machine, update the containers:

```
$ docker-compose -f docker-compose-dev.yml up -d
```

Run the tests:

```
$ testcafe chrome e2e
```

Ensure the tests fail.

### Component

Add a new functional component, called `Message` to a new component file called *Message.jsx*, which is responsible *only* for displaying a message:

```
import React from 'react';

const Message = (props) => {
```

```
    return (
      <div className={`alert alert-${props.messageType}`}>
        <span
          className="glyphicon glyphicon-exclamation-sign"
          aria-hidden="true">
        </span>
        <span> {props.messageName}</span>
        <button
          className='close'
          data-dismiss='alert'
        >&times;</button>
      </div>
    )
  };

  export default Message;
```

Now that the component is ready to go, let's wire it up to the `App` component:

1. Add `messageName` and `messageType` to the state:

   ```
   this.state = {
     users: [],
     title: 'TestDriven.io',
     isAuthenticated: false,
     messageName: null,
     messageType: null,
   };
   ```

2. Import the `Message` component:

   ```
   import Message from './components/Message';
   ```

3. Render the component, just below the `NavBar` :

   ```
   <div>
     <NavBar
       title={this.state.title}
       isAuthenticated={this.state.isAuthenticated}
     />
     <div className="container">
       {this.state.messageName && this.state.messageType &&
         <Message
           messageName={this.state.messageName}
           messageType={this.state.messageType}
         />
       }
   ...
   ```

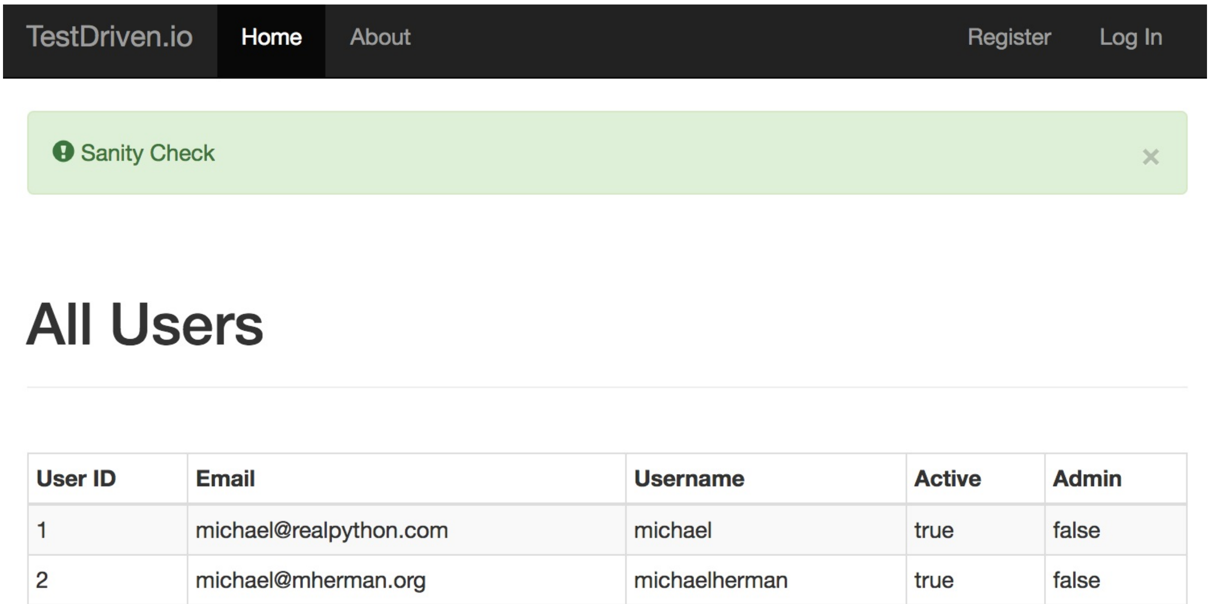
4. Finally, add a `createMessage` method, with default parameters, to the `App` component:

```
createMessage(name='Sanity Check', type='success') {
  this.setState({
    messageName: name,
    messageType: type
  });
};
```

Call it in the `componentDidMount` Lifecycle Method.

```
componentDidMount() {
  this.getUsers();
  this.createMessage();
};
```

Re-build, and then manually test in the browser. You should see the alert on every route.




To get the tests to pass though, we need to dynamically create the message.

Remove the call in `componentDidMount()`, and, instead, call the method in `loginUser()`:

```
loginUser(token) {
  window.localStorage.setItem('authToken', token);
  this.setState({ isAuthenticated: true });
  this.getUsers();
  this.createMessage('Welcome!', 'success');
};
```

Update the containers and run the end-to-end tests again. They should pass.

```
/
 ✓ should display the page correctly if a user is not logged in

 /login
 ✓ should display the sign in form
 ✓ should allow a user to sign in

 /register
 ✓ should display the registration form
 ✓ should allow a user to register
 ✓ should validate the password field

 /status
 ✓ should not display user info if a user is not logged in
 ✓ should display user info if a user is logged in


 8 passed (20s)
```

Now, turn to the tests. What else do we need to test?

Update *should display the page correctly if a user is not logged in* to ensure the message is not displayed on page load:

```
test(`should display the page correctly if a user is not logged in`, async (t) => {
  await t
    .navigateTo(TEST_URL)
    .expect(Selector('H1').withText('All Users').exists).ok()
    .expect(Selector('a').withText('User Status').exists).notOk()
    .expect(Selector('a').withText('Log Out').exists).notOk()
    .expect(Selector('a').withText('Register').exists).ok()
    .expect(Selector('a').withText('Log In').exists).ok()
    .expect(Selector('.alert').exists).notOk()
});
```

Make sure the tests still pass. Then, run the client-side tests as well since we made changes to the code:

```
$ docker-compose -f docker-compose-dev.yml \
  run client npm test -- --verbose
```

## Error Messages

Let's use the flash message system to properly handle errors...

### End-to-End Test

`/register` :

1. should throw an error if the username is taken

```
test(`should throw an error if the username is taken`, async (t) => {

  // register user with duplicate user name
  await t
    .navigateTo(`${TEST_URL}/register`)
    .typeText('input[name="username"]', username)
    .typeText('input[name="email"]', `${email}unique`)
    .typeText('input[name="password"]', password)
    .click(Selector('input[type="submit"]'))

  // assert user registration failed
  await t
    .expect(Selector('H1').withText('Register').exists).ok()
    .expect(Selector('a').withText('User Status').exists).notOk()
    .expect(Selector('a').withText('Log Out').exists).notOk()
    .expect(Selector('a').withText('Register').exists).ok()
    .expect(Selector('a').withText('Log In').exists).ok()
    .expect(Selector('.alert-success').exists).notOk()
    .expect(Selector('.alert-danger').withText(
      'That user already exists.').exists).ok()

});
```

2. should throw an error if the email is taken

```
test(`should throw an error if the email is taken`, async (t) => {

  // register user with duplicate email
  await t
    .navigateTo(`${TEST_URL}/register`)
    .typeText('input[name="username"]', `${username}unique`)
    .typeText('input[name="email"]', email)
    .typeText('input[name="password"]', password)
    .click(Selector('input[type="submit"]'))

  // assert user registration failed
  await t
    .expect(Selector('H1').withText('Register').exists).ok()
    .expect(Selector('a').withText('User Status').exists).notOk()
    .expect(Selector('a').withText('Log Out').exists).notOk()
    .expect(Selector('a').withText('Register').exists).ok()
    .expect(Selector('a').withText('Log In').exists).ok()
    .expect(Selector('.alert-success').exists).notOk()
    .expect(Selector('.alert-danger').withText(
      'That user already exists.').exists).ok()
```

```
});
```

`/login` :

1. should throw an error if the credentials are incorrect

```
test(`should throw an error if the credentials are incorrect`, async (t) => {

  // attempt to log in
  await t
    .navigateTo(`${TEST_URL}/login`)
    .typeText('input[name="email"]', 'incorrect@email.com')
    .typeText('input[name="password"]', password)
    .click(Selector('input[type="submit"]'))

  // assert user login failed
  await t
    .expect(Selector('H1').withText('Login').exists).ok()
    .expect(Selector('a').withText('User Status').exists).notOk()
    .expect(Selector('a').withText('Log Out').exists).notOk()
    .expect(Selector('a').withText('Register').exists).ok()
    .expect(Selector('a').withText('Log In').exists).ok()
    .expect(Selector('.alert-success').exists).notOk()
    .expect(Selector('.alert-danger').withText(
      'User does not exist.').exists).ok()

  // attempt to log in
  await t
    .navigateTo(`${TEST_URL}/login`)
    .typeText('input[name="email"]', email)
    .typeText('input[name="password"]', 'incorrectpassword')
    .click(Selector('input[type="submit"]'))

  // assert user login failed
  await t
    .expect(Selector('H1').withText('Login').exists).ok()
    .expect(Selector('a').withText('User Status').exists).notOk()
    .expect(Selector('a').withText('Log Out').exists).notOk()
    .expect(Selector('a').withText('Register').exists).ok()
    .expect(Selector('a').withText('Log In').exists).ok()
    .expect(Selector('.alert-success').exists).notOk()
    .expect(Selector('.alert-danger').withText(
      'User does not exist.').exists).ok()

});
```

### Component

Add `createMessage()` to the `Form` component via the props:

```
...
<Route exact path='/register' render={() => (
  <Form
    formType={'register'}
    isAuthenticated={this.state.isAuthenticated}
    loginUser={this.loginUser}
    createMessage={this.createMessage}
  />
)} />
<Route exact path='/login' render={() => (
  <Form
    formType={'login'}
    isAuthenticated={this.state.isAuthenticated}
    loginUser={this.loginUser}
    createMessage={this.createMessage}
  />
)} />
...
```

Bind the method in the constructor:

```
this.createMessage = this.createMessage.bind(this);
```

Then update `handleUserFormSubmit()`:

```
handleUserFormSubmit(event) {
  event.preventDefault();
  const formType = this.props.formType
  let data;
  if (formType === 'login') {
    data = {
      email: this.state.formData.email,
      password: this.state.formData.password
    };
  };
  if (formType === 'register') {
    data = {
      username: this.state.formData.username,
      email: this.state.formData.email,
      password: this.state.formData.password
    };
  };
  const url = `${process.env.REACT_APP_USERS_SERVICE_URL}/auth/${formType}`;
  axios.post(url, data)
  .then((res) => {
    this.clearForm();
    this.props.loginUser(res.data.auth_token);
  })
  .catch((err) => {
```

```
    if (formType === 'login') {
      this.props.createMessage('User does not exist.', 'danger');
    };
    if (formType === 'register') {
      this.props.createMessage('That user already exists.', 'danger');
    };
  });
};
```

Update the containers, and then test.

> `User does not exist` isn't really accurate if the error was due to an incorrect password. `Login failed` is probably a better generic error message. Check your understanding and update this on your own.

## Delete Message

Next, the message should disappear when any of these events occur-

1. An end user clicks the `x`, on the right side of the message
2. A new message is flashed
3. Three seconds passes

### End-to-End Test

Create a new test file called *message.test.js*:

```
import { Selector } from 'testcafe';

const randomstring = require('randomstring');

const username = randomstring.generate();
const email = `${username}@test.com`;
const password = 'greaterthanten';

const TEST_URL = process.env.TEST_URL;

fixture('/register').page(`${TEST_URL}/register`);

test(`should display flash messages correctly`, async (t) => {

  // register user
  await t
    .navigateTo(`${TEST_URL}/register`)
    .typeText('input[name="username"]', username)
    .typeText('input[name="email"]', email)
    .typeText('input[name="password"]', password)
    .click(Selector('input[type="submit"]'))
```

```
// assert flash messages are removed when user clicks the 'x'
await t
  .expect(Selector('.alert-success').withText('Welcome!').exists).ok()
  .click(Selector('.alert > button'))
  .expect(Selector('.alert-success').withText('Welcome!').exists).notOk()

// log a user out
await t
  .click(Selector('a').withText('Log Out'))

// attempt to log in
await t
  .navigateTo(`${TEST_URL}/login`)
  .typeText('input[name="email"]', 'incorrect@email.com')
  .typeText('input[name="password"]', password)
  .click(Selector('input[type="submit"]'))

// assert correct message is flashed
await t
  .expect(Selector('.alert-success').exists).notOk()
  .expect(Selector('.alert-danger').withText(
    'User does not exist.').exists).ok()

// log a user in
await t
  .navigateTo(`${TEST_URL}/login`)
  .typeText('input[name="email"]', email)
  .typeText('input[name="password"]', password)
  .click(Selector('input[type="submit"]'))

// assert flash message is removed when a new message is flashed
await t
  .expect(Selector('.alert-success').withText('Welcome!').exists).ok()
  .expect(Selector('.alert-danger').withText(
    'User does not exist.').exists).notOk()

// log a user out
await t
  .click(Selector('a').withText('Log Out'))

// log a user in
await t
  .navigateTo(`${TEST_URL}/login`)
  .typeText('input[name="email"]', email)
  .typeText('input[name="password"]', password)
  .click(Selector('input[type="submit"]'))

// assert flash message is removed after three seconds
await t
  .expect(Selector('.alert-success').withText('Welcome!').exists).ok()
  .wait(4000)
```

```
    .expect(Selector('.alert-success').withText('Welcome!').exists).notOk()

});
```

## Component

To get the first set of expects - `assert flash messages are removed when user clicks the 'x'` - to pass, add a `removeMessage` method to the `App` component:

```
removeMessage() {
  this.setState({
    messageName: null,
    messageType: null
  });
};
```

Pass it down on the `props` :

```
...
<div className="container">
  {this.state.messageName && this.state.messageType &&
    <Message
      messageName={this.state.messageName}
      messageType={this.state.messageType}
      removeMessage={this.removeMessage}
    />
  }
...
```

Bind:

```
constructor() {
  super();
  this.state = {
    users: [],
    title: 'TestDriven.io',
    isAuthenticated: false,
    messageName: null,
    messageType: null,
  };
  this.logoutUser = this.logoutUser.bind(this);
  this.loginUser = this.loginUser.bind(this);
  this.createMessage = this.createMessage.bind(this);
  this.removeMessage = this.removeMessage.bind(this);
}
```

Then update the `button` , so that the `removeMessage` method is fired on click:

```
const Message = (props) => {
  return (
    <div className={`alert alert-${props.messageType}`}>
      <span
        className="glyphicon glyphicon-exclamation-sign"
        aria-hidden="true">
      </span>
      <span> {props.messageName}</span>
      <button
        className='close'
        data-dismiss='alert'
        onClick={()=>{props.removeMessage()}}
      >&times;</button>
    </div>
  )
};
```

Run the tests again.

Did you notice that the next set of expects passed - `assert flash message is removed when a new message is flashed`? To get the last set to pass, add a `setTimeout` to `createMessage()`:

```
createMessage(name='Sanity Check', type='success') {
  this.setState({
    messageName: name,
    messageType: type
  });
  setTimeout(() => {
    this.removeMessage();
  }, 3000);
};
```

> Is there any way to mock the wait time so that the test doesn't *actually* take an extra four seconds to run?

Re-build. Run your tests.

```
/
✓ should display the page correctly if a user is not logged in

/login
✓ should display the sign in form
✓ should allow a user to sign in
✓ should throw an error if the credentials are incorrect

/register
✓ should display flash messages correctly

/register
```

```
✓ should display the registration form
✓ should allow a user to register
✓ should validate the password field
✓ should throw an error if the username is taken
✓ should throw an error if the email is taken

/status
✓ should not display user info if a user is not logged in
✓ should display user info if a user is logged in


12 passed (50s)
```

## Unit Tests

Before moving on, let's write four quick client-side tests for the `Message` component:

1. When given a success message Message renders properly
2. When given a success message Message renders a snapshot properly
3. When given a danger message Message renders properly
4. When given a danger message Message renders a snapshot properly

Add a new test file called *Message.test.jsx*:

```
import React from 'react';
import { shallow } from 'enzyme';
import renderer from 'react-test-renderer';
import { MemoryRouter as Router } from 'react-router-dom';

import Message from '../Message';


describe('When given a success message', () => {

  const removeMessage = jest.fn();

  const messageSuccessProps = {
    messageName: 'Hello, World!',
    messageType: 'success',
    removeMessage: removeMessage,
  }

  it(`Message renders properly`, () => {
    const wrapper = shallow(<Message {...messageSuccessProps} />);
    const element = wrapper.find('.alert-success');
    expect(element.length).toBe(1);
    const span = wrapper.find('span');
    expect(span.length).toBe(2);
    expect(span.get(1).props.children[1]).toContain(
```

```
      messageSuccessProps.messageName);
    const button = wrapper.find('button');
    expect(button.length).toBe(1);
    expect(removeMessage).toHaveBeenCalledTimes(0);
    button.simulate('click');
    expect(removeMessage).toHaveBeenCalledTimes(1);
  });

  test('Message renders a snapshot properly', () => {
    const tree = renderer.create(
      <Message {...messageSuccessProps} />
    ).toJSON();
    expect(tree).toMatchSnapshot();
  });

});

describe('When given a danger message', () => {

  const removeMessage = jest.fn();

  const messageDangerProps = {
    messageName: 'Hello, World!',
    messageType: 'danger',
    removeMessage: removeMessage,
  }

  it(`Message renders properly`, () => {
    const wrapper = shallow(<Message {...messageDangerProps} />);
    const element = wrapper.find('.alert-danger');
    expect(element.length).toBe(1);
    const span = wrapper.find('span');
    expect(span.length).toBe(2);
    expect(span.get(1).props.children[1]).toContain(
      messageDangerProps.messageName);
    const button = wrapper.find('button');
    expect(button.length).toBe(1);
    expect(removeMessage).toHaveBeenCalledTimes(0);
    button.simulate('click');
    expect(removeMessage).toHaveBeenCalledTimes(1);
  });

  test('Message renders a snapshot properly', () => {
    const tree = renderer.create(
      <Message {...messageDangerProps} />
    ).toJSON();
    expect(tree).toMatchSnapshot();
  });

});
```

Make sure to review the code, and then ensure the tests pass:

```
PASS  src/components/__tests__/App.test.jsx
 ✓ App renders without crashing (6ms)
 ✓ App will call componentWillMount when mounted (38ms)

PASS  src/components/__tests__/UsersList.test.jsx
 ✓ UsersList renders properly (8ms)
 ✓ UsersList renders a snapshot properly (3ms)

PASS  src/components/__tests__/Form.test.jsx
 When not authenticated
   ✓ register Form renders properly (4ms)
   ✓ register Form submits the form properly (2ms)
   ✓ register Form renders a snapshot properly (3ms)
   ✓ register Form should be disabled by default (1ms)
   ✓ login Form renders properly (2ms)
   ✓ login Form submits the form properly (7ms)
   ✓ login Form renders a snapshot properly (3ms)
   ✓ login Form should be disabled by default (2ms)
 When authenticated
   ✓ register redirects properly (1ms)
   ✓ login redirects properly

PASS  src/components/__tests__/NavBar.test.jsx
 ✓ NavBar renders properly (4ms)
 ✓ NavBar renders a snapshot properly (8ms)

PASS  src/components/__tests__/Logout.test.jsx
 ✓ Logout renders properly (2ms)
 ✓ Logout renders a snapshot properly (2ms)

PASS  src/components/__tests__/FormErrors.test.jsx
 ✓ FormErrors (with register form) renders properly (4ms)
 ✓ FormErrors (with register form) renders a snapshot properly (3ms)
 ✓ FormErrors (with login form) renders properly (1ms)
 ✓ FormErrors (with login form) renders a snapshot properly (2ms)

PASS  src/components/__tests__/Message.test.jsx
 When given a success message
   ✓ Message renders properly (4ms)
   ✓ Message renders a snapshot properly (1ms)
 When given a danger message
   ✓ Message renders properly (2ms)
   ✓ Message renders a snapshot properly (1ms)

PASS  src/components/__tests__/About.test.jsx
 ✓ About renders properly (2ms)
 ✓ About renders a snapshot properly (1ms)
```

```
PASS  src/components/__tests__/AddUser.test.jsx
 ✓ AddUser renders properly (14ms)
 ✓ AddUser renders a snapshot properly (2ms)

Test Suites: 9 passed, 9 total
Tests:       30 passed, 30 total
Snapshots:   11 passed, 11 total
Time:        3.602s, estimated 4s
Ran all test suites.
```

Commit your code.

# Swagger Setup

In this lesson, we'll document the user-service API with Swagger...

---

Swagger, which is now the OpenAPI Specification, is a specification for describing, producing, consuming, testing, and visualizing a RESTful API. It comes packed with a number of tools for automatically generating documentation based on a given endpoint. The focus of this lesson will be on one of those tools - Swagger UI, which is used to build client-side API docs.

> New to Swagger? Review the What Is Swagger? guide from the official documentation.

## New Service

Let's set up a new service for this.

Create a new directory in "services" called "swagger" and add a *Dockerfile-dev* to the new directory to pull in the base Nginx image from Docker Hub, install Swagger UI, update Nginx, and then run *start.sh*:

```
FROM nginx:1.13.8

ENV SWAGGER_UI_VERSION 3.9.3
ENV URL **None**

RUN apt-get update \
    && apt-get install -y curl \
    && curl -L https://github.com/swagger-api/swagger-ui/archive/v${SWAGGER_UI_VERS
ION}.tar.gz | tar -zxv -C /tmp \
    && cp -R /tmp/swagger-ui-${SWAGGER_UI_VERSION}/dist/* /usr/share/nginx/html \
    && rm -rf /tmp/*

RUN rm /etc/nginx/conf.d/default.conf
COPY /nginx.conf /etc/nginx/conf.d

# COPY swagger.json /usr/share/nginx/html/swagger.json
COPY start.sh /start.sh

RUN ["chmod", "+x", "/start.sh"]
CMD ["/start.sh"]
```

Add the *nginx.conf* file:

```
server {
  listen 8080;
  root /usr/share/nginx/html/;
```

```
  location / {
    try_files $uri /index.html;
  }
}
```

And finally, add *start.sh*:

```
#!/bin/bash

if [ $URL != "**None**" ]; then
  sed -i -e 's@http://petstore.swagger.io/v2/swagger.json@'"$URL"'@g' /usr/share/ng
inx/html/index.html
fi

exec nginx -g 'daemon off;'
```

Take note of the `if` statement. Here, if the `URL` environment variable exists, we're replacing any occurrences of `http://petstore.swagger.io/v2/swagger.json` with that URL in */usr/share/nginx/html/index.html*.

Add the new service to the *docker-compose-dev.yml* file:

```
swagger:
  container_name: swagger
  build:
    context: ./services/swagger
    dockerfile: Dockerfile-dev
  ports:
    - '3008:8080'
  environment:
    - URL=http://petstore.swagger.io/v2/swagger.json
  depends_on:
    - users
```

Spin up the new container:

```
$ docker-compose -f docker-compose-dev.yml up -d --build
```

Once up, ensure you can see the sample API docs (Swagger Petstore) in your browser at http://DOCKER_MACHINE_DEV_IP:3008/.

Now, add a new `location` block to *services/nginx/dev.conf* and *services/nginx/prod.conf*:

```
location /swagger {
  proxy_pass        http://swagger:8080;
  proxy_redirect    default;
```

```
    proxy_set_header  Host $host;
    proxy_set_header  X-Real-IP $remote_addr;
    proxy_set_header  X-Forwarded-For $proxy_add_x_forwarded_for;
    proxy_set_header  X-Forwarded-Host $server_name;
  }
```

Update the containers:

```
$ docker-compose -f docker-compose-dev.yml up -d --build
```

Ensure http://DOCKER_MACHINE_DEV_IP/swagger works.

## Spec File

Next, we simply need to provide our own custom spec file. We could add additional logic to the Flask app, to automatically generate the spec from the route handlers, but this is quite a bit of work. For now, let's just create this file by hand, based on the following routes:

| Endpoint | HTTP Method | Authenticated? | Result |
|---|---|---|---|
| /auth/register | POST | No | register user |
| /auth/login | POST | No | log in user |
| /auth/logout | GET | Yes | log out user |
| /auth/status | GET | Yes | check user status |
| /users | GET | No | get all users |
| /users/:id | GET | No | get single user |
| /users | POST | Yes (admin) | add a user |
| /users/ping | GET | No | sanity check |

Add a *swagger.json* file to "services/swagger":

```
{
  "openapi": "3.0.0",
  "info": {
    "version": "0.0.1",
    "title": "Users Service",
    "description": "Swagger spec for documenting the users service"
  },
  "servers": [
    {
      "url": "http://UPDATE_ME"
    }
  ],
  "paths": {
  },
```

```
  "components": {
    "schemas": {
    }
  }
}
```

Be sure to update the value of `url` with your local `DOCKER_MACHINE_DEV_IP`.

Here, we defined some basic metadata about the `users` API. Be sure to review the official spec documentation for more info.

> The configuration file can be written in YAML as well - i.e., *swagger.yaml*. The JSON to YAML online convertor can be used to convert the examples to YAML.

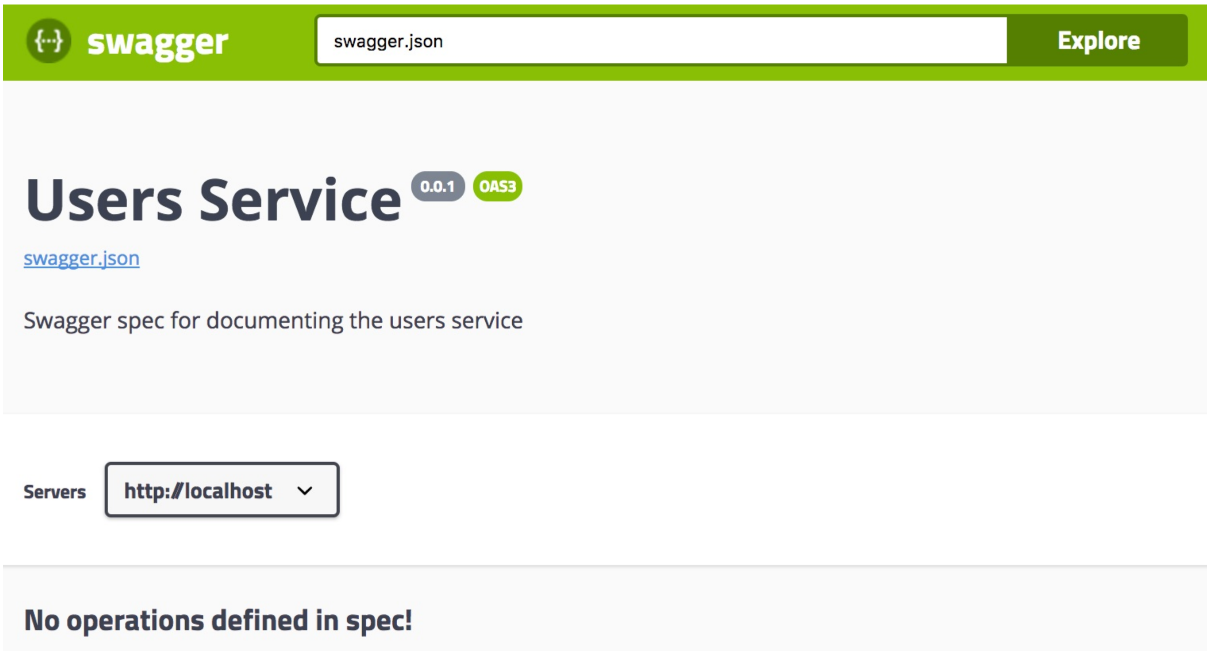
Update the environment variable in *docker-compose-dev.yml* and add a volume:

```
swagger:
  container_name: swagger
  build:
    context: ./services/swagger
    dockerfile: Dockerfile-dev
  volumes:
    - './services/swagger/swagger.json:/usr/share/nginx/html/swagger.json'
  ports:
    - '3008:8080'
  environment:
    - URL=swagger.json
  depends_on:
    - users
```

Uncomment the following line in *services/swagger/Dockerfile-dev*:

```
# COPY swagger.json /usr/share/nginx/html/swagger.json
```

Update the container. Test it out in the browser.




## Unauthenticated Routes

Add each of these as properties to the `paths` object in the *swagger.json* file...

`/ping` :

```
"/users/ping": {
  "get": {
    "summary": "Just a sanity check",
    "responses": {
      "200": {
        "description": "Will return 'pong!'"
      }
    }
  }
}
```

`/users` :

```
"/users": {
  "get": {
    "summary": "Returns all users",
    "responses": {
      "200": {
        "description": "user object"
      }
    }
  }
}
```

`/users/:id` :

```
"/users/{id}": {
  "get": {
    "summary": "Returns a user based on a single user ID",
    "parameters": [
      {
        "name": "id",
        "in": "path",
        "description": "ID of user to fetch",
        "required": true,
        "schema": {
          "type": "integer",
          "format": "int64"
        }
      }
    ],
    "responses": {
      "200": {
        "description": "user object"
      }
    }
  }
}
```

`/auth/register` :

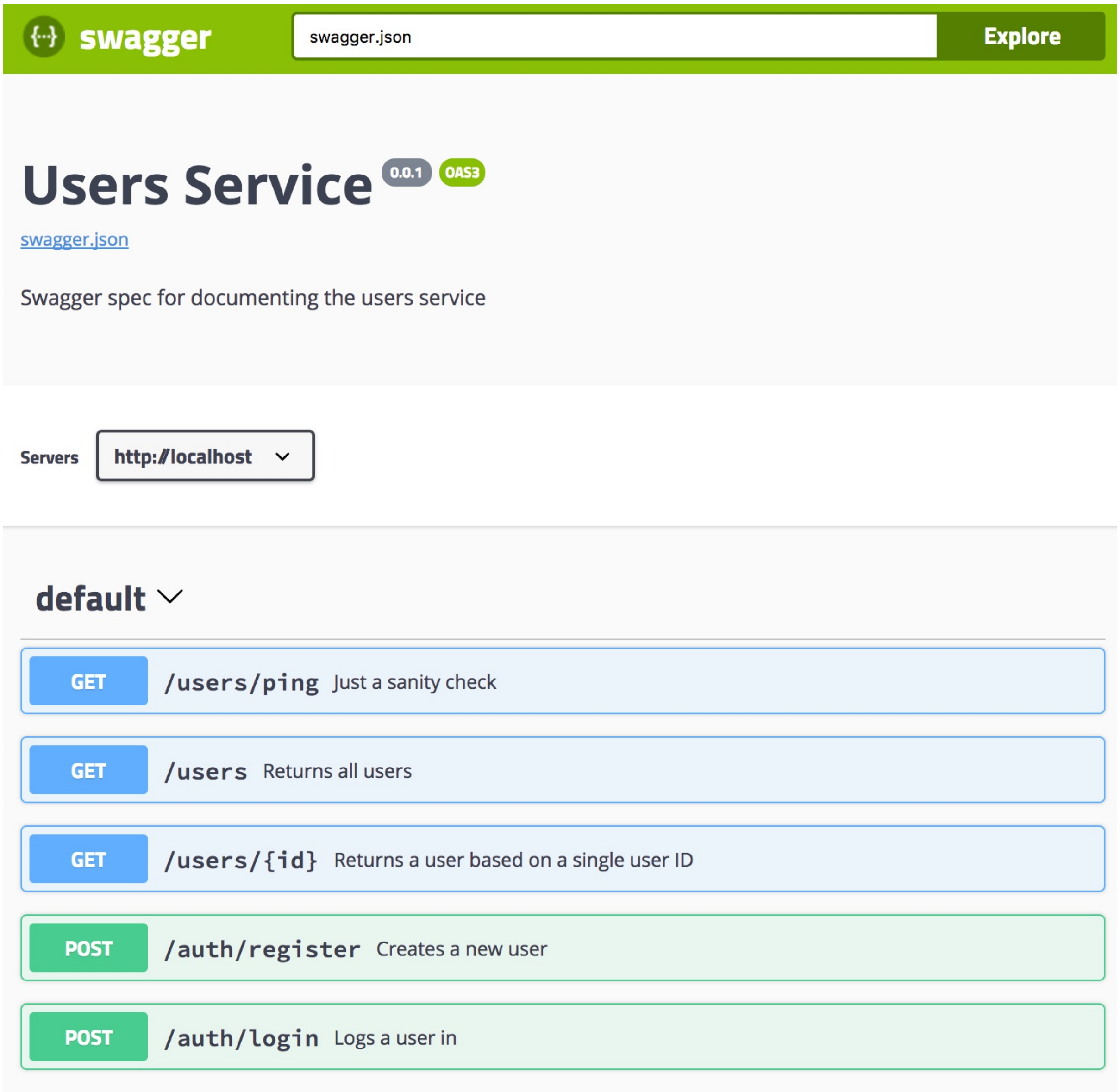
```
"/auth/register": {
  "post": {
    "summary": "Creates a new user",
    "requestBody": {
      "description": "User to add",
      "required": true,
      "content": {
        "application/json": {
          "schema": {
            "type": "object",
            "required": [
              "username",
              "email",
              "password"
            ],
            "properties": {
              "username": {
                "type": "string"
              },
              "email": {
                "type": "string"
              },
```

```
                "password": {
                  "type": "string"
                }
              }
            }
          }
        }
      },
      "responses": {
        "200": {
          "description": "user object"
        }
      }
    }
  }
```

`/auth/login` :

```
  "/auth/login": {
    "post": {
      "summary": "Logs a user in",
      "requestBody": {
        "description": "User to log in",
        "required": true,
        "content": {
          "application/json": {
            "schema": {
            }
          }
        }
      },
      "responses": {
        "200": {
          "description": "Successfully logged in"
        }
      }
    }
  }
```

Refresh the browser to see the changes.




## Schemas

To keep things DRY, let's abstract out the schema definitions via a reference object. Add two new schemas to the `components` :

```
"components": {
  "schemas": {
    "user": {
      "properties": {
        "email": {
          "type": "string"
        },
        "password": {
          "type": "string"
        }
      }
```

```
      },
      "user-full": {
        "properties": {
          "username": {
            "type": "string"
          },
          "email": {
            "type": "string"
          },
          "password": {
            "type": "string"
          }
        }
      }
    }
  }
```

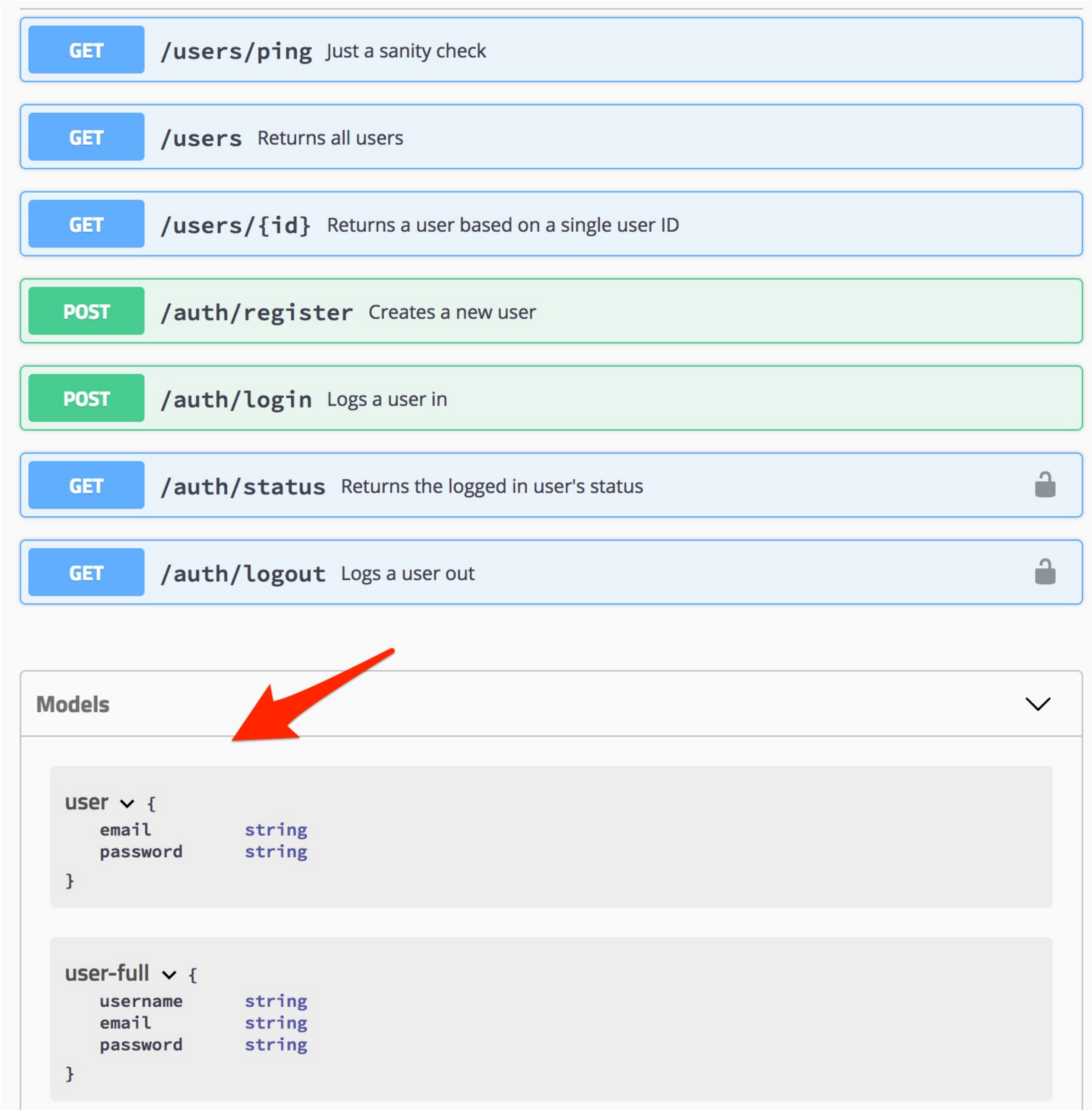
Now, turn back to the `/auth/login` route and update the `schema` like so:

```
"schema": {
  "$ref": "#/components/schemas/user"
}
```

Then, update the `schema` in the `/auth/register` route:

```
"schema": {
  "$ref": "#/components/schemas/user-full"
}
```

These schema definition can now be re-used.




## Authenticated Routes

To access authenticated routes, we need to add a Bearer token to the request header. Fortunately Swagger supports this out of the box.

Start by adding a securitySchemes object to the `components` :

```
"components": {
  "securitySchemes": {
    "bearerAuth": {
      "type": "http",
      "scheme": "bearer"
    }
  },
  "schemas": {
    "user": {
```

```
      "properties": {
        "email": {
          "type": "string"
        },
        "password": {
          "type": "string"
        }
      }
    }
  }
}
```

Now, we can provide a security property to paths that require authentication...

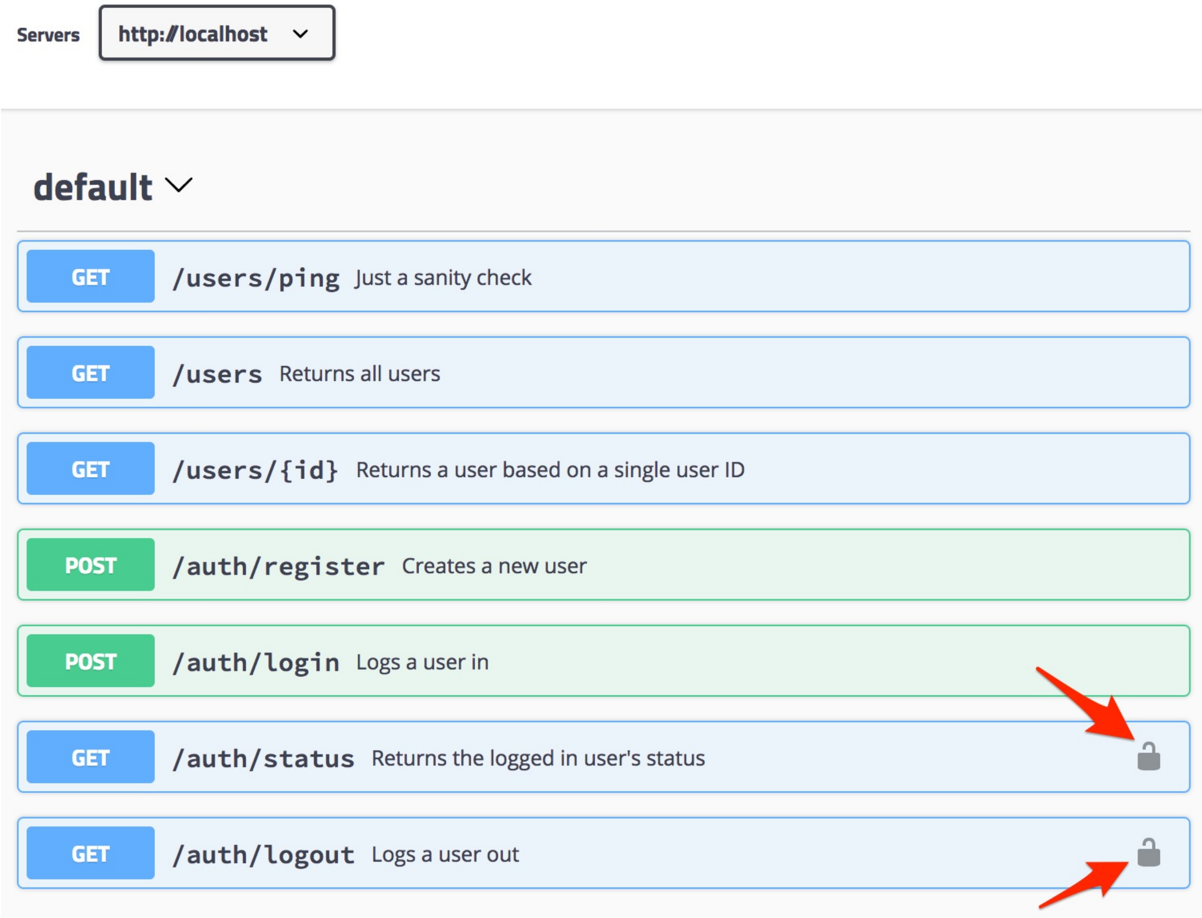
`/auth/status` :

```
"/auth/status": {
  "get": {
    "summary": "Returns the logged in user's status",
    "security": [
      {
        "bearerAuth": []
      }
    ],
    "responses": {
      "200": {
        "description": "user object"
      }
    }
  }
}
```

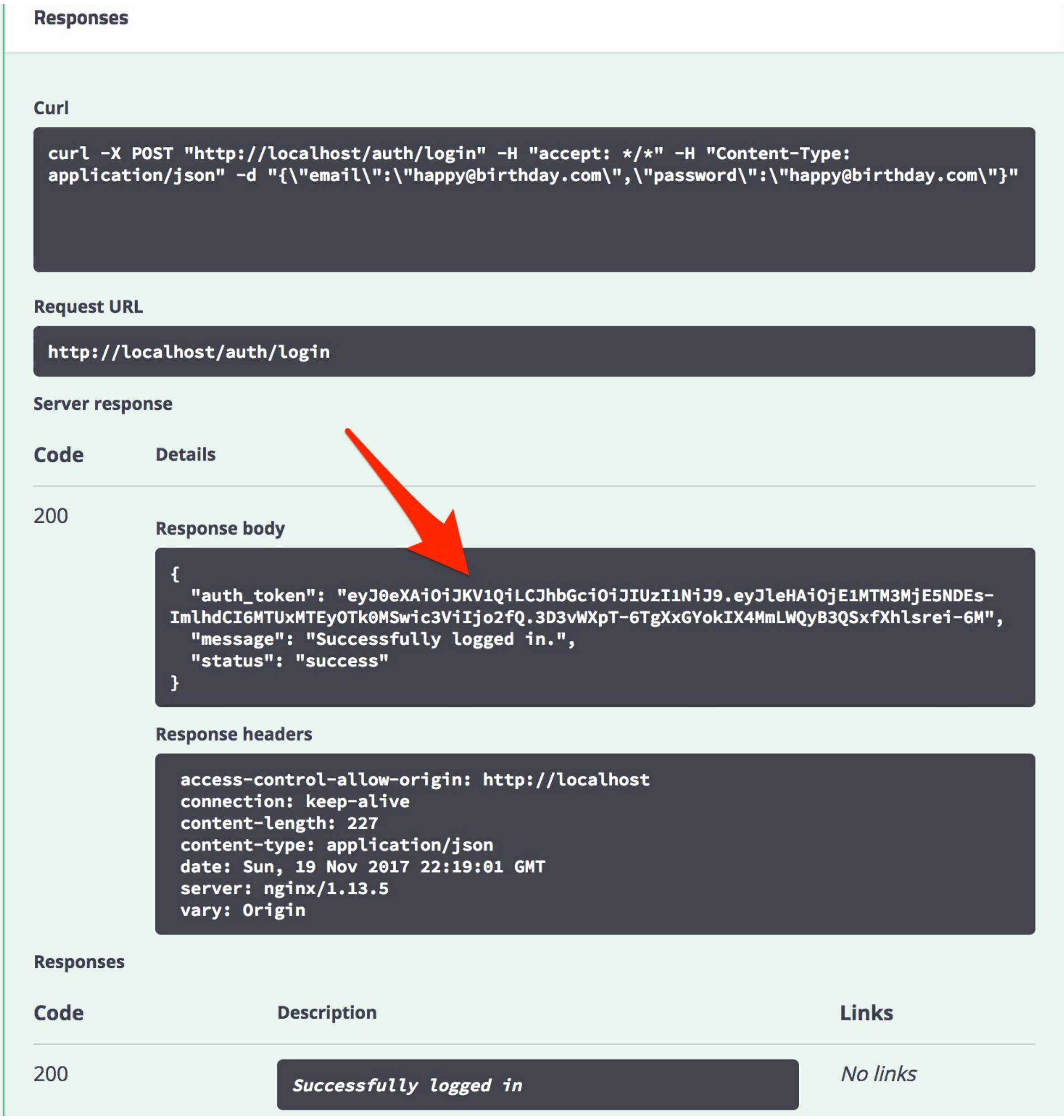
`/auth/logout` :

```
"/auth/logout": {
  "get": {
    "summary": "Logs a user out",
    "security": [
      {
        "bearerAuth": []
      }
    ],
    "responses": {
      "200": {
        "description": "Successfully logged out"
      }
    }
  }
}
```

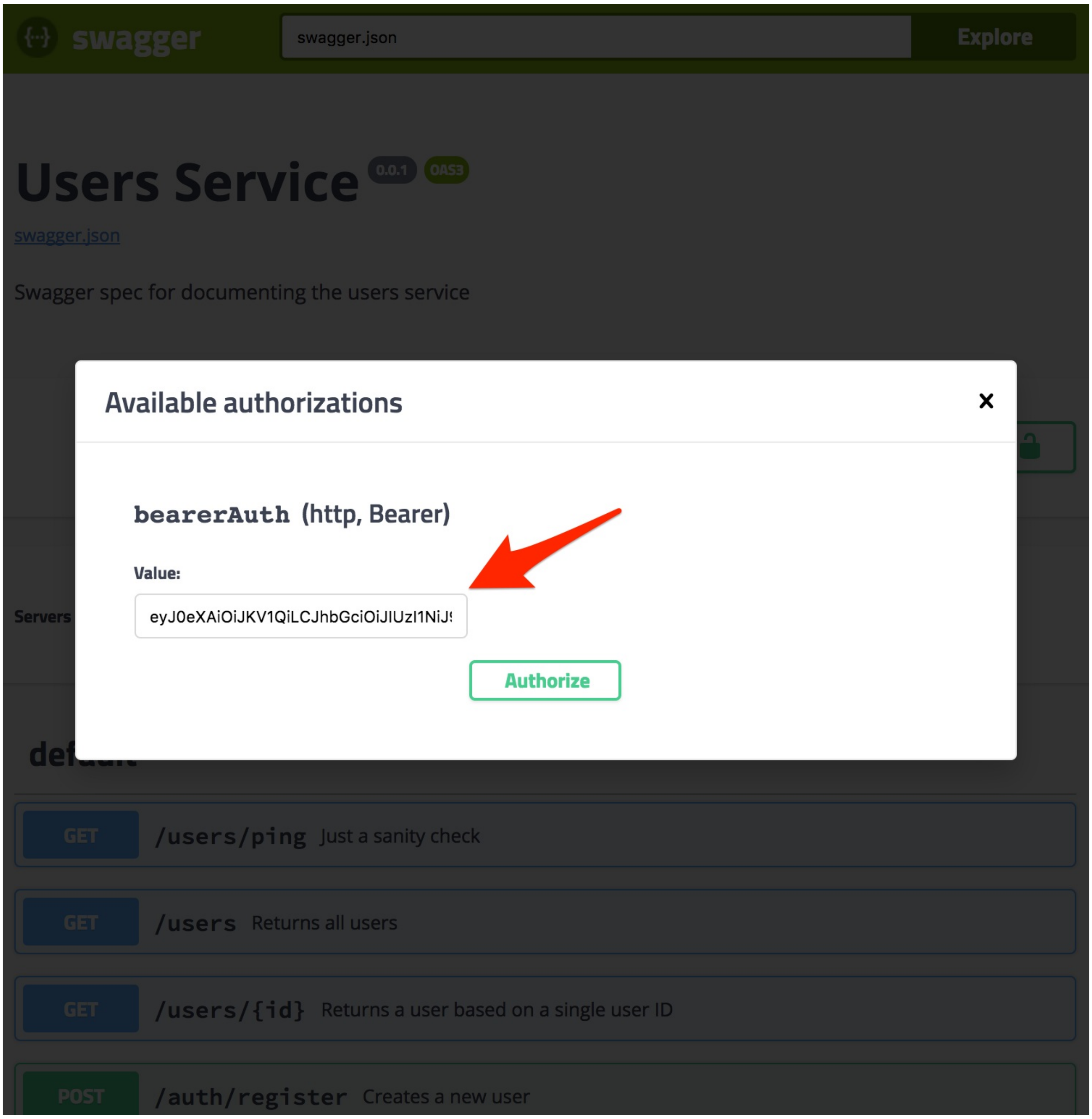
Refresh the browser.




To test, first log a user in and grab the provided token.




Then click the "Authorize" button at the top of the page and add the token to the input box.




Finally, for the `/users` route, since we already defined a `users` path, we can just add a new request method to the current object:

```
"/users": {
  "get": {
    "summary": "Returns all users",
    "responses": {
      "200": {
        "description": "user object"
      }
    }
  },
  "post": {
    "summary": "Adds a new user",
    "requestBody": {
      "description": "User to add",
```

```
        "required": true,
        "content": {
          "application/json": {
            "schema": {
              "$ref": "#/components/schemas/user-full"
            }
          }
        }
      },
      "security": [
        {
          "bearerAuth": []
        }
      ],
      "responses": {
        "200": {
          "description": "User added"
        }
      }
    }
  },
```

Remember: To test this route, you will need to be authenticated as an admin.

## Next Steps

Before moving on, add error handling to the `responses` for each path, based on the actual error responses from the users service.

Commit and push your code.

# Staging Environment

In this lesson, we'll set up a staging environment on AWS...

---

It's important to test applications out in an environment as close to production as possible when deploying to avoid hitting unexpected, environment-specific bugs in production. Docker containers help to eliminate much of the disparity between development and production, but problems can (and will) still arise. So, let's set up a staging environment.

## Docker Machine

Create a new Docker machine:

```
$ docker-machine create --driver amazonec2 testdriven-stage
```

## Docker Compose

While the new EC2 instance is being provisioned, create a new file called *docker-compose-stage.yml*:

```
version: '3.4'

services:

  users:
    container_name: users
    build:
      context: ./services/users
      dockerfile: Dockerfile-stage
    expose:
      - 5000
    environment:
      - APP_SETTINGS=project.config.StagingConfig
      - DATABASE_URL=postgres://postgres:postgres@users-db:5432/users_stage
      - DATABASE_TEST_URL=postgres://postgres:postgres@users-db:5432/users_test
      - SECRET_KEY=my_precious
    depends_on:
      - users-db
    links:
      - users-db

  users-db:
    container_name: users-db
    build:
      context: ./services/users/project/db
      dockerfile: Dockerfile
```

```
    expose:
      - 5432
    environment:
      - POSTGRES_USER=postgres
      - POSTGRES_PASSWORD=postgres

  client:
    container_name: client
    build:
      context: ./services/client
      dockerfile: Dockerfile-stage
      args:
        - NODE_ENV=production
        - REACT_APP_USERS_SERVICE_URL=${REACT_APP_USERS_SERVICE_URL}
    expose:
      - 80
    depends_on:
      - users
    links:
      - users

  nginx:
    container_name: nginx
    build:
      context: ./services/nginx
      dockerfile: Dockerfile-stage
    restart: always
    ports:
      - 80:80
    depends_on:
      - users
      - client

  swagger:
    container_name: swagger
    build:
      context: ./services/swagger
      dockerfile: Dockerfile-stage
    expose:
      - 8080
    environment:
      - URL=swagger.json
    depends_on:
      - users
```

Take note of any changes, and then update *services/users/project/db/create.sql*:

```
CREATE DATABASE users_prod;
CREATE DATABASE users_stage;
CREATE DATABASE users_dev;
```

```
CREATE DATABASE users_test;
```

Then, add a `StagingConfig` to *services/users/project/config.py*:

```
class StagingConfig(BaseConfig):
    """Staging configuration"""
    SQLALCHEMY_DATABASE_URI = os.environ.get('DATABASE_URL')
```

## Dockerfiles

Add four new *Dockerfiles*:

1. *services/users/Dockerfile-stage*
2. *services/client/Dockerfile-stage*
3. *services/swagger/Dockerfile-stage*
4. *services/nginx/Dockerfile-stage*

*services/users/Dockerfile-stage*

```
FROM python:3.6.4

# install environment dependencies
RUN apt-get update -yqq \
  && apt-get install -yqq --no-install-recommends \
    netcat \
  && apt-get -q clean

# set working directory
RUN mkdir -p /usr/src/app
WORKDIR /usr/src/app

# add requirements
COPY ./requirements.txt /usr/src/app/requirements.txt

# install requirements
RUN pip install -r requirements.txt

# add entrypoint.sh
COPY ./entrypoint-prod.sh /usr/src/app/entrypoint-prod.sh

# add app
COPY . /usr/src/app

# run server
CMD ["./entrypoint-prod.sh"]
```

*services/client/Dockerfile-stage*

```
# build environment
FROM node:9.4 as builder
RUN mkdir /usr/src/app
WORKDIR /usr/src/app
ENV PATH /usr/src/app/node_modules/.bin:$PATH
ARG REACT_APP_USERS_SERVICE_URL
ARG NODE_ENV
ENV NODE_ENV $NODE_ENV
ENV REACT_APP_USERS_SERVICE_URL $REACT_APP_USERS_SERVICE_URL
COPY package.json /usr/src/app/package.json
RUN npm install --silent
RUN npm install react-scripts@1.1.0 -g --silent
COPY . /usr/src/app
RUN npm run build

# production environment
FROM nginx:1.13.5-alpine
RUN rm -rf /etc/nginx/conf.d
COPY conf /etc/nginx
COPY --from=builder /usr/src/app/build /usr/share/nginx/html
EXPOSE 80
CMD ["nginx", "-g", "daemon off;"]
```

*services/swagger/Dockerfile-stage*

```
FROM nginx:1.13.8

ENV SWAGGER_UI_VERSION 3.9.3
ENV URL **None**

RUN apt-get update \
    && apt-get install -y curl \
    && curl -L https://github.com/swagger-api/swagger-ui/archive/v${SWAGGER_UI_VERS
ION}.tar.gz | tar -zxv -C /tmp \
    && cp -R /tmp/swagger-ui-${SWAGGER_UI_VERSION}/dist/* /usr/share/nginx/html \
    && rm -rf /tmp/*

RUN rm /etc/nginx/conf.d/default.conf
COPY /nginx.conf /etc/nginx/conf.d

COPY swagger.json /usr/share/nginx/html/swagger.json
COPY start.sh /start.sh

RUN ["chmod", "+x", "/start.sh"]
CMD ["/start.sh"]
```

*services/nginx/Dockerfile-stage*

```
FROM nginx:1.13.8

RUN rm /etc/nginx/conf.d/default.conf
COPY /prod.conf /etc/nginx/conf.d
```

### Swagger

Next, we need to update the host URL in *services/swagger/swagger.json*:

```
...
"servers": [
  {
    "url": "http://UPDATE_ME"
  }
],
...
```

Let's write a script for this.

```
import os
import sys
import json


def update_json_file(url):
    full_path = os.path.abspath('services/swagger/swagger.json')
    with open(full_path, 'r') as file:
        data = json.load(file)
    data['servers'][0]['url'] = url
    with open(full_path, 'w') as file:
        json.dump(data, file)
    return True


if __name__ == '__main__':
    try:
        update_json_file(sys.argv[1])
    except IndexError:
        print('Please provide a URL.')
        print('USAGE: python update-spec.py URL')
        sys.exit()
```

Save this as *update-spec.py* to the "swagger" directory. Grab the IP for the `staging` machine, and then run the script:

```
$ python services/swagger/update-spec.py http://DOCKER_MACHINE_STAGING_IP
```

## Deploy

Update the `REACT_APP_USERS_SERVICE_URL` environment variable:

```
$ export REACT_APP_USERS_SERVICE_URL=http://DOCKER_MACHINE_STAGING_IP
```

Spin up the containers:

```
$ docker-compose -f docker-compose-stage.yml \
  up -d --build
```

Create and seed the database and run the following tests:

```
$ docker-compose -f docker-compose-stage.yml \
  run users python manage.py recreate_db

$ docker-compose -f docker-compose-stage.yml \
  run users python manage.py seed_db

$ docker-compose -f docker-compose-stage.yml \
  run users python manage.py test

$ docker-compose -f docker-compose-stage.yml \
  run users flake8 project
```

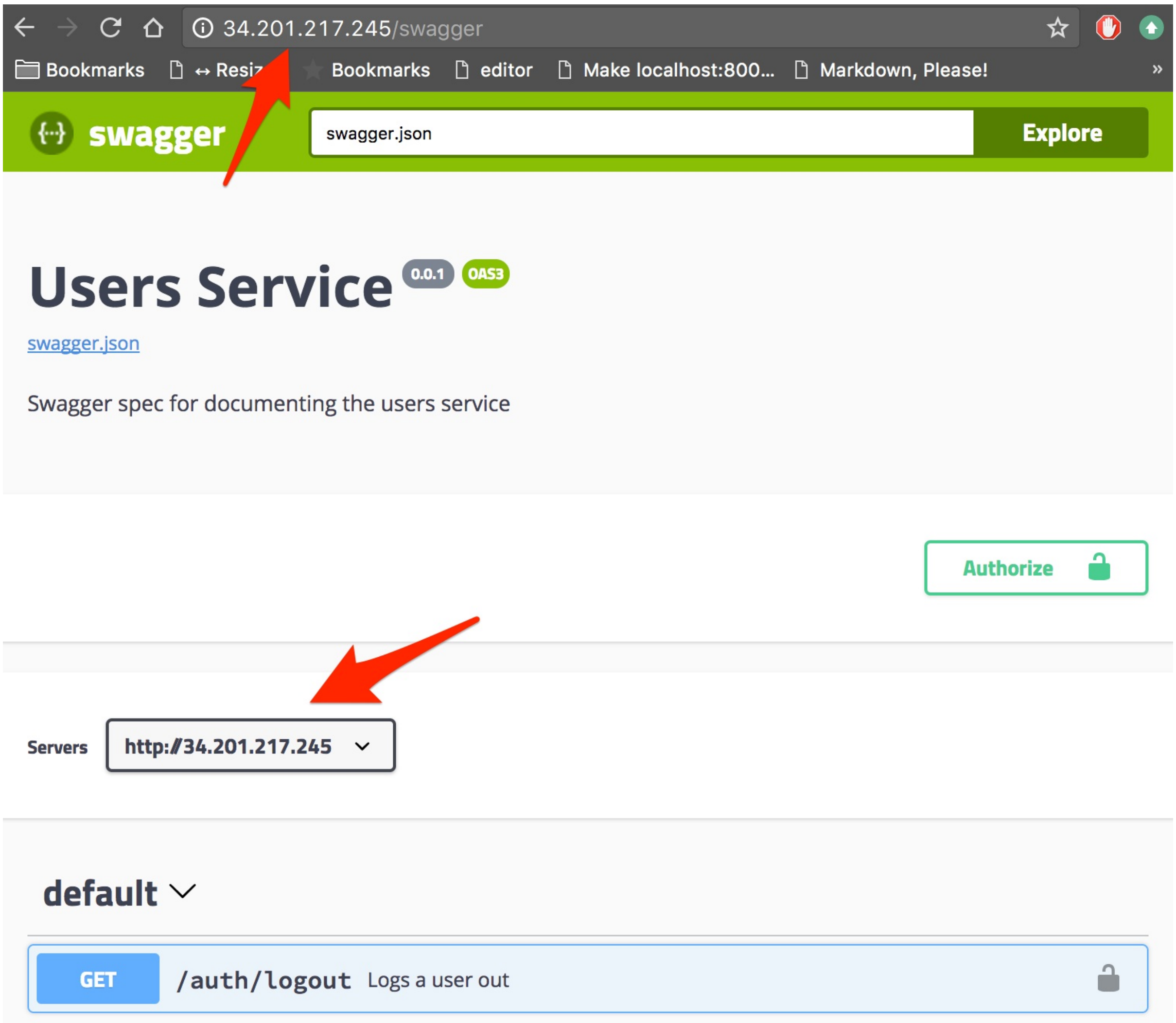
Set the `TEST_URL` variable for the end-to-end tests:

```
$ export TEST_URL=http://DOCKER_MACHINE_STAGING_IP
```

Run the end-to-end tests:

```
$ testcafe chrome e2e
```

Make sure the Swagger docs are up and running as well.


34.201.217.245/swagger
Bookmarks
↔ Resiz
Bookmarks
editor
Make localhost:800...
Markdown, Please!
swagger
swagger.json
Explore
Users Service
0.0.1
OAS3
swagger.json
Swagger spec for documenting the users service
Authorize
Servers
http://34.201.217.245
default
GET
/auth/logout
Logs a user out

# Production Environment

In this lesson, we'll update the production environment on AWS...

---

Think about the steps needed to update the production environment:

1. Add the Swagger service to *docker-compose-prod.yml*
2. Add *Dockerfile-prod* to "services/swagger"
3. Change to the `testdriven-prod` machine, taking note of the IP address
4. Run *update-spec.py*
5. Set the proper environment variables
6. Update the containers and run the automated tests

Try doing this on your own!

## Docker Compose

Add the service to *docker-compose-prod.yml*:

```
version: '3.4'

services:

  users:
    container_name: users
    build:
      context: ./services/users
      dockerfile: Dockerfile-prod
    expose:
      - 5000
    environment:
      - APP_SETTINGS=project.config.ProductionConfig
      - DATABASE_URL=postgres://postgres:postgres@users-db:5432/users_prod
      - DATABASE_TEST_URL=postgres://postgres:postgres@users-db:5432/users_test
      - SECRET_KEY=${SECRET_KEY}
    depends_on:
      - users-db
    links:
      - users-db

  users-db:
    container_name: users-db
    build:
      context: ./services/users/project/db
      dockerfile: Dockerfile
    expose:
      - 5432
```

```
    environment:
      - POSTGRES_USER=postgres
      - POSTGRES_PASSWORD=postgres

  client:
    container_name: client
    build:
      context: ./services/client
      dockerfile: Dockerfile-prod
      args:
        - NODE_ENV=production
        - REACT_APP_USERS_SERVICE_URL=${REACT_APP_USERS_SERVICE_URL}
    expose:
      - 80
    depends_on:
      - users
    links:
      - users

  nginx:
    container_name: nginx
    build:
      context: ./services/nginx
      dockerfile: Dockerfile-prod
    restart: always
    ports:
      - 80:80
    depends_on:
      - users
      - client

  swagger:
    container_name: swagger
    build:
      context: ./services/swagger
      dockerfile: Dockerfile-prod
    expose:
      - 8080
    environment:
      - URL=swagger.json
    depends_on:
      - users
```

## Dockerfile

Create a new file called *services/swagger/Dockerfile-prod*:

```
FROM nginx:1.13.8

ENV SWAGGER_UI_VERSION 3.9.3
```

```
ENV URL **None**

RUN apt-get update \
    && apt-get install -y curl \
    && curl -L https://github.com/swagger-api/swagger-ui/archive/v${SWAGGER_UI_VERS
ION}.tar.gz | tar -zxv -C /tmp \
    && cp -R /tmp/swagger-ui-${SWAGGER_UI_VERSION}/dist/* /usr/share/nginx/html \
    && rm -rf /tmp/*

RUN rm /etc/nginx/conf.d/default.conf
COPY /nginx.conf /etc/nginx/conf.d

COPY swagger.json /usr/share/nginx/html/swagger.json
COPY start.sh /start.sh

RUN ["chmod", "+x", "/start.sh"]
CMD ["/start.sh"]
```

## Docker Machine

Set `testdriven-prod` as the active Docker Machine:

```
$ docker-machine env testdriven-prod
$ eval $(docker-machine env testdriven-prod)
```

Grab the IP address:

```
$ docker-machine ip testdriven-prod
```

## Swagger

Update *swagger.json*:

```
$ python services/swagger/update-spec.py http://DOCKER_MACHINE_PROD_IP
```

## Environment Variables

Set the following environment variables:

```
$ export SERVER_KEY=test
$ export REACT_APP_USERS_SERVICE_URL=http://DOCKER_MACHINE_PROD_IP
$ export TEST_URL=http://DOCKER_MACHINE_PROD_IP
```

## Deploy

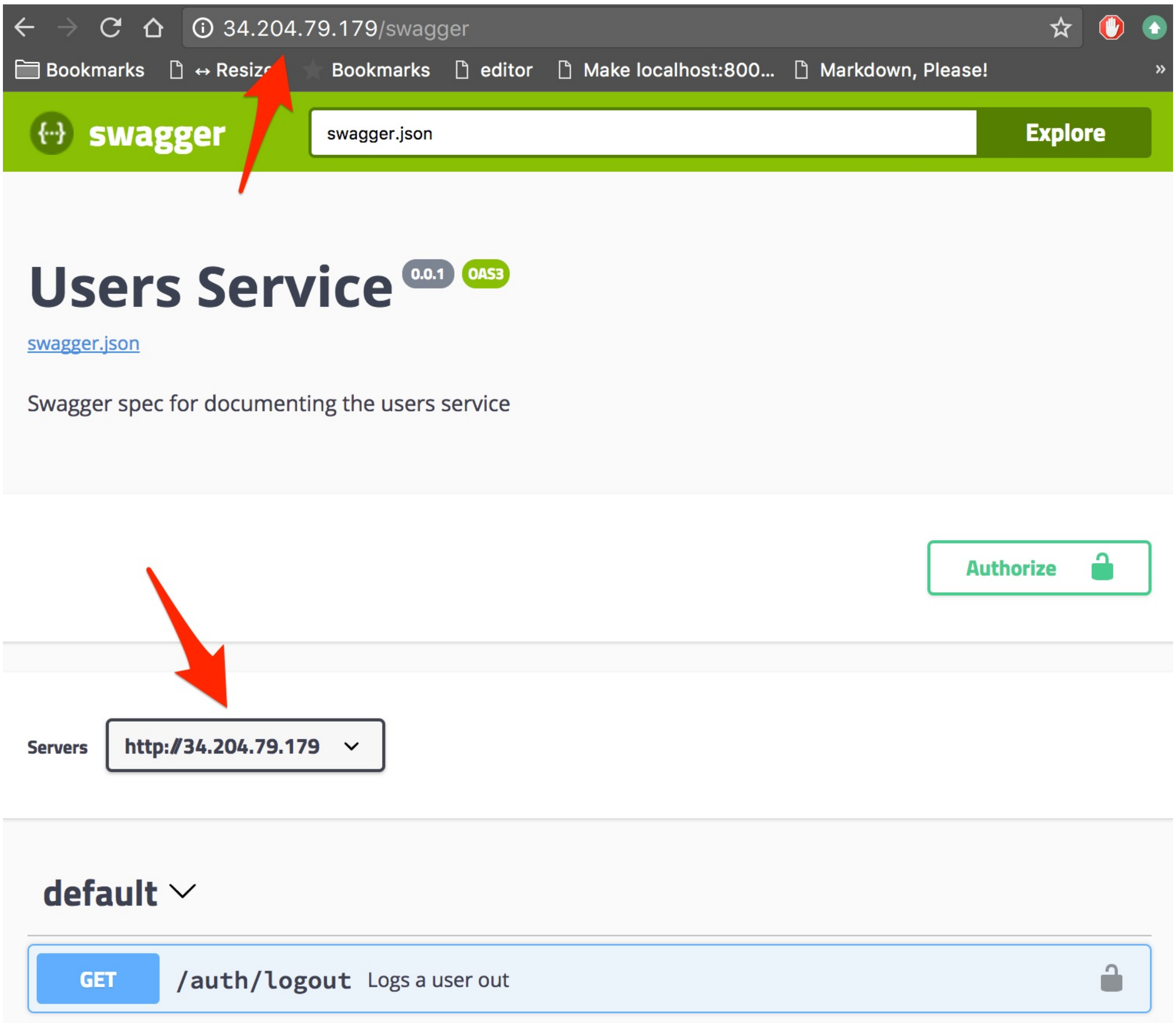
Update:

```
$ docker-compose -f docker-compose-prod.yml \
  up -d --build
```

Ensure all is well:

```
$ docker-compose -f docker-compose-prod.yml \
  run users python manage.py recreate_db

$ docker-compose -f docker-compose-prod.yml \
  run users python manage.py seed_db

$ docker-compose -f docker-compose-prod.yml \
  run users python manage.py test

$ docker-compose -f docker-compose-prod.yml \
  run users flake8 project

$ testcafe chrome e2e
```

Make sure the Swagger docs are up and running. Commit and push your code! The build should now pass.


34.204.79.179/swagger
Bookmarks
↔ Resize
Bookmarks
editor
Make localhost:800...
Markdown, Please!
swagger
swagger.json
Explore
Users Service
0.0.1
OAS3
swagger.json
Swagger spec for documenting the users service
Authorize
Servers
http://34.204.79.179
default
GET
/auth/logout
Logs a user out

# Workflow

Updated reference guide...

## All Services

The following commands are for spinning up all the containers...

### Environment Variables

Development:

```
$ export REACT_APP_USERS_SERVICE_URL=http://DOCKER_MACHINE_DEV_IP
$ export TEST_URL=http://DOCKER_MACHINE_DEV_IP
```

Staging:

```
$ export REACT_APP_USERS_SERVICE_URL=http://DOCKER_MACHINE_STAGING_IP
$ export TEST_URL=http://DOCKER_MACHINE_STAGING_IP
```

Production:

```
$ export REACT_APP_USERS_SERVICE_URL=http://DOCKER_MACHINE_PROD_IP
$ export SECRET_KEY=SOMETHING_SUPER_SECRET
$ export TEST_URL=http://DOCKER_MACHINE_PROD_IP
```

### Start

Build the images:

```
$ docker-compose -f docker-compose-dev.yml build
```

Run the containers:

```
$ docker-compose -f docker-compose-dev.yml up -d
```

Create and seed the database:

```
$ docker-compose -f docker-compose-dev.yml \
  run users python manage.py recreate_db

$ docker-compose -f docker-compose-dev.yml \
  run users python manage.py seed_db
```

Run the unit and integration tests:

```
$ docker-compose -f docker-compose-dev.yml \
  run users python manage.py test
```

Lint:

```
$ docker-compose -f docker-compose-dev.yml \
  run users flake8 project
```

Run the client-side tests:

```
$ docker-compose -f docker-compose-dev.yml \
  run client npm test -- --verbose
```

Run the e2e tests:

```
$ testcafe chrome e2e
```

### Stop

Stop the containers:

```
$ docker-compose -f docker-compose-dev.yml stop
```

Bring down the containers:

```
$ docker-compose -f docker-compose-dev.yml down
```

Remove images:

```
$ docker rmi $(docker images -q)
```

## Individual Services

The following commands are for spinning up individual containers...

### Users DB

Build and run:

```
$ docker-compose -f docker-compose-dev.yml up -d --build users-db
```

Test:

```
$ docker exec -ti users-db psql -U postgres -W
```

## Users

Build and run:

```
$ docker-compose -f docker-compose-dev.yml up -d --build users
```

Create and seed the database:

```
$ docker-compose -f docker-compose-dev.yml \
  run users python manage.py recreate_db
$ docker-compose -f docker-compose-dev.yml \
  run users python manage.py seed_db
```

Run the unit and integration tests:

```
$ docker-compose -f docker-compose-dev.yml run users python manage.py test
```

Lint:

```
$ docker-compose -f docker-compose-dev.yml \
  run users flake8 project
```

## Client

Set env variable:

```
$ export REACT_APP_USERS_SERVICE_URL=http://DOCKER_MACHINE_DEV_IP
```

Build and run:

```
$ docker-compose -f docker-compose-dev.yml up -d --build web-service
```

To test, navigate to http://DOCKER_MACHINE_DEV_IP:3007 in your browser.

> Keep in mind that you won't be able to register or log in until Nginx is set up.

Run the client-side tests:

```
$ docker-compose -f docker-compose-dev.yml \
  run client npm test -- --verbose
```

## Nginx

Build and run:

```
$ docker-compose -f docker-compose-dev.yml up -d --build nginx
```

To test, navigate to http://DOCKER_MACHINE_DEV_IP in your browser. Also, run the e2e tests:

```
$ export TEST_URL=http://DOCKER_MACHINE_DEV_IP
$ testcafe chrome e2e
```

### Swagger

Build and run:

```
$ docker-compose -f docker-compose-dev.yml up -d --build swagger
```

To test, navigate to http://DOCKER_MACHINE_DEV_IP:3008 in your browser.

## Aliases

To save some precious keystrokes, create aliases for both the `docker-compose` and `docker-machine` commands - `dc` and `dm` , respectively.

Simply add the following lines to your *.bashrc* file:

```
alias dc='docker-compose'
alias dm='docker-machine'
```

Save the file, then execute it:

```
$ source ~/.bashrc
```

Test out the new aliases!

> On Windows? You will first need to create a PowerShell Profile (if you don't already have one), and then you can add the aliases to it using Set-Alias - i.e., `Set-Alias dc docker-compose` .

## "Saved" State

Is the VM stuck in a "Saved" state?

```
$ docker-machine ls

NAME    ACTIVE    DRIVER       STATE      URL                          SWARM    DOCKER
      ERRORS
aws      *        amazonec2    Running    tcp://34.207.173.181:2376             v17.05.0
```

```
-ce
dev     -         virtualbox   Saved                                          Unknown
```

To break out of this, you'll need to power off the VM:

1. Start virtualbox - `virtualbox`
2. Select the VM and click "start"
3. Exit the VM and select "Power off the machine"
4. Exit virtualbox

The VM should now have a "Stopped" state:

```
$ docker-machine ls

NAME   ACTIVE   DRIVER       STATE     URL                         SWARM   DOCKER
      ERRORS
aws     *       amazonec2    Running   tcp://34.207.173.181:2376            v17.05.0
-ce
dev     -       virtualbox   Stopped                                        Unknown
```

Now you can start the machine:

```
$ docker-machine start dev
```

It should be "Running":

```
$ docker-machine ls

NAME   ACTIVE   DRIVER       STATE     URL                         SWARM   DOCKER
      ERRORS
aws     *       amazonec2    Running   tcp://34.207.173.181:2376            v17.05.0
-ce
dev     -       virtualbox   Running   tcp://192.168.99.100:2376            v17.05.0
-ce
```

## Other Commands

Want to force a build?

```
$ docker-compose -f docker-compose-dev.yml build --no-cache
```

Remove images:

```
$ docker rmi $(docker images -q)
```

Reset Docker environment back to localhost, unsetting all Docker environment variables:

```
$ eval $(docker-machine env -u)
```

# Structure

At the end of part 4, your project structure should look like this:

```
├── README.md
├── docker-compose-dev.yml
├── docker-compose-prod.yml
├── docker-compose-stage.yml
├── e2e
│   ├── index.test.js
│   ├── login.test.js
│   ├── message.test.js
│   ├── register.test.js
│   └── status.test.js
├── package.json
├── services
│   ├── client
│   │   ├── Dockerfile-dev
│   │   ├── Dockerfile-prod
│   │   ├── Dockerfile-stage
│   │   ├── README.md
│   │   ├── build
│   │   ├── conf
│   │   │   └── conf.d
│   │   │       └── default.conf
│   │   ├── coverage
│   │   ├── package.json
│   │   ├── public
│   │   │   ├── favicon.ico
│   │   │   ├── index.html
│   │   │   └── manifest.json
│   │   └── src
│   │       ├── App.jsx
│   │       ├── components
│   │       │   ├── About.jsx
│   │       │   ├── AddUser.jsx
│   │       │   ├── Logout.jsx
│   │       │   ├── Message.jsx
│   │       │   ├── NavBar.jsx
│   │       │   ├── UserStatus.jsx
│   │       │   ├── UsersList.jsx
│   │       │   ├── __tests__
│   │       │   │   ├── About.test.jsx
│   │       │   │   ├── AddUser.test.jsx
│   │       │   │   ├── App.test.jsx
│   │       │   │   ├── Form.test.jsx
│   │       │   │   ├── FormErrors.test.jsx
│   │       │   │   ├── Logout.test.jsx
│   │       │   │   ├── Message.test.jsx
```

```
│   │       │   │   ├── NavBar.test.jsx
│   │       │   │   ├── UsersList.test.jsx
│   │       │   │   └── __snapshots__
│   │       │   │       ├── About.test.jsx.snap
│   │       │   │       ├── AddUser.test.jsx.snap
│   │       │   │       ├── Form.test.jsx.snap
│   │       │   │       ├── FormErrors.test.jsx.snap
│   │       │   │       ├── Logout.test.jsx.snap
│   │       │   │       ├── Message.test.jsx.snap
│   │       │   │       ├── NavBar.test.jsx.snap
│   │       │   │       └── UsersList.test.jsx.snap
│   │       │   └── forms
│   │       │       ├── Form.jsx
│   │       │       ├── FormErrors.css
│   │       │       ├── FormErrors.jsx
│   │       │       └── form-rules.js
│   │       ├── index.js
│   │       ├── logo.svg
│   │       ├── registerServiceWorker.js
│   │       └── setupTests.js
│   ├── nginx
│   │   ├── Dockerfile-dev
│   │   ├── Dockerfile-prod
│   │   ├── Dockerfile-stage
│   │   ├── dev.conf
│   │   └── prod.conf
│   ├── swagger
│   │   ├── Dockerfile-dev
│   │   ├── Dockerfile-prod
│   │   ├── Dockerfile-stage
│   │   ├── nginx.conf
│   │   ├── start.sh
│   │   ├── swagger.json
│   │   └── update-spec.py
│   └── users
│       ├── Dockerfile-dev
│       ├── Dockerfile-prod
│       ├── Dockerfile-stage
│       ├── entrypoint-prod.sh
│       ├── entrypoint.sh
│       ├── htmlcov
│       ├── manage.py
│       ├── migrations
│       ├── project
│       │   ├── __init__.py
│       │   ├── api
│       │   │   ├── __init__.py
│       │   │   ├── auth.py
│       │   │   ├── models.py
│       │   │   ├── templates
│       │   │   │   └── index.html
```

```
│       │   │   ├── users.py
│       │   │   └── utils.py
│       │   ├── config.py
│       │   ├── db
│       │   │   ├── Dockerfile
│       │   │   └── create.sql
│       │   └── tests
│       │       ├── __init__.py
│       │       ├── base.py
│       │       ├── test_auth.py
│       │       ├── test_config.py
│       │       ├── test_user_model.py
│       │       ├── test_users.py
│       │       └── utils.py
│       └── requirements.txt
└── test.sh
```

Code for part 4: https://github.com/testdrivenio/testdriven-app-2.2/releases/tag/part4

# Part 5

In part 5, we'll dive into *container orchestration* with Amazon ECS as we move our staging and production environments to a more scaleable infrastructure. We'll also add Amazon Elastic Container Registry along with Amazon's Elastic Load Balancing for *load balancing* and Amazon's Relational Database Service for *data persistence*.

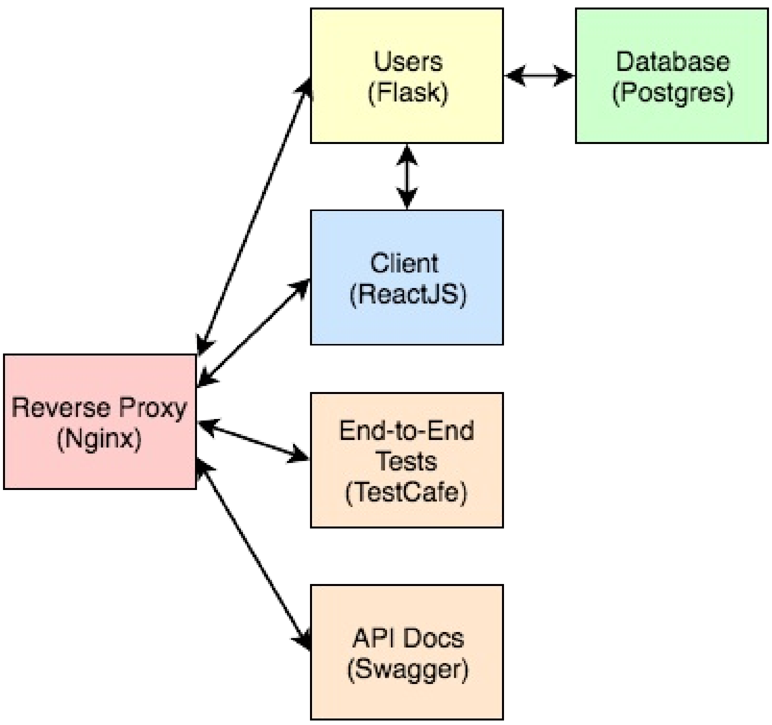
## Objectives

By the end of part 5, you will be able to...

1. Explain what container orchestration is and why you may need to use an orchestration tool to manage deployments
2. Discuss the pros and cons of using Elastic Container Service (ECS) over other orchestration tools like Kubernetes, Mesos, and Docker Swarm
3. Set up an IAM user
4. Configure an Application Load Balancer (ALB) along with ECS to run a set of microservices
5. Integrate Amazon Elastic Container Registry (ECR) into the deployment process
6. Send container logs to CloudWatch
7. Update a running container via a zero-downtime deployment strategy to not disrupt the current users or your application
8. Explain the types of scaling that are available to you within ECS
9. Add Relational Database Service, for data persistence, to our production stack

---

## App



Check out the live apps, running on EC2 -

1. Production
2. Staging

You can also test out the following endpoints...

| Endpoint | HTTP Method | Authenticated? | Result |
| --- | --- | --- | --- |
| /auth/register | POST | No | register user |
| /auth/login | POST | No | log in user |
| /auth/logout | GET | Yes | log out user |
| /auth/status | GET | Yes | check user status |
| /users | GET | No | get all users |
| /users/:id | GET | No | get single user |
| /users | POST | Yes (admin) | add a user |
| /users/ping | GET | No | sanity check |

Finished code for part 5: https://github.com/testdrivenio/testdriven-app-2.2/releases/tag/part5

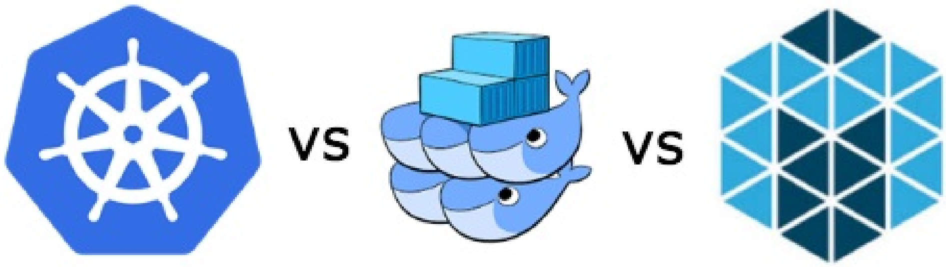
## Dependencies

No new dependencies.

# Container Orchestration

## What is Container Orchestration?

As you move from deploying containers on a single machine to deploying them across a number of machines, you need an orchestration tool to manage the arrangement and coordination of the containers across the entire system. This is where ECS fits in along with a number of other orchestration tools - like Kubernetes, Mesos, and Docker Swarm.




## Why ECS?

ECS is simpler to set up and easier to use and you have the full power of AWS behind it, so you can easily integrate it into other AWS services (which we will be doing shortly). In short, you get scheduling, service discovery, load balancing, and auto-scaling out-of-the-box. Plus, you can take full advantage of EC2's multiple availability-zones.

If you're already on AWS and have no desire to leave, then it makes sense to use AWS.

Keep in mind, that ECS is often lagging behind Kubernetes, in terms of features, though. If you're looking for the most features and portability and you don't mind installing and managing the tool, then Kubernetes, Docker Swarm, or Mesos may be right for you.

One last thing to take note of is that since ECS is closed-source, there isn't a true way to run an environment locally in order to achieve development-to-production parity.

> For more, review the Choosing the Right Containerization and Cluster Management Tool blog post.

## Orchestration Feature Wish-List

Most orchestration tools come with a core set of features. You can find those features below along with the associated AWS service...

| Feature | Info | AWS Service |
|---|---|---|
| Health checks | Verify when a task is ready to accept traffic | ALB |

| | | |
|---|---|---|
| Path-based routing | Forward requests based on the URL path | ALB |
| Dynamic port-mapping | Ports are assigned dynamically when a new container is spun up | ALB |
| Zero-downtime deployments | Deployments do not disrupt the users | ALB |
| Service discovery | Automatic detection of new containers and services | ALB, ECS |
| High availability | Containers are evenly distributed across Availability Zones | ECS |
| Auto scaling | Automatically scaling resources up or down based on fluctuations in traffic patterns or metrics (like CPU usage) | ECS |
| Provisioning | New containers should select hosts based on resources and configuration | ECS |
| Container storage | Private image storage and management | ECR |
| Container logs | Centralized storage of container logs | CloudWatch |
| Monitoring | Ability to monitor basic stats like CPU usage, memory, I/O, and network usage as well as set alarms and create events | CloudWatch |
| Secrets management | Sensitive info should be encrypted and stored in a centralized store | Parameter Store, KMS, IAM |

> If you're completely new to ECS, please review the Getting Started with Amazon ECS guide.

# IAM

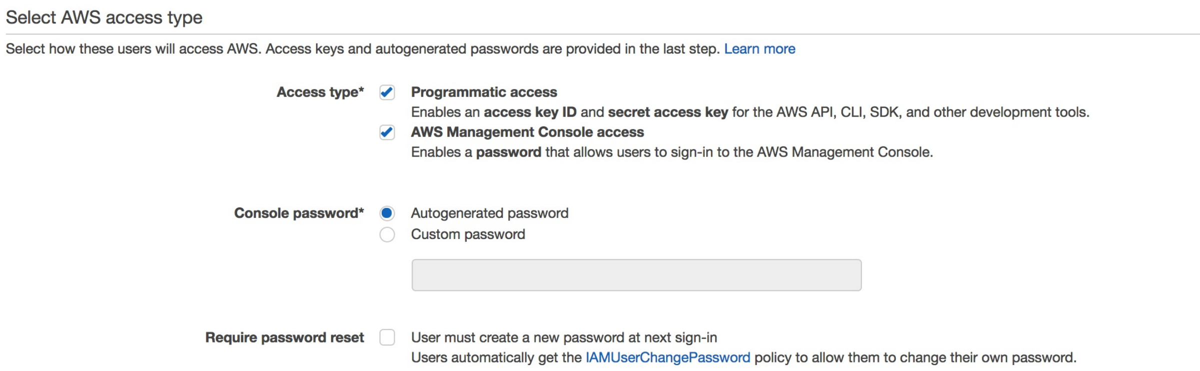
In this lesson, we'll configure a new AWS user with IAM...

---

Instead of using the root user (assuming you already set one up), let's configure a new user that has a bit less access.

> We will be using the `us-west-1` region throughout this course.

Navigate to the Amazon IAM dashboard, click "Users" and then "Add user". Add a username, select both of the checkmarks next to the "Access type", and then uncheck "Require password reset":




On the "Permissions" page, click "Attach existing policies directly" and check both the "Administrator Access" and "Billing" policies. Click "Next" to review and then create the new user.


Then, update your *~/.aws/credentials* file. Take note of the generated password and log in, with your new user, at https://YOUR_AWS_ACCOUNT_ID.signin.aws.amazon.com/console.

> Although not required, it's a good idea to set up another new IAM User and Role specifically for container instances. For more, review the following guide from Amazon.

# Elastic Container Registry

In this lesson, we'll add the Elastic Container Registry (ECR), a private image registry into the CI process...

---

> It's a good idea to set up a Billing Alert via CloudWatch to alert you if you're AWS usage costs exceed a certain amount. Review Creating a Billing Alarm for more info.

## Image Registry

A container image registry is used to store and distribute container images. Docker Hub is one of the more popular image registry services for public images - basically GitHub for Docker images.

> Review the following Stack Overflow article for more info on Docker Hub and image registries in general.

## ECR

Why Elastic Container Registry?

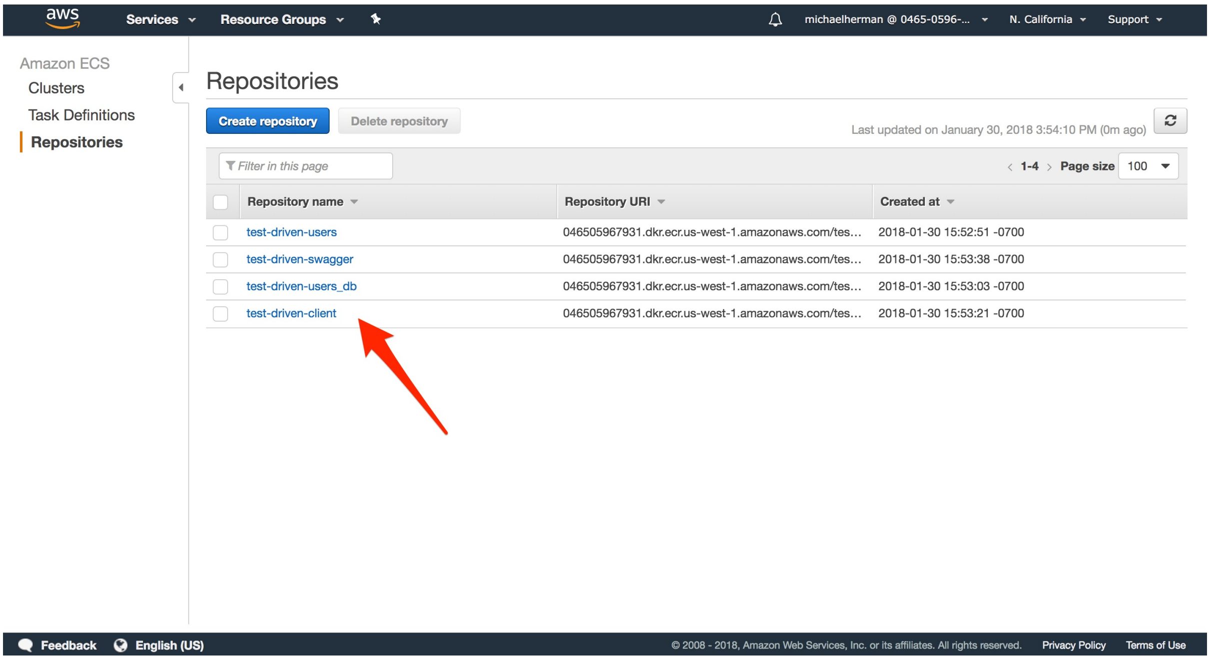
1. We do not want to add any sensitive info to the images on Docker Hub since they are publicly available
2. ECR plays nice with the Elastic Container Service (which we'll be setting up shortly)

Navigate to Amazon ECS, click "Repositories", and then add four new repositories:

1. *test-driven-users*
2. *test-driven-users_db*
3. *test-driven-client*
4. *test-driven-swagger*

> *Why only four images?* We'll use the Application Load Balancer instead of Nginx in our stack so we won't need that image or container.

You can ignore the "build, tag, and push" instructions. For now, just set up the images.




> You can also create a new repository with the AWS CLI:
>
> ```
> $ aws ecr create-repository --repository-name REPOSITORY_NAME
> ```

Instead of building the images locally, let's incorporate the process into our current CI workflow...

## Update Travis

Add a new file to the project root called *docker-push.sh*:

```
#!/bin/sh

if [ -z "$TRAVIS_PULL_REQUEST" ] || [ "$TRAVIS_PULL_REQUEST" == "false" ]
then

  if [ "$TRAVIS_BRANCH" == "staging" ] || \
     [ "$TRAVIS_BRANCH" == "production" ]
  then
    curl "https://s3.amazonaws.com/aws-cli/awscli-bundle.zip" -o "awscli-bundle.zip"

    unzip awscli-bundle.zip
    ./awscli-bundle/install -b ~/bin/aws
    export PATH=~/bin:$PATH
    # add AWS_ACCOUNT_ID, AWS_ACCESS_KEY_ID, AWS_SECRET_ACCESS_KEY env vars
    eval $(aws ecr get-login --region us-west-1 --no-include-email)
    export TAG=$TRAVIS_BRANCH
    export REPO=$AWS_ACCOUNT_ID.dkr.ecr.us-west-1.amazonaws.com
  fi

  if [ "$TRAVIS_BRANCH" == "staging" ] || \
     [ "$TRAVIS_BRANCH" == "production" ]
```

```
  then
    # users
    docker build $USERS_REPO -t $USERS:$COMMIT -f Dockerfile-prod
    docker tag $USERS:$COMMIT $REPO/$USERS:$TAG
    docker push $REPO/$USERS:$TAG
    # users db
    docker build $USERS_DB_REPO -t $USERS_DB:$COMMIT -f Dockerfile
    docker tag $USERS_DB:$COMMIT $REPO/$USERS_DB:$TAG
    docker push $REPO/$USERS_DB:$TAG
    # client
    docker build $CLIENT_REPO -t $CLIENT:$COMMIT -f Dockerfile-prod --build-arg REA
CT_APP_USERS_SERVICE_URL=TBD
    docker tag $CLIENT:$COMMIT $REPO/$CLIENT:$TAG
    docker push $REPO/$CLIENT:$TAG
    # swagger
    docker build $SWAGGER_REPO -t $SWAGGER:$COMMIT -f Dockerfile-prod $SWAGGER_DIR
    docker tag $SWAGGER:$COMMIT $REPO/$SWAGGER:$TAG
    docker push $REPO/$SWAGGER:$TAG
  fi
fi
```

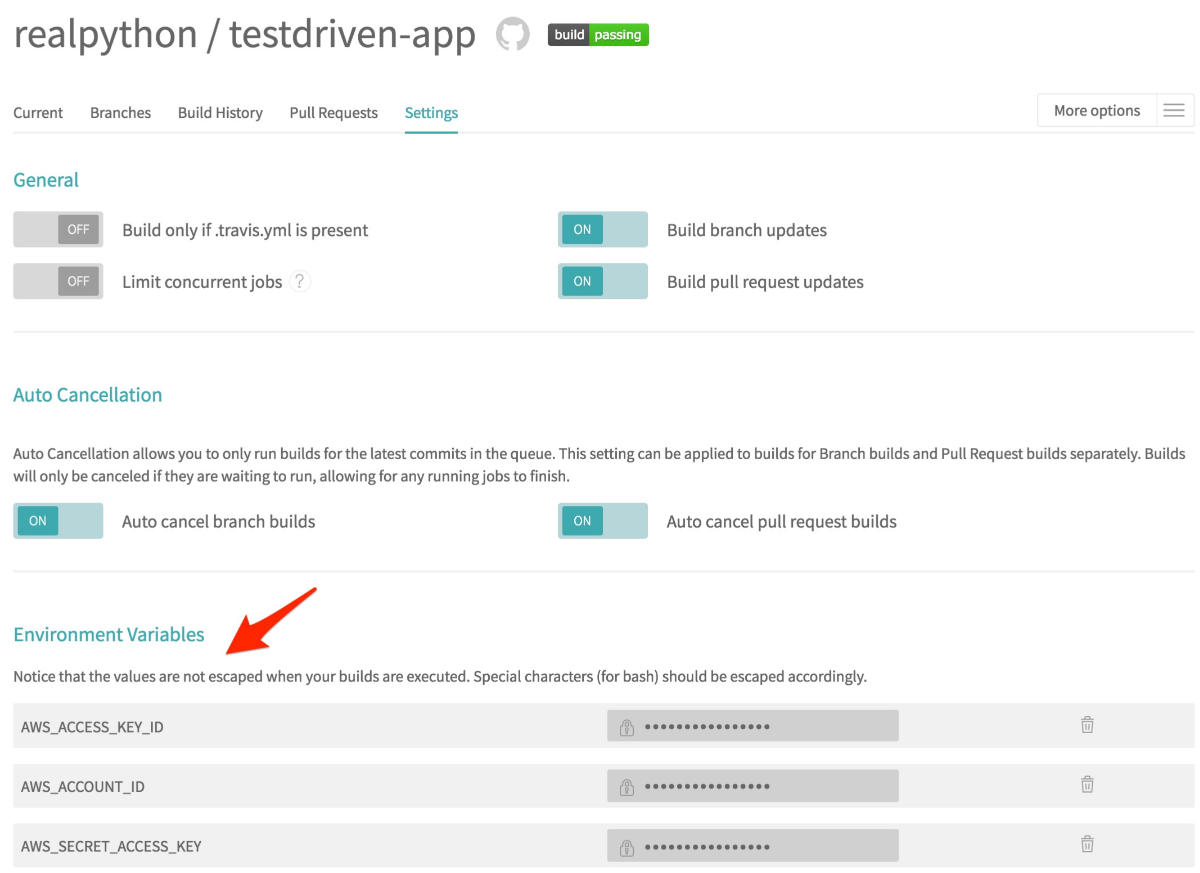
So, if the branch is `staging` or `production` and it's not a pull request, we download the AWS CLI, log in to AWS, and then set the appropriate `TAG` and `REPO` .

Grab your AWS credentials from the *~/.aws/credentials* file:

```
$ cat ~/.aws/credentials
```

Set them as environment variables within the Repository Settings of your *testdriven-app* on Travis:

1. AWS_ACCOUNT_ID - YOUR_ACCOUNT_ID
2. AWS_ACCESS_KEY_ID - YOUR_ACCCES_KEY_ID
3. AWS_SECRET_ACCESS_KEY - YOUR_SECRET_ACCESS_KEY




Update *.travis.yml*, adding in the necessary environment variables and an `after_success` step:

```
language: node_js
node_js: '9'

before_install:
  - stty cols 80

dist: trusty
sudo: required

addons:
  apt:
    sources:
     - google-chrome
    packages:
     - google-chrome-stable

services:
  - docker

env:
  global:
    - DOCKER_COMPOSE_VERSION=1.18.0
    - COMMIT=${TRAVIS_COMMIT::8}
    - MAIN_REPO=https://github.com/testdrivenio/testdriven-app-2.2.git
```

```
    - USERS=test-driven-users
    - USERS_REPO=${MAIN_REPO}#${TRAVIS_BRANCH}:services/users
    - USERS_DB=test-driven-users_db
    - USERS_DB_REPO=${MAIN_REPO}#${TRAVIS_BRANCH}:services/users/project/db
    - CLIENT=test-driven-client
    - CLIENT_REPO=${MAIN_REPO}#${TRAVIS_BRANCH}:services/client
    - SWAGGER=test-driven-swagger
    - SWAGGER_REPO=${MAIN_REPO}#${TRAVIS_BRANCH}:services/swagger

before_install:
  - sudo rm /usr/local/bin/docker-compose
  - curl -L https://github.com/docker/compose/releases/download/${DOCKER_COMPOSE_VE
RSION}/docker-compose-`uname -s`-`uname -m` > docker-compose
  - chmod +x docker-compose
  - sudo mv docker-compose /usr/local/bin

before_script:
  - export TEST_URL=http://127.0.0.1
  - export REACT_APP_USERS_SERVICE_URL=http://127.0.0.1
  - export SECRET_KEY=my_precious
  - export DISPLAY=:99.0
  - sh -e /etc/init.d/xvfb start
  - sleep 3
  - docker-compose -f docker-compose-prod.yml up --build -d

script:
  - bash test.sh stage

after_script:
  - docker-compose -f docker-compose-prod.yml down

after_success:
  - bash ./docker-push.sh
```

Did you notice the `COMMIT` variable?

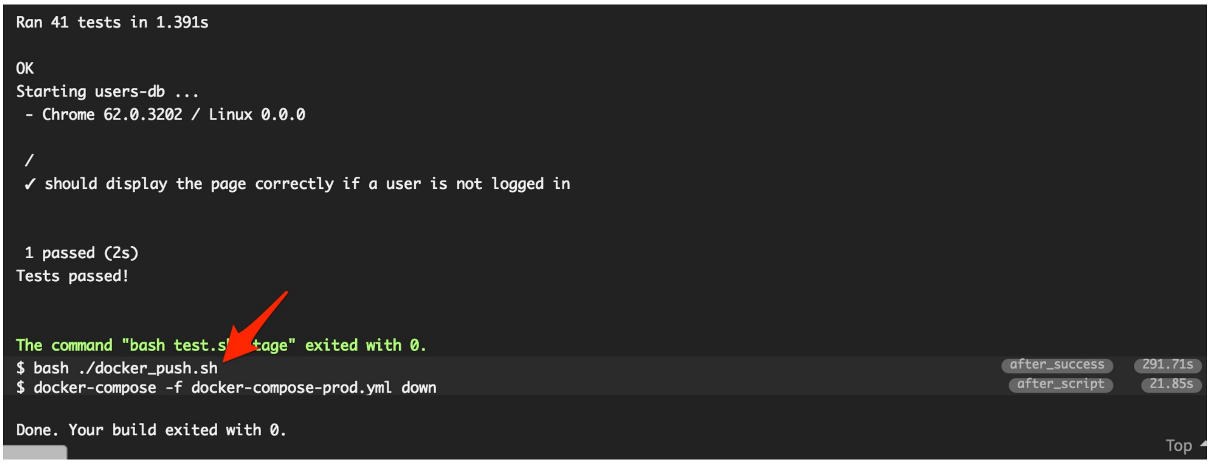
```
COMMIT=${TRAVIS_COMMIT::8}
```

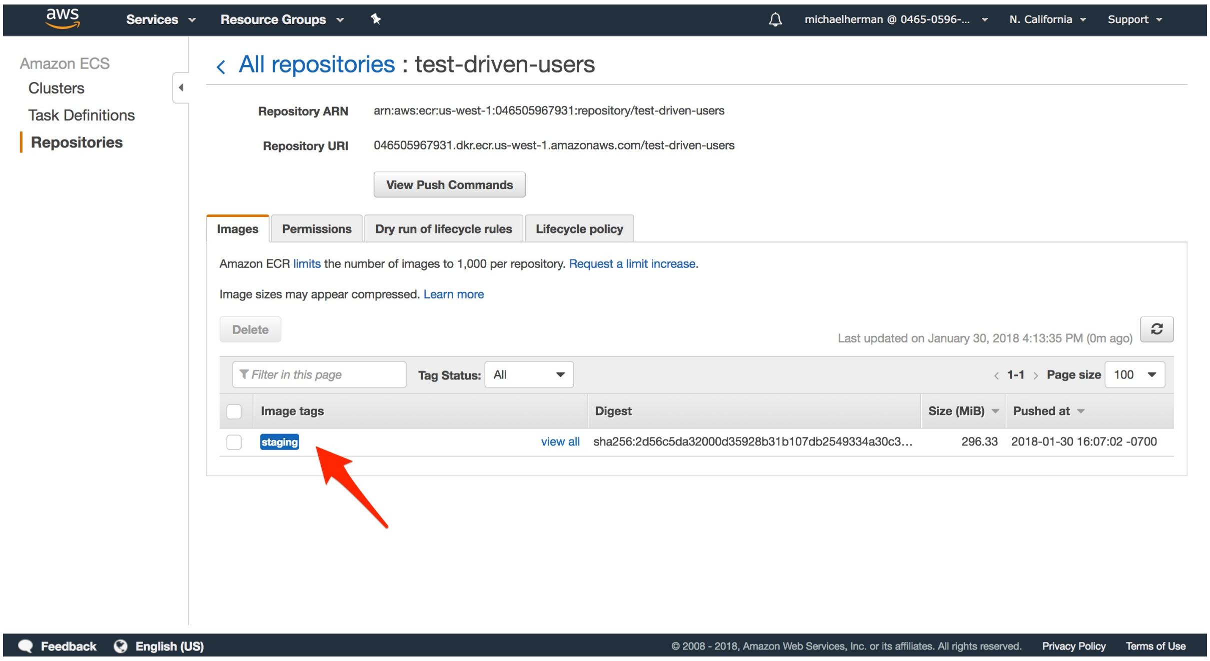
This sets a new environment variable, which contains the first 8 characters of the git commit hash. We not only have a unique name with the image, we can now tie it back to a commit in case we need to troubleshoot the code in the image.

Let's test this out. Create a `staging` branch, commit your code, and then push it to GitHub.

```
$ git checkout -b staging
$ git add -A
$ git commit -m "added ecr into the ci process (part 5)"
$ git push origin staging
```

If all goes well, the build should pass and a new image should be added to each of the repositories.




> We're not currently handling errors in the *docker-push.sh* script. How would you add some sort of error handling so that the build fails if any of the commands in that script fail? Try this on your own.

## Build

Regardless of whether the build is for `staging` or `production`, we're using *docker-compose-prod.yml* in *.travis.yml*:

```
docker-compose -f docker-compose-prod.yml up --build -d
```

To make this dynamic, let's update *.travis.yml* to dynamically set the `DOCKER_ENV` environment variable based on the `TRAVIS_BRANCH`:

```
language: node_js
```

```
node_js: '9'

before_install:
  - stty cols 80

dist: trusty
sudo: required

addons:
  apt:
    sources:
     - google-chrome
    packages:
     - google-chrome-stable

services:
  - docker

env:
  global:
    - DOCKER_COMPOSE_VERSION=1.18.0
    - COMMIT=${TRAVIS_COMMIT::8}
    - MAIN_REPO=https://github.com/testdrivenio/testdriven-app-2.2.git
    - USERS=test-driven-users
    - USERS_REPO=${MAIN_REPO}#${TRAVIS_BRANCH}:services/users
    - USERS_DB=test-driven-users_db
    - USERS_DB_REPO=${MAIN_REPO}#${TRAVIS_BRANCH}:services/users/project/db
    - CLIENT=test-driven-client
    - CLIENT_REPO=${MAIN_REPO}#${TRAVIS_BRANCH}:services/client
    - SWAGGER=test-driven-swagger
    - SWAGGER_REPO=${MAIN_REPO}#${TRAVIS_BRANCH}:services/swagger

before_install:
  - sudo rm /usr/local/bin/docker-compose
  - curl -L https://github.com/docker/compose/releases/download/${DOCKER_COMPOSE_VE
RSION}/docker-compose-`uname -s`-`uname -m` > docker-compose
  - chmod +x docker-compose
  - sudo mv docker-compose /usr/local/bin

before_script:
  - export TEST_URL=http://127.0.0.1
  - export REACT_APP_USERS_SERVICE_URL=http://127.0.0.1
  - export SECRET_KEY=my_precious
  - export DISPLAY=:99.0
  - export DOCKER_ENV=$(if [ "$TRAVIS_BRANCH" == "production" ]; then echo "prod";
else echo "stage"; fi)
  - sh -e /etc/init.d/xvfb start
  - sleep 3
  - docker-compose -f docker-compose-$DOCKER_ENV.yml up --build -d

script:
```

```
  - bash test.sh $DOCKER_ENV

after_script:
  - docker-compose -f docker-compose-$DOCKER_ENV.yml down

after_success:
  - bash ./docker-push.sh
```

Then, update *docker-push.sh*:

```
#!/bin/sh

if [ -z "$TRAVIS_PULL_REQUEST" ] || [ "$TRAVIS_PULL_REQUEST" == "false" ]
then

  if [ "$TRAVIS_BRANCH" == "staging" ] || \
     [ "$TRAVIS_BRANCH" == "production" ]
  then
    curl "https://s3.amazonaws.com/aws-cli/awscli-bundle.zip" -o "awscli-bundle.zip"

    unzip awscli-bundle.zip
    ./awscli-bundle/install -b ~/bin/aws
    export PATH=~/bin:$PATH
    # add AWS_ACCOUNT_ID, AWS_ACCESS_KEY_ID, AWS_SECRET_ACCESS_KEY env vars
    eval $(aws ecr get-login --region us-west-1 --no-include-email)
    export TAG=$TRAVIS_BRANCH
    export REPO=$AWS_ACCOUNT_ID.dkr.ecr.us-west-1.amazonaws.com
  fi

  if [ "$TRAVIS_BRANCH" == "staging" ] || \
     [ "$TRAVIS_BRANCH" == "production" ]
  then
    # users
    docker build $USERS_REPO -t $USERS:$COMMIT -f Dockerfile-$DOCKER_ENV
    docker tag $USERS:$COMMIT $REPO/$USERS:$TAG
    docker push $REPO/$USERS:$TAG
    # users db
    docker build $USERS_DB_REPO -t $USERS_DB:$COMMIT -f Dockerfile
    docker tag $USERS_DB:$COMMIT $REPO/$USERS_DB:$TAG
    docker push $REPO/$USERS_DB:$TAG
    # client
    docker build $CLIENT_REPO -t $CLIENT:$COMMIT -f Dockerfile-$DOCKER_ENV --build-
arg REACT_APP_USERS_SERVICE_URL=TBD
    docker tag $CLIENT:$COMMIT $REPO/$CLIENT:$TAG
    docker push $REPO/$CLIENT:$TAG
    # swagger
    docker build $SWAGGER_REPO -t $SWAGGER:$COMMIT -f Dockerfile-$DOCKER_ENV $SWAGG
ER_DIR
    docker tag $SWAGGER:$COMMIT $REPO/$SWAGGER:$TAG
    docker push $REPO/$SWAGGER:$TAG
```

```
  fi
fi
```

## Sanity Check

Commit your code.

Assuming you're on the `staging` branch, create two new branches, `development` and `production` :

```
$ git branch development
$ git checkout development
$ git push origin development

$ git branch production
$ git checkout production
$ git push origin production
```

Then, merge `staging` into `master` :

```
$ git checkout master
$ git merge staging
```

Now, test out the following workflow:

**Development**

1. Create a new feature branch from the `master` branch, make an arbitrary change, commit and push it up to GitHub:

   ```
   $ git checkout -b feature-branch
   $ git push origin feature-branch
   ```

2. After the build passes, open a PR against the `development` branch to trigger a new build on Travis

3. Merge the PR after the build passes

**Staging**

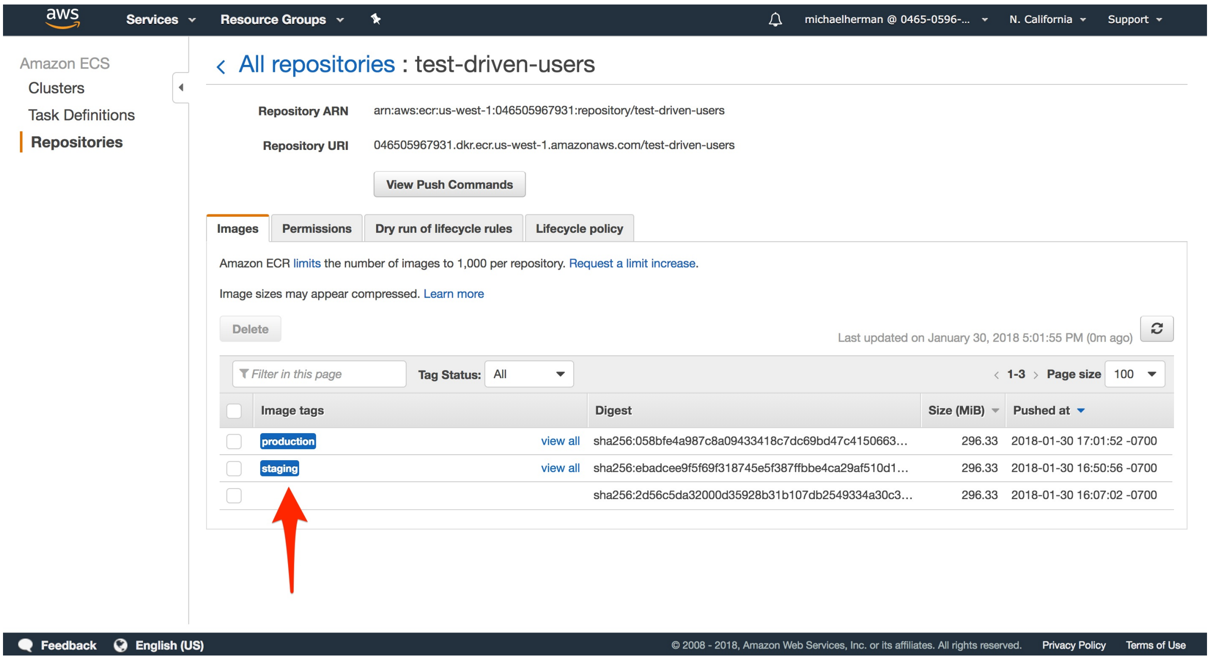
1. Open PR from the `development` branch against the `staging` branch to trigger a new build on Travis

2. Merge the PR after the build passes to trigger a new build

3. After the build passes, images are created, tagged `staging` , and pushed to ECR

### Production

1. Open PR from the `staging` branch against the `production` branch to trigger a new build on Travis
2. Merge the PR after the build passes to trigger a new build
3. After the build passes, images are created, tagged `production` , and pushed to ECR
4. Merge the changes into the `master` branch




## Docker Cache

With ECR configured, you can speed up builds by first pulling from ECR. Try this on your own.

# Elastic Load Balancer

In the lesson, we'll add load balancing via Elastic Load Balancing to distribute traffic and create a more reliable app with automatic scaling and failover...

---

The Elastic Load Balancer distributes incoming application traffic and scales resources as needed to meet traffic needs.

A load balancer is one of (if not) the most important parts of your application since it needs to always be up, routing traffic to healthy back-ends, and ready to scale at a moment's notice.

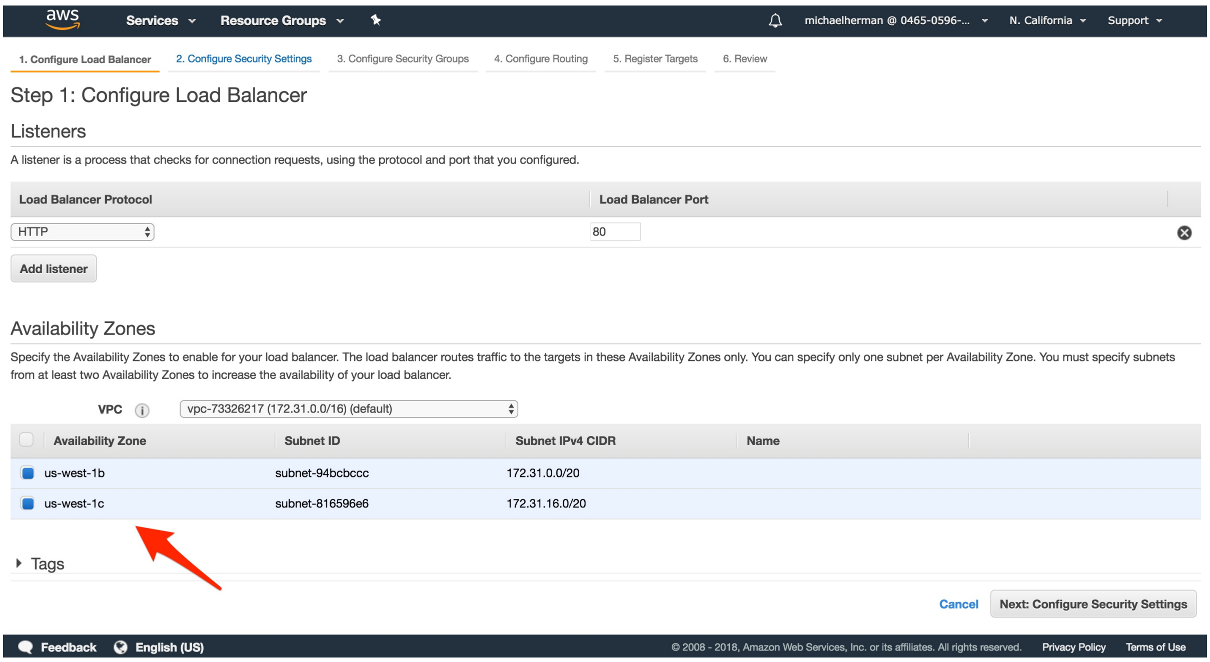
There are currently three types of Elastic Load Balancers to choose from. We'll be using the *Application Load Balancer* since it provides support for path-based routing and dynamic port-mapping and it also enables zero-downtime deployments and support for A/B testing. The Application Load Balancer is one of those AWS services that makes ECS so powerful. In fact, before it's release, ECS was not a viable orchestration solution.

## Configure ALB

Navigate to Amazon EC2, click "Load Balancers" on the sidebar, and then click the "Create Load Balancer" button. Select the "Create" button under "Application Load Balancer".

### Step 1: Configure Load Balancer

1. "Name": `testdriven-staging-alb`
2. "VPC": Select the default VPC to keep things simple
3. "Availability Zones": Select at least two available subnets

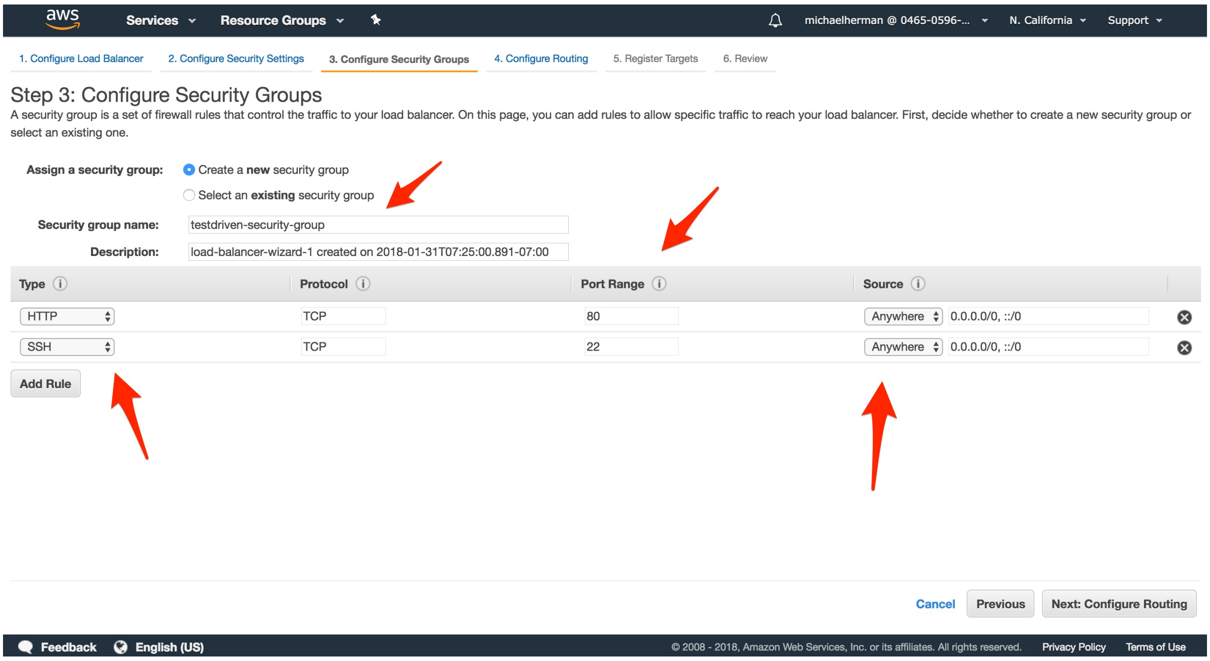


### Step 2: Configure Security Settings

Skip this for now.

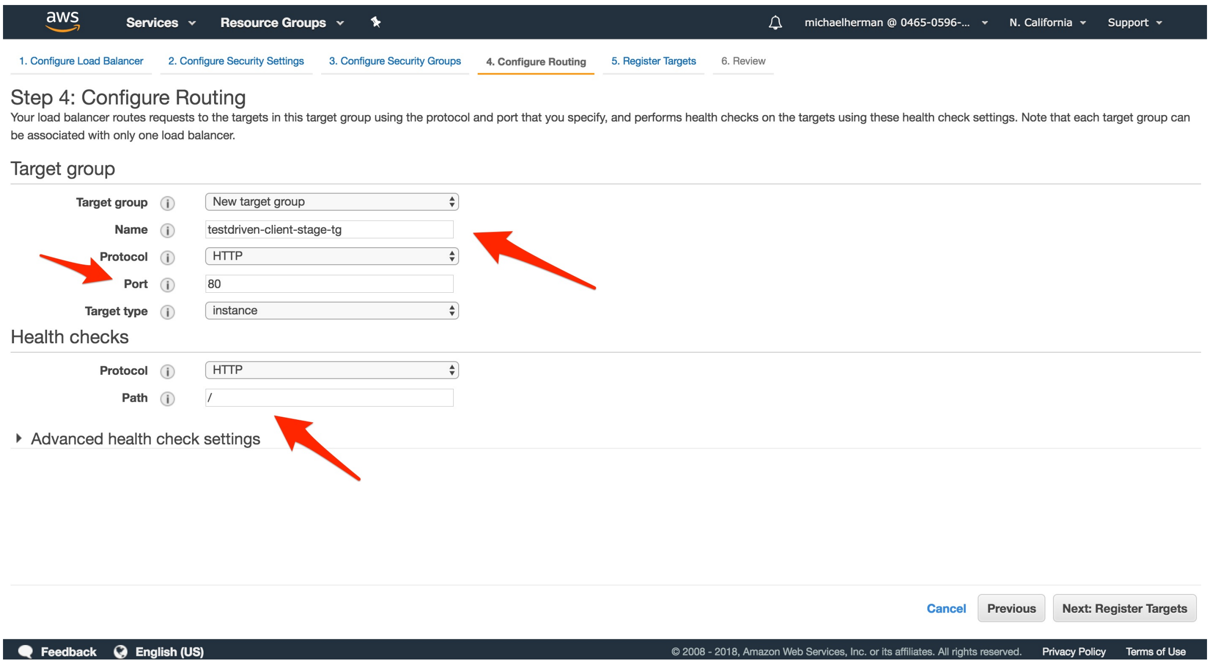
### Step 3: Configure Security Groups

Select an existing Security Group or create a new Security Group called `testdriven-security-group`, making sure at least HTTP 80 and SSH 22 are open.




### Step 4: Configure Routing

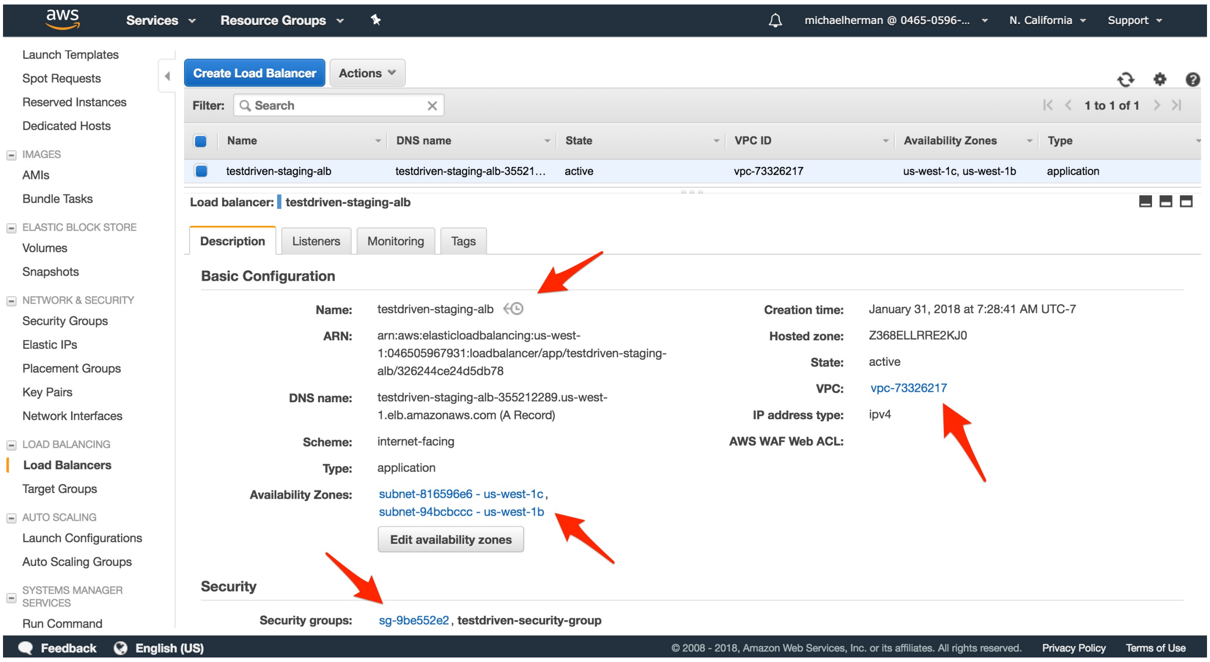
1. "Name": `testdriven-client-stage-tg`
2. "Port": `80`
3. "Path": `/`



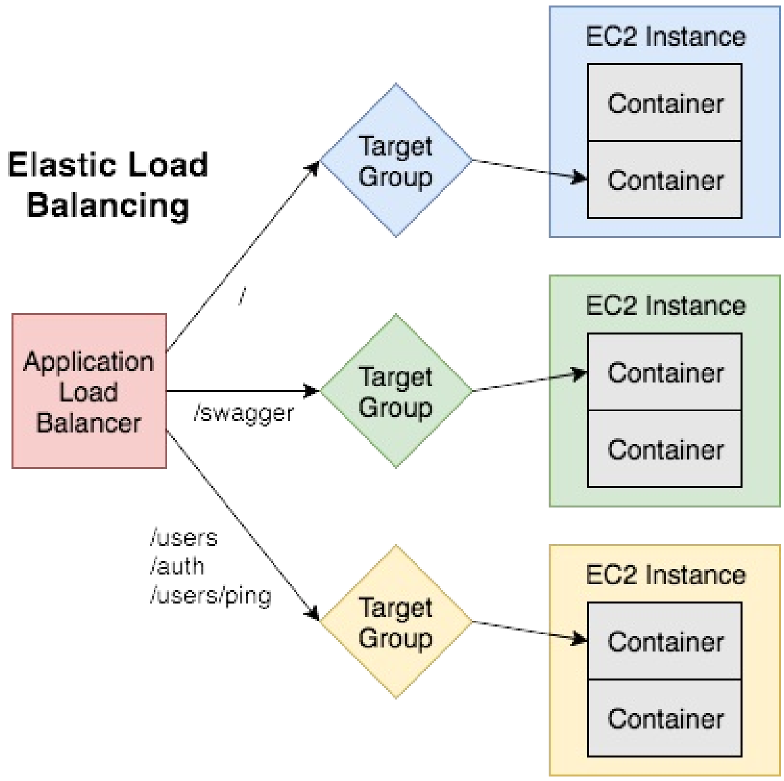
### Step 5: Register Targets

Do not assign any instances manually since this will be managed by ECS. Review and then create the new load balancer.

Once created, take note of the new Security Group:




With that, we also need to set up Target Groups and Listeners:




## Target Groups

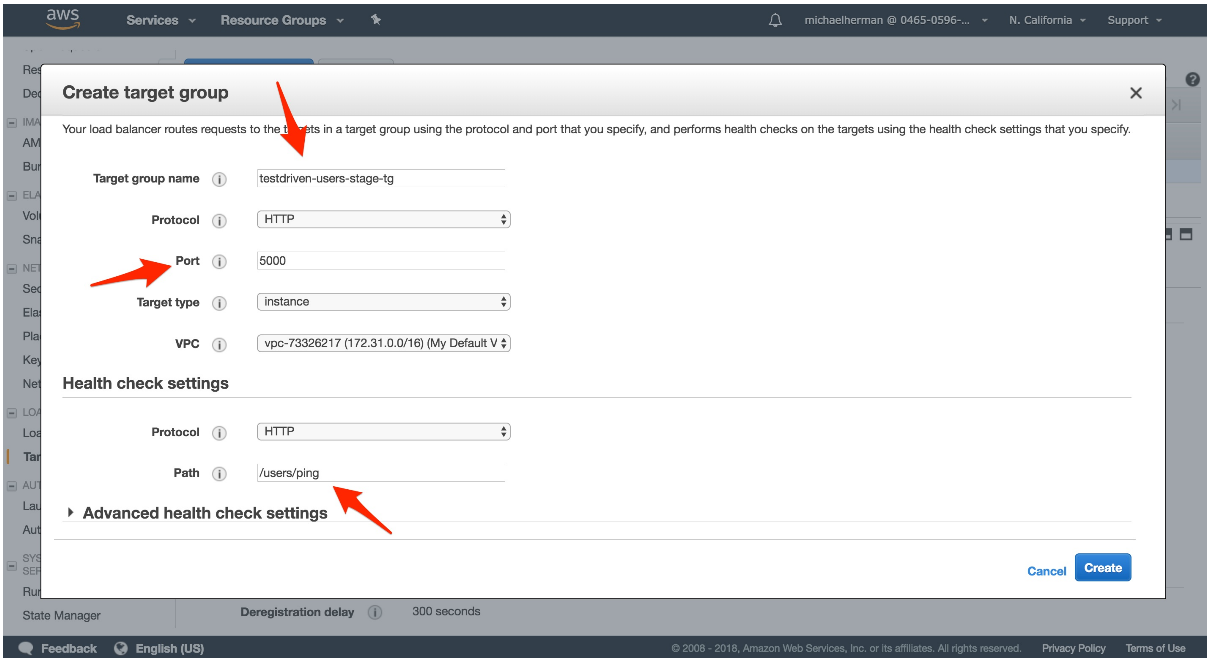
Target Groups are attached to the Application Load Balancer and are used to route traffic to the containers found in the ECS Service.

You may have already noticed, but a Target Group called `testdriven-client-stage-tg` was already created (which we'll use for the `client` app) when we set up the Application Load Balancer, so we just need to set up two more.

Within the EC2 Dashboard, click "Target Groups", and then create the following Target Groups...

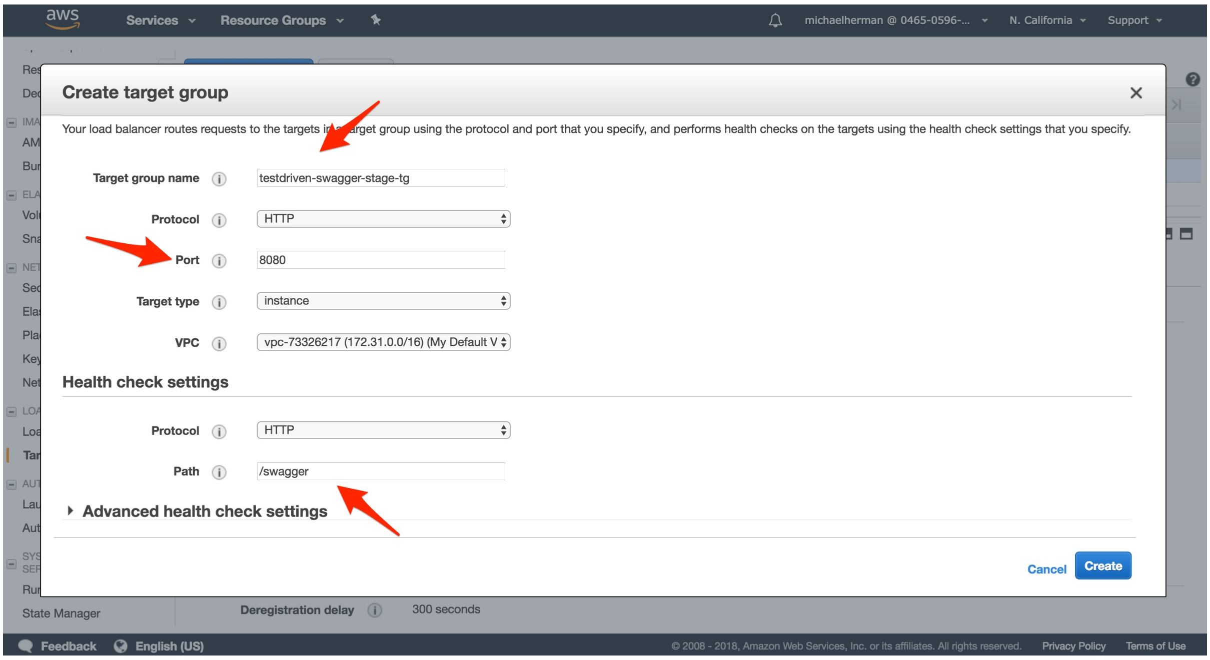
#### Target Group 1: `users`

1. "Target group name": `testdriven-users-stage-tg`
2. "Port": `5000`
3. Then, under "Health check settings" set the "Path" to `/users/ping` .

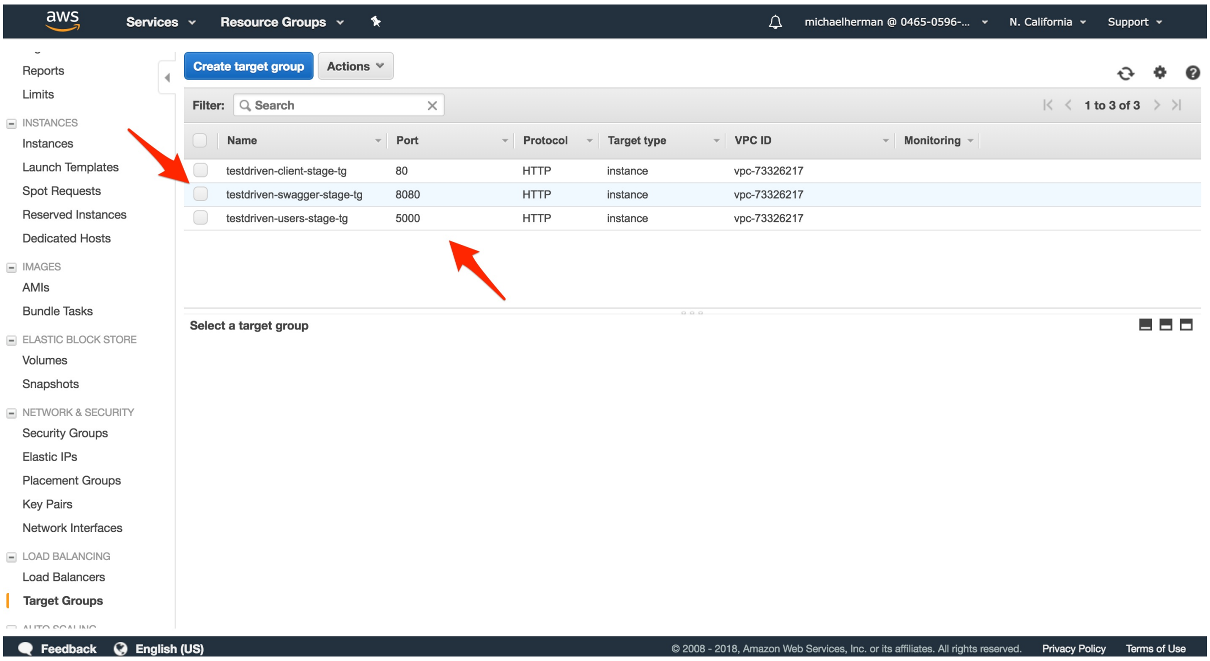



#### Target Group 2: `swagger`

1. "Target group name": `testdriven-swagger-stage-tg`
2. "Port": `8080`
3. Then, under "Health check settings" set the "Path" to `/swagger` .




You should now have the following Target Groups:



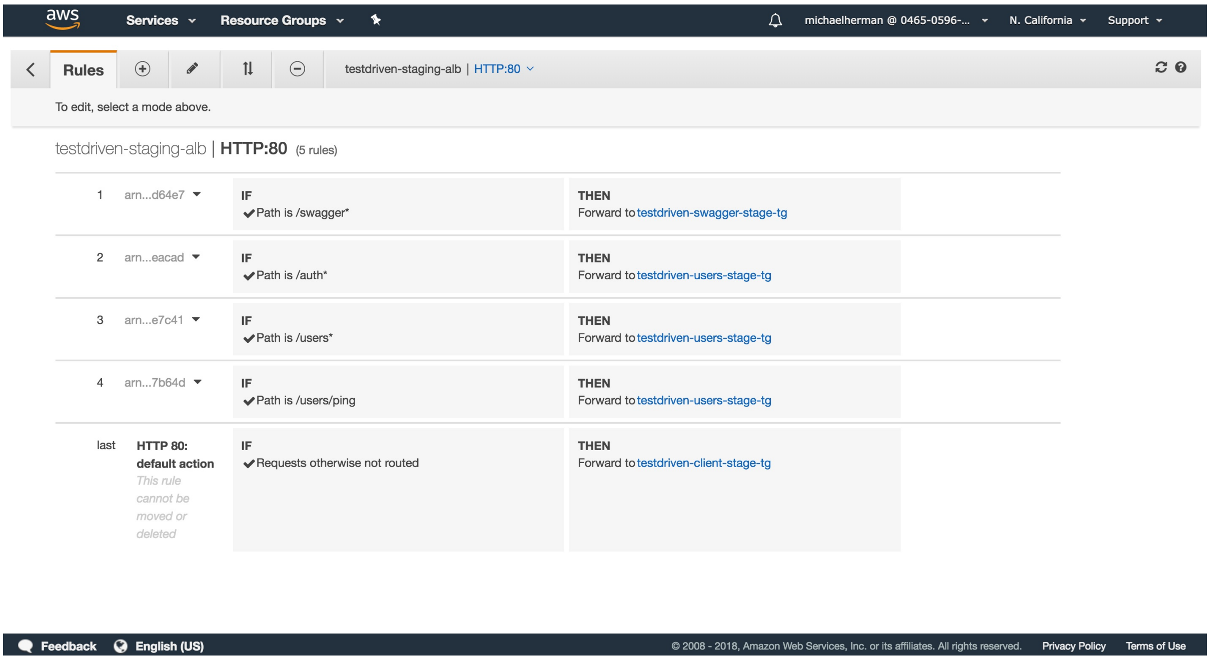

## Listeners

Back on the "Load Balancers" page within the EC2 Dashboard, select the `testdriven-staging-alb` Load Balancer, and then click the "Listeners" tab. Here, we can add Listeners to the load balancer, which are then forwarded to a specific Target Group.

There should already be a listener for "HTTP : 80". Click the "View/edit rules >" link, and then insert four new rules:

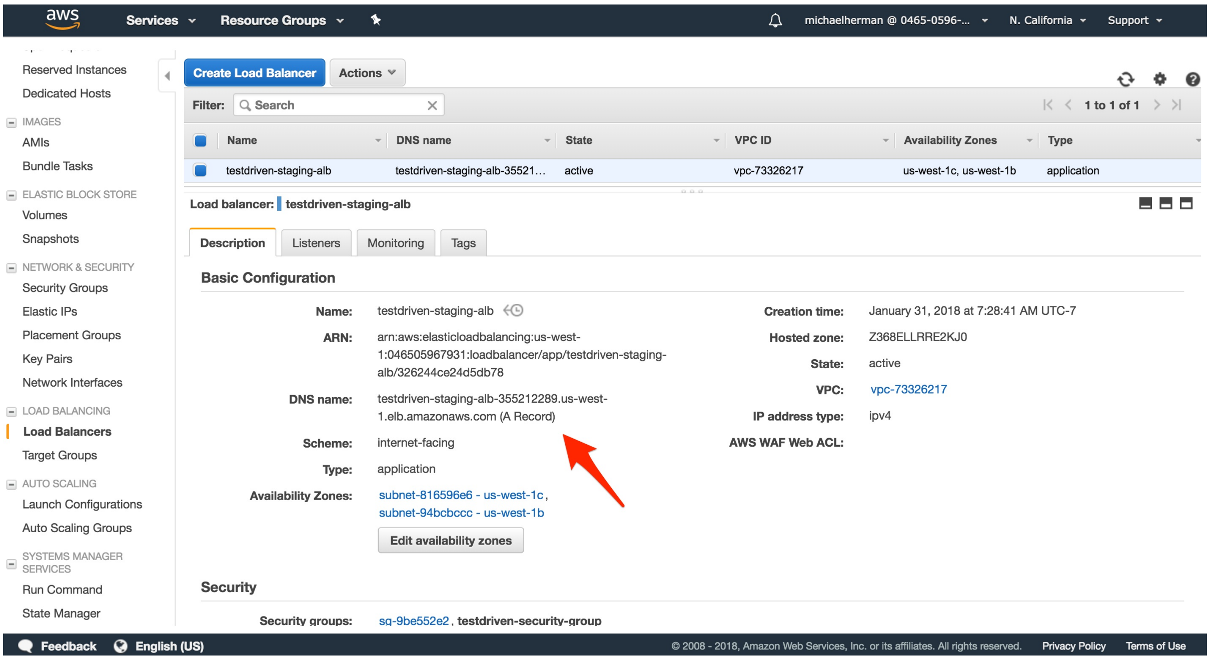
1. If `/swagger*` , Then `testdriven-swagger-stage-tg`
2. If `/auth*` , Then `testdriven-users-stage-tg`

3. If `/users*` , Then `testdriven-users-stage-tg`
4. If `/users/ping` , Then `testdriven-users-stage-tg`




## Update Travis

Finally, navigate back to the Load Balancer and grab the "DNS name" from the "Description" tab:




We need to set this as the value of `REACT_APP_USERS_SERVICE_URL` in *docker-push.sh*:

```
if [ "$TRAVIS_BRANCH" == "staging" ]
then
  export REACT_APP_USERS_SERVICE_URL="http://LOAD_BALANCER_STAGE_DNS_NAME"
fi
```

We'll use this value for the build-time arg for the client service:

```
docker build $CLIENT_REPO -t $CLIENT:$COMMIT -f Dockerfile-$DOCKER_ENV --build-arg
REACT_APP_USERS_SERVICE_URL=$REACT_APP_USERS_SERVICE_URL
```

Did you notice that we are setting the value of `REACT_APP_USERS_SERVICE_URL` in the `before_script` step of *.travis.yml*:

```
export REACT_APP_USERS_SERVICE_URL=http://127.0.0.1
```

This value needs to be initially set since we build the images for our tests from the *docker-compose-stage.yml* file. We then need to update the value for the building of the images before the push to ECR.

Updated script:

```
#!/bin/sh

if [ -z "$TRAVIS_PULL_REQUEST" ] || [ "$TRAVIS_PULL_REQUEST" == "false" ]
then

  if [ "$TRAVIS_BRANCH" == "staging" ] || \
     [ "$TRAVIS_BRANCH" == "production" ]
  then
    curl "https://s3.amazonaws.com/aws-cli/awscli-bundle.zip" -o "awscli-bundle.zip"

    unzip awscli-bundle.zip
    ./awscli-bundle/install -b ~/bin/aws
    export PATH=~/bin:$PATH
    # add AWS_ACCOUNT_ID, AWS_ACCESS_KEY_ID, AWS_SECRET_ACCESS_KEY env vars
    eval $(aws ecr get-login --region us-west-1 --no-include-email)
    export TAG=$TRAVIS_BRANCH
    export REPO=$AWS_ACCOUNT_ID.dkr.ecr.us-west-1.amazonaws.com
  fi

  if [ "$TRAVIS_BRANCH" == "staging" ]
  then
    export REACT_APP_USERS_SERVICE_URL="http://LOAD_BALANCER_STAGE_DNS_NAME"
  fi

  if [ "$TRAVIS_BRANCH" == "staging" ] || \
     [ "$TRAVIS_BRANCH" == "production" ]
  then
    # users
    docker build $USERS_REPO -t $USERS:$COMMIT -f Dockerfile-$DOCKER_ENV
    docker tag $USERS:$COMMIT $REPO/$USERS:$TAG
    docker push $REPO/$USERS:$TAG
```

```
    # users db
    docker build $USERS_DB_REPO -t $USERS_DB:$COMMIT -f Dockerfile
    docker tag $USERS_DB:$COMMIT $REPO/$USERS_DB:$TAG
    docker push $REPO/$USERS_DB:$TAG
    # client
    docker build $CLIENT_REPO -t $CLIENT:$COMMIT -f Dockerfile-$DOCKER_ENV --build-
arg REACT_APP_USERS_SERVICE_URL=$REACT_APP_USERS_SERVICE_URL
    docker tag $CLIENT:$COMMIT $REPO/$CLIENT:$TAG
    docker push $REPO/$CLIENT:$TAG
    # swagger
    docker build $SWAGGER_REPO -t $SWAGGER:$COMMIT -f Dockerfile-$DOCKER_ENV $SWAGG
ER_DIR
    docker tag $SWAGGER:$COMMIT $REPO/$SWAGGER:$TAG
    docker push $REPO/$SWAGGER:$TAG
  fi
fi
```

## Update ECR

Assuming you're on the `staging` branch, update the Swagger spec:

```
$ python services/swagger/update-spec.py http://LOAD_BALANCER_STAGE_DNS_NAME
```

Commit and push your code to trigger a new Travis build. Make sure new images are added to ECR once the build is done.

---

With that, we can turn our attention to ECS...

# Elastic Container Service

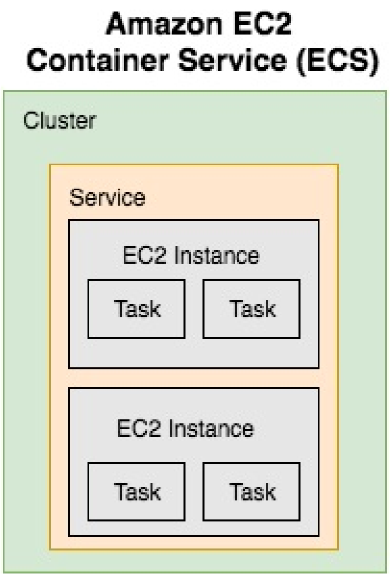
Let's configure a Task Definition along with a Cluster and a Service within Elastic Container Service (ECS)...

---

ECS is a container orchestration system used for managing and deploying Docker-based containers. It has four main components:

1. Task Definitions
2. Tasks
3. Services
4. Clusters

In short, Task Definitions are used to spin up Tasks that get assigned to a Service, which is then assigned to a Cluster.




## Task Definition

Task Definitions define which containers make up the overall app and how much resources are allocated to each container. You can think of them as blueprints, similar to a Docker Compose file.

Navigate to Amazon ECS, click "Task Definitions", and then click the button "Create new Task Definition".

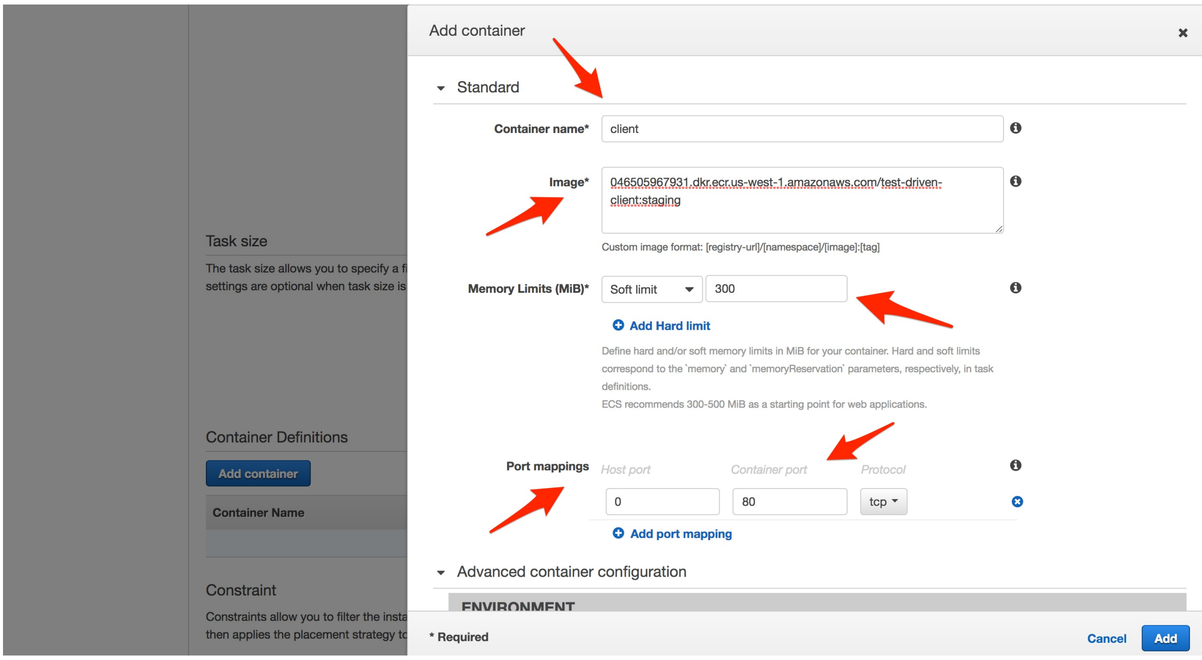
#### Target Definition 1: `client`

First, Update the "Task Definition Name" to `testdriven-client-stage-td` and then add a new container:

1. "Container name": `client`

2. "Image": `YOUR_AWS_ACCOUNT_ID.dkr.ecr.us-west-1.amazonaws.com/test-driven-client:staging`
3. "Memory Limits (MB)": `300` soft limit
4. "Port mappings": `0` host, `80` container

> We set the host port for the service to `0` so that a port is dynamically assigned when the Task is spun up.

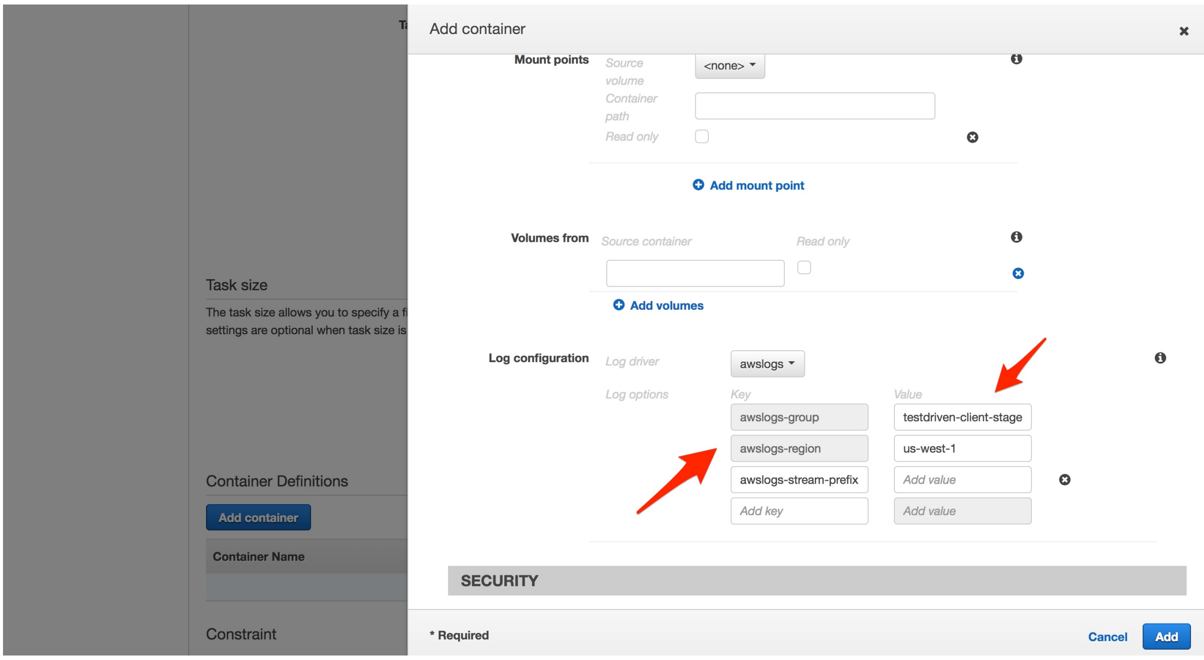



It's important to note that you will not need to add the `REACT_APP_USERS_SERVICE_URL` environment variable in the container definition. This variable is required at the build-time, not the run-time, and is added during the building of the image on Travis:

```
docker build $CLIENT_REPO -t $CLIENT:$COMMIT -f Dockerfile-$DOCKER_ENV --build-arg
REACT_APP_USERS_SERVICE_URL=$REACT_APP_USERS_SERVICE_URL
```

It's a good idea to configure logs, via LogConfiguration, to pipe logs to CloudWatch.

To set up, we need to create a new Log Group. Simply navigate to CloudWatch, click "Logs" on the navigation pane, click the "Actions" drop-down button, and then select "Create log group". Name the group `testdriven-client-stage`.




**Target Definition 2:** `users`

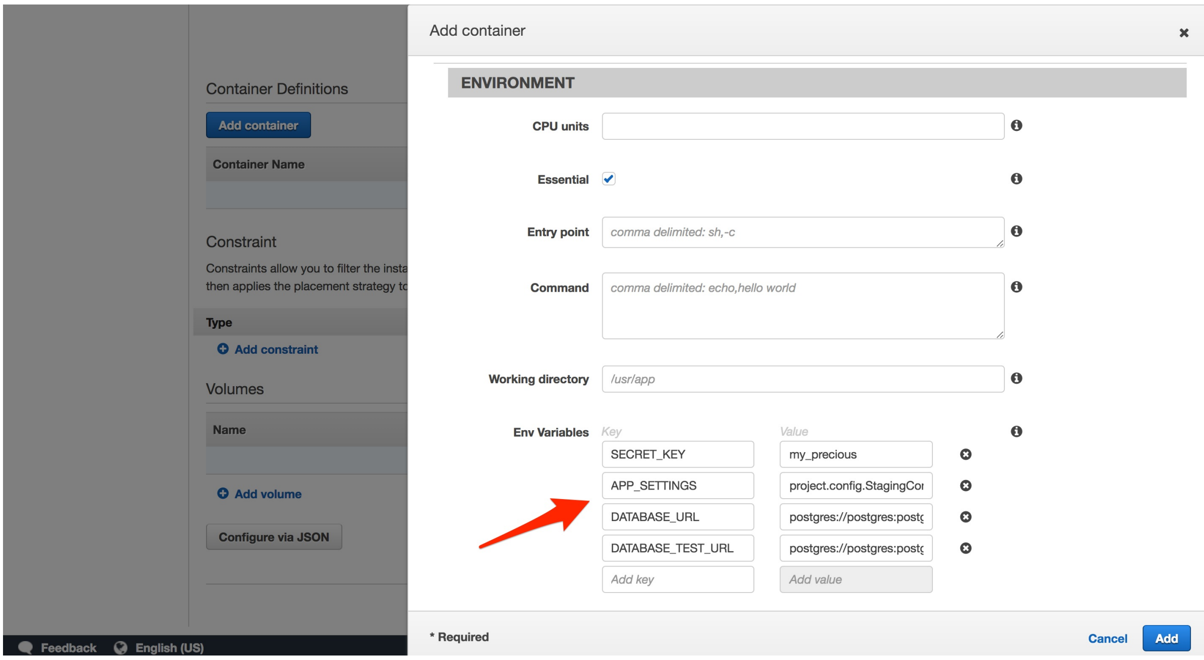
Set up a single Task Definition for the `users` service with the name `testdriven-users-stage-td`. Then, add two containers:

*Container 1:*

1. "Container name": `users`
2. "Image": `YOUR_AWS_ACCOUNT_ID.dkr.ecr.us-west-1.amazonaws.com/test-driven-users:staging`
3. "Memory Limits (MB)": `300` soft limit
4. "Port mappings": `0` host, `5000` container
5. "Links": `users-db`
6. "Log configuration": `testdriven-users-stage`

Also, add the following environment variables:

1. `SECRET_KEY` - `my_precious`
2. `APP_SETTINGS` - `project.config.StagingConfig`
3. `DATABASE_URL` - `postgres://postgres:postgres@users-db:5432/users_stage`
4. `DATABASE_TEST_URL` - `postgres://postgres:postgres@users-db:5432/users_test`




*Container 2:*

1. "Container name": `users-db`
2. "Image": `YOUR_AWS_CCOUNT_ID.dkr.ecr.us-west-1.amazonaws.com/test-driven-users_db:staging`
3. "Memory Limits (MB)": `300` soft limit
4. "Port mappings": `5432` container
5. "Env Variables":
   i. `POSTGRES_USER` - `postgres`
   ii. `POSTGRES_PASSWORD` - `postgres`
6. "Log configuration": `testdriven-users_db-stage`

**Target Definition 3: `swagger`**

Add a final Task Definition for the `swagger` service with the name `testdriven-swagger-stage-td` .

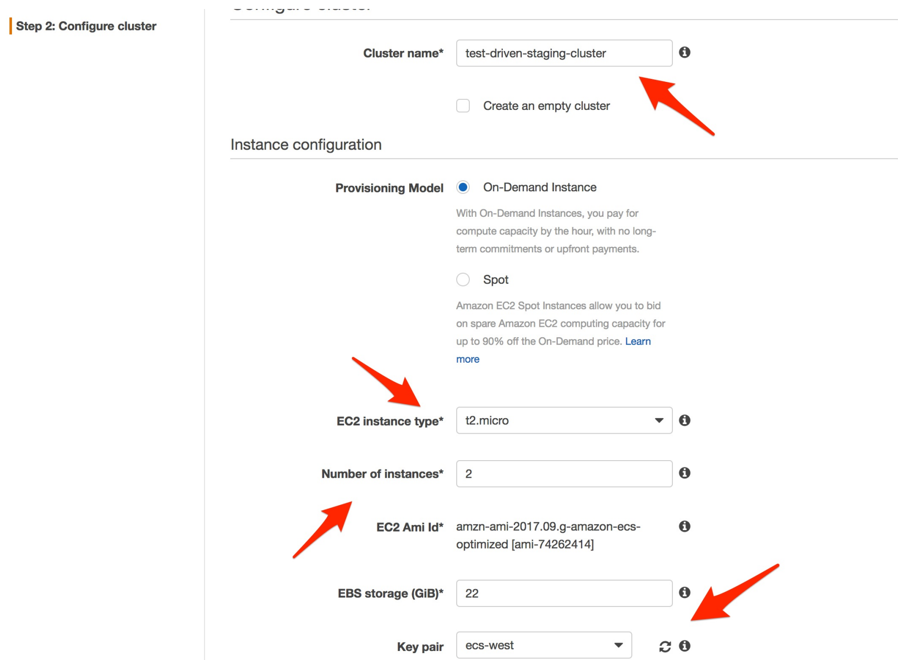
1. "Container name": `swagger`
2. "Image": `YOUR_AWS_ACCOUNT_ID.dkr.ecr.us-west-1.amazonaws.com/test-driven-swagger:staging`
3. "Memory Limits (MB)": `300` soft limit
4. "Port mappings": `0` host, `8080` container
5. "Env Variables":
   i. `URL` - `swagger.json`
6. "Log configuration": `testdriven-swagger-stage`

## Cluster

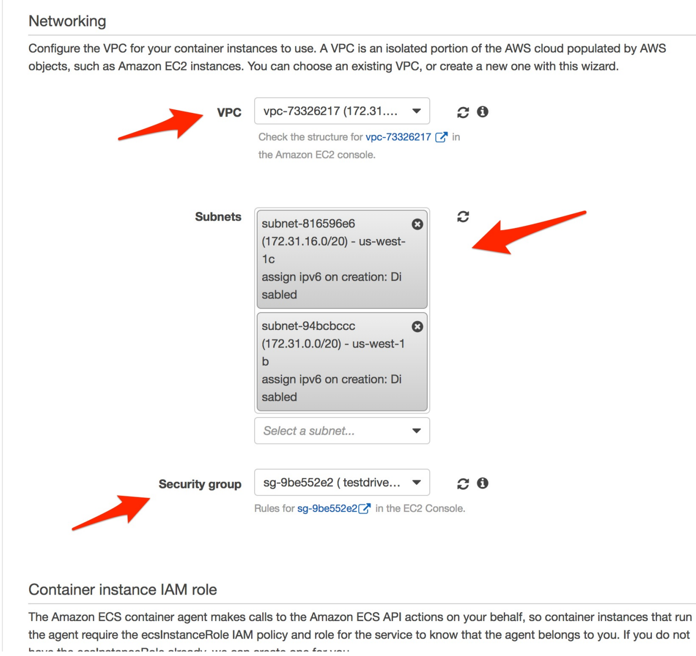
Clusters are where the actual containers run. They are just groups of EC2 instances that run Docker containers managed by ECS. To create a Cluster, click "Clusters" on the ECS Console sidebar, and then click the "Create Cluster" button. Select "EC2 Linux + Networking".

Add:

1. "Cluster name": `test-driven-staging-cluster`
2. "EC2 instance type": `t2.micro`
3. "Number of instances": `2`
4. "Key pair": Select an existing Key Pair or create a new one (see below for details on how to create a new Key Pair)




Select the default VPC and the previously created Security Group along with the appropriate subnets.




It will take a few minutes to setup the EC2 resources.

## Key Pair

To create a new EC2 Key Pair, so we can SSH into the EC2 instances managed by ECS, navigate to Amazon EC2, click "Key Pairs" on the sidebar, and then click the "Create Key Pair" button.

Name the the new key pair `ecs` and add it to "~/.ssh".

## Service

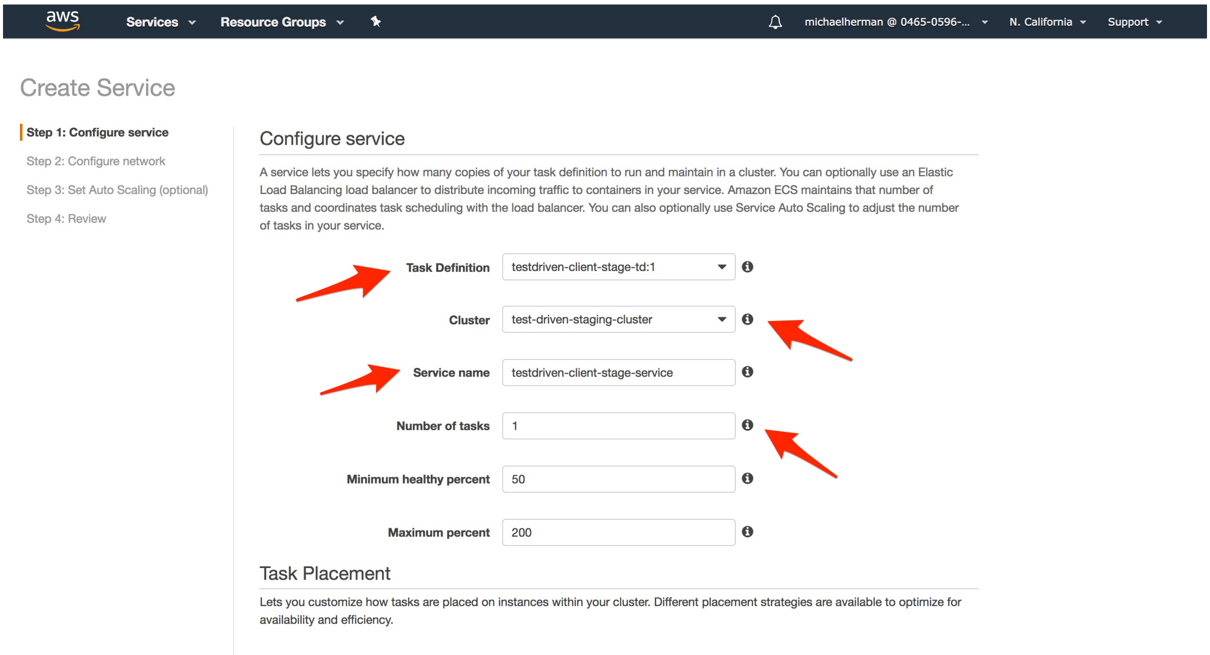
Services instantiate the containers from the Task Definitions and run them on EC2 boxes within the ECS Cluster. Such instances are called Tasks. To define a Service, on the "Services" tab within the newly created Cluster, click "Create".

Create the following Services...

### Client

*Configure service*:

1. "Task Definition": `testdriven-client-stage-td:LATEST_REVISION_NUMBER`
2. "Service name": `testdriven-client-stage-service`
3. "Number of tasks": `1`



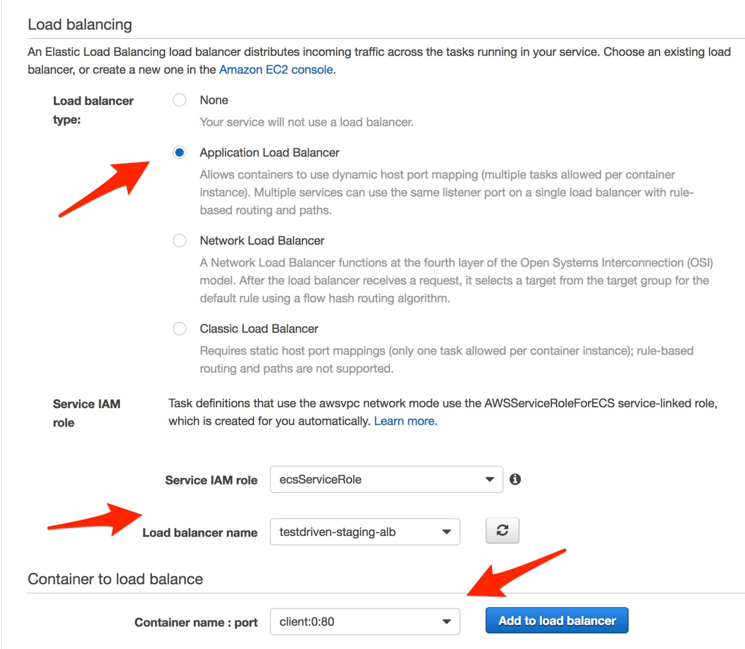

You can configure how and where new Tasks are placed in a Cluster via "Task Placement" Strategies. We will use the basic "AZ Balanced Spread" in this course, which spreads Tasks evenly across Availability Zones (AZ), and, then within each AZ, Tasks are spread evenly among Instances. For more, review [Amazon ECS Task Placement Strategies](#)

Click "Next".

*Configure network*:

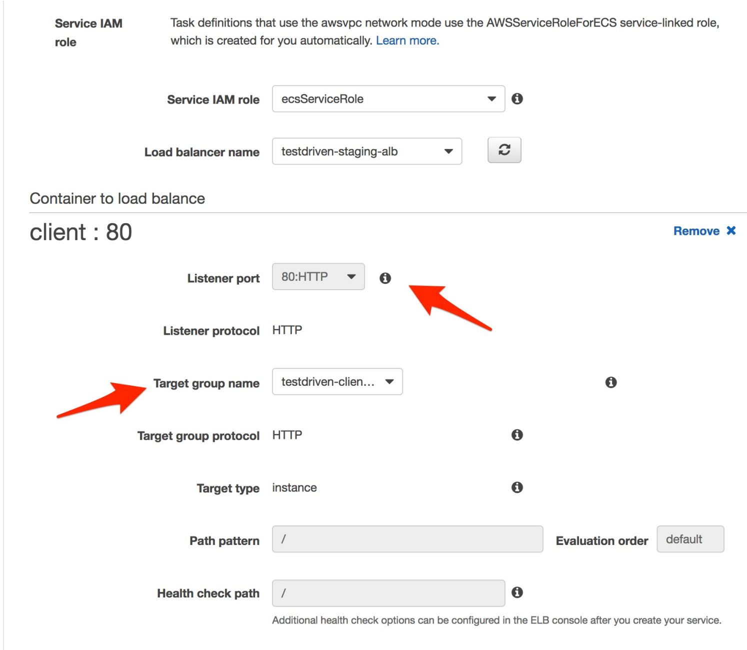
Select the "Application Load Balancer" under "Load balancer type".

1. "Load balancer name": `testdriven-staging-alb`
2. "Container name : port": `client:0:80`



Click "Add to load balancer".

1. "Listener port": `80:HTTP`
2. "Target group name": `testdriven-client-stage-tg`




Click the next button a few times, and then "Create Service".

#### Users

*Configure service*:

1. "Task Definition": `testdriven-users-stage-td:LATEST_REVISION_NUMBER`
2. "Service name": `testdriven-users-stage-service`
3. "Number of tasks": `1`

Click "Next".

*Configure network*:

Select the "Application Load Balancer" under "Load balancer type".

1. "Load balancer name": `testdriven-staging-alb`
2. "Container name : port": `users-service:0:5000`

Click "Add to load balancer".

1. "Listener port": `80:HTTP`
2. "Target group name": `testdriven-users-stage-tg`

Click the next button a few times, and then "Create Service".

#### Swagger

*Configure service*:

1. "Task Definition": `testdriven-swagger-stage-td:LATEST_REVISION_NUMBER`
2. "Service name": `testdriven-swagger-stage-service`
3. "Number of tasks": `1`

Click "Next".

*Configure network*:

Select the "Application Load Balancer" under "Load balancer type".

1. "Load balancer name": `testdriven-staging-alb`
2. "Container name : port": `swagger:0:8080`

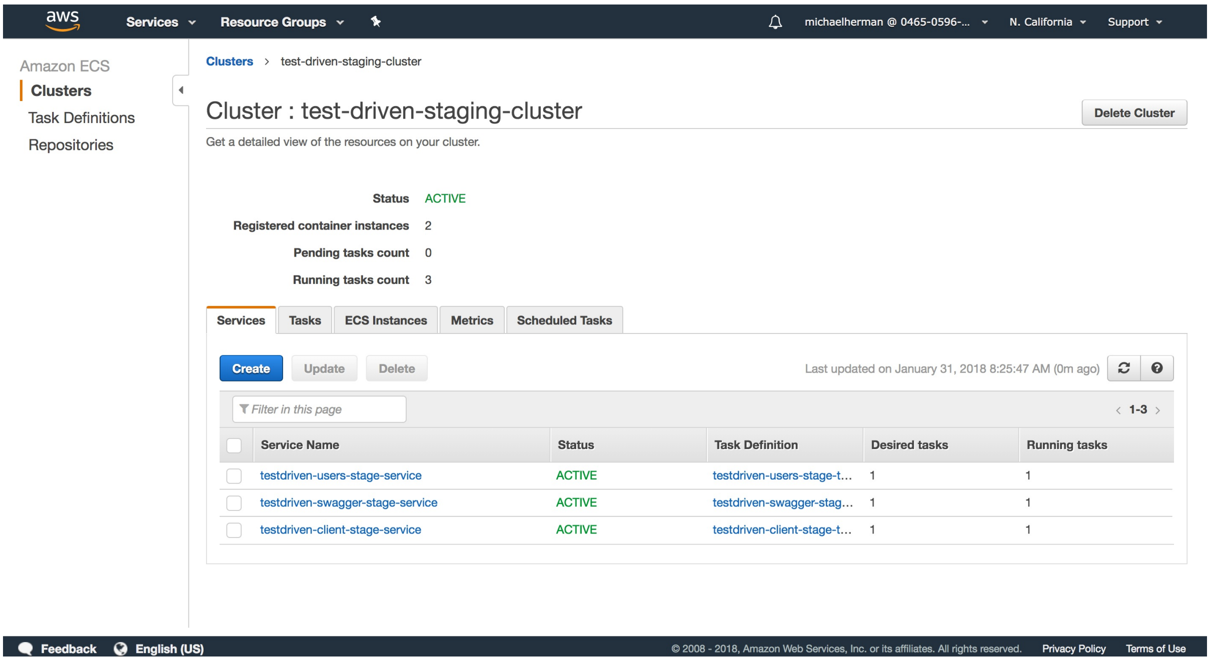
Click "Add to load balancer".

1. "Listener port": `80:HTTP`
2. "Target group name": `testdriven-swagger-stage-tg`

Click the next button a few times, and then "Create Service".

---

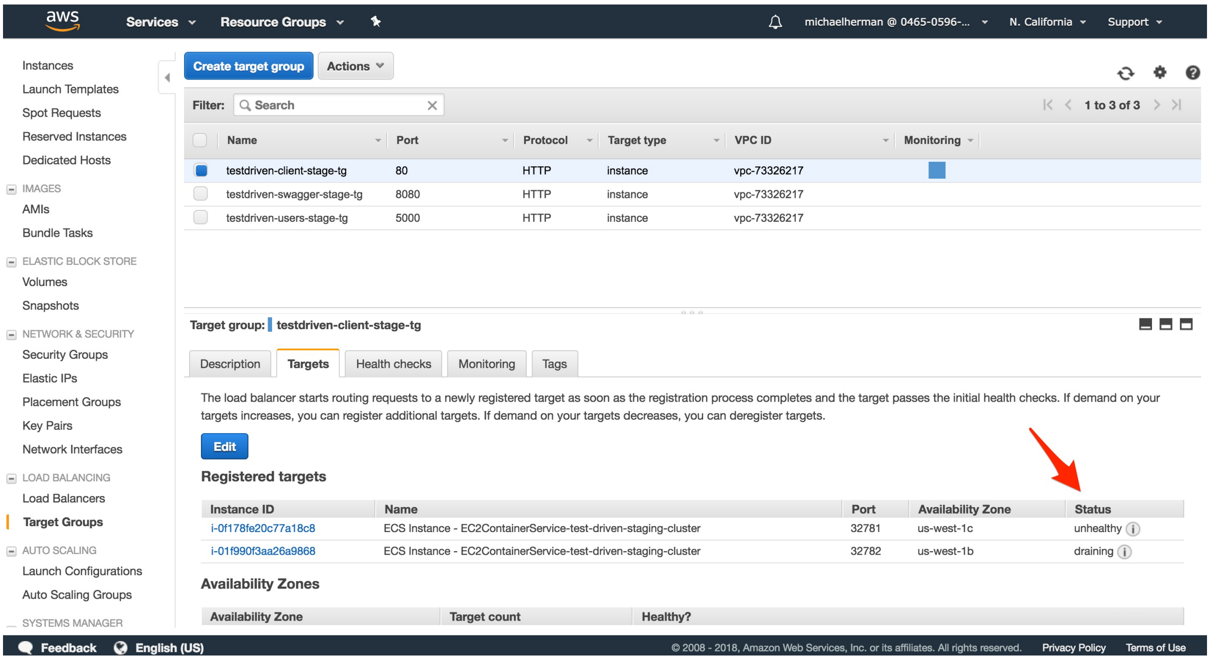
You should now have the following Services running, each with a single Task:




## Sanity Check

Navigate to the EC2 Dashboard, and click "Target Groups".

Make sure `testdriven-client-stage-tg`, `testdriven-users-stage-tg`, and `testdriven-swagger-stage-tg` have a single registered instance each. Each of the instances should be *unhealthy* because they failed their respective health checks.



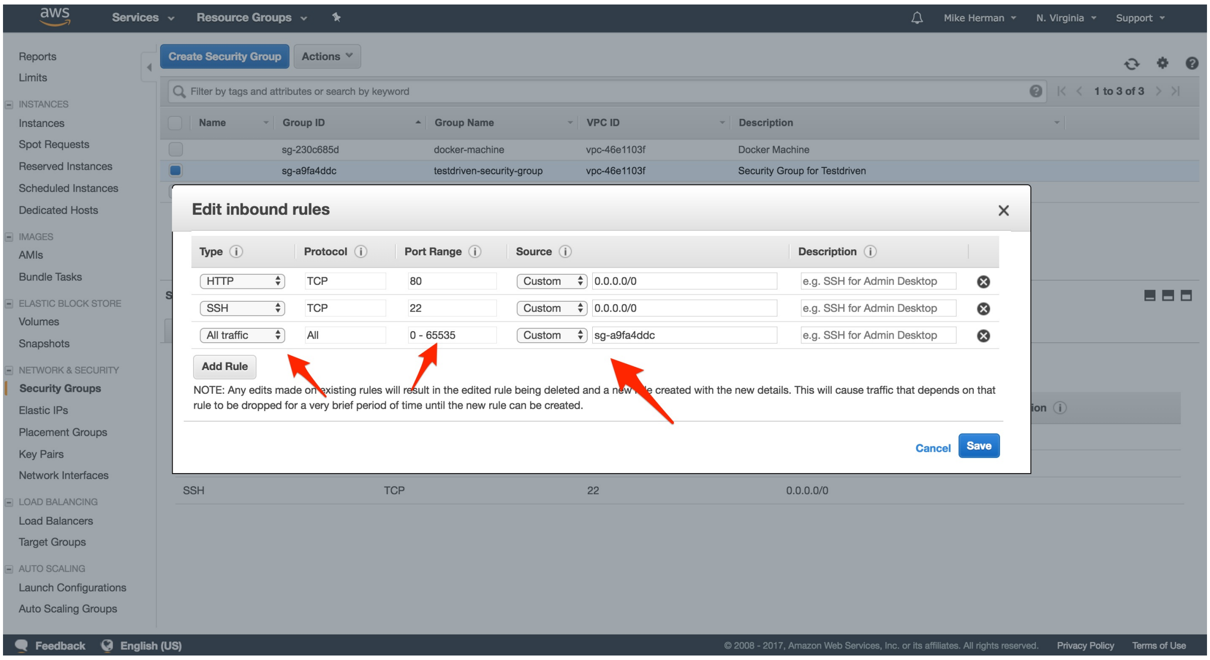

To get them to pass the health checks, we need to add another inbound rule to the Security Group associated with the containers (which we defined when we configured the Cluster), allowing traffic from the Load Balancer to reach the containers.

## Inbound Rule

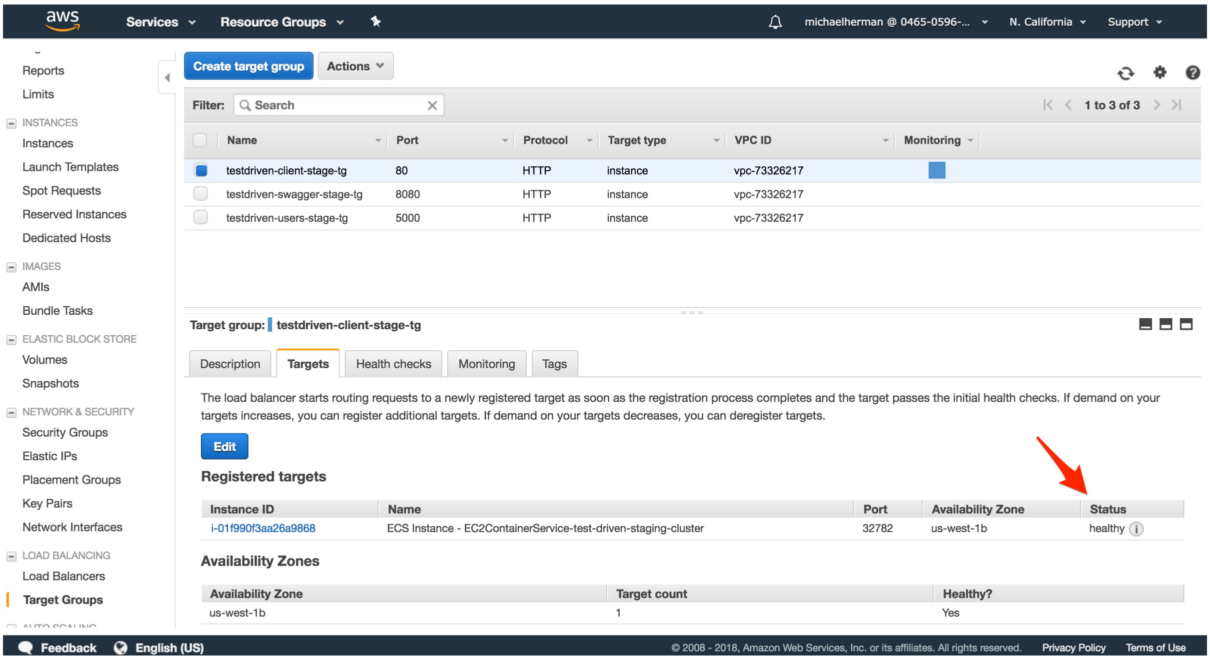
Within the EC2 Dashboard, click "Security Groups" and select the Security Group associated with the containers, which is the same group assigned to the Load Balancer. Click the "Inbound" tab and then click "Edit"

Add a new rule:

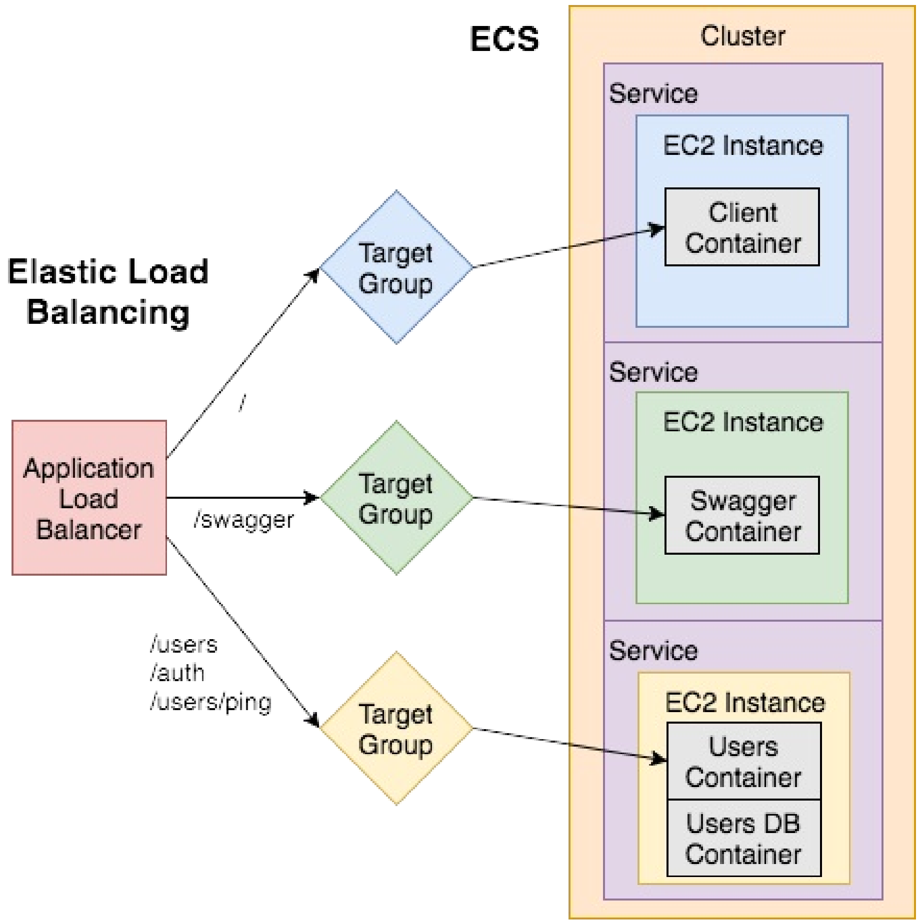
1. "Type": `All traffic`
2. "Port Range": `0 - 65535`
3. "Source": Choose `Custom`, then add the Security Group ID



Once added, the next time a container is added to each of the Target Groups, the instance should be *healthy*:



Essentially, when the Service was spun up, ECS automatically discovered and associated the new Cluster instances with the Application Load Balancer.




Next, navigate back to the Load Balancer and grab the "DNS name" from the "Description" tab, and navigate to http://LOAD_BALANCER_STAGE_DNS_NAME/users/ping in your browser.

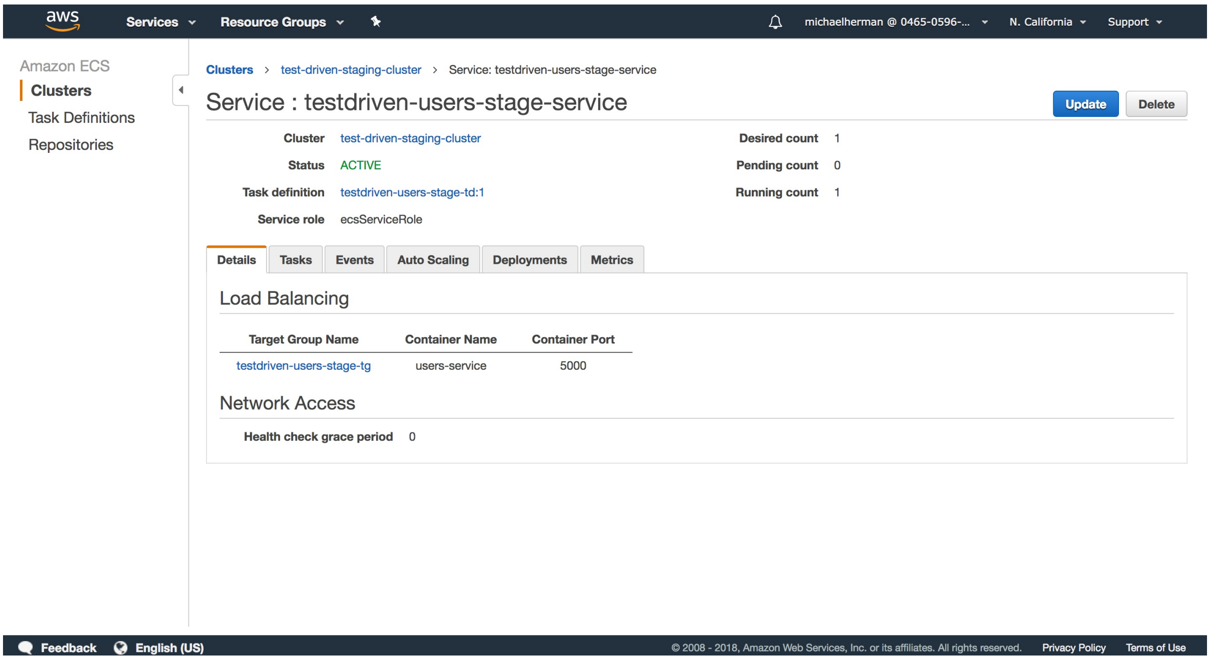
If all went well, you should see:

```
{
  "message": "pong!",
  "status": "success"
}
```

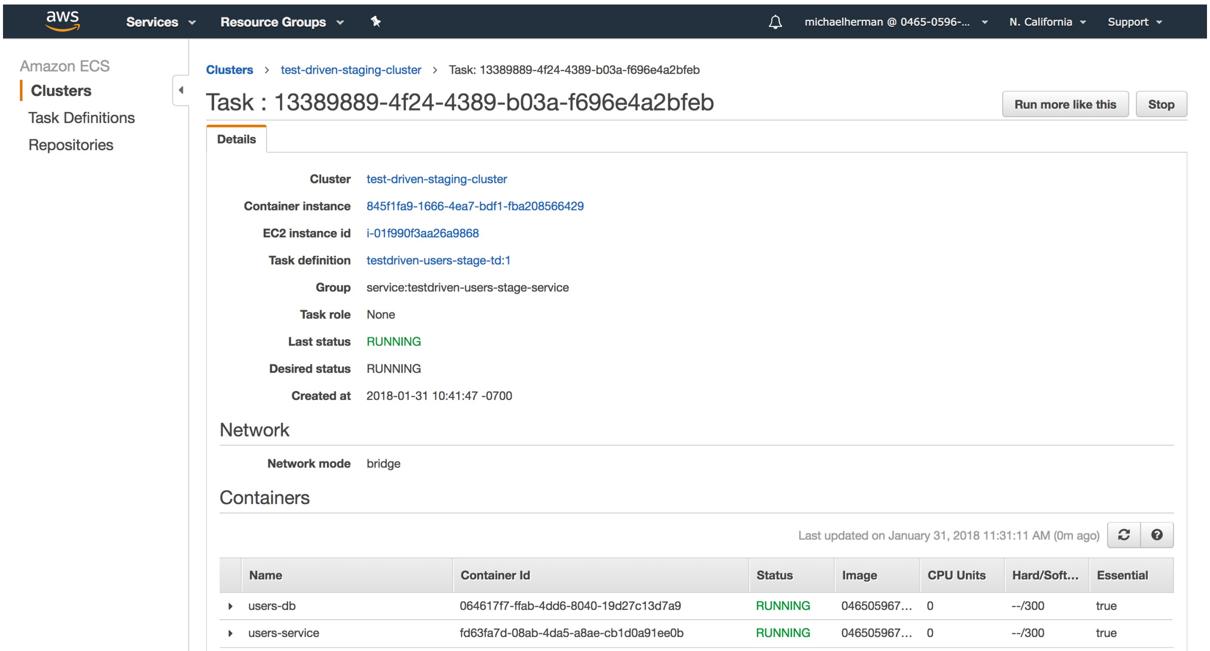
Try the `/users` endpoint at http://LOAD_BALANCER_STAGE_DNS_NAME/users. You should see a 500 error since the migrations have not been ran.

## Migrations

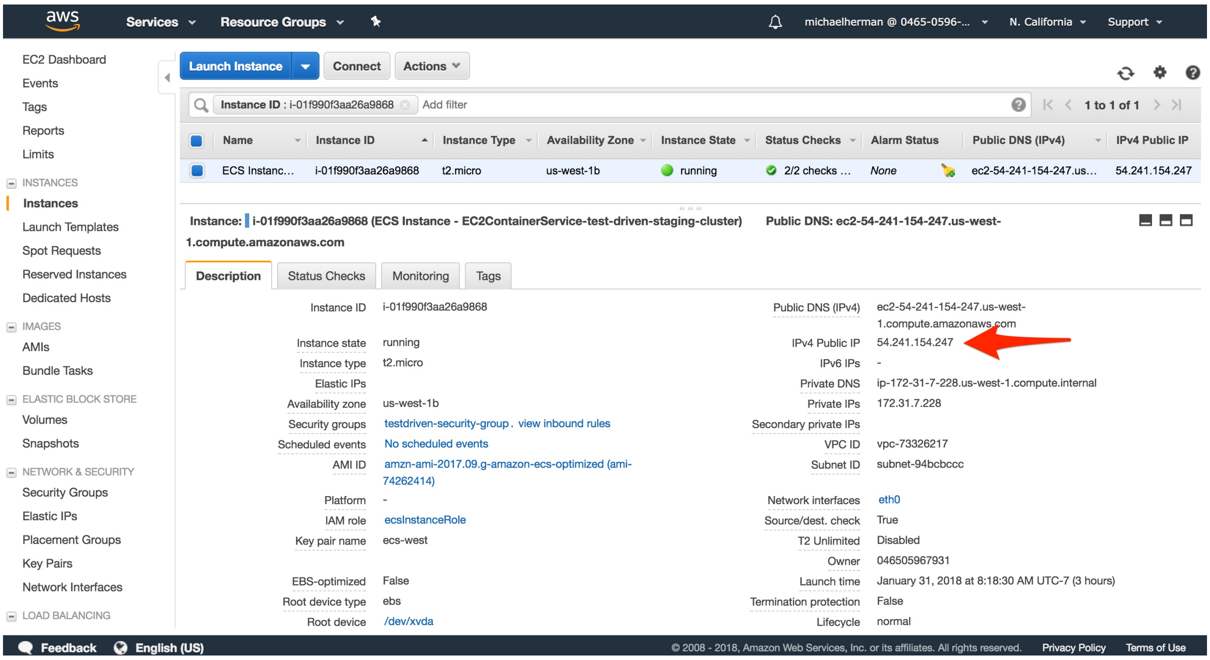
We'll need to SSH into the EC2 instance associated with the `users-db` service to apply the migrations. First, on the "Services" tab within the created Cluster, click the link for the `testdriven-users-stage-service` service.




From there, click on the "Tasks" tab and click the link for the associated Task.




Then, click the link for the EC2 Instance ID and grab the public IP:



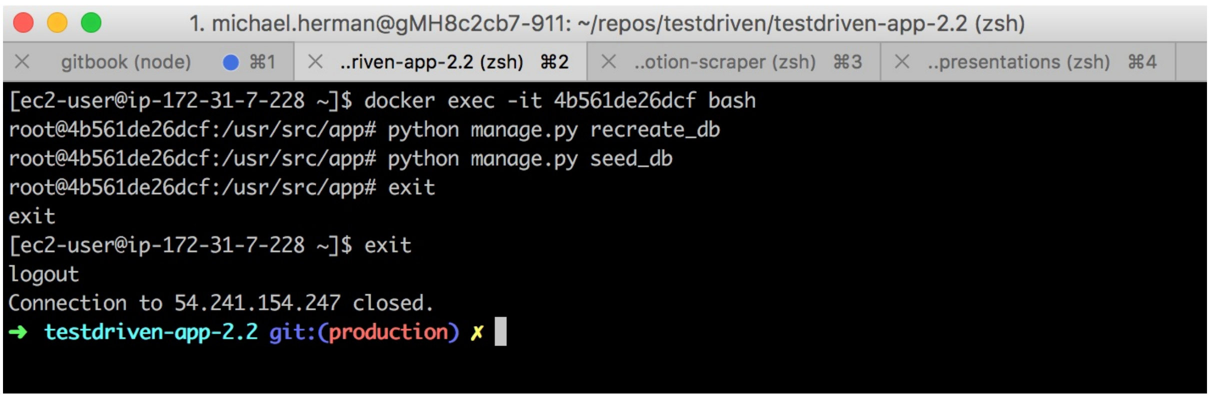

SSH into the instance:

```
$ ssh -i ~/.ssh/ecs-west.pem ec2-user@@EC2_PUBLIC_IP
```

> You may need to update the permissions on the Pem file - i.e., `chmod 400 ~/.ssh/ecs-west.pem`.

Next, grab the Container ID for `users` (via `docker ps`), enter the shell within the running container, and then update the database:

```
$ docker exec -it Container_ID bash
# python manage.py recreate_db
# python manage.py seed_db
```




Navigate to http://LOAD_BALANCER_STAGE_DNS_NAME/users again and you should see the users. Then, test the remaining GET endpoints in your browser:

1. http://LOAD_BALANCER_STAGE_DNS_NAME
2. http://LOAD_BALANCER_STAGE_DNS_NAME/swagger

## Test

Finally, let's point our local end-to-end tests at the new instance on AWS:

```
$ export TEST_URL=http://LOAD_BALANCER_STAGE_DNS_NAME
$ testcafe chrome e2e
```

They should pass:

```
/
✓ should display the page correctly if a user is not logged in

/login
✓ should display the sign in form
✓ should allow a user to sign in
✓ should throw an error if the credentials are incorrect

/register
✓ should display flash messages correctly

/register
✓ should display the registration form
✓ should allow a user to register
✓ should validate the password field
✓ should throw an error if the username is taken
✓ should throw an error if the email is taken

/status
✓ should not display user info if a user is not logged in
✓ should display user info if a user is logged in


12 passed (1m 22s)
```

# ECS Staging

In the lesson, we'll update the CI/CD workflow to add a new revision to the Task Definition and update the Service...

---

## Zero Downtime Deployments

Before jumping in, check your understanding by updating the app on your own:

1. From the `staging` branch, make a quick change to the app locally.
2. Commit and push your code to GitHub.
3. Once the Travis build passes, the new images will be built, tagged, and pushed to ECR.
4. Once done, add a new revision to the applicable Task Definitions.
5. Update the Service.

Once you update the Service, ECS will automatically pick up on these changes and instantiate the Task Definitions, creating new Tasks that will spin up on the Cluster instances.

ALB will run health checks on the new instances once they are up:

1. *Pass?* If the health checks pass, traffic is forwarded appropriately to the new Tasks while the old Tasks are spun down.
2. *Fail?* If the health checks fail, the new Tasks are spun down.

Try this again while pinging the service in a background terminal tab. Does the application go down at all. It shouldn't. Zero-downtime.

Now, let's automate that process...

## Task Definitions

Let's create JSON files for the Task Definitions in a new folder at the project root called "ecs".

1. *ecs_client_stage_taskdefinition.json*
2. *ecs_users_stage_taskdefinition.json*
3. *ecs_swagger_stage_taskdefinition.json*

#### Client

```
{
  "containerDefinitions": [
    {
      "name": "client",
      "image": "%s.dkr.ecr.us-west-1.amazonaws.com/test-driven-client:staging",
      "essential": true,
      "memoryReservation": 300,
```

```
        "portMappings": [
          {
            "hostPort": 0,
            "protocol": "tcp",
            "containerPort": 80
          }
        ],
        "logConfiguration": {
          "logDriver": "awslogs",
          "options": {
            "awslogs-group": "testdriven-client-stage",
            "awslogs-region": "us-west-1"
          }
        }
      }
    ],
    "family": "testdriven-client-stage-td"
}
```

#### Users

```
{
  "containerDefinitions": [
    {
      "name": "users",
      "image": "%s.dkr.ecr.us-west-1.amazonaws.com/test-driven-users:staging",
      "essential": true,
      "memoryReservation": 300,
      "portMappings": [
        {
          "hostPort": 0,
          "protocol": "tcp",
          "containerPort": 5000
        }
      ],
      "environment": [
        {
          "name": "APP_SETTINGS",
          "value": "project.config.StagingConfig"
        },
        {
          "name": "DATABASE_TEST_URL",
          "value": "postgres://postgres:postgres@users-db:5432/users_test"
        },
        {
          "name": "DATABASE_URL",
          "value": "postgres://postgres:postgres@users-db:5432/users_stage"
        },
        {
          "name": "SECRET_KEY",
```

```
            "value": "my_precious"
          }
        ],
        "links": [
          "users-db"
        ],
        "logConfiguration": {
          "logDriver": "awslogs",
          "options": {
            "awslogs-group": "testdriven-users-stage",
            "awslogs-region": "us-west-1"
          }
        }
      },
      {
        "name": "users-db",
        "image": "%s.dkr.ecr.us-west-1.amazonaws.com/test-driven-users_db:staging",
        "essential": true,
        "memoryReservation": 300,
        "portMappings": [
          {
            "hostPort": 0,
            "protocol": "tcp",
            "containerPort": 5432
          }
        ],
        "environment": [
          {
            "name": "POSTGRES_PASSWORD",
            "value": "postgres"
          },
          {
            "name": "POSTGRES_USER",
            "value": "postgres"
          }
        ],
        "logConfiguration": {
          "logDriver": "awslogs",
          "options": {
            "awslogs-group": "testdriven-users_db-stage",
            "awslogs-region": "us-west-1"
          }
        }
      }
    ],
    "family": "testdriven-users-stage-td"
  }
```

**Swagger**

```
{
  "containerDefinitions": [
    {
      "name": "swagger",
      "image": "%s.dkr.ecr.us-west-1.amazonaws.com/test-driven-swagger:staging",
      "essential": true,

      "logConfiguration": {
        "logDriver": "awslogs",
        "options": {
          "awslogs-group": "testdriven-swagger-stage",
          "awslogs-region": "us-west-1"
        }
      },
      "portMappings": [
        {
          "hostPort": 0,
          "protocol": "tcp",
          "containerPort": 8080
        }
      ],
      "environment": [
        {
          "name": "URL",
          "value": "swagger.json"
        }
      ],
      "memoryReservation": 300
    }
  ],
  "family": "testdriven-swagger-stage-td"
}
```

## Travis - update task definition

Add a new file to the root called *docker-deploy-stage.sh*:

```
#!/bin/sh

if [ -z "$TRAVIS_PULL_REQUEST" ] || [ "$TRAVIS_PULL_REQUEST" == "false" ]
then

  if [ "$TRAVIS_BRANCH" == "staging" ]
  then

    JQ="jq --raw-output --exit-status"

    configure_aws_cli() {
        aws --version
```

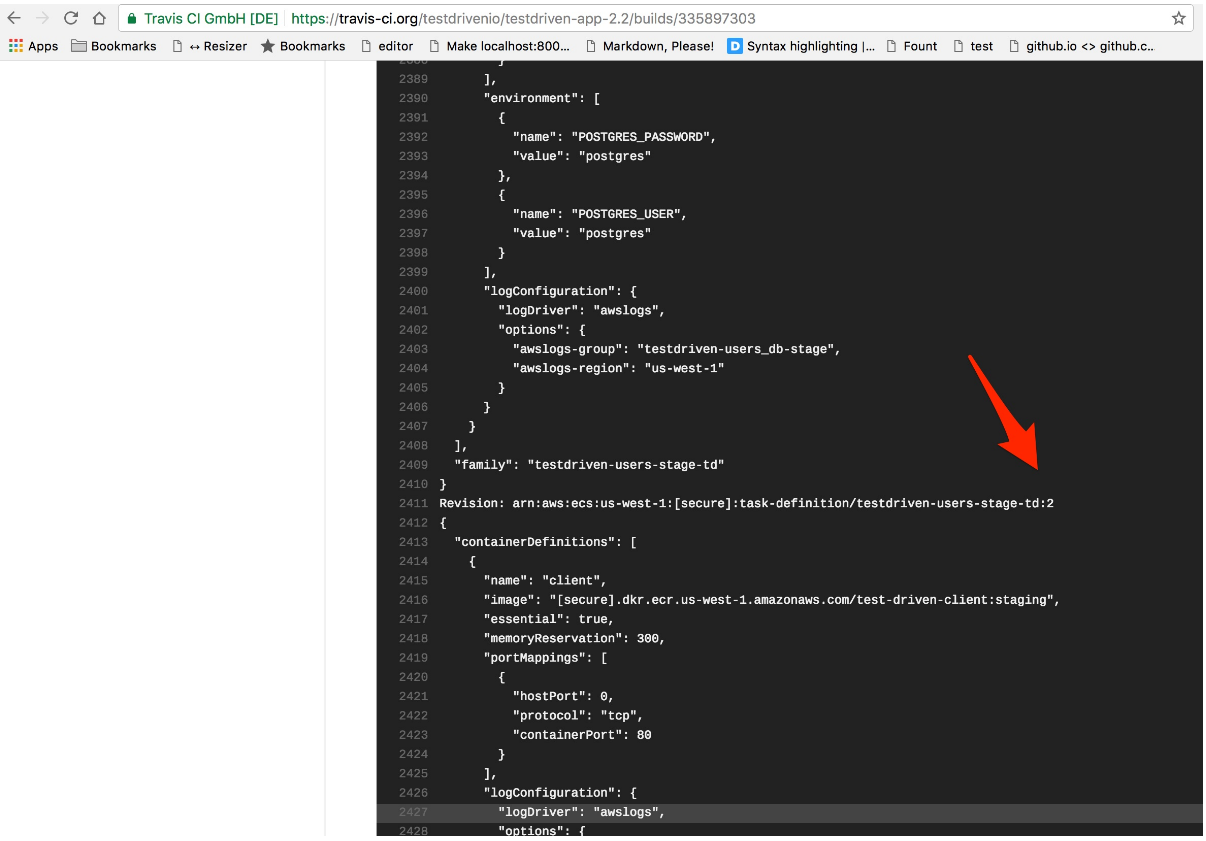
```
        aws configure set default.region us-west-1
        aws configure set default.output json
        echo "AWS Configured!"
    }

    register_definition() {
      if revision=$(aws ecs register-task-definition --cli-input-json "$task_def" |
$JQ '.taskDefinition.taskDefinitionArn'); then
        echo "Revision: $revision"
      else
        echo "Failed to register task definition"
        return 1
      fi
    }

    deploy_cluster() {

      # users
      template="ecs_users_stage_taskdefinition.json"
      task_template=$(cat "ecs/$template")
      task_def=$(printf "$task_template" $AWS_ACCOUNT_ID $AWS_ACCOUNT_ID)
      echo "$task_def"
      register_definition

      # client
      template="ecs_client_stage_taskdefinition.json"
      task_template=$(cat "ecs/$template")
      task_def=$(printf "$task_template" $AWS_ACCOUNT_ID)
      echo "$task_def"
      register_definition

      # swagger
      template="ecs_swagger_stage_taskdefinition.json"
      task_template=$(cat "ecs/$template")
      task_def=$(printf "$task_template" $AWS_ACCOUNT_ID)
      echo "$task_def"
      register_definition

    }

    configure_aws_cli
    deploy_cluster

  fi

fi
```

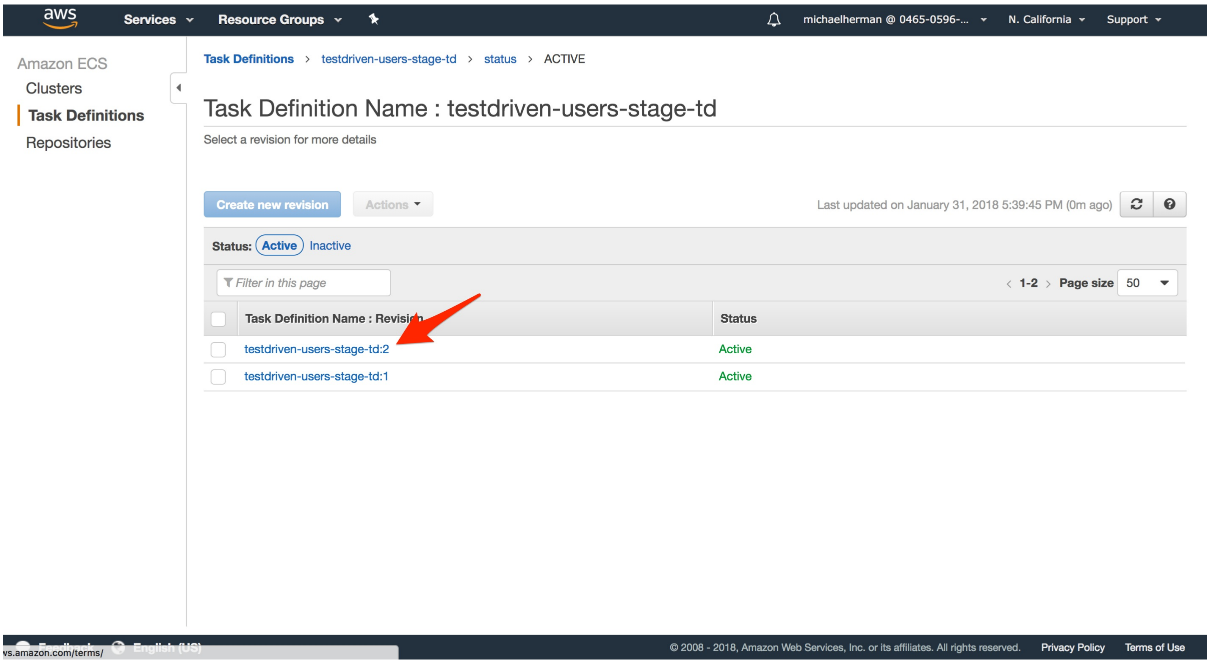
Here, if the branch is `staging` and it's not a a pull request, the AWS CLI is configured and then the `deploy_cluster` function is fired, which updates the existing Task Definitions with the definitions found in the JSON files we just created.

Update the `after_success` in *.travis.yml*:

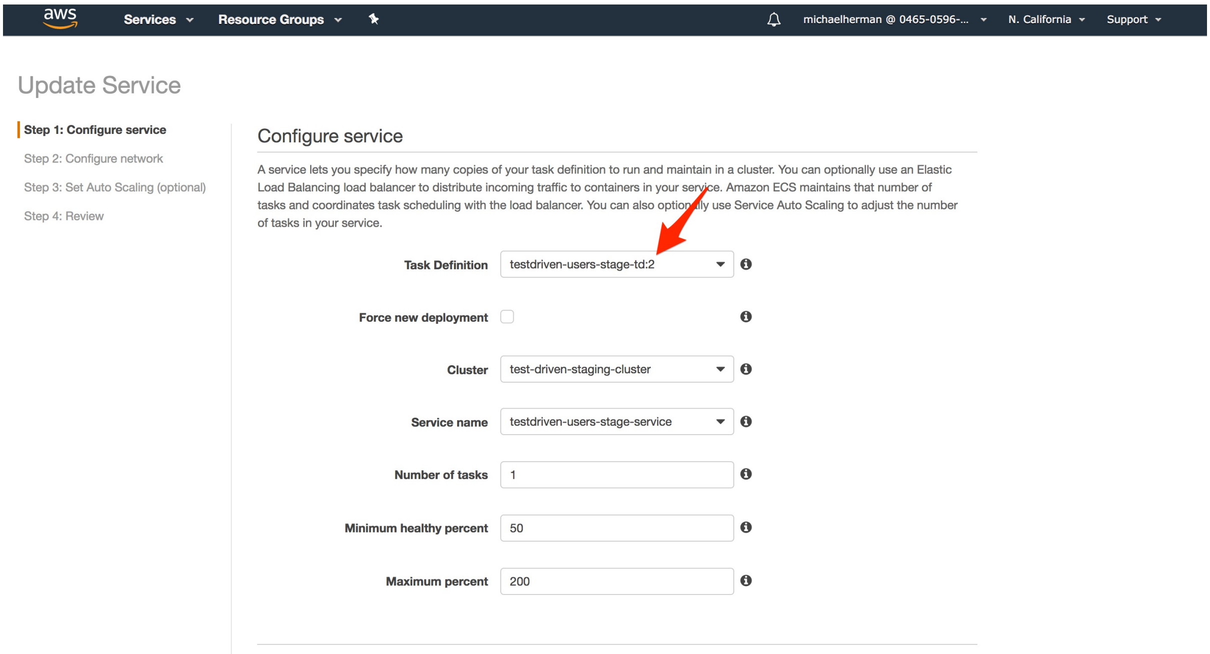
```
after_success:
  - bash ./docker-push.sh
  - bash ./docker-deploy-stage.sh
```

Assuming you're still on the `staging` branch, commit and push your code to GitHub. After the Travis build passes, make sure new images were created and revisions to the Task Definitions were added.






Then, navigate to the Cluster. Update each of the Services so that they use the new Task Definitions.

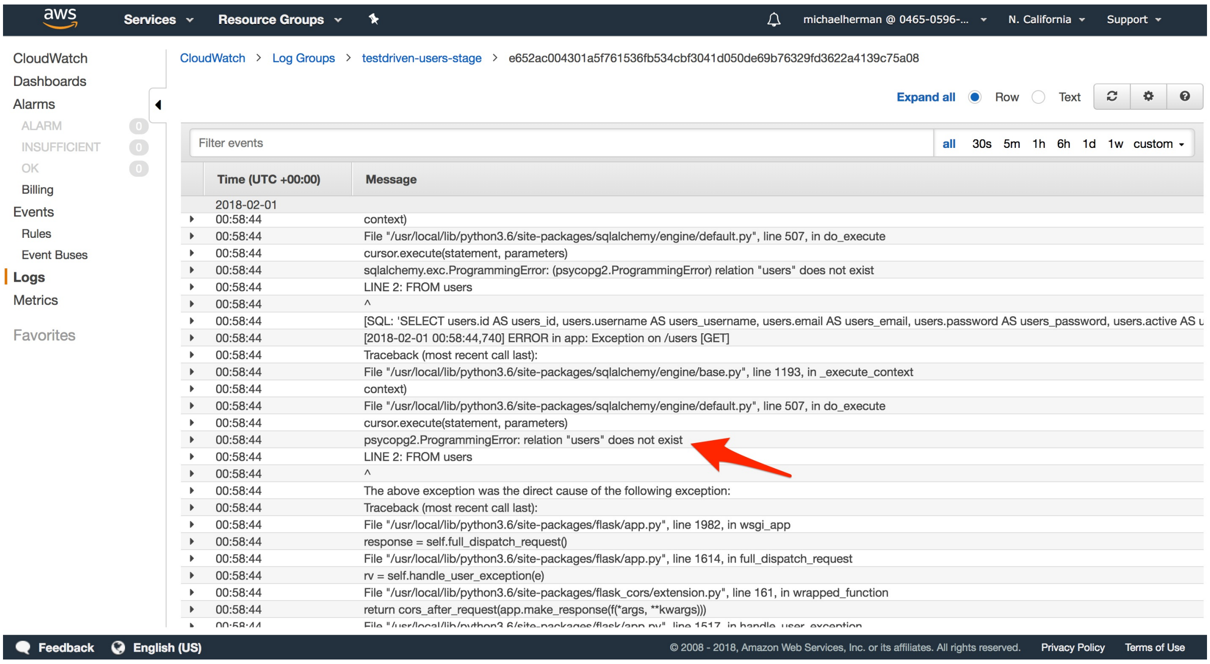



Again, ECS will instantiate the Task Definitions, creating new Tasks that will spin up on the Cluster instances. Then, as long as the health checks pass, the load balancer will start sending traffic to them.

Make sure the instances associated with the Target Groups - `testdriven-client-stage-tg`, `testdriven-users-stage-tg`, and `testdriven-swagger-stage-tg` - are healthy. And then run the end-to-end tests. You should see several errors since we didn't run the migrations:

```
4/12 failed (1m 23s)
```

You can confirm that it is a database migration issue by checking the logs in CloudWatch:




We could run the migrations manually, but let's automate this as well by updating *services/users/entrypoint-prod.sh*:

```
#!/bin/sh

echo "Waiting for postgres..."

while ! nc -z users-db 5432; do
  sleep 0.1
done

echo "PostgreSQL started"

python manage.py recreate_db
python manage.py seed_db
gunicorn -b 0.0.0.0:5000 manage:app
```

Here, we wait for the `users-db` service to be up before updating and seeding the database and then firing the server.

Commit and push your code, and after the build passes ensure:

1. New images were created
2. Revisions were added to the Task Definitions

Then, update each of the Services so that they reference the new Task Definitions. Run the end-to-end tests after the Tasks spin up and new instances are added to the Target Groups:

```
/
✓ should display the page correctly if a user is not logged in
```

```
/login
✓ should display the sign in form
✓ should allow a user to sign in
✓ should throw an error if the credentials are incorrect

/register
✓ should display flash messages correctly

/register
✓ should display the registration form
✓ should allow a user to register
✓ should validate the password field
✓ should throw an error if the username is taken
✓ should throw an error if the email is taken

/status
✓ should not display user info if a user is not logged in
✓ should display user info if a user is logged in


12 passed (53s)
```

## Travis - update service

Now, update *docker-deploy-stage.sh*, like so, to automatically update the Services after new revisions are added to the Task Definitions:

```
#!/bin/sh

if [ -z "$TRAVIS_PULL_REQUEST" ] || [ "$TRAVIS_PULL_REQUEST" == "false" ]
then

  if [ "$TRAVIS_BRANCH" == "staging" ]
  then

    JQ="jq --raw-output --exit-status"

    configure_aws_cli() {
        aws --version
        aws configure set default.region us-west-1
        aws configure set default.output json
        echo "AWS Configured!"
    }

    register_definition() {
      if revision=$(aws ecs register-task-definition --cli-input-json "$task_def" |
$JQ '.taskDefinition.taskDefinitionArn'); then
        echo "Revision: $revision"
      else
```

```
        echo "Failed to register task definition"
        return 1
      fi
    }

    update_service() {
      if [[ $(aws ecs update-service --cluster $cluster --service $service --task-d
efinition $revision | $JQ '.service.taskDefinition') != $revision ]]; then
        echo "Error updating service."
        return 1
      fi
    }

    deploy_cluster() {

      cluster="test-driven-staging-cluster"

      # users
      service="testdriven-users-stage-service"
      template="ecs_users_stage_taskdefinition.json"
      task_template=$(cat "ecs/$template")
      task_def=$(printf "$task_template" $AWS_ACCOUNT_ID $AWS_ACCOUNT_ID)
      echo "$task_def"
      register_definition
      update_service

      # client
      service="testdriven-client-stage-service"
      template="ecs_client_stage_taskdefinition.json"
      task_template=$(cat "ecs/$template")
      task_def=$(printf "$task_template" $AWS_ACCOUNT_ID)
      echo "$task_def"
      register_definition
      update_service

      # swagger
      service="testdriven-swagger-stage-service"
      template="ecs_swagger_stage_taskdefinition.json"
      task_template=$(cat "ecs/$template")
      task_def=$(printf "$task_template" $AWS_ACCOUNT_ID)
      echo "$task_def"
      register_definition
      update_service

    }

    configure_aws_cli
    deploy_cluster

  fi
```

```
fi
```

If you haven't already, update the `URL` in the *services/swagger/swagger.json* file:

```
$ python services/swagger/update-spec.py http://LOAD_BALANCER_STAGE_DNS_NAME
```

Commit and push your code to GitHub to trigger a new Travis build. Once done, you should see a new revision associated with each Task Definition and the Services should now be running a new Task based on that revision.

Test everything out again, manually and with the e2e tests!

# Setting up RDS

Before adding our production Cluster, let's set up Amazon RDS...

---

First off, why should we set up Amazon Relational Database Service (RDS)? Why should we not just manage Postgres within the Cluster itself?

1. Since the recommended means of service discovery in ECS is load balancing, we'll have to register the Postgres instance with the ALB. There's additional costs and overhead associated with this. We'll also have to assign a public IP to it and expose it to the internet. It's best to keep it private. Now, we could add Route 53 in front of the ALB, to restrict traffic to specific endpoints (like Postgres), but this is yet another service.
2. Data integrity is an issue as well. What happens if the container crashes?
3. In the end, you will save time and money using RDS rather than managing your own Postgres instance on a server somewhere

> For more, check out this Reddit post.

## RDS Setup

Navigate to Amazon RDS, click "Instances" on the sidebar, and then click the "Launch DB Instance" button.

#### Step 1: Select engine

> You *probably* want to click the "Only enable options eligible for RDS Free Usage Tier". More info.

Select the "PostgreSQL" engine and click "Next".

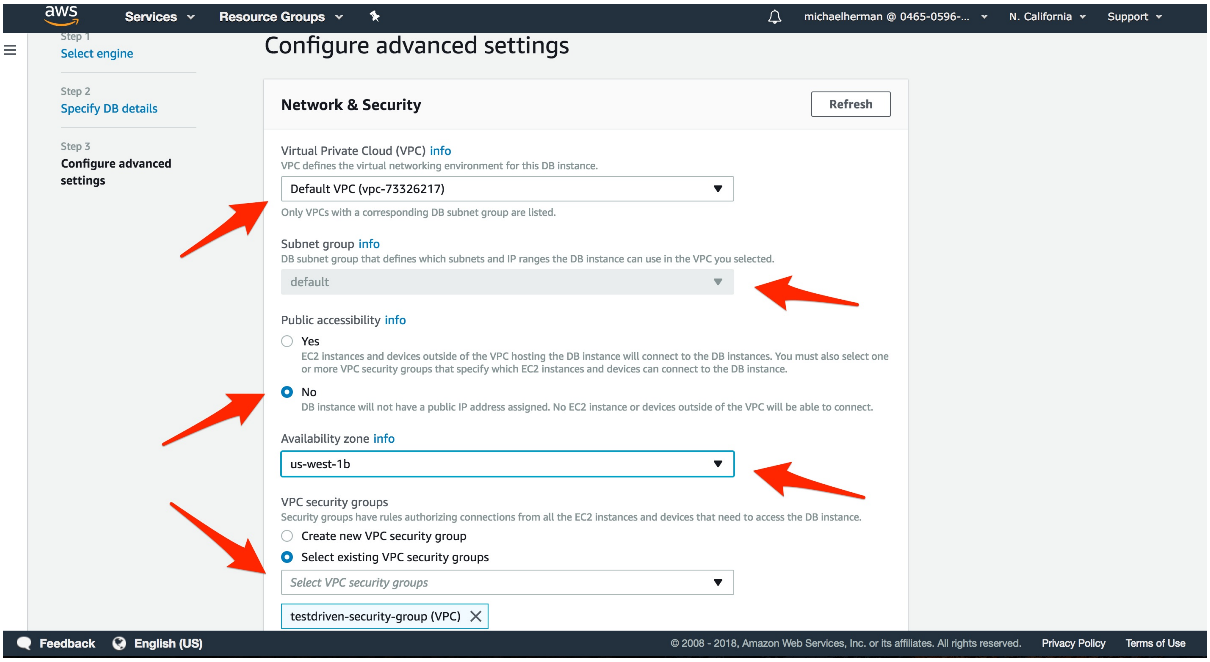
#### Step 2: Specify DB details

1. "DB Engine Version": `PostgreSQL 9.6.5-R1`
2. "DB Instance Class": `db.t2.micro`
3. "Multi-AZ Deployment": `No`
4. "Storage Type": `General Purpose (SSD)`
5. "Allocated Storage": `20 GB`
6. "DB Instance Identifier": `testdriven-production`
7. "Master Username": `webapp`
8. "Master Password": `something_super_secret`

Click "Next".

#### Step 3: Configure advanced settings

Under "Network & Security", make sure to pick the "VPC" and "Security group" associated with ALB. Select one of the available "Subnets" as well - either `us-west-1b` or `us-west-1c`.




Change the DB name to `users_prod` and then create the new database.

You can quickly check the status via:

```
$ aws --region us-west-1 rds describe-db-instances \
  --db-instance-identifier testdriven-production \
  --query 'DBInstances[].{DBInstanceStatus:DBInstanceStatus}'

[
    {
        "DBInstanceStatus": "creating"
    }
]
```

Then, once the status is "available", grab the address:

```
$ aws --region us-west-1 rds describe-db-instances \
  --db-instance-identifier testdriven-production \
  --query 'DBInstances[].{Address:Endpoint.Address}'
```

Take note of the production URI:

```
postgres://webapp:YOUR_PASSWORD@YOUR_ADDRESS:5432/users_prod
```

> Keep in mind that you cannot access the DB outside the VPC. So, if you want to connect to the instance, you will need to use SSH tunneling via SSHing into an EC2 instance on the same VPC and, from there, connecting to the database. We'll go through this process in a future

> lesson.

# ECS Production Setup

In this lesson, we'll set up our production Cluster on ECS...

---

Start by reviewing the Staging Cluster. Which AWS resources do we need to set up for the production Cluster? Think about the steps we have to take, starting with the manual setup of the resources...

## Setup Steps

#### Load Balancer

1. Create an Application Load Balancer (ALB)
2. Configure Target Groups
3. Add Listeners to the ALB
4. Update *docker-push.sh*

#### ECS

1. Configure Task Definitions
2. Add images to ECR
3. Create an ECS Cluster
4. Update *entrypoint-prod.sh*
5. Create Services

Switch to the `production` branch. Let's get to it!

> This is a great time to check your understanding. There are a number of steps, but the big difference between production and staging is the RDS database. Do your best to configure everything on your own before reviewing the lesson.

## Load Balancer

### Application Load Balancer

Navigate to the Amazon EC2 Dashboard. Click "Load Balancers" on the sidebar, and then click the "Create Load Balancer" button. Select "Application Load Balancer".

#### Step 1: Configure Load Balancer

1. "Name": `testdriven-production-alb`
2. "VPC": Select the default VPC to keep things simple
3. "Availability Zones": Select at least two available subnets

#### Step 2: Configure Security Settings

Skip this for now.

#### Step 3: Configure Security Groups

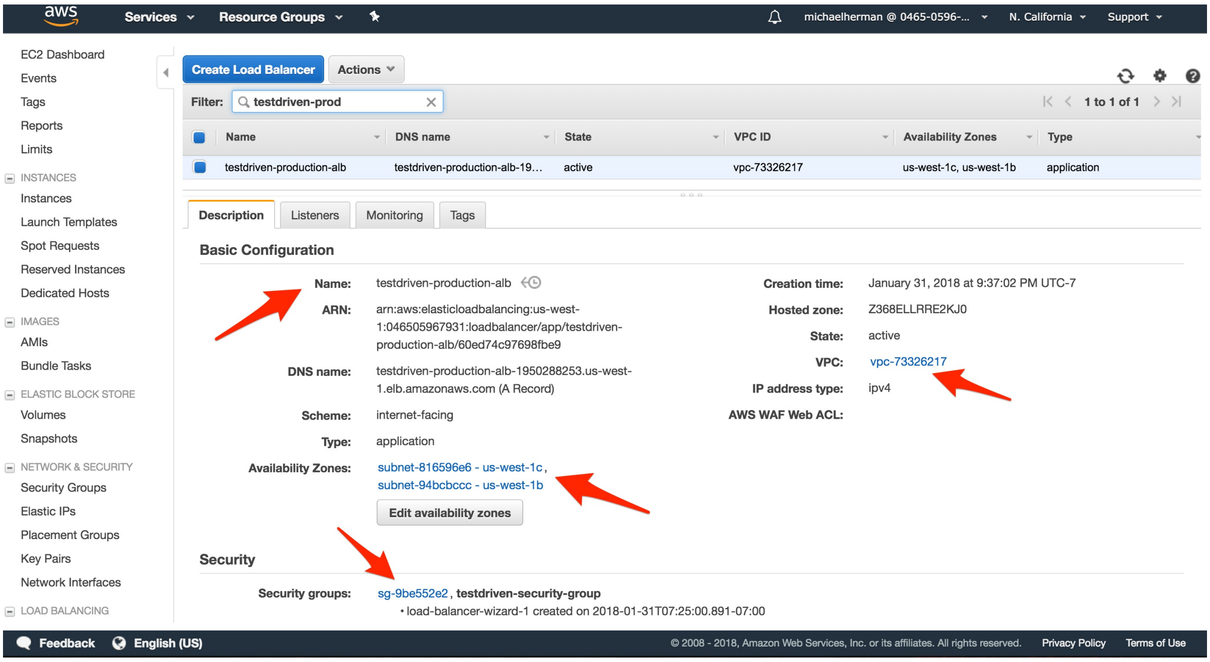
Select the `testdriven-security-group` .

#### Step 4: Configure Routing

1. "Name": `testdriven-client-prod-tg`
2. "Port": `80`
3. "Path": `/`

#### Step 5: Register Targets

Do not assign any instances manually since this will be managed by ECS. Review and then create the new load balancer.




## Target Groups

Next, set up new Target Groups for `swagger` and the `users` service. Within Amazon EC2, click "Target Groups", and then create the following Target Groups:

#### Target Group 1: `users`

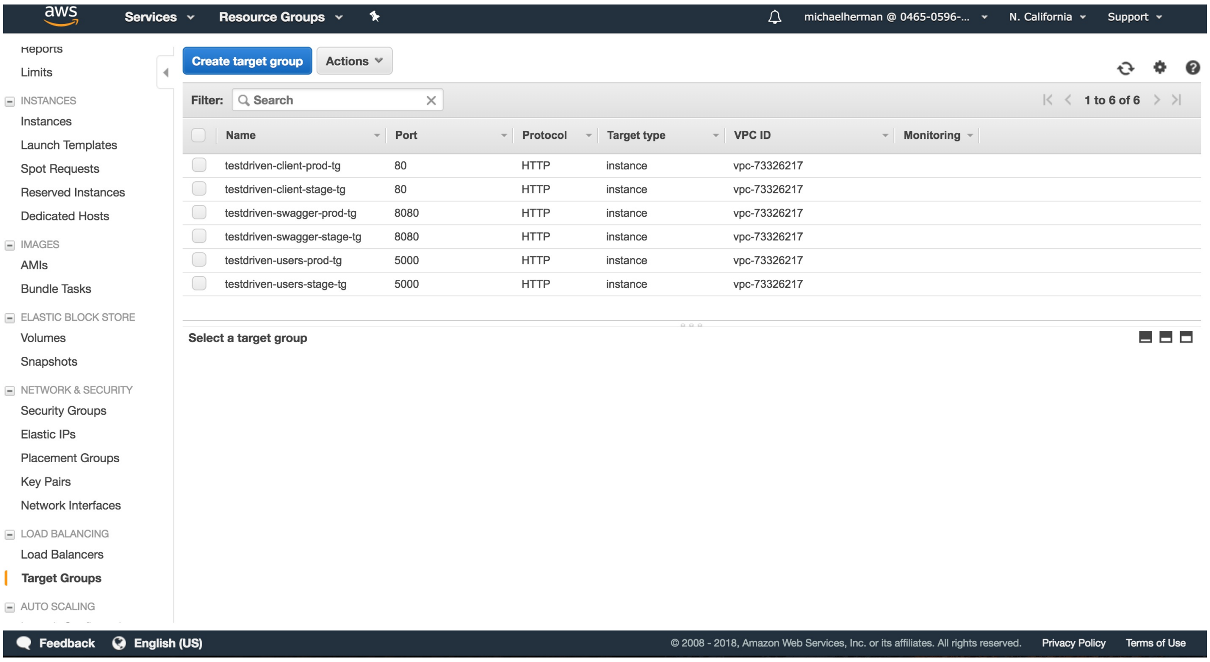
1. "Target group name": `testdriven-users-prod-tg`
2. "Port": `5000`
3. Then, under "Health check settings" set the "Path" to `/users/ping` .

#### Target Group 2: `swagger`

1. "Target group name": `testdriven-swagger-prod-tg`

2. "Port": `8080`
3. Then, under "Health check settings" set the "Path" to `/swagger` .

You should now have the following Target Groups:

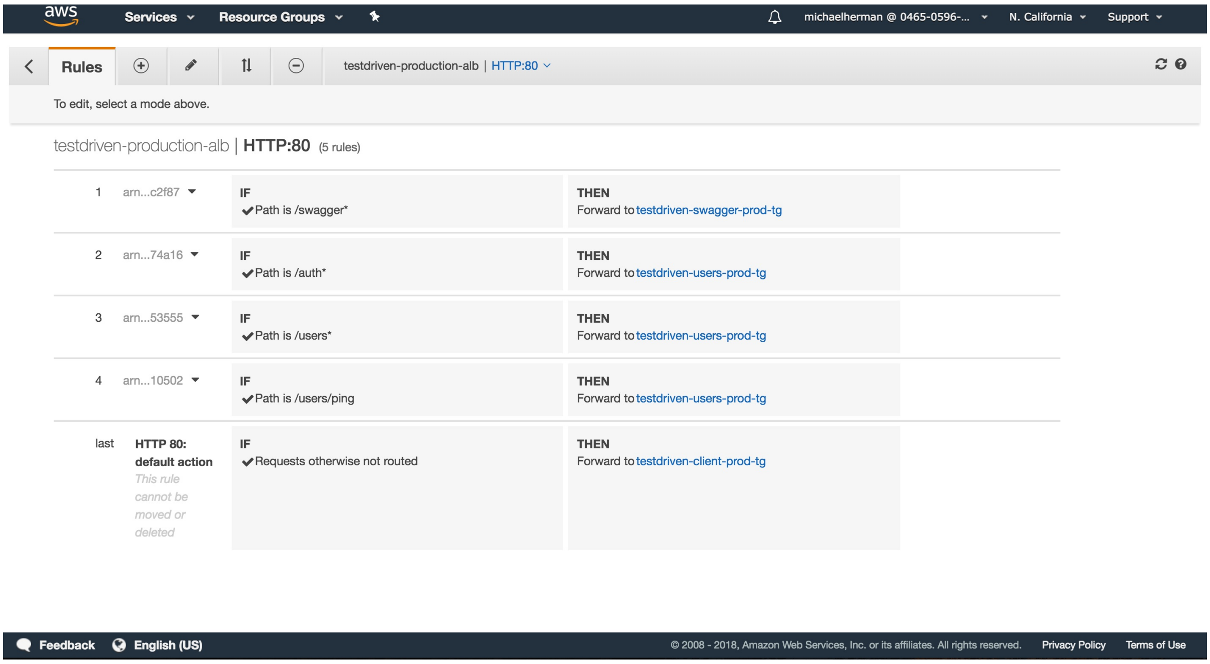



## Listeners

Back on the "Load Balancers" page, click the `testdriven-production-alb` Load Balancer, and then select the "Listeners" tab. Here, we can add Listeners to the ALB, which forward traffic to a specific Target Group.

There should already be a listener for "HTTP : 80". Click the "View/edit rules >" link, and then insert four new rules:

1. If `/swagger*` , Then `testdriven-swagger-prod-tg`
2. If `/auth*` , Then `testdriven-users-prod-tg`
3. If `/users*` , Then `testdriven-users-prod-tg`
4. If `/users/ping` , Then `testdriven-users-prod-tg`




## Update *docker-push.sh*

Navigate back to the Load Balancer and grab the "DNS name" from the "Description" tab. We need to set this as the value of `REACT_APP_USERS_SERVICE_URL` in *docker-push.sh*, so that the build-time arg for the `client` service is set correctly.

Updated script:

```
#!/bin/sh

if [ -z "$TRAVIS_PULL_REQUEST" ] || [ "$TRAVIS_PULL_REQUEST" == "false" ]
then

  if [ "$TRAVIS_BRANCH" == "staging" ] || \
     [ "$TRAVIS_BRANCH" == "production" ]
  then
    curl "https://s3.amazonaws.com/aws-cli/awscli-bundle.zip" -o "awscli-bundle.zip"

    unzip awscli-bundle.zip
    ./awscli-bundle/install -b ~/bin/aws
    export PATH=~/bin:$PATH
    # add AWS_ACCOUNT_ID, AWS_ACCESS_KEY_ID, AWS_SECRET_ACCESS_KEY env vars
    eval $(aws ecr get-login --region us-west-1 --no-include-email)
    export TAG=$TRAVIS_BRANCH
    export REPO=$AWS_ACCOUNT_ID.dkr.ecr.us-west-1.amazonaws.com
  fi
  if [ "$TRAVIS_BRANCH" == "staging" ]
  then
    export REACT_APP_USERS_SERVICE_URL="http://LOAD_BALANCER_STAGE_DNS_NAME"
  fi
  if [ "$TRAVIS_BRANCH" == "production" ]
```

```
  then
    export REACT_APP_USERS_SERVICE_URL="http://LOAD_BALANCER_PROD_DNS_NAME"
  fi
  if [ "$TRAVIS_BRANCH" == "staging" ] || \
     [ "$TRAVIS_BRANCH" == "production" ]
  then
    # users
    docker build $USERS_REPO -t $USERS:$COMMIT -f Dockerfile-$DOCKER_ENV
    docker tag $USERS:$COMMIT $REPO/$USERS:$TAG
    docker push $REPO/$USERS:$TAG
    # users db
    docker build $USERS_DB_REPO -t $USERS_DB:$COMMIT -f Dockerfile
    docker tag $USERS_DB:$COMMIT $REPO/$USERS_DB:$TAG
    docker push $REPO/$USERS_DB:$TAG
    # client
    docker build $CLIENT_REPO -t $CLIENT:$COMMIT -f Dockerfile-$DOCKER_ENV --build-
arg REACT_APP_USERS_SERVICE_URL=$REACT_APP_USERS_SERVICE_URL
    docker tag $CLIENT:$COMMIT $REPO/$CLIENT:$TAG
    docker push $REPO/$CLIENT:$TAG
    # swagger
    docker build $SWAGGER_REPO -t $SWAGGER:$COMMIT -f Dockerfile-$DOCKER_ENV $SWAGG
ER_DIR
    docker tag $SWAGGER:$COMMIT $REPO/$SWAGGER:$TAG
    docker push $REPO/$SWAGGER:$TAG
  fi
fi
```

# ECS

## Task Definitions

Navigate to Amazon ECS, click "Task Definitions", and then click the button "Create new Task Definition".

> Make sure you set up the production logs. To set up, navigate to CloudWatch, click "Logs", click the "Actions" drop-down button, and then select "Create log group".

**Target Definition 1: `client`**

First, Update the "Task Definition Name" to `testdriven-client-prod-td` and then add a new container:

1. "Container name": `client`
2. "Image": `YOUR_AWS_ACCOUNT_ID.dkr.ecr.us-west-1.amazonaws.com/test-driven-client:production`
3. "Memory Limits (MB)": `300` soft limit
4. "Port mappings": `0` host, `80` container
5. "Log configuration": `testdriven-client-prod`

#### Target Definition 2: `users`

For the `users` service, use the name `testdriven-users-prod-td`, and then add a single container:

1. "Container name": `users`
2. "Image": `YOUR_AWS_ACCOUNT_ID.dkr.ecr.us-west-1.amazonaws.com/test-driven-users:production`
3. "Memory Limits (MB)": `300` soft limit
4. "Port mappings": `0` host, `5000` container
5. "Log configuration": `testdriven-users-prod`

Also, add the following environment variables:

1. `SECRET_KEY` - `my_precious`
2. `APP_SETTINGS` - `project.config.ProductionConfig`
3. `DATABASE_URL` - `YOUR_RDS_URI`
4. `DATABASE_TEST_URL` - `postgres://postgres:postgres@users-db:5432/users_test`

> We'll update the `SECRET_KEY`, when we add the automation script.

#### Target Definition 3: `swagger`

Add a final Task Definition for the `swagger` service with the name `testdriven-swagger-prod-td`.

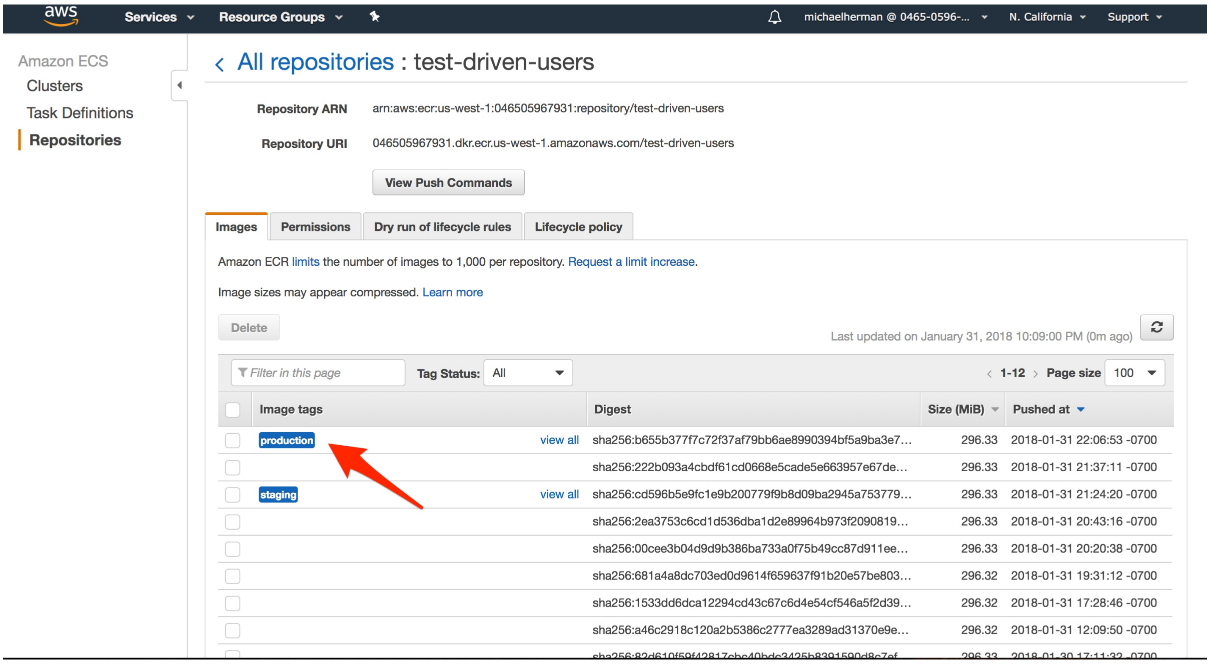
1. "Container name": `swagger`
2. "Image": `YOUR_AWS_ACCOUNT_ID.dkr.ecr.us-west-1.amazonaws.com/test-driven-swagger:production`
3. "Memory Limits (MB)": `300` soft limit
4. "Port mappings": `0` host, `8080` container
5. "Env Variables":
    i. `URL` - `swagger.json`
6. "Log configuration": `testdriven-swagger-prod`

## ECR

Update the Swagger spec:

```
$ python services/swagger/update-spec.py http://LOAD_BALANCER_PROD_DNS_NAME
```

To create the images, commit and push to GitHub. Make sure the images were created and tagged as `production` on ECR, after the build passes.




> Since we're not using `users_db` in production, you may want to update the *docker-push.sh* file so it is not built, tagged, or pushed when the branch is `production`.

## Cluster

Navigate to Amazon ECS, and create a new Cluster (make sure to select "EC2 Linux + Networking"):

1. "Cluster name": `test-driven-production-cluster`
2. "EC2 instance type": `t2.micro`
3. "Number of instances": `2`
4. "Key pair": Select an existing Key Pair, like `ecs-west`, or create a new one
5. Make sure to pick the "VPC" and "Security group" associated with ALB along with the appropriate "Subnets" - `us-west-1b` and `us-west-1c`.

> Although it doesn't matter so much for this course, it is best practice to use a different key pair for production, especially for large development teams.

It will take a few minutes to setup the EC2 resources.

## Update *entrypoint-prod.sh*

First, rename *entrypoint-prod.sh* to *entrypoint-stage.sh* within the "users" service, and then update the run service command in *services/users/Dockerfile-stage* to:

```
# run server
CMD ["./entrypoint-stage.sh"]
```

Make sure you add the correct file as well:

```
# add entrypoint-stage.sh
```

```
COPY ./entrypoint-stage.sh /usr/src/app/entrypoint-stage.sh
```

Then, update the run service command in *services/users/Dockerfile-prod* to:

```
# run server
CMD gunicorn -b 0.0.0.0:5000 manage:app
```

You can remove this layer:

```
# add entrypoint.sh
COPY ./entrypoint-prod.sh /usr/src/app/entrypoint-prod.sh
```

So, instead of running an *entrypoint* file, we are now just running Gunicorn in production. Why? Well, first off, we will not be using a `users-db` container in production. Also, we only want to create the database and seed it once, rather than on every deploy, to persist the data.

This will introduce a race condition on Travis if we use the *docker-compose-prod.yml* file since it's no longer waiting for the database to spin up before it starts.

To fix, in the `before_script` step in *.travis.yml*, change:

```
- docker-compose -f docker-compose-$DOCKER_ENV.yml up --build -d
```

To:

```
- docker-compose -f docker-compose-stage.yml up --build -d
```

Also, change the `after_script` from:

```
after_script:
  - docker-compose -f docker-compose-$DOCKER_ENV.yml down
```

To:

```
after_script:
  - docker-compose -f docker-compose-stage.yml down
```

## Services

Create the following Services...

#### Client

*Configure service*:

1. "Task Definition": `testdriven-client-prod-td:LATEST_REVISION_NUMBER`

2. "Service name": `testdriven-client-prod-service`
3. "Number of tasks": `1`

Click "Next".

*Configure network*:

Select the "Application Load Balancer" under "Load balancer type".

1. "Load balancer name": `testdriven-production-alb`
2. "Container name : port": `client:0:80`

Click "Add to load balancer".

1. "Listener port": `80:HTTP`
2. "Target group name": `testdriven-client-prod-tg`

Click the next button a few times, and then "Create Service".

#### Users

*Configure service*:

1. "Task Definition": `testdriven-users-prod-td:LATEST_REVISION_NUMBER`
2. "Service name": `testdriven-users-prod-service`
3. "Number of tasks": `1`

Click "Next".

*Configure network*:

Select the "Application Load Balancer" under "Load balancer type".

1. "Load balancer name": `testdriven-production-alb`
2. "Container name : port": `users:0:5000`

Click "Add to load balancer".

1. "Listener port": `80:HTTP`
2. "Target group name": `testdriven-users-prod-tg`

Click the next button a few times, and then "Create Service".

#### Swagger

*Configure service*:

1. "Task Definition": `testdriven-swagger-prod-td:LATEST_REVISION_NUMBER`
2. "Service name": `testdriven-swagger-prod-service`
3. "Number of tasks": `1`

Click "Next".

*Configure network*:

Select the "Application Load Balancer" under "Load balancer type".

1. "Load balancer name": `testdriven-production-alb`
2. "Container name : port": `swagger:0:8080`

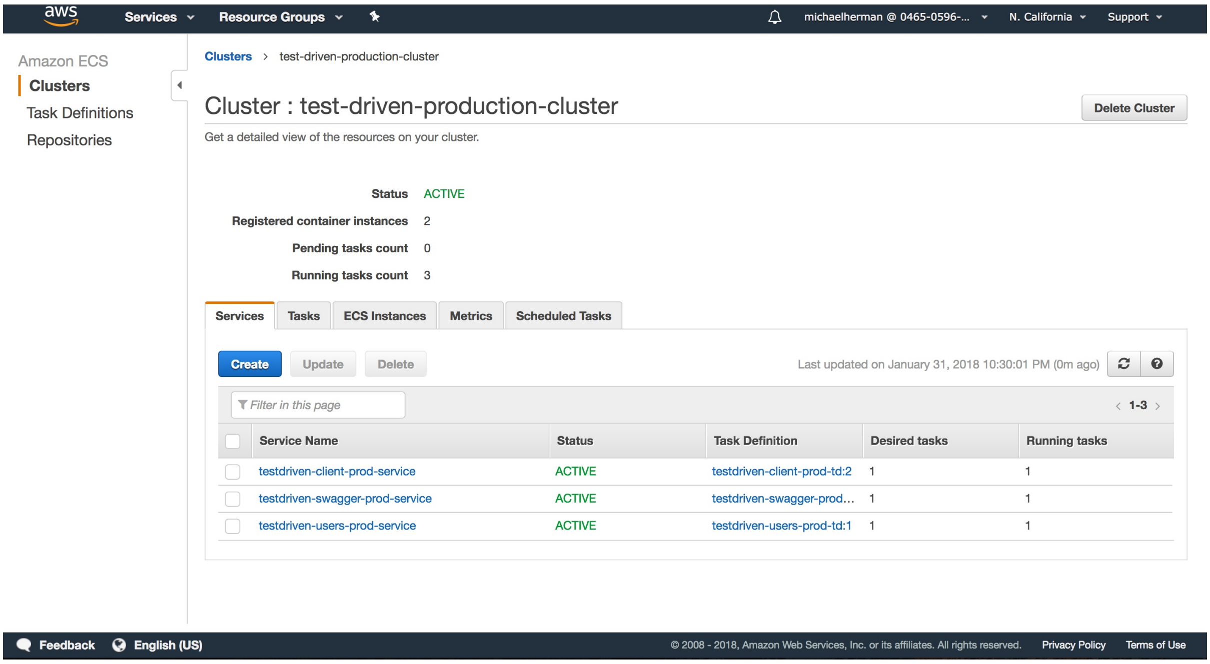
Click "Add to load balancer".

1. "Listener port": `80:HTTP`
2. "Target group name": `testdriven-swagger-prod-tg`

Click the next button a few times, and then "Create Service".

---

You should now have the following Services running, each with a single Task:




## Sanity Check (take one)

Navigate to Amazon EC2, and click "Target Groups". Make sure `testdriven-client-prod-tg`, `testdriven-users-prod-tg`, and `testdriven-swagger-prod-tg` have a single registered instance each. They should all be healthy.

Then, navigate back to the Load Balancer and grab the "DNS name" from the "Description" tab. Test each in your browser:

1. http://LOAD_BALANCER_PROD_DNS_NAME
2. http://LOAD_BALANCER_PROD_DNS_NAME/users/ping

Try the `/users` endpoint at http://LOAD_BALANCER_PROD_DNS_NAME/users. You should see a 500 error since the migrations have not been ran. To do this, let's SSH into the EC2 instance associated with the `testdriven-users-prod-tg` Target Group:

```
$ ssh -i ~/.ssh/ecs.pem ec2-user@EC2_PUBLIC_IP
```

> You may need to update the permissions on the Pem file - i.e., `chmod 400 ~/.ssh/ecs.pem`.

Next, grab the Container ID for `users` (via `docker ps`), enter the shell within the running container, and then update the RDS database:

```
$ docker exec -it Container_ID bash
# python manage.py recreate_db
# python manage.py seed_db
```

Navigate to http://LOAD_BALANCER_PROD_DNS_NAME/users again and you should see the users.

Now for the real sanity check - run the e2e tests!

Set the `TEST_URL` environment variable and then run the tests:

```
$ export TEST_URL=http://LOAD_BALANCER_PROD_DNS_NAME
$ testcafe chrome e2e
```

They should pass!

# ECS Production Automation

In the lesson, we'll update the CI/CD workflow to add a new revision to the Task Definition and update the Service for the production Cluster on ECS...

---

Again, try this on your own before reviewing the lesson.

## Steps

1. Create local Task Definition JSON files
2. Update creation of Task Definitions on AWS via Travis
3. Update Service via Travis

## Task Definitions

Create JSON files for the Task Definitions in the "ecs" folder.

1. *ecs_client_prod_taskdefinition.json*
2. *ecs_users_prod_taskdefinition.json*
3. *ecs_swagger_prod_taskdefinition.json*

#### Client

```
{
  "containerDefinitions": [
    {
      "name": "client",
      "image": "%s.dkr.ecr.us-west-1.amazonaws.com/test-driven-client:production",
      "essential": true,
      "memoryReservation": 300,
      "portMappings": [
        {
          "hostPort": 0,
          "protocol": "tcp",
          "containerPort": 80
        }
      ],
      "logConfiguration": {
        "logDriver": "awslogs",
        "options": {
          "awslogs-group": "testdriven-client-prod",
          "awslogs-region": "us-west-1"
        }
      }
    }
```

```
  ],
  "family": "testdriven-client-prod-td"
}
```

**Users**

```
{
  "containerDefinitions": [
    {
      "name": "users",
      "image": "%s.dkr.ecr.us-west-1.amazonaws.com/test-driven-users:production",
      "essential": true,
      "memoryReservation": 300,
      "portMappings": [
        {
          "hostPort": 0,
          "protocol": "tcp",
          "containerPort": 5000
        }
      ],
      "environment": [
        {
          "name": "APP_SETTINGS",
          "value": "project.config.ProductionConfig"
        },
        {
          "name": "DATABASE_TEST_URL",
          "value": "postgres://postgres:postgres@users-db:5432/users_test"
        },
        {
          "name": "DATABASE_URL",
          "value": "%s"
        },
        {
          "name": "SECRET_KEY",
          "value": "%s"
        }
      ],
      "logConfiguration": {
        "logDriver": "awslogs",
        "options": {
          "awslogs-group": "testdriven-users-prod",
          "awslogs-region": "us-west-1"
        }
      }
    }
  ],
  "family": "testdriven-users-prod-td"
}
```

**Swagger**

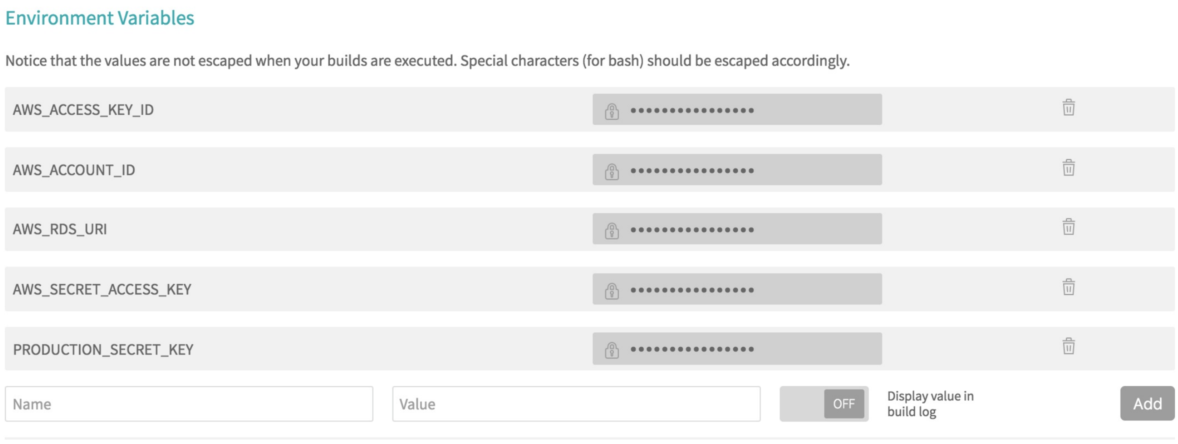
```
{
  "containerDefinitions": [
    {
      "name": "swagger",
      "image": "%s.dkr.ecr.us-west-1.amazonaws.com/test-driven-swagger:production",
      "essential": true,

      "logConfiguration": {
        "logDriver": "awslogs",
        "options": {
          "awslogs-group": "testdriven-swagger-prod",
          "awslogs-region": "us-west-1"
        }
      },
      "portMappings": [
        {
          "hostPort": 0,
          "protocol": "tcp",
          "containerPort": 8080
        }
      ],
      "environment": [
        {
          "name": "URL",
          "value": "swagger.json"
        }
      ],
      "memoryReservation": 300
    }
  ],
  "family": "testdriven-swagger-prod-td"
}
```

## Travis - update task definition

Start by updating the environment variables for `production` in *docker-push.sh*:

```
if [ "$TRAVIS_BRANCH" == "production" ]
then
  export REACT_APP_USERS_SERVICE_URL="http://LOAD_BALANCER_PROD_DNS_NAME"
  export DATABASE_URL="$AWS_RDS_URI"
  export SECRET_KEY="$PRODUCTION_SECRET_KEY"
fi
```

Add the `AWS_RDS_URI` and `PRODUCTION_SECRET_KEY` environment variables to the Travis project.




To create a key, open the Python shell and run:

```
>>> import binascii
>>> import os
>>> binascii.hexlify(os.urandom(24))
b'958185f1b6ec1290d5aec4eb4dc77e67846ce85cdb7a212a'
```

Next, create a new file called *docker-deploy-prod.sh*:

```
#!/bin/sh

if [ -z "$TRAVIS_PULL_REQUEST" ] || [ "$TRAVIS_PULL_REQUEST" == "false" ]
then

  if [ "$TRAVIS_BRANCH" == "production" ]
  then

    JQ="jq --raw-output --exit-status"

    configure_aws_cli() {
        aws --version
        aws configure set default.region us-west-1
        aws configure set default.output json
        echo "AWS Configured!"
    }

    register_definition() {
      if revision=$(aws ecs register-task-definition --cli-input-json "$task_def" | $JQ '.taskDefinition.taskDefinitionArn'); then
        echo "Revision: $revision"
      else
        echo "Failed to register task definition"
        return 1
      fi
    }
```

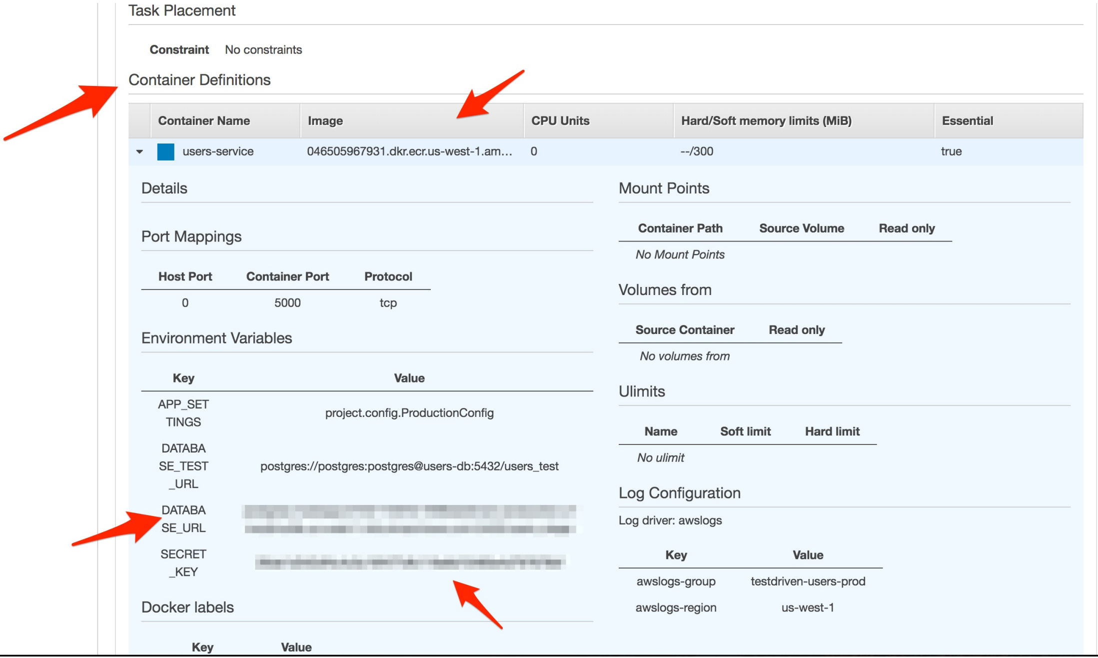
```
    deploy_cluster() {

      # users
      template="ecs_users_prod_taskdefinition.json"
      task_template=$(cat "ecs/$template")
      task_def=$(printf "$task_template" $AWS_ACCOUNT_ID $AWS_RDS_URI $PRODUCTION_S
ECRET_KEY)
      echo "$task_def"
      register_definition

      # client
      template="ecs_client_prod_taskdefinition.json"
      task_template=$(cat "ecs/$template")
      task_def=$(printf "$task_template" $AWS_ACCOUNT_ID)
      echo "$task_def"
      register_definition

      # swagger
      template="ecs_swagger_prod_taskdefinition.json"
      task_template=$(cat "ecs/$template")
      task_def=$(printf "$task_template" $AWS_ACCOUNT_ID)
      echo "$task_def"
      register_definition

    }

    configure_aws_cli
    deploy_cluster

  fi

fi
```

Update the `after_success` step in *.travis.yml*:

```
after_success:
  - bash ./docker-push.sh
  - bash ./docker-deploy-stage.sh
  - bash ./docker-deploy-prod.sh
```

Commit and push your code to GitHub. After the Travis build passes, make sure new images were created and revisions to the Task Definitions were added.

You can ensure that the correct Task Definition JSON files were used to create the revisions by reviewing the latest revisions added to each of the Task Definitions. For example, open the latest revision for `testdriven-users-prod-td` . Under the "Container Definitions", click the drop-down next to the `users` container. Make sure the `DATABASE_URL` and `SECRET_KEY` environment variables are correct:




Next, navigate to the Cluster. Update each of the production Services so that they use the new Task Definitions. After the new instances are spun up and recognized by the Load Balancer, test everything out again manually and with the e2e tests.

## Travis - update service

Update *docker-deploy-prod.sh* to automatically update the Services after new revisions are added to the Task Definitions:

```
#!/bin/sh

if [ -z "$TRAVIS_PULL_REQUEST" ] || [ "$TRAVIS_PULL_REQUEST" == "false" ]
then

  if [ "$TRAVIS_BRANCH" == "production" ]
  then

    JQ="jq --raw-output --exit-status"

    configure_aws_cli() {
        aws --version
        aws configure set default.region us-west-1
        aws configure set default.output json
        echo "AWS Configured!"
    }

    register_definition() {
      if revision=$(aws ecs register-task-definition --cli-input-json "$task_def" |
```

```
$JQ '.taskDefinition.taskDefinitionArn'); then
        echo "Revision: $revision"
      else
        echo "Failed to register task definition"
        return 1
      fi
    }

    update_service() {
      if [[ $(aws ecs update-service --cluster $cluster --service $service --task-d
efinition $revision | $JQ '.service.taskDefinition') != $revision ]]; then
        echo "Error updating service."
        return 1
      fi
    }

    deploy_cluster() {

      cluster="test-driven-production-cluster"

      # users
      service="testdriven-users-prod-service"
      template="ecs_users_prod_taskdefinition.json"
      task_template=$(cat "ecs/$template")
      task_def=$(printf "$task_template" $AWS_ACCOUNT_ID $AWS_RDS_URI $PRODUCTION_S
ECRET_KEY)
      echo "$task_def"
      register_definition
      update_service

      # client
      service="testdriven-client-prod-service"
      template="ecs_client_prod_taskdefinition.json"
      task_template=$(cat "ecs/$template")
      task_def=$(printf "$task_template" $AWS_ACCOUNT_ID)
      echo "$task_def"
      register_definition
      update_service

      # swagger
      service="testdriven-swagger-prod-service"
      template="ecs_swagger_prod_taskdefinition.json"
      task_template=$(cat "ecs/$template")
      task_def=$(printf "$task_template" $AWS_ACCOUNT_ID)
      echo "$task_def"
      register_definition
      update_service

    }

    configure_aws_cli
```

```
    deploy_cluster

  fi

fi
```

> Compare this file to *docker-deploy-stage.sh*. What are the differences?

Be sure to update the `URL` in the *services/swagger/swagger.json* file:

```
$ python services/swagger/update-spec.py http://LOAD_BALANCER_PROD_DNS_NAME
```

Commit and push your code to GitHub to trigger a new Travis build. Once done, you should see a new revision associated with the each Task Definition and the Services should now be running a new Task based on that revision.

Test everything out again!

| Endpoint | HTTP Method | Authenticated? | Result |
|---|---|---|---|
| /auth/register | POST | No | register user |
| /auth/login | POST | No | log in user |
| /auth/logout | GET | Yes | log out user |
| /auth/status | GET | Yes | check user status |
| /users | GET | No | get all users |
| /users/:id | GET | No | get single user |
| /users | POST | Yes (admin) | add a user |
| /users/ping | GET | No | sanity check |

```
/
✓ should display the page correctly if a user is not logged in

/login
✓ should display the sign in form
✓ should allow a user to sign in
✓ should throw an error if the credentials are incorrect

/register
✓ should display flash messages correctly

/register
✓ should display the registration form
✓ should allow a user to register
✓ should validate the password field
✓ should throw an error if the username is taken
```

```
✓ should throw an error if the email is taken

/status
✓ should not display user info if a user is not logged in
✓ should display user info if a user is logged in


12 passed (57s)
```

## Domain Name

Route 53 can be used to link a domain name to the instances running on the Cluster. The setup is fairly simple. Review Routing Traffic to an ELB Load Balancer for more details.

## Docker Machines

Go ahead and spin down the `testdriven-prod` and `testdriven-stage` Docker Machines:

```
$ docker-machine stop testdriven-prod
$ docker-machine stop testdriven-stage
```

Make sure both EC2 instances were brought down as well.

# Workflow

Updated reference guide...

## Development Environment

The following commands are for spinning up all the containers in your `development` environment...

### Docker Machine

Set `testdriven-dev` as the active Docker Machine:

```
$ docker-machine env testdriven-dev
$ eval $(docker-machine env testdriven-dev)
```

### Environment Variables

```
$ export REACT_APP_USERS_SERVICE_URL=http://DOCKER_MACHINE_DEV_IP
$ export TEST_URL=http://DOCKER_MACHINE_DEV_IP
```

### Start

Update *swagger.json*:

```
$ python services/swagger/update-spec.py http://DOCKER_MACHINE_DEV_IP
```

Build the images:

```
$ docker-compose -f docker-compose-dev.yml build
```

Run the containers:

```
$ docker-compose -f docker-compose-dev.yml up -d
```

Create and seed the database:

```
$ docker-compose -f docker-compose-dev.yml \
  run users python manage.py recreate_db

$ docker-compose -f docker-compose-dev.yml \
  run users python manage.py seed_db
```

Run the unit and integration tests:

```
$ docker-compose -f docker-compose-dev.yml \
  run users python manage.py test
```

Lint:

```
$ docker-compose -f docker-compose-dev.yml \
  run users flake8 project
```

Run the client-side tests:

```
$ docker-compose -f docker-compose-dev.yml \
  run client npm test -- --verbose
```

Run the e2e tests:

```
$ testcafe chrome e2e
```

Enter psql:

```
$ docker exec -ti users-db psql -U postgres -W
```

#### Stop

Stop the containers:

```
$ docker-compose -f docker-compose-dev.yml stop
```

Bring down the containers:

```
$ docker-compose -f docker-compose-dev.yml down
```

### Aliases

To save some precious keystrokes, create aliases for both the `docker-compose` and `docker-machine` commands - `dc` and `dm` , respectively.

Simply add the following lines to your *.bashrc* file:

```
alias dc='docker-compose'
alias dm='docker-machine'
```

Save the file, then execute it:

```
$ source ~/.bashrc
```

Test out the new aliases!

> On Windows? You will first need to create a PowerShell Profile (if you don't already have one), and then you can add the aliases to it using Set-Alias - i.e., `Set-Alias dc docker-compose`.

### "Saved" State

Is the VM stuck in a "Saved" state?

```
$ docker-machine ls

NAME    ACTIVE    DRIVER        STATE     URL                         SWARM    DOCKER
      ERRORS
aws      *        amazonec2     Running   tcp://34.207.173.181:2376            v17.05.0
-ce
dev      -        virtualbox    Saved                                          Unknown
```

To break out of this, you'll need to power off the VM:

1. Start virtualbox - `virtualbox`
2. Select the VM and click "start"
3. Exit the VM and select "Power off the machine"
4. Exit virtualbox

The VM should now have a "Stopped" state:

```
$ docker-machine ls

NAME    ACTIVE    DRIVER        STATE     URL                         SWARM    DOCKER
      ERRORS
aws      *        amazonec2     Running   tcp://34.207.173.181:2376            v17.05.0
-ce
dev      -        virtualbox    Stopped                                        Unknown
```

Now you can start the machine:

```
$ docker-machine start dev
```

It should be "Running":

```
$ docker-machine ls

NAME    ACTIVE    DRIVER        STATE     URL                         SWARM    DOCKER
      ERRORS
aws      *        amazonec2     Running   tcp://34.207.173.181:2376            v17.05.0
-ce
dev      -        virtualbox    Running   tcp://192.168.99.100:2376            v17.05.0
```

```
-ce
```

## Other Commands

Want to force a build?

```
$ docker-compose build --no-cache
```

Remove exited containers:

```
$ docker rm -v $(docker ps -a -q -f status=exited)
```

Remove images:

```
$ docker rmi $(docker images -q)
```

Remove untagged images:

```
$ docker rmi $(docker images | grep “^<none>” | awk ‘{print $3}’)
```

Reset Docker environment back to localhost, unsetting all Docker environment variables:

```
$ eval $(docker-machine env -u)
```

## Development Workflow

Try out the following development workflow...

**Development**:

1. Create a new feature branch from the `master` branch
2. Make an arbitrary change; commit and push it up to GitHub
3. After the build passes, open a PR against the `development` branch to trigger a new build on Travis
4. Merge the PR after the build passes

**Staging**:

1. Open PR from the `development` branch against the `staging` branch to trigger a new build on Travis
2. Merge the PR after the build passes to trigger a new build
3. After the build passes, images are created, tagged `staging`, and pushed to ECR, revisions are added to the Task Definitions, and the Service is updated

**Production**:

1. Open PR from the `staging` branch against the `production` branch to trigger a new build on Travis
2. Merge the PR after the build passes to trigger a new build
3. After the build passes, images are created, tagged `production`, and pushed to ECR, revisions are added to the Task Definitions, and the Service is updated
4. Merge the changes into the `master` branch

# Structure

At the end of part 5, your project structure should look like this:

```
├── README.md
├── docker-compose-dev.yml
├── docker-compose-prod.yml
├── docker-compose-stage.yml
├── docker-deploy-prod.sh
├── docker-deploy-stage.sh
├── docker-push.sh
├── e2e
│   ├── index.test.js
│   ├── login.test.js
│   ├── message.test.js
│   ├── register.test.js
│   └── status.test.js
├── ecs
│   ├── ecs_client_prod_taskdefinition.json
│   ├── ecs_client_stage_taskdefinition.json
│   ├── ecs_swagger_prod_taskdefinition.json
│   ├── ecs_swagger_stage_taskdefinition.json
│   ├── ecs_users_prod_taskdefinition.json
│   └── ecs_users_stage_taskdefinition.json
├── package.json
├── services
│   ├── client
│   │   ├── Dockerfile-dev
│   │   ├── Dockerfile-prod
│   │   ├── Dockerfile-stage
│   │   ├── README.md
│   │   ├── build
│   │   ├── conf
│   │   │   └── conf.d
│   │   │       └── default.conf
│   │   ├── coverage
│   │   ├── package.json
│   │   ├── public
│   │   │   ├── favicon.ico
│   │   │   ├── index.html
│   │   │   └── manifest.json
│   │   └── src
│   │       ├── App.jsx
│   │       ├── components
│   │       │   ├── About.jsx
│   │       │   ├── AddUser.jsx
│   │       │   ├── Logout.jsx
│   │       │   ├── Message.jsx
│   │       │   ├── NavBar.jsx
```

```
│   │       │   ├── UserStatus.jsx
│   │       │   ├── UsersList.jsx
│   │       │   ├── __tests__
│   │       │   │   ├── About.test.jsx
│   │       │   │   ├── AddUser.test.jsx
│   │       │   │   ├── App.test.jsx
│   │       │   │   ├── Form.test.jsx
│   │       │   │   ├── FormErrors.test.jsx
│   │       │   │   ├── Logout.test.jsx
│   │       │   │   ├── Message.test.jsx
│   │       │   │   ├── NavBar.test.jsx
│   │       │   │   ├── UsersList.test.jsx
│   │       │   │   └── __snapshots__
│   │       │   │       ├── About.test.jsx.snap
│   │       │   │       ├── AddUser.test.jsx.snap
│   │       │   │       ├── Form.test.jsx.snap
│   │       │   │       ├── FormErrors.test.jsx.snap
│   │       │   │       ├── Logout.test.jsx.snap
│   │       │   │       ├── Message.test.jsx.snap
│   │       │   │       ├── NavBar.test.jsx.snap
│   │       │   │       └── UsersList.test.jsx.snap
│   │       │   └── forms
│   │       │       ├── Form.jsx
│   │       │       ├── FormErrors.css
│   │       │       ├── FormErrors.jsx
│   │       │       └── form-rules.js
│   │       ├── index.js
│   │       ├── logo.svg
│   │       ├── registerServiceWorker.js
│   │       └── setupTests.js
│   ├── nginx
│   │   ├── Dockerfile-dev
│   │   ├── Dockerfile-prod
│   │   ├── Dockerfile-stage
│   │   ├── dev.conf
│   │   └── prod.conf
│   ├── swagger
│   │   ├── Dockerfile-dev
│   │   ├── Dockerfile-prod
│   │   ├── Dockerfile-stage
│   │   ├── nginx.conf
│   │   ├── start.sh
│   │   ├── swagger.json
│   │   └── update-spec.py
│   └── users
│       ├── Dockerfile-dev
│       ├── Dockerfile-prod
│       ├── Dockerfile-stage
│       ├── entrypoint-stage.sh
│       ├── entrypoint.sh
│       ├── htmlcov
```

```
│       ├── manage.py
│       ├── migrations
│       ├── project
│       │   ├── __init__.py
│       │   ├── api
│       │   │   ├── __init__.py
│       │   │   ├── auth.py
│       │   │   ├── models.py
│       │   │   ├── templates
│       │   │   │   └── index.html
│       │   │   ├── users.py
│       │   │   └── utils.py
│       │   ├── config.py
│       │   ├── db
│       │   │   ├── Dockerfile
│       │   │   └── create.sql
│       │   └── tests
│       │       ├── __init__.py
│       │       ├── base.py
│       │       ├── test_auth.py
│       │       ├── test_config.py
│       │       ├── test_user_model.py
│       │       ├── test_users.py
│       │       └── utils.py
│       └── requirements.txt
└── test.sh
```

Code for part 5: https://github.com/testdrivenio/testdriven-app-2.2/releases/tag/part5

# Part 6

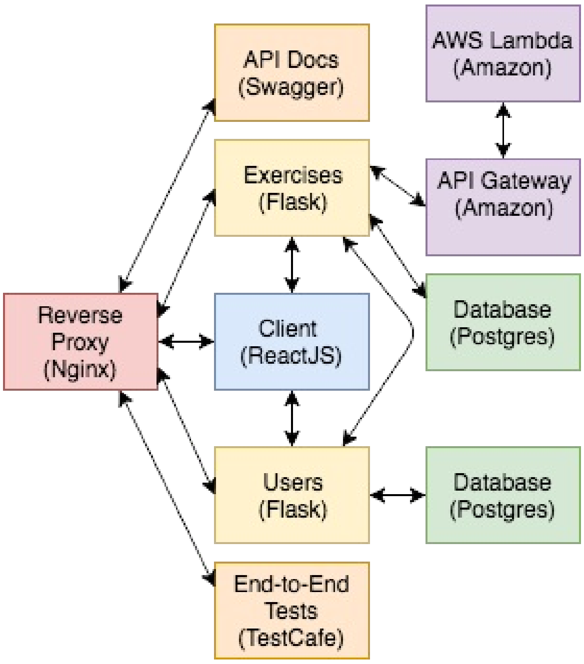
In part 6, we'll focus our attention on adding a new *Flask* service, with two RESTful-resources, to evaluate user-submitted code. Along the way, we'll tie in *AWS Lambda* and *API Gateway* and spend a bit of time refactoring *React* and the *end-to-end* test suite. Finally, we'll update the staging and production environments on ECS.

## Objectives

By the end of part 6, you will be able to...

1. Practice test driven development while refactoring code
2. Integrate a new microservice in the existing set of services
3. Explain what AWS Lambda and API Gateway are and why would would want to use them
4. Develop a RESTful API endpoint with API Gateway that triggers an AWS Lambda function
5. Update the staging and production environments on Amazon ECS

## App




Check out the live apps, running on EC2 -

1. Production
2. Staging

You can also test out the following endpoints...

| Endpoint | HTTP Method | Authenticated? | Result |
| --- | --- | --- | --- |
| /auth/register | POST | No | register user |
| /auth/login | POST | No | log in user |

| | | | |
|---|---|---|---|
| /auth/logout | GET | Yes | log out user |
| /auth/status | GET | Yes | check user status |
| /users | GET | No | get all users |
| /users/:id | GET | No | get single user |
| /users | POST | Yes (admin) | add a user |
| /users/ping | GET | No | sanity check |
| /exercises | GET | No | get all exercises |
| /exercises | POST | Yes (admin) | add an exercise |

Finished code for part 6: https://github.com/testdrivenio/testdriven-app-2.2/releases/tag/part6

## Dependencies

You will use the following dependencies in part 6:

1. React-Ace v5.9.0
2. Flask v0.12.2
3. Flask-SQLAlchemy v2.3.2
4. psycopg2 v2.7.3.2
5. Flask-Testing v0.6.2
6. Gunicorn v19.7.1
7. Coverage.py v4.4.2
8. flake8 v3.5.0
9. Flask Debug Toolbar v0.10.1
10. Flask-CORS v3.0.3
11. Flask-Migrate v2.1.1
12. Requests v2.18.4

# React Refactor

Before we start work on the new services, let's refactor a number of React components...

---

## Docker Machine

Set `testdriven-dev` as the active Docker Machine:

```
$ docker-machine env testdriven-dev
$ eval $(docker-machine env testdriven-dev)
```

Update the `REACT_APP_USERS_SERVICE_URL` environment variable:

```
$ export REACT_APP_USERS_SERVICE_URL=http://DOCKER_MACHINE_DEV_IP
```

Spin up the app:

```
$ docker-compose -f docker-compose-dev.yml up -d --build
```

Update the database:

```
$ docker-compose -f docker-compose-dev.yml \
  run users python manage.py recreate_db

$ docker-compose -f docker-compose-dev.yml \
  run users python manage.py seed_db
```

Ensure the app is working in the browser, and then run the tests:

```
$ docker-compose -f docker-compose-dev.yml \
  run users python manage.py test

$ docker-compose -f docker-compose-dev.yml \
  run users flake8 project

$ docker-compose -f docker-compose-dev.yml \
  run client npm test -- --verbose
```

Set the `TEST_URL` variable for the end-to-end tests:

```
$ export TEST_URL=http://DOCKER_MACHINE_DEV_IP
```

Run the end-to-end tests:

```
$ testcafe chrome e2e
```

## Workflow

For each component, you'll want to follow this workflow:

**Development**:

1. Create a new feature branch from the `master` branch
2. Write tests, ensuring they fail (red)
3. Update code
4. Run the tests again, ensuring they pass (green)
5. Refactor (if necessary)
6. Commit and push your code up to GitHub
7. After the build passes, open a PR against the `development` branch to trigger a new build on Travis
8. Merge the PR after the build passes

**Staging**:

1. Open PR from the `development` branch against the `staging` branch to trigger a new build on Travis
2. Merge the PR after the build passes to trigger a new build
3. After the build passes, images are created, tagged `staging`, and pushed to ECR, revisions are added to the Task Definitions, and the Service is updated

**Production**:

1. Open PR from the `staging` branch against the `production` branch to trigger a new build on Travis
2. Merge the PR after the build passes to trigger a new build
3. After the build passes, images are created, tagged `production`, and pushed to ECR, revisions are added to the Task Definitions, and the Service is updated
4. Merge the changes into the `master` branch

## NavBar

For the `NavBar`, let's update the styles and also add a link to the Swagger docs.

### Test

Update `should display the page correctly if a user is not logged in` in *e2e/index.test.js*:

```
test(`should display the page correctly if a user is not logged in`, async (t) => {
  await t
```

```
    .navigateTo(TEST_URL)
    .expect(Selector('H1').withText('All Users').exists).ok()
    .expect(Selector('a').withText('User Status').exists).notOk()
    .expect(Selector('a').withText('Log Out').exists).notOk()
    .expect(Selector('a').withText('Register').exists).ok()
    .expect(Selector('a').withText('Log In').exists).ok()
    .expect(Selector('a').withText('Swagger').exists).ok()
    .expect(Selector('.alert').exists).notOk()
});
```

Let's also add a new test to assert that the Swagger docs load and reference the correct endpoint URL. Add a new file to "e2e" called *swagger.test.js*:

```
import { Selector } from 'testcafe';

const TEST_URL = process.env.TEST_URL;
const SERVER_URL = process.env.SERVER_URL;


fixture('/swagger').page(`${TEST_URL}/`);

test(`should display the swagger docs correctly`, async (t) => {
  await t
    .navigateTo(TEST_URL)
    .click(Selector('a').withText('Swagger'))
    .expect(Selector('select > option').withText(SERVER_URL).exists).ok()
});
```

Set the environment variable:

```
$ export SERVER_URL=http://DOCKER_MACHINE_DEV_IP
```

Ensure the tests fail.

## Code

Within "services/client/src/components", add a new file called *NavBar.css*:

```
.navbar {
  border-radius: 0;
  color: rgba(255, 255, 255, .5);
}

.navbar-brand {
  color: #777 !important;
}

.navbar-brand:hover {
```

```
  cursor: pointer;
  color: #5e5e5e !important;
}
```

Then, update *NavBar.jsx*:

```
import React from 'react';
import { Navbar, Nav, NavItem } from 'react-bootstrap';
import { LinkContainer } from 'react-router-bootstrap';

import './NavBar.css';

const NavBar = (props) => (
  <Navbar collapseOnSelect>
    <Navbar.Header>
      <Navbar.Brand>
        <LinkContainer to="/">
          <span>{props.title}</span>
        </LinkContainer>
      </Navbar.Brand>
      <Navbar.Toggle />
    </Navbar.Header>
    <Navbar.Collapse>
      <Nav>
        <LinkContainer to="/about">
          <NavItem eventKey={2}>About</NavItem>
        </LinkContainer>
        {props.isAuthenticated &&
          <LinkContainer to="/status">
            <NavItem eventKey={4}>User Status</NavItem>
          </LinkContainer>
        }
      </Nav>
      <ul className="nav navbar-nav">
        <li><a href="/swagger">Swagger</a></li>
      </ul>
      <Nav pullRight>
        {!props.isAuthenticated &&
          <LinkContainer to="/register">
            <NavItem eventKey={1}>Register</NavItem>
          </LinkContainer>
        }
        {!props.isAuthenticated &&
          <LinkContainer to="/login">
            <NavItem eventKey={2}>Log In</NavItem>
          </LinkContainer>
        }
        {props.isAuthenticated &&
          <LinkContainer to="/logout">
            <NavItem eventKey={3}>Log Out</NavItem>
```

```
          </LinkContainer>
        }
      </Nav>
    </Navbar.Collapse>
  </Navbar>
)

export default NavBar;
```

`LinkContainer` does not play nice with external links, so we had to use a regular `<a>` tag with custom styles.

Update *swagger.json*:

```
$ python services/swagger/update-spec.py http://DOCKER_MACHINE_DEV_IP
```

Ensure the tests pass!

## Footer

Next, let's add a basic footer component.

### Test

We'll stick with a unit test for this one, but feel free to update the `should display the page correctly if a user is not logged in` end-to-end test spec as well.

*services/client/src/components/__tests__/Footer.test.js*:

```
import React from 'react';
import { shallow } from 'enzyme';
import renderer from 'react-test-renderer';

import Footer from '../Footer';

test('Footer renders properly', () => {
  const wrapper = shallow(<Footer/>);
  const element = wrapper.find('span');
  expect(element.length).toBe(1);
  expect(element.text()).toBe('Copyright 2018 TestDriven.io.');
});

test('Footer renders a snapshot properly', () => {
  const tree = renderer.create(<Footer/>).toJSON();
  expect(tree).toMatchSnapshot();
});
```

### Code

Add a new file to "src/components" called *Footer.jsx*:

```
import React from 'react';

import './Footer.css';

const Footer = (props) => (
  <footer className="footer">
    <div className="container">
      <small className="text-muted">
        <span>Copyright 2018 <a href="http://testdriven.io">TestDriven.io</a>.</span
>
      </small>
    </div>
  </footer>
)

export default Footer;
```

Create a *Footer.css* file as well:

```
.footer {
  bottom: 0;
  width: 100%;
  height: 50px;
  line-height: 50px;
  margin-top: 50px;
}
```

Add the import to *App.jsx*:

```
import Footer from './components/Footer';
```

Then, add the component in the `render()`, just before the closing `div`:

```
<Footer/>
```

## Users

Next, let's move the `UsersList` component to a new route.

### Test

Update `should display the page correctly if a user is not logged in` from e2e/index.test.js:

```
test(`should display the page correctly if a user is not logged in`, async (t) => {
  await t
    .navigateTo(TEST_URL)
    .expect(Selector('a').withText('User Status').exists).notOk()
    .expect(Selector('a').withText('Log Out').exists).notOk()
    .expect(Selector('a').withText('Register').exists).ok()
    .expect(Selector('a').withText('Log In').exists).ok()
    .expect(Selector('a').withText('Swagger').exists).ok()
    .expect(Selector('a').withText('Users').exists).ok()
    .expect(Selector('.alert').exists).notOk()
});
```

Update `should allow a user to sign in` from *e2e/login.test.js*:

```
test(`should allow a user to sign in`, async (t) => {

  // register user
  await t
    .navigateTo(`${TEST_URL}/register`)
    .typeText('input[name="username"]', username)
    .typeText('input[name="email"]', email)
    .typeText('input[name="password"]', password)
    .click(Selector('input[type="submit"]'))

  // log a user out
  await t
    .click(Selector('a').withText('Log Out'))

  // log a user in
  await t
    .navigateTo(`${TEST_URL}/login`)
    .typeText('input[name="email"]', email)
    .typeText('input[name="password"]', password)
    .click(Selector('input[type="submit"]'))

  // assert user is redirected to '/'
  // assert '/all-users' is displayed properly
  const tableRow = Selector('td').withText(username).parent();
  await t
    .expect(Selector('.alert-success').withText('Welcome!').exists).ok()
    .navigateTo(`${TEST_URL}/all-users`)
    .expect(Selector('H1').withText('All Users').exists).ok()
    .expect(tableRow.child().withText(username).exists).ok()
    .expect(tableRow.child().withText(email).exists).ok()
    .expect(Selector('a').withText('User Status').exists).ok()
    .expect(Selector('a').withText('Log Out').exists).ok()
    .expect(Selector('a').withText('Register').exists).notOk()
    .expect(Selector('a').withText('Log In').exists).notOk()

  // log a user out
```

```
  await t
    .click(Selector('a').withText('Log Out'))

  // assert '/logout' is displayed properly
  await t
    .expect(Selector('p').withText('You are now logged out').exists).ok()
    .expect(Selector('a').withText('User Status').exists).notOk()
    .expect(Selector('a').withText('Log Out').exists).notOk()
    .expect(Selector('a').withText('Register').exists).ok()
    .expect(Selector('a').withText('Log In').exists).ok()

});
```

Update `should allow a user to register` from *e2e/register.test.js*:

```
test(`should allow a user to register`, async (t) => {

   // register user
   await t
     .navigateTo(`${TEST_URL}/register`)
     .typeText('input[name="username"]', username)
     .typeText('input[name="email"]', email)
     .typeText('input[name="password"]', password)
     .click(Selector('input[type="submit"]'))

   // assert user is redirected to '/'
   // assert '/all-users' is displayed properly
   const tableRow = Selector('td').withText(username).parent();
   await t
     .navigateTo(`${TEST_URL}/all-users`)
     .expect(Selector('H1').withText('All Users').exists).ok()
     .expect(tableRow.child().withText(username).exists).ok()
     .expect(tableRow.child().withText(email).exists).ok()
     .expect(Selector('a').withText('User Status').exists).ok()
     .expect(Selector('a').withText('Log Out').exists).ok()
     .expect(Selector('a').withText('Register').exists).notOk()
     .expect(Selector('a').withText('Log In').exists).notOk()

});
```

Add a new test case to a new file called *e2e/users.test.js*:

```
import { Selector } from 'testcafe';

const TEST_URL = process.env.TEST_URL;


fixture('/all-users').page(`${TEST_URL}/all-users`);
```

```
test(`should display the all-users page correctly if a user is not logged in`, async
 (t) => {
  await t
    .navigateTo(`${TEST_URL}/all-users`)
    .expect(Selector('H1').withText('All Users').exists).ok()
    .expect(Selector('a').withText('User Status').exists).notOk()
    .expect(Selector('a').withText('Log Out').exists).notOk()
    .expect(Selector('a').withText('Register').exists).ok()
    .expect(Selector('a').withText('Log In').exists).ok()
    .expect(Selector('a').withText('Swagger').exists).ok()
    .expect(Selector('a').withText('Users').exists).ok()
    .expect(Selector('.alert').exists).notOk()
});
```

### Code

Within the `render()` in *src/App.jsx*, update the main route to:

```
<Route exact path='/' render={() => (
  <p>Something.</p>
)} />
```

Then, create a new route for the `UsersList` component:

```
<Route exact path='/all-users' render={() => (
  <UsersList
    users={this.state.users}
  />
)} />
```

Finally, add a new link just below the `/about` link in *src/components/NavBar.jsx*:

```
<LinkContainer to="/all-users">
  <NavItem>Users</NavItem>
</LinkContainer>
```

## Custom Font

Let's add the Roboto font. First, add the stylesheet to the `head` in *index.html*, just below the Bootstrap stylesheet:

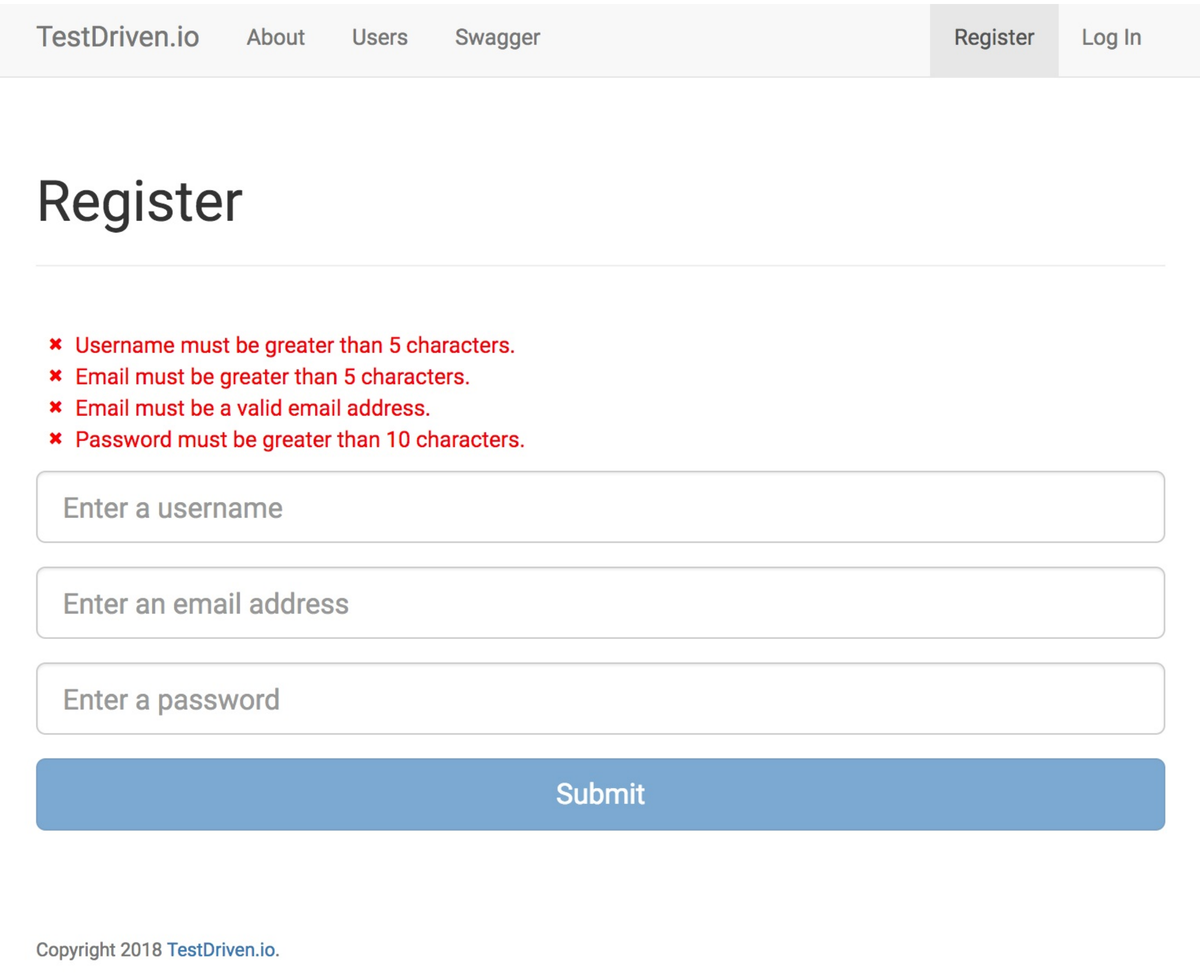
```
<link href="//fonts.googleapis.com/css?family=Roboto" rel="stylesheet">
```

To apply the font, create a new file called *main.css* in the "public" directory:

```
html, body {
  font-family: 'Roboto', sans-serif !important;
}
```

Again, add the stylesheet to the *index.html* file, below the Roboto stylesheet:

```
<link rel="stylesheet" href="main.css">
```




Add more tests as needed. Ensure they are all green before moving on.

```
/
✓ should display the page correctly if a user is not logged in

/login
✓ should display the sign in form
✓ should allow a user to sign in
✓ should throw an error if the credentials are incorrect

/register
```

```
✓ should display flash messages correctly

/register
✓ should display the registration form
✓ should allow a user to register
✓ should validate the password field
✓ should throw an error if the username is taken
✓ should throw an error if the email is taken

/status
✓ should not display user info if a user is not logged in
✓ should display user info if a user is logged in

/swagger
✓ should display the swagger docs correctly

/all-users
✓ should display the all-users page correctly if a user is not logged in


14 passed (56s)
```

```
PASS  src/components/__tests__/App.test.jsx
 ✓ App renders without crashing (6ms)
 ✓ App will call componentWillMount when mounted (26ms)

PASS  src/components/__tests__/NavBar.test.jsx
 ✓ NavBar renders properly (17ms)
 ✓ NavBar renders a snapshot properly (10ms)

PASS  src/components/__tests__/UsersList.test.jsx
 ✓ UsersList renders properly (9ms)
 ✓ UsersList renders a snapshot properly (15ms)

PASS  src/components/__tests__/Form.test.jsx
 When not authenticated
   ✓ register Form renders properly (10ms)
   ✓ register Form submits the form properly (12ms)
   ✓ register Form renders a snapshot properly (6ms)
   ✓ register Form should be disabled by default (1ms)
   ✓ login Form renders properly (3ms)
   ✓ login Form submits the form properly (26ms)
   ✓ login Form renders a snapshot properly (3ms)
   ✓ login Form should be disabled by default (1ms)
 When authenticated
   ✓ register redirects properly (1ms)
   ✓ login redirects properly (1ms)

PASS  src/components/__tests__/Message.test.jsx
 When given a success message
```

```
    ✓ Message renders properly (5ms)
    ✓ Message renders a snapshot properly (1ms)
  When given a danger message
    ✓ Message renders properly (2ms)
    ✓ Message renders a snapshot properly (2ms)

PASS  src/components/__tests__/Footer.test.jsx
  ✓ Footer renders properly (2ms)
  ✓ Footer renders a snapshot properly (1ms)

PASS  src/components/__tests__/AddUser.test.jsx
  ✓ AddUser renders properly (6ms)
  ✓ AddUser renders a snapshot properly (1ms)

PASS  src/components/__tests__/About.test.jsx
  ✓ About renders properly (2ms)
  ✓ About renders a snapshot properly (2ms)

PASS  src/components/__tests__/FormErrors.test.jsx
  ✓ FormErrors (with register form) renders properly (17ms)
  ✓ FormErrors (with register form) renders a snapshot properly (3ms)
  ✓ FormErrors (with login form) renders properly (1ms)
  ✓ FormErrors (with login form) renders a snapshot properly (2ms)

PASS  src/components/__tests__/Logout.test.jsx
  ✓ Logout renders properly (2ms)
  ✓ Logout renders a snapshot properly (3ms)

Test Suites: 10 passed, 10 total
Tests:       32 passed, 32 total
Snapshots:   12 passed, 12 total
Time:        4.959s, estimated 5s
Ran all test suites.
```

# React Ace Code Editor

In this lesson, we'll add code exercises to the client-side using the Ace code editor plugin...

## Exercise Component

Let's add a class-based component to the React app to display the exercises.

### Test

Start by adding a new file in "services/client/src/components/__tests__/" called *Exercises.test.jsx*:

```
import React from 'react';
import { shallow, mount } from 'enzyme';
import renderer from 'react-test-renderer';

import Exercises from '../Exercises';

test('Exercises renders properly', () => {
  const wrapper = shallow(<Exercises/>);
  const element = wrapper.find('h4');
  expect(element.length).toBe(1);
});

test('Exercises renders a snapshot properly', () => {
  const tree = renderer.create(<Exercises/>).toJSON();
  expect(tree).toMatchSnapshot();
});

test('Exercises will call componentWillMount when mounted', () => {
  const onWillMount = jest.fn();
  Exercises.prototype.componentWillMount = onWillMount;
  const wrapper = mount(<Exercises/>);
  expect(onWillMount).toHaveBeenCalledTimes(1)
});
```

### Code

Add a new class-based component called *Exercises.jsx* to "services/client/src/components":

```
import React, { Component } from 'react';

class Exercises extends Component {
  constructor (props) {
    super(props)
    this.state = {};
```

```
  };
  render() {
    return (
      <div>
        <p>Hello, world!</p>
      </div>
    )
  };
};

export default Exercises;
```

Update the main route in *src/App.jsx*:

```
<Route exact path='/' render={() => (
  <Exercises />
)} />
```

Add the import at the top:

```
import Exercises from './components/Exercises';
```

You should see `Hello, world!` in your browser.

Next, we'll need to write up an AJAX call to grab the exercises. Since we don't have the `exercises` service wired up, we'll hard-code some dummy exercises.

Add a new method to the component:

```
getExercises() {
  const exercises = [
    {
      id: 0,
      body: `Define a function called sum that takes
      two integers as arguments and returns their sum.`
    },
    {
      id: 1,
      body: `Define a function called reverse that takes a string
      as an argument and returns the string in reversed order.`
    },
    {
      id: 2,
      body: `Define a function called factorial that takes a random
      number as an argument and then returns the factorial of that
      given number.`,
    }
  ];
  this.setState({ exercises: exercises });
```

```
};
```

Add `exercises` to the state:

```
this.state = {
  exercises: []
};
```

Call `getExercises()` in the `componentDidMount` Lifecycle method:

```
componentDidMount() {
  this.getExercises();
}
```

Finally, update the `render()` :

```
render() {
  return (
    <div>
      <h1>Exercises</h1>
      <hr/><br/>
      {this.state.exercises.length &&
        <div key={this.state.exercises[0].id}>
          <h4>{this.state.exercises[0].body}</h4>
        </div>
      }
    </div>
  )
};
```

Make sure the tests pass before moving on.

## Ace Code Editor

Ace is an embeddable code editor, which we'll use to allow end users to submit their exercise solutions directly in the browser. We'll use a pre-configured component for Ace called React-Ace.

Add Ace to the *services/client/package.json* file:

```
"dependencies": {
  "axios": "^0.17.1",
  "react": "^16.2.0",
  "react-ace": "^5.9.0",
  "react-bootstrap": "^0.32.1",
  "react-dom": "^16.2.0",
  "react-router-bootstrap": "^0.24.4",
  "react-router-dom": "^4.2.2",
```

```
  "react-scripts": "1.1.0"
},
```

Add the imports to *services/client/src/components/Exercises.jsx*:

```
import AceEditor from 'react-ace';
import 'brace/mode/python';
import 'brace/theme/solarized_dark';
```

Update the `render()` :

```
render() {
  return (
    <div>
      <h1>Exercises</h1>
      <hr/><br/>
        {this.state.exercises.length &&
          <div key={this.state.exercises[0].id}>
            <h4>{this.state.exercises[0].body}</h4>
              <AceEditor
                mode="python"
                theme="solarized_dark"
                name={(this.state.exercises[0].id).toString()}
                onLoad={this.onLoad}
                fontSize={14}
                height={'175px'}
                showPrintMargin={true}
                showGutter={true}
                highlightActiveLine={true}
                value={'# Enter your code here.'}
                style={{
                  marginBottom: '10px'
                }}
              />
            <Button bsStyle="primary" bsSize="small">Run Code</Button>
            <br/><br/>
          </div>
        }
    </div>
  )
};
```

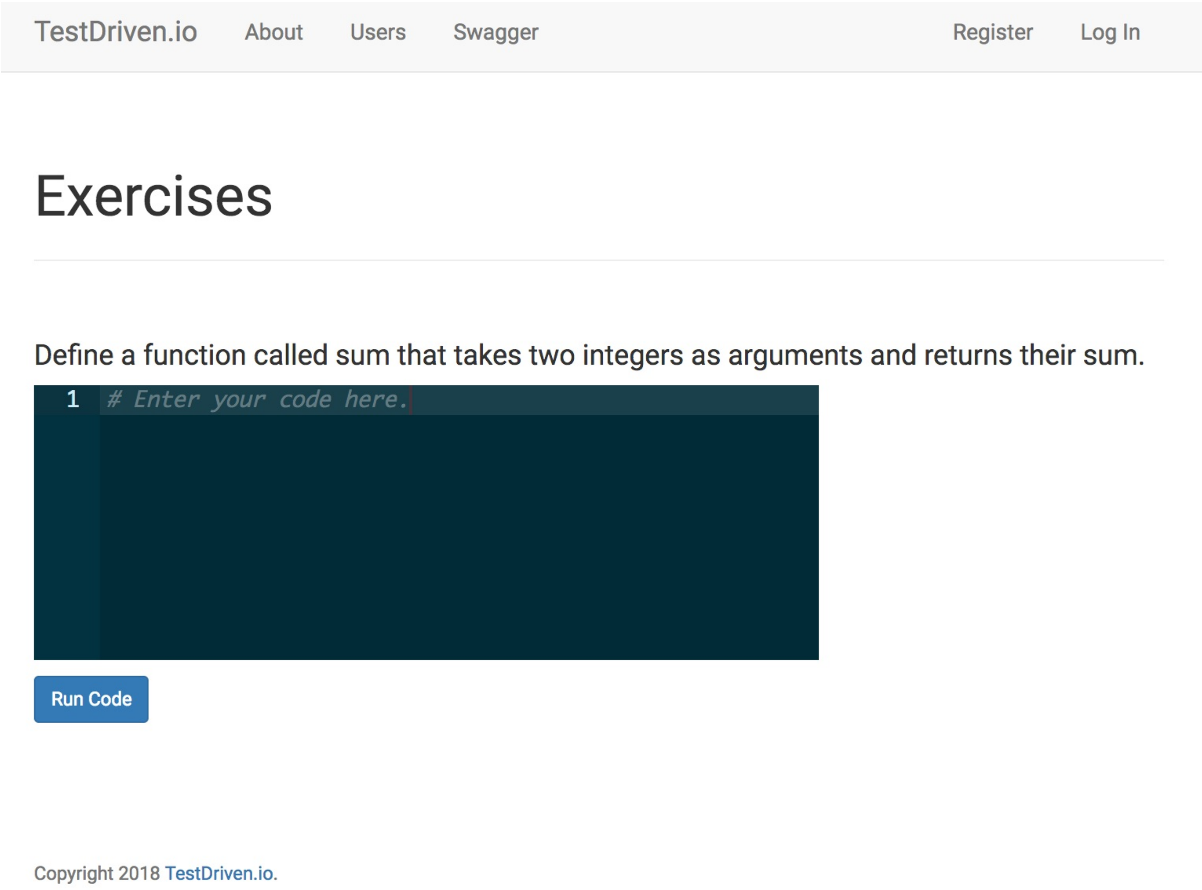
Take note of how we created a new instance of the Ace Editor. Experiment with the available props if you'd like. We also added a Bootstrap-styled button with React Bootstrap. Make sure you add the import:

```
import { Button } from 'react-bootstrap';
```

Update the components:

```
$ docker-compose -f docker-compose-prod.yml up -d --build
```

Jump back to the browser. You should see something similar to:




Run the tests:

```
$ docker-compose -f docker-compose-dev.yml \
  run client npm test -- --verbose
```

You should see a few failures since the `AceEditor` is not being rendered properly in the tests:

```
Cannot find module 'react-ace' from 'Exercises.jsx'
```

Let's mock the entire module.

*Exercises.test.jsx*:

```
import React from 'react';
import { shallow, mount } from 'enzyme';
```

```
import renderer from 'react-test-renderer';

import AceEditor from 'react-ace';
jest.mock('react-ace');

import Exercises from '../Exercises';

test('Exercises renders properly', () => {
  const wrapper = shallow(<Exercises/>);
  const element = wrapper.find('h4');
  expect(element.length).toBe(1);
});

test('Exercises renders a snapshot properly', () => {
  const tree = renderer.create(<Exercises/>).toJSON();
  expect(tree).toMatchSnapshot();
});

test('Exercises will call componentWillMount when mounted', () => {
  const onWillMount = jest.fn();
  Exercises.prototype.componentWillMount = onWillMount;
  const wrapper = mount(<Exercises/>);
  expect(onWillMount).toHaveBeenCalledTimes(1)
});
```

*App.test.jsx*:

```
import React from 'react';
import { shallow, mount } from 'enzyme';
import { MemoryRouter as Router } from 'react-router-dom';
import AceEditor from 'react-ace';
jest.mock('react-ace');

import App from '../../App';

beforeAll(() => {
  global.localStorage = {
    getItem: () => 'someToken'
   };
});

test('App renders without crashing', () => {
  const wrapper = shallow(<App/>);
});

test('App will call componentWillMount when mounted', () => {
  const onWillMount = jest.fn();
  App.prototype.componentWillMount = onWillMount;
  App.prototype.AceEditor = jest.fn();
  const wrapper = mount(<Router><App/></Router>);
```

```
  expect(onWillMount).toHaveBeenCalledTimes(1)
});
```

## Ensure Authenticated

Next, let's only display the button if a user is logged in.

### Test

Update *e2e/index.test.js*:

```
import { Selector } from 'testcafe';

const randomstring = require('randomstring');

const username = randomstring.generate();
const email = `${username}@test.com`;
const password = 'greaterthanten';

const TEST_URL = process.env.TEST_URL;


fixture('/').page(`${TEST_URL}/`);

test(`should display the page correctly if a user is not logged in`, async (t) => {
  await t
    .navigateTo(TEST_URL)
    .expect(Selector('h1').withText('Exercises').exists).ok()
    .expect(Selector('a').withText('User Status').exists).notOk()
    .expect(Selector('a').withText('Log Out').exists).notOk()
    .expect(Selector('a').withText('Register').exists).ok()
    .expect(Selector('a').withText('Log In').exists).ok()
    .expect(Selector('a').withText('Swagger').exists).ok()
    .expect(Selector('a').withText('Users').exists).ok()
    .expect(Selector('button').withText('Run Code').exists).notOk()
    .expect(Selector('.alert-warning').withText(
      'Please log in to submit an exercise.').exists).ok()
});

test(`should display the page correctly if a user is logged in`, async (t) => {
  await t
    .navigateTo(`${TEST_URL}/register`)
    .typeText('input[name="username"]', username)
    .typeText('input[name="email"]', email)
    .typeText('input[name="password"]', password)
    .click(Selector('input[type="submit"]'))
    .navigateTo(TEST_URL)
    .expect(Selector('h1').withText('Exercises').exists).ok()
    .expect(Selector('a').withText('User Status').exists).ok()
```

```
    .expect(Selector('a').withText('Log Out').exists).ok()
    .expect(Selector('a').withText('Register').exists).notOk()
    .expect(Selector('a').withText('Log In').exists).notOk()
    .expect(Selector('a').withText('Swagger').exists).ok()
    .expect(Selector('a').withText('Users').exists).ok()
    .expect(Selector('.alert').exists).notOk()
    .expect(Selector('button').withText('Run Code').exists).ok()
    .expect(Selector('.alert-warning').withText(
      'Please log in to submit an exercise.').exists).notOk()
});
```

Then, update *Exercises.test.jsx*:

```
import React from 'react';
import { shallow, mount } from 'enzyme';
import renderer from 'react-test-renderer';

import AceEditor from 'react-ace';
jest.mock('react-ace');

import Exercises from '../Exercises';

test('Exercises renders properly when not authenticated', () => {
  const wrapper = shallow(<Exercises isAuthenticated={false}/>);
  const heading = wrapper.find('h4');
  expect(heading.length).toBe(1);
  const alert = wrapper.find('.alert');
  expect(alert.length).toBe(1);
  expect(alert.get(0).props.children[1].props.children).toContain(
    'Please log in to submit an exercise.')
});

test('Exercises renders properly when authenticated', () => {
  const wrapper = shallow(<Exercises isAuthenticated={true}/>);
  const heading = wrapper.find('h4');
  expect(heading.length).toBe(1);
  const alert = wrapper.find('.alert');
  expect(alert.length).toBe(0);
});

test('Exercises renders a snapshot properly', () => {
  const tree = renderer.create(<Exercises/>).toJSON();
  expect(tree).toMatchSnapshot();
});

test('Exercises will call componentWillMount when mounted', () => {
  const onWillMount = jest.fn();
  Exercises.prototype.componentWillMount = onWillMount;
  const wrapper = mount(<Exercises/>);
  expect(onWillMount).toHaveBeenCalledTimes(1)
```

```
});
```

### Code

```
{this.props.isAuthenticated &&
  <Button bsStyle="primary" bsSize="small">Run Code</Button>
}
```

Pass the appropriate prop in by updating the route in *src/App.jsx*:

```
<Route exact path='/' render={() => (
  <Exercises
    isAuthenticated={this.state.isAuthenticated}
  />
)} />
```

Let's also add a message for those not logged in. Update the `render()` method in *src/components/Exercises.jsx*

```
render() {
  return (
    <div>
      <h1>Exercises</h1>
      <hr/><br/>
        {!this.props.isAuthenticated &&
          <div>
            <div className="alert alert-warning">
              <span
                className="glyphicon glyphicon-exclamation-sign"
                aria-hidden="true">
              </span>
              <span> Please log in to submit an exercise.</span>
            </div>
            <br/>
          </div>
        }
        {this.state.exercises.length &&
          <div key={this.state.exercises[0].id}>
            <h4>{this.state.exercises[0].body}</h4>
              <AceEditor
                mode="python"
                theme="solarized_dark"
                name={(this.state.exercises[0].id).toString()}
                onLoad={this.onLoad}
                fontSize={14}
                height={'175px'}
                showPrintMargin={true}
                showGutter={true}
```

```
                  highlightActiveLine={true}
                  value={'# Enter your code here.'}
                  style={{
                    marginBottom: '10px'
                  }}
                />
                {this.props.isAuthenticated &&
                  <Button bsStyle="primary" bsSize="small">Run Code</Button>
                }
              <br/><br/>
            </div>
          }
      </div>
    )
  };
```

## Event Handler

Before moving on, let's add two event handlers that will add the actual value to the code editor.

### Code

Start by adding the `value` to the state:

```
this.state = {
  exercises: [],
  editor: {
    value: '# Enter your code here.'
  }
};
```

Then, update the `value` property of the `AceEditor` :

```
value={this.state.editor.value}
```

Next, add an `onChange` prop to the `AceEditor` as well:

```
onChange={this.onChange}
```

Bind it in the constructor:

```
this.onChange = this.onChange.bind(this);
```

onChange, which is an event used to retrieve the current content of the editor, can be used to fire the following function to update the state:

```
onChange(value) {
  this.setState({
    editor: {
      value: value
    }
  });
};
```

Add it, and then add an `onClick` handler to the button:

```
{this.props.isAuthenticated &&
  <Button
    bsStyle="primary"
    bsSize="small"
    onClick={this.submitExercise}
  >Run Code</Button>
}
```

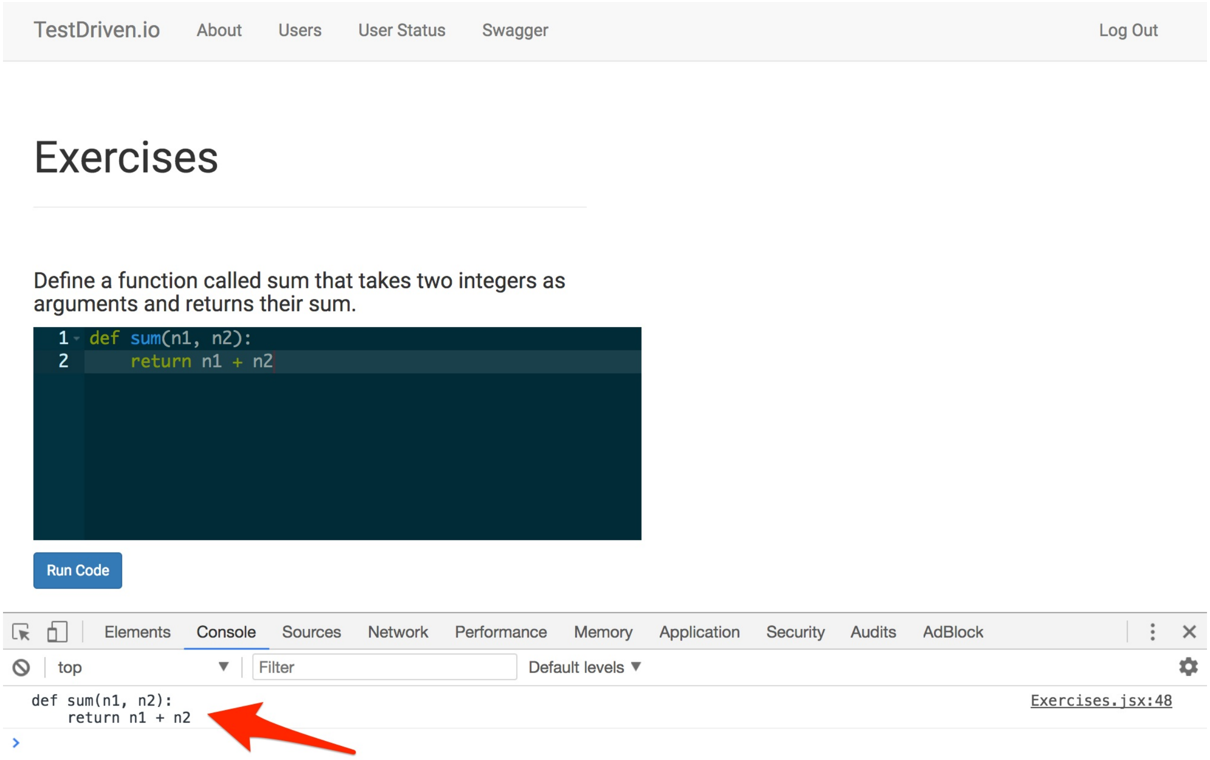
Add the bind to the constructor:

```
this.submitExercise = this.submitExercise.bind(this);
```

Add the `submitExercise` method to the component:

```
submitExercise(event) {
  event.preventDefault();
  console.log(this.state.editor.value);
};
```

We'll just log the value for now. Re-build the containers to manually test.




Commit and push your code to GitHub once done.

# Exercises Service Setup

In this lesson, we'll quickly wire up a new Flask microservice that is responsible for maintaining exercises...

---

From the project root, download the base project directory and unzip it:

```
$ curl https://raw.githubusercontent.com/testdriven/testdriven-app-2.2/master/base.
zip \
  --output base.zip
$ unzip base.zip
```

Open the project in your code editor of choice, and quickly review the code. Then, copy and rename:

```
$ cp -r base services/exercises
```

Add the service to *docker-compose-dev.yml*:

```
exercises:
  container_name: exercises
  build:
    context: ./services/exercises
    dockerfile: Dockerfile-dev
  volumes:
    - './services/exercises:/usr/src/app'
  ports:
    - 5002:5000
  environment:
    - FLASK_DEBUG=1
    - APP_SETTINGS=project.config.DevelopmentConfig
    - USERS_SERVICE_URL=http://users:5000
    - SECRET_KEY=my_precious
  depends_on:
    - users
  links:
    - users
```

Take note of the `USERS_SERVICE_URL` above. Then, jump to the `ensure_authenticated` function in *services/exercises/project/api/utils.py*:

```
def ensure_authenticated(token):
    if current_app.config['TESTING']:
        return True
    url = '{0}/auth/status'.format(current_app.config['USERS_SERVICE_URL'])
```

```
    bearer = 'Bearer {0}'.format(token)
    headers = {'Authorization': bearer}
    response = requests.get(url, headers=headers)
    data = json.loads(response.text)
    if response.status_code == 200 and \
       data['status'] == 'success' and \
       data['data']['active']:
        return data
    else:
        return False
```

In this case, we'll need to make a request from one container to another so we need to reference the container name rather than the Docker Machine IP.

> Did you notice that we simply return `True` in the `ensure_authenticated` function in test mode? It's probably better to mock the `authenticate` function in the test suite to separate the source code from the test code. Refactor on your own.

Next, to spin up the containers, first set `testdriven-dev` as the active machine:

```
$ docker-machine env testdriven-dev
$ eval $(docker-machine env testdriven-dev)
```

Set the environment variables:

```
$ export REACT_APP_USERS_SERVICE_URL=http://DOCKER_MACHINE_DEV_IP
$ export TEST_URL=http://DOCKER_MACHINE_DEV_IP
```

Update *swagger.json*:

```
$ python services/swagger/update-spec.py http://DOCKER_MACHINE_DEV_IP
```

Update the file permissions:

```
$ chmod +x services/exercises/entrypoint.sh
```

Fire up the containers:

```
$ docker-compose -f docker-compose-dev.yml up -d --build
```

Create and seed the database:

```
$ docker-compose -f docker-compose-dev.yml \
  run users python manage.py recreate_db

$ docker-compose -f docker-compose-dev.yml \
```

```
  run users python manage.py seed_db
```

Update *test.sh*:

```
#!/bin/bash

env=$1
file=""
fails=""

if [[ "${env}" == "stage" ]]; then
  file="docker-compose-stage.yml"
elif [[ "${env}" == "dev" ]]; then
  file="docker-compose-dev.yml"
elif [[ "${env}" == "prod" ]]; then
  file="docker-compose-prod.yml"
else
  echo "USAGE: sh test.sh environment_name"
  echo "* environment_name: must either be 'dev', 'stage', or 'prod'"
  exit 1
fi

inspect() {
  if [ $1 -ne 0 ]; then
    fails="${fails} $2"
  fi
}

/bin/sleep 5

docker-compose -f $file run users python manage.py test
inspect $? users
docker-compose -f $file run users flake8 project
inspect $? users-lint

docker-compose -f $file run exercises python manage.py test
inspect $? exercises
docker-compose -f $file run exercises flake8 project
inspect $? exercises-lint

if [[ "${env}" == "dev" ]]; then
  docker-compose -f $file run client npm test -- --coverage
  inspect $? client
  testcafe chrome e2e
  inspect $? e2e
else
  testcafe chrome e2e/index.test.js
  inspect $? e2e
fi
```

```
if [ -n "${fails}" ]; then
  echo "Tests failed: ${fails}"
  exit 1
else
  echo "Tests passed!"
  exit 0
fi
```

Run the tests:

```
$ sh test.sh dev
```

# Exercises Database

In this lesson, we'll test-drive the development of the exercises API starting with the database...

## Tasks

It's highly, highly recommended to implement this *all* on your own. Put your skills to test! The end goal is to set up an `Exercise` model with the following columns:

| Name | Type | Example |
|---|---|---|
| `id` | integer | `1` |
| `body` | string | `Define a function called sum that takes two integers as arguments and returns their sum.` |
| `test_code` | string | `sum(2, 2)` |
| `test_code_solution` | string | `4` |

Steps:

1. Configure Docker Machine
2. Write a test
3. Add SQLAlchemy
4. Update Docker
5. Add the model
6. Update *manage.py*
7. Update *.travis.yml*

## Docker Machine

Set `testdriven-dev` as the active Docker Machine:

```
$ docker-machine env testdriven-dev
$ eval $(docker-machine env testdriven-dev)
```

Set the environment variables:

```
$ export REACT_APP_USERS_SERVICE_URL=http://DOCKER_MACHINE_DEV_IP
$ export TEST_URL=http://DOCKER_MACHINE_DEV_IP
```

Update *swagger.json*:

```
$ python services/swagger/update-spec.py http://DOCKER_MACHINE_DEV_IP
```

Fire up the containers:

```
$ docker-compose -f docker-compose-dev.yml up -d --build
```

Create and seed the database:

```
$ docker-compose -f docker-compose-dev.yml \
  run users python manage.py recreate_db

$ docker-compose -f docker-compose-dev.yml \
  run users python manage.py seed_db
```

Run the tests:

```
$ sh test.sh dev
```

## Write a test

Add a new file called *test_exercises_model.py* to "services/exercises/project/tests":

```
# services/exercises/project/tests/test_exercises_model.py


from project.tests.base import BaseTestCase
from project.tests.utils import add_exercise


class TestExerciseModel(BaseTestCase):

    def test_add_exercise(self):
        exercise = add_exercise()
        self.assertTrue(exercise.id)
        self.assertTrue(exercise.body)
        self.assertEqual(exercise.test_code, 'sum(2, 2)')
        self.assertEqual(exercise.test_code_solution, '4')
```

Then, create the *utils.py* file as well in "services/exercises/project/tests":

```
# services/exercises/project/tests/utils.py


from project import db
from project.api.models import Exercise


def add_exercise(
        body=('Define a function called sum that takes two integers as '
```

```
                'arguments and returns their sum'),
            test_code='sum(2, 2)',
            test_code_solution='4'):
    exercise = Exercise(
        body=body,
        test_code=test_code,
        test_code_solution=test_code_solution,
    )
    db.session.add(exercise)
    db.session.commit()
    return exercise
```

Ensure the tests fail before moving on:

```
$ docker-compose -f docker-compose-dev.yml \
  run exercises python manage.py test
```

## Add SQLAlchemy

Update *services/exercises/project/config.py*:

```
# services/exercises/project/config.py


import os


class BaseConfig:
    """Base configuration"""
    TESTING = False
    DEBUG_TB_ENABLED = False
    DEBUG_TB_INTERCEPT_REDIRECTS = False
    USERS_SERVICE_URL = os.environ.get('USERS_SERVICE_URL')
    SECRET_KEY = os.environ.get('SECRET_KEY')
    SQLALCHEMY_TRACK_MODIFICATIONS = False


class DevelopmentConfig(BaseConfig):
    """Development configuration"""
    DEBUG_TB_ENABLED = True
    SQLALCHEMY_DATABASE_URI = os.environ.get('DATABASE_URL')


class TestingConfig(BaseConfig):
    """Testing configuration"""
    TESTING = True
    SQLALCHEMY_DATABASE_URI = os.environ.get('DATABASE_TEST_URL')
```

```
class StagingConfig(BaseConfig):
    """Staging configuration"""
    SQLALCHEMY_DATABASE_URI = os.environ.get('DATABASE_URL')


class ProductionConfig(BaseConfig):
    """Production configuration"""
    SQLALCHEMY_DATABASE_URI = os.environ.get('DATABASE_URL')
```

Update *services/exercises/project/__init__.py* to create new instances of SQLAlchemy and Flask-Migrate:

```
# services/exercises/project/__init__.py


import os

from flask import Flask
from flask_cors import CORS
from flask_debugtoolbar import DebugToolbarExtension
from flask_sqlalchemy import SQLAlchemy
from flask_migrate import Migrate


# instantiate the extensions
db = SQLAlchemy()
migrate = Migrate()
toolbar = DebugToolbarExtension()


def create_app(script_info=None):

    # instantiate the app
    app = Flask(__name__)

    # enable CORS
    CORS(app)

    # set config
    app_settings = os.getenv('APP_SETTINGS')
    app.config.from_object(app_settings)

    # set up extensions
    toolbar.init_app(app)
    db.init_app(app)
    migrate.init_app(app, db)

    # register blueprints
    from project.api.base import base_blueprint
    app.register_blueprint(base_blueprint)
```

```
    # shell context for flask cli
    app.shell_context_processor({'app': app, 'db': db})
    return app
```

Next, update *services/exercises/project/tests/base.py* to create the database in the `setUp()` and then drop it in the `tearDown()` :

```
# services/exercises/project/tests/base.py


from flask_testing import TestCase

from project import create_app, db

app = create_app()


class BaseTestCase(TestCase):
    def create_app(self):
        app.config.from_object('project.config.TestingConfig')
        return app

    def setUp(self):
        db.create_all()
        db.session.commit()

    def tearDown(self):
        db.session.remove()
        db.drop_all()
```

Finally, update *services/exercises/project/tests/test_config.py*:

```
# services/exercises/project/tests/test_config.py


import unittest

from flask import current_app
from flask_testing import TestCase

from project import create_app

app = create_app()


class TestDevelopmentConfig(TestCase):
    def create_app(self):
        app.config.from_object('project.config.DevelopmentConfig')
```

```
        return app

    def test_app_is_development(self):
        self.assertFalse(current_app is None)


class TestTestingConfig(TestCase):
    def create_app(self):
        app.config.from_object('project.config.TestingConfig')
        return app

    def test_app_is_testing(self):
        self.assertTrue(app.config['TESTING'])
        self.assertFalse(app.config['PRESERVE_CONTEXT_ON_EXCEPTION'])


class TestProductionConfig(TestCase):
    def create_app(self):
        app.config.from_object('project.config.ProductionConfig')
        return app

    def test_app_is_production(self):
        self.assertFalse(app.config['TESTING'])


if __name__ == '__main__':
    unittest.main()
```

## Update Docker

Add a "db" directory to "services/exercises/project", and then add a *create.sql* file in "db":

```
CREATE DATABASE exercises_prod;
CREATE DATABASE exercises_stage;
CREATE DATABASE exercises_dev;
CREATE DATABASE exercises_test;
```

Add a *Dockerfile* to the "db" directory as well:

```
FROM postgres

# run create.sql on init
ADD create.sql /docker-entrypoint-initdb.d
```

Then, add the service to *docker-compose-dev.yml*:

```
exercises-db:
  container_name: exercises-db
```

```
  build:
    context: ./services/exercises/project/db
    dockerfile: Dockerfile
  ports:
    - 5436:5432
  environment:
    - POSTGRES_USER=postgres
    - POSTGRES_PASSWORD=postgres
```

Update the `exercises` service to link the `exercises-db` to it and set the `DATABASE_URL` and `DATABASE_TEST_URL` environment variables:

```
exercises:
  container_name: exercises
  build:
    context: ./services/exercises
    dockerfile: Dockerfile-dev
  volumes:
    - './services/exercises:/usr/src/app'
  ports:
    - 5002:5000
  environment:
    - FLASK_DEBUG=1
    - APP_SETTINGS=project.config.DevelopmentConfig
    - USERS_SERVICE_URL=http://users:5000
    - SECRET_KEY=my_precious
    - DATABASE_URL=postgres://postgres:postgres@exercises-db:5432/exercises_dev
    - DATABASE_TEST_URL=postgres://postgres:postgres@exercises-db:5432/exercises_te
st
  depends_on:
    - users
    - exercises-db
  links:
    - users
    - exercises-db
```

Then, update *services/exercises/entrypoint.sh*:

```
#!/bin/sh

echo "Waiting for postgres..."

while ! nc -z exercises-db 5432; do
  sleep 0.1
done

echo "PostgreSQL started"

python manage.py run -h 0.0.0.0
```

## Add the model

Add a new file to "services/exercises/project/api" called *models.py*:

```
# services/exercises/project/api/models.py


from project import db


class Exercise(db.Model):
    __tablename__ = "exercises"
    id = db.Column(db.Integer, primary_key=True, autoincrement=True)
    body = db.Column(db.String, nullable=False)
    test_code = db.Column(db.String, nullable=False)
    test_code_solution = db.Column(db.String, nullable=False)

    def __init__(self, body, test_code, test_code_solution):
        self.body = body
        self.test_code = test_code
        self.test_code_solution = test_code_solution

    def to_json(self):
        return {
            'id': self.id,
            'body': self.body,
            'test_code': self.test_code,
            'test_code_solution': self.test_code_solution
        }
```

Run the tests to ensure they pass:

```
$ docker-compose -f docker-compose-dev.yml \
  run exercises python manage.py test
```

## Update *manage.py*

Now, let's update *manage.py*:

```
# services/exercises/manage.py


import unittest
import coverage

from flask.cli import FlaskGroup
```

```python
from project import create_app, db
from project.api.models import Exercise


app = create_app()
cli = FlaskGroup(create_app=create_app)

COV = coverage.coverage(
    branch=True,
    include='project/*',
    omit=[
        'project/tests/*',
        'project/config.py',
    ]
)
COV.start()


@cli.command()
def test():
    """ Runs the tests without code coverage"""
    tests = unittest.TestLoader().discover('project/tests', pattern='test*.py')
    result = unittest.TextTestRunner(verbosity=2).run(tests)
    if result.wasSuccessful():
        return 0
    return 1


@cli.command()
def cov():
    """Runs the unit tests with coverage."""
    tests = unittest.TestLoader().discover('project/tests')
    result = unittest.TextTestRunner(verbosity=2).run(tests)
    if result.wasSuccessful():
        COV.stop()
        COV.save()
        print('Coverage Summary:')
        COV.report()
        COV.html_report()
        COV.erase()
        return 0
    return 1


@cli.command()
def recreate_db():
    db.drop_all()
    db.create_all()
    db.session.commit()
```

```
if __name__ == '__main__':
    cli()
```

Apply the model to the dev database:

```
$ docker-compose -f docker-compose-dev.yml \
  run exercises python manage.py recreate_db
```

Did it work?

```
$ docker exec -ti $(docker ps -aqf "name=exercises-db") psql -U postgres

# \c exercises_dev
You are now connected to database "exercises_dev" as user "postgres".

# \dt
          List of relations
Schema |    Name    | Type  |  Owner
--------+-----------+-------+----------
public | exercises | table | postgres
(1 row)

# \q
```

Commit. Push your code to GitHub. Ensure the Travis build passes.

# Exercises API

In this lesson, we'll add an API to the exercises service...

---

## Tasks

Again, try this on our own to check your understanding.

### Routes

| Endpoint | HTTP Method | Authenticated? | Result |
|---|---|---|---|
| /exercises | GET | No | get all exercises |
| /exercises | POST | Yes (admin) | add an exercise |

Process:

1. write a test
2. run the test to ensure it fails (**red**)
3. write just enough code to get the test to pass (**green**)
4. **refactor** (if necessary)

Files:

1. Test - *services/exercises/project/tests/test_exercises_api.py*
2. API - *services/exercises/project/api/exercises.py*

## GET all exercises

Test:

```
# services/exercises/project/tests/test_exercises_api.py


import json
import unittest

from project.tests.base import BaseTestCase
from project.tests.utils import add_exercise


class TestExercisesService(BaseTestCase):
    """Tests for the Exercises Service."""

    def test_all_exercises(self):
        """Ensure get all exercises behaves correctly."""
```

```
        add_exercise()
        add_exercise(
            'Just a sample', 'print("Hello, World!")', 'Hello, World!')
        with self.client:
            response = self.client.get('/exercises')
            data = json.loads(response.data.decode())
            self.assertEqual(response.status_code, 200)
            self.assertEqual(len(data['data']['exercises']), 2)
            self.assertIn(
                'Define a function called sum',
                data['data']['exercises'][0]['body'])
            self.assertEqual(
                'Just a sample',
                data['data']['exercises'][1]['body'])
            self.assertEqual(
                'sum(2, 2)', data['data']['exercises'][0]['test_code'])
            self.assertEqual(
                'print("Hello, World!")',
                data['data']['exercises'][1]['test_code'])
            self.assertEqual(
                '4', data['data']['exercises'][0]['test_code_solution'])
            self.assertEqual(
                'Hello, World!',
                data['data']['exercises'][1]['test_code_solution'])
            self.assertIn('success', data['status'])


if __name__ == '__main__':
    unittest.main()
```

Route:

```
# services/exercises/project/api/exercises.py


from sqlalchemy import exc
from flask import Blueprint, jsonify, request

from project import db
from project.api.models import Exercise
from project.api.utils import authenticate


exercises_blueprint = Blueprint('exercises', __name__)


@exercises_blueprint.route('/exercises', methods=['GET'])
def get_all_exercises():
    """Get all exercises"""
    response_object = {
```

```
        'status': 'success',
        'data': {
            'exercises': [ex.to_json() for ex in Exercise.query.all()]
        }
    }
    return jsonify(response_object), 200
```

Be sure to wire up the Blueprint in *services/exercises/project/__init__.py*:

```
from project.api.exercises import exercises_blueprint
app.register_blueprint(exercises_blueprint)
```

## POST

Tests:

```
def test_add_exercise(self):
    """Ensure a new exercise can be added to the database."""
    with self.client:
        response = self.client.post(
            '/exercises',
            data=json.dumps({
                'body': 'Sample sample',
                'test_code': 'get_sum(2, 2)',
                'test_code_solution': '4',
            }),
            content_type='application/json',
            headers=({'Authorization': 'Bearer test'})
        )
        data = json.loads(response.data.decode())
        self.assertEqual(response.status_code, 201)
        self.assertIn('New exercise was added!', data['message'])
        self.assertIn('success', data['status'])

def test_add_exercise_invalid_json(self):
    """Ensure error is thrown if the JSON object is empty."""
    with self.client:
        response = self.client.post(
            '/exercises',
            data=json.dumps({}),
            content_type='application/json',
            headers=({'Authorization': 'Bearer test'})
        )
        data = json.loads(response.data.decode())
        self.assertEqual(response.status_code, 400)
        self.assertIn('Invalid payload.', data['message'])
        self.assertIn('fail', data['status'])

def test_add_exercise_invalid_json_keys(self):
```

```
    """Ensure error is thrown if the JSON object is invalid."""
    with self.client:
        response = self.client.post(
            '/exercises',
            data=json.dumps({'body': 'test'}),
            content_type='application/json',
            headers=({'Authorization': 'Bearer test'})
        )
        data = json.loads(response.data.decode())
        self.assertEqual(response.status_code, 400)
        self.assertIn('Invalid payload.', data['message'])
        self.assertIn('fail', data['status'])

def test_add_exercise_no_header(self):
    """Ensure error is thrown if 'Authorization' header is empty."""
    response = self.client.post(
        '/exercises',
        data=json.dumps({
            'body': 'Sample sample',
            'test_code': 'get_sum(2, 2)',
            'test_code_solution': '4',
        }),
        content_type='application/json'
    )
    data = json.loads(response.data.decode())
    self.assertEqual(response.status_code, 403)
    self.assertIn('Provide a valid auth token.', data['message'])
    self.assertIn('error', data['status'])
```

Route:

```
@exercises_blueprint.route('/exercises', methods=['POST'])
@authenticate
def add_exercise(resp):
    """Add exercise"""
    if not resp['admin']:
        response_object = {
            'status': 'error',
            'message': 'You do not have permission to do that.'
        }
        return jsonify(response_object), 401
    post_data = request.get_json()
    if not post_data:
        response_object = {
            'status': 'fail',
            'message': 'Invalid payload.'
        }
        return jsonify(response_object), 400
    body = post_data.get('body')
    test_code = post_data.get('test_code')
```

```
    test_code_solution = post_data.get('test_code_solution')
    try:
        db.session.add(Exercise(
            body=body,
            test_code=test_code,
            test_code_solution=test_code_solution))
        db.session.commit()
        response_object = {
            'status': 'success',
            'message': 'New exercise was added!'
        }
        return jsonify(response_object), 201
    except (exc.IntegrityError, ValueError) as e:
        db.session().rollback()
        response_object = {
            'status': 'fail',
            'message': 'Invalid payload.'
        }
        return jsonify(response_object), 400
```

Update the `ensure_authenticated` function in *project/api/utils.py* as well:

```
def ensure_authenticated(token):
    if current_app.config['TESTING']:
        test_response = {
            'data': {'id': 998877},
            'status': 'success',
            'admin': True
        }
        return test_response
    url = '{0}/auth/status'.format(current_app.config['USERS_SERVICE_URL'])
    bearer = 'Bearer {0}'.format(token)
    headers = {'Authorization': bearer}
    response = requests.get(url, headers=headers)
    data = json.loads(response.text)
    if response.status_code == 200 and \
       data['status'] == 'success' and \
       data['data']['active']:
        print(data)
        return data
    else:
        return False
```

Instead of returning `True`, we are now returning a test object. So, there's even more test code polluting the source code. Refactor this!

## Sanity Check

Set `testdriven-dev` as the active Docker Machine:

```
$ docker-machine env testdriven-dev
$ eval $(docker-machine env testdriven-dev)
```

Set the environment variables:

```
$ export REACT_APP_USERS_SERVICE_URL=http://DOCKER_MACHINE_DEV_IP
$ export TEST_URL=http://DOCKER_MACHINE_DEV_IP
```

Update *swagger.json*:

```
$ python services/swagger/update-spec.py http://DOCKER_MACHINE_DEV_IP
```

Fire up the containers:

```
$ docker-compose -f docker-compose-dev.yml up -d --build
```

Create and seed the database:

```
$ docker-compose -f docker-compose-dev.yml \
  run exercises python manage.py recreate_db

$ docker-compose -f docker-compose-dev.yml \
  run users python manage.py recreate_db

$ docker-compose -f docker-compose-dev.yml \
  run users python manage.py seed_db
```

Run the tests:

```
$ sh test.sh dev
```

How about test coverage?

```
$ docker-compose -f docker-compose-dev.yml \
  run exercises python manage.py cov

OK
Coverage Summary:
Name                        Stmts   Miss Branch BrPart  Cover
-------------------------------------------------------------
project/__init__.py            23     10      0      0    57%
project/api/base.py             6      5      0      0    17%
project/api/exercises.py       30     11      6      1    67%
project/api/models.py          13      9      0      0    31%
```

```
project/api/utils.py          33      19       8       2    44%
------------------------------------------------------------
TOTAL                        105      54      14       3    50%
```

Write additional routes and tests as needed. Once done, commit and push your code to GitHub.

# Code Evaluation with AWS Lambda

In this lesson, we'll set up a RESTful API with AWS Lambda and API Gateway to handle code evaluation...

---

It's a good idea to move long-running processes (like code evaluation) outside of the direct HTTP request/response cycle to improve performance of the web app. This is typically handled by Redis or RabbitMQ along with Celery. We're going to take a different approach with AWS Lambda.

With AWS Lambda, we can run scripts without having to provision or manage servers in response to an HTTP POST request.

## What is AWS Lambda?

Amazon Web Services (AWS) Lambda is an on-demand compute service that lets you run code in response to events or HTTP requests.

Use cases:

| Event | Action |
| --- | --- |
| Image added to S3 | Image is processed |
| HTTP Request via API Gateway | HTTP Response |
| Log file added to Cloudwatch | Analyze the log |
| Scheduled event | Back up files |
| Scheduled event | Synchronization of files |

For more examples, review the Examples of How to Use AWS Lambda guide from AWS.

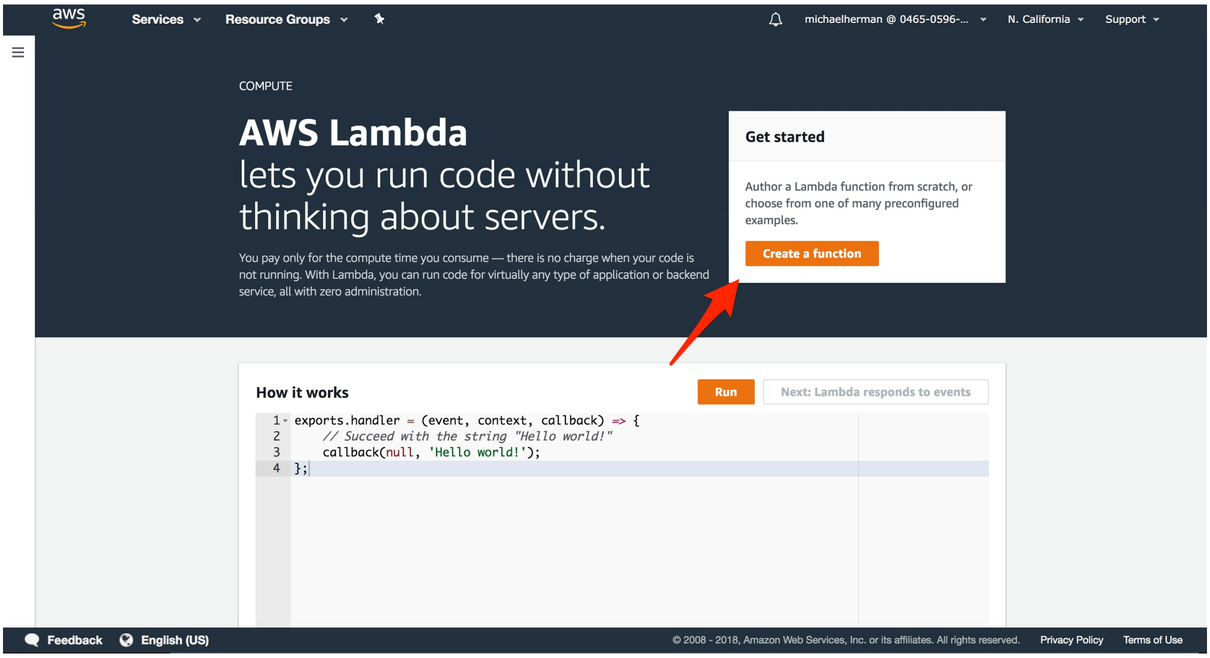
You can run scripts and apps without having to provision or manage servers in a seemingly infinitely-scalable environment where you pay only for usage. This is "serverless" computing in a nut shell. For our purposes, AWS Lambda is a perfect solution for running user-supplied code quickly, securely, and cheaply.

As of writing, Lambda supports code written in JavaScript (Node.js), Python, Java, GO, and C#.

We'll start by simply setting up an HTTP endpoint with API Gateway, which is used to trigger the Lambda function. Keep in mind that you would probably want to set up a message queuing service, like Redis or SQS, as well. To quickly get up and running we'll skip the message queue... for now.

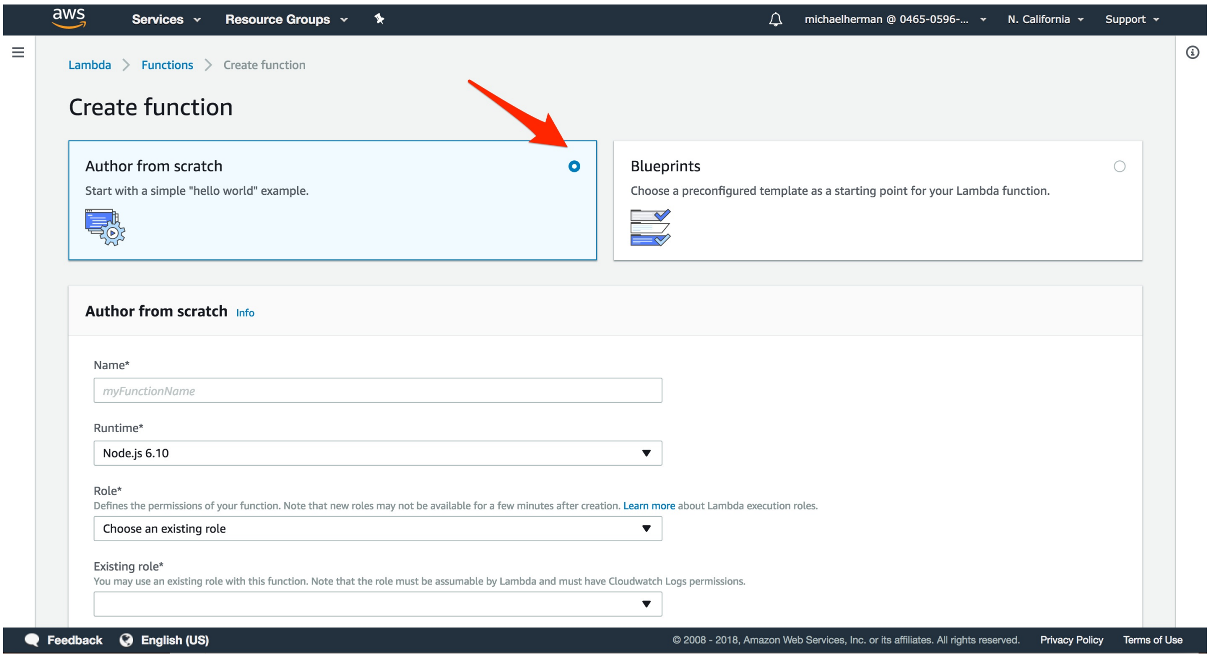
## AWS Lambda Setup

Within the AWS Console, navigate to the main Lambda page and click "Create a function":

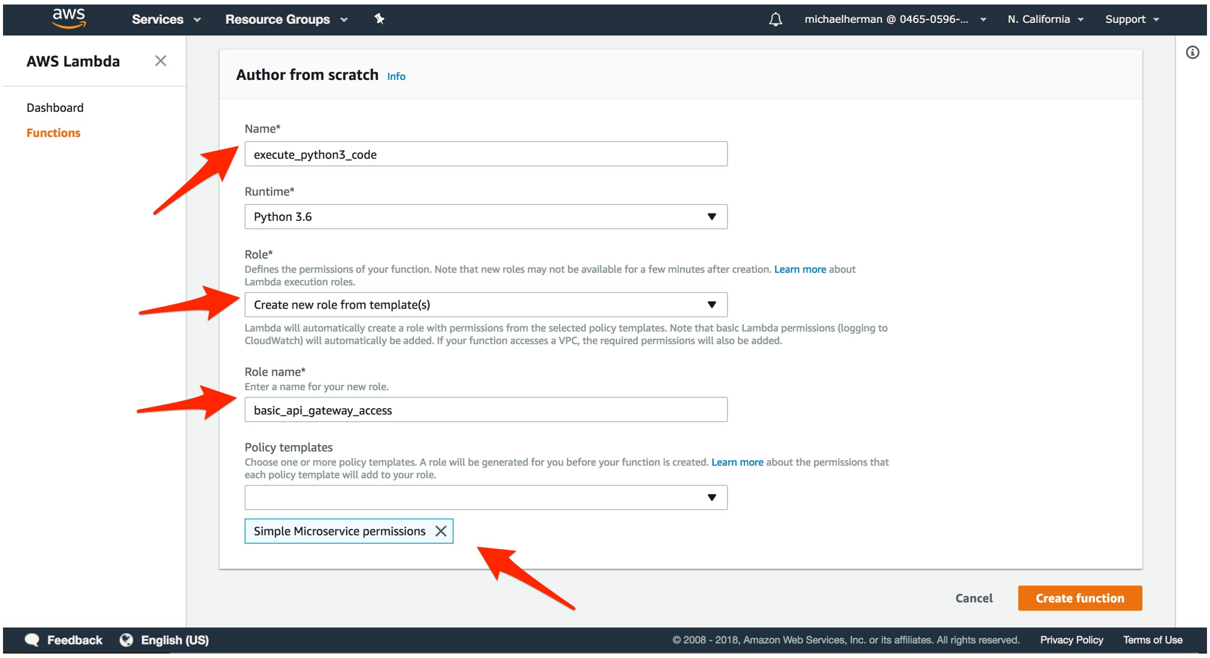



## Create function

Click the "Author from scratch" pane to start with a blank function:




Name the function `execute_python3_code`, select "Python 3.6" in the "Runtime" drop-down, and select "Create a new Role from template(s)" from the drop-down for the Role. Then, enter `basic_api_gateway_access` for the "Role name". To provide access to the API Gateway, select "Simple Microservice permissions" for the "Policy templates".

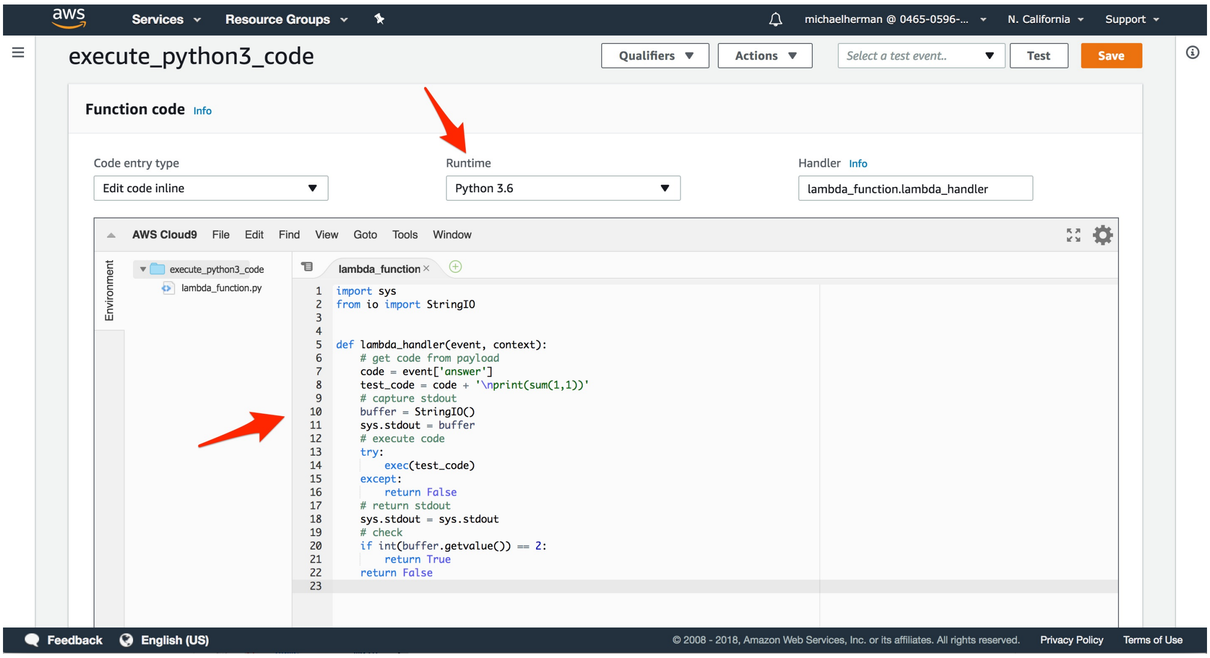



Click "Create function".

## Function code

Within the inline code editor, update the `lambda_handler` function definition with:

```
import sys
from io import StringIO


def lambda_handler(event, context):
    # get code from payload
    code = event['answer']
    test_code = code + '\nprint(sum(1,1))'
    # capture stdout
    buffer = StringIO()
    sys.stdout = buffer
    # execute code
    try:
        exec(test_code)
    except:
        return False
    # return stdout
    sys.stdout = sys.stdout
    # check
    if int(buffer.getvalue()) == 2:
        return True
    return False
```




Here, within `lambda_handler`, which is the default entry point for Lambda, we parse the JSON request body, passing the supplied code along with some test code – `sum(1,1)` – to the exec function – which executes the string as Python code. Then, we simply ensure the actual results are the same as what's expected – e.g., `2` – and return the appropriate response.

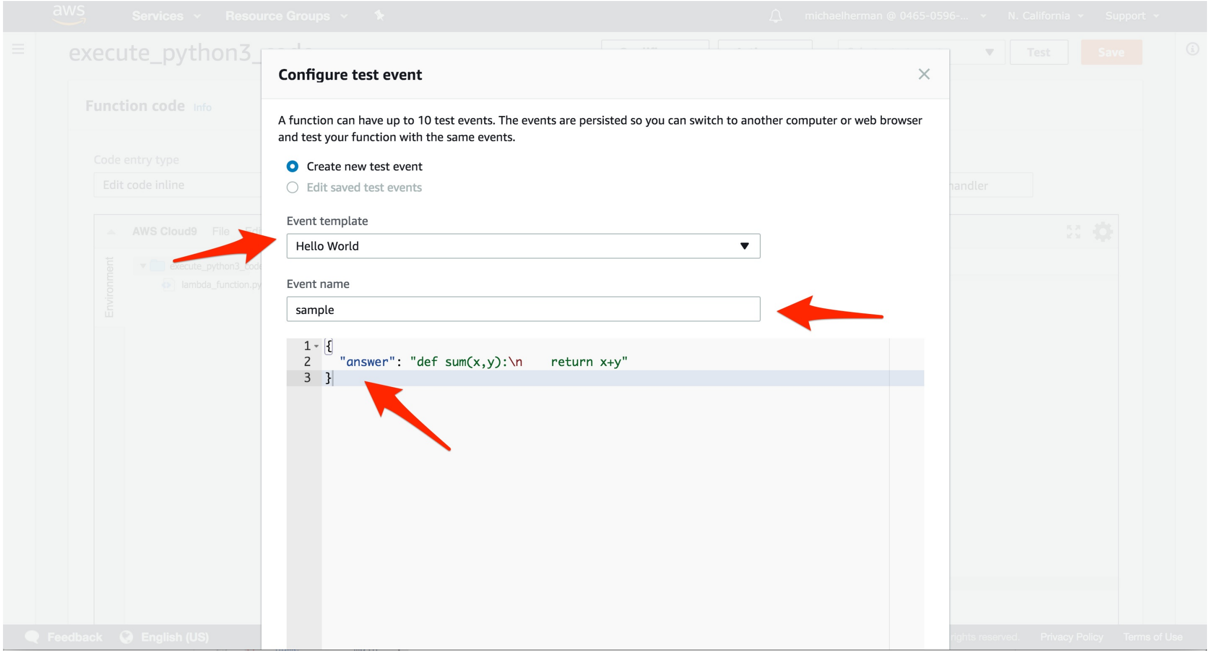
Save a copy of the `lambda_handler` function in *services/lambda/handler.py*.

## Test

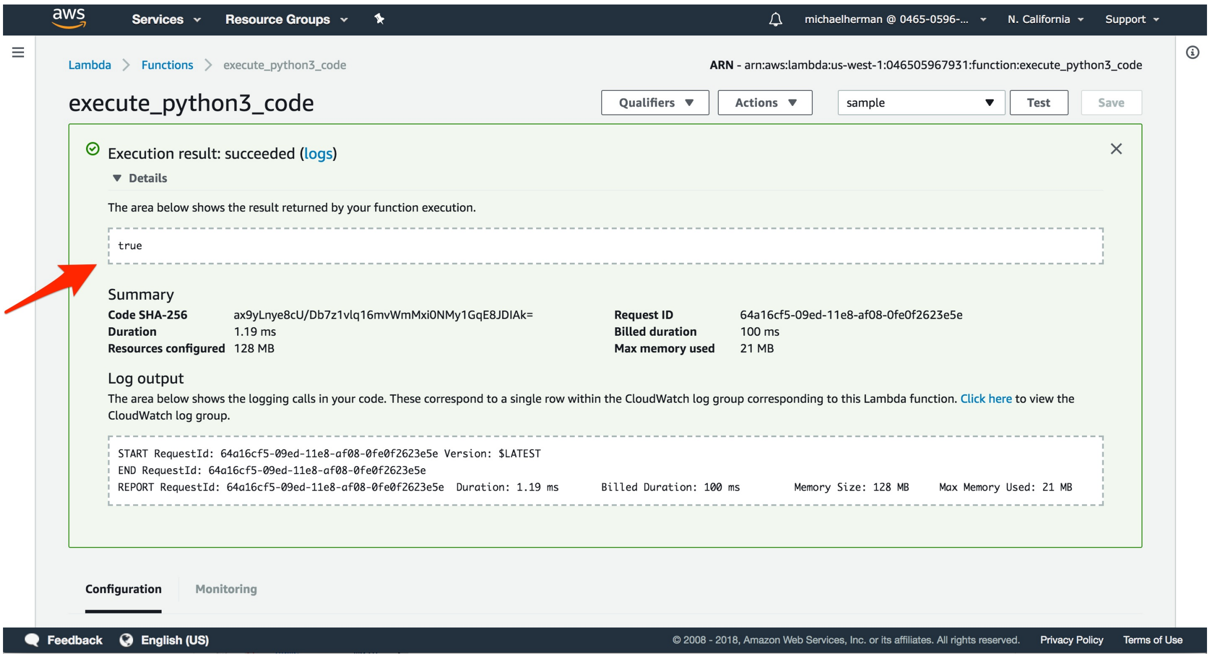
Next click on the "Test" button to execute the newly created Lambda. Using the "Hello World" event template, replace the sample with:

```
{
  "answer": "def sum(x,y):\n    return x+y"
}
```

Use `sample` for the event name.




Click the "Create" button at the bottom of the modal to run the test. Once done, you should see something similar to:




With that, we can move on to configuring the API Gateway to trigger the Lambda from user-submitted POST requests...

## API Gateway Setup

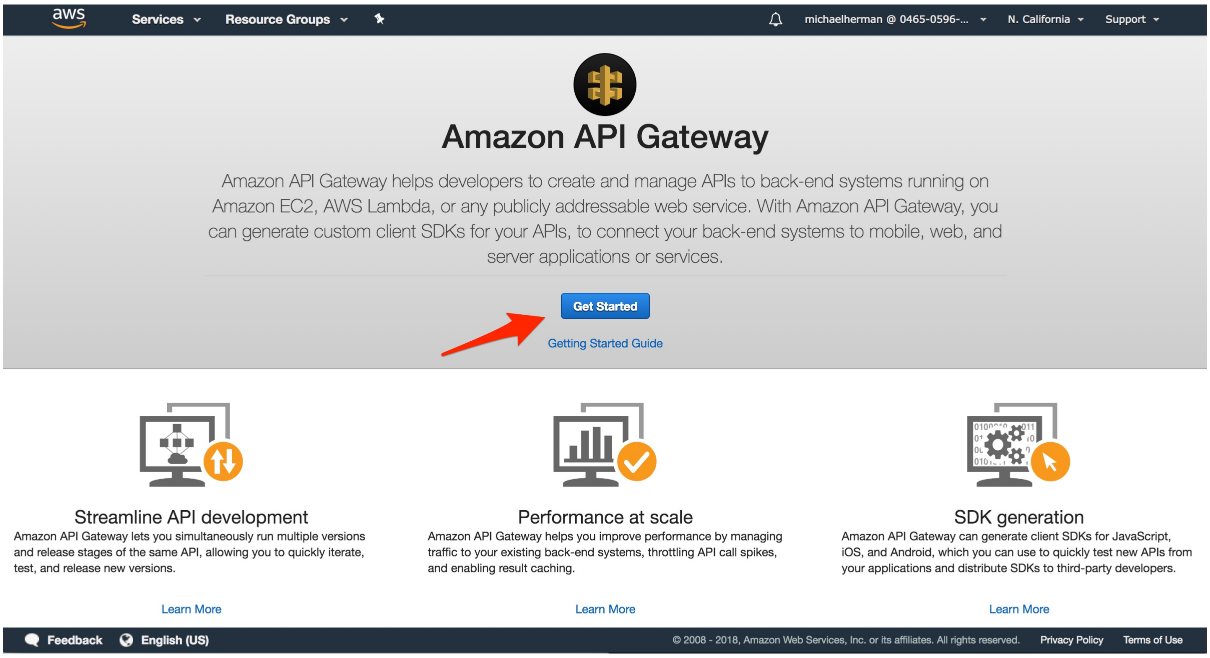
API Gateway is used to define and host APIs. In our example, we'll create a single HTTP POST endpoint that triggers the Lambda function when an HTTP request is received and then responds with the results of the Lambda function, either `true` or `false`.

Steps:

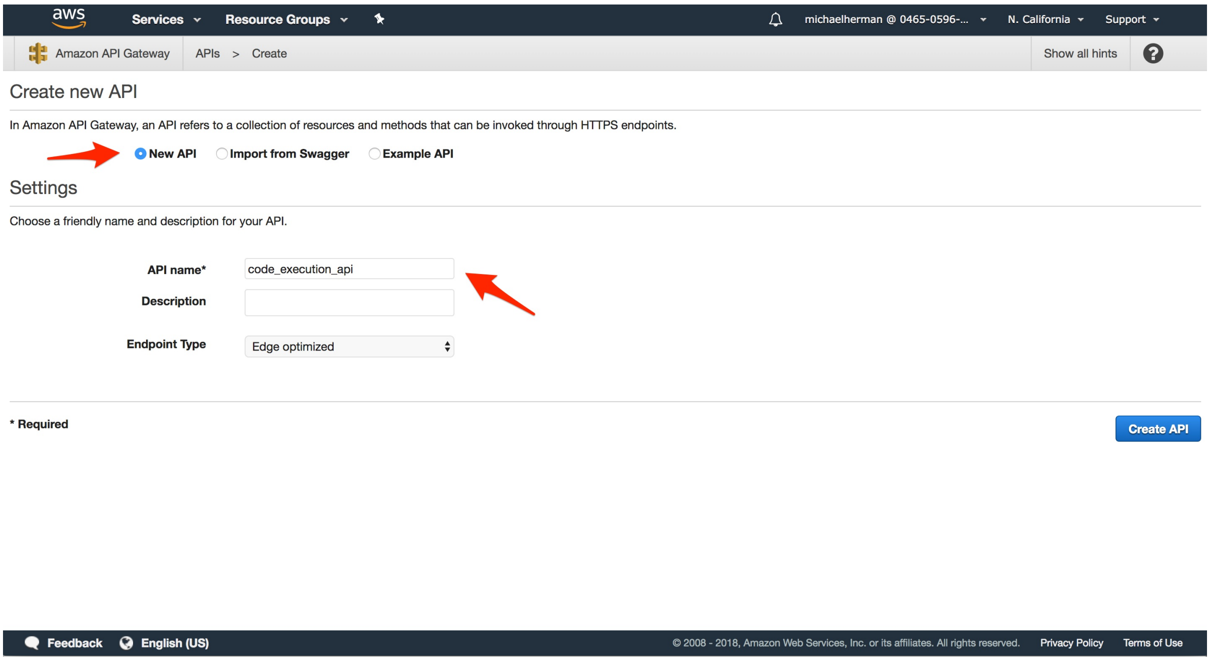
1. Create the API
2. Test it manually
3. Enable CORS
4. Deploy the API
5. Test via cURL

## Create the API

To start, from the API Gateway page, click the "Get Started" button to create a new API:

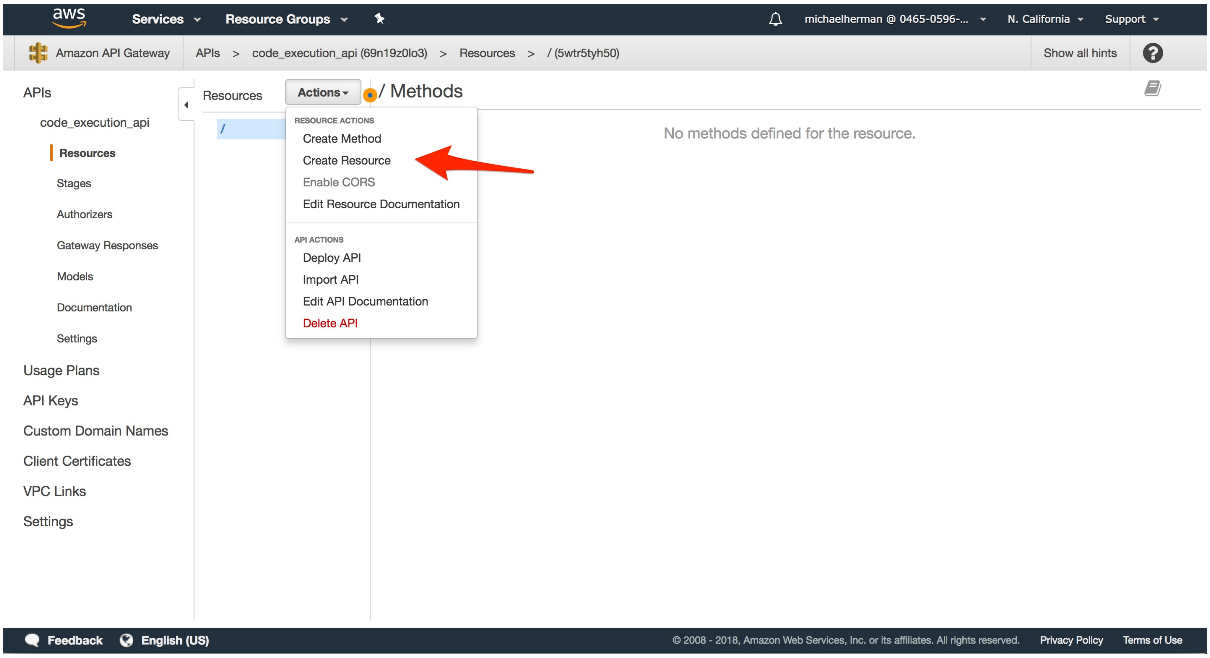



Select "New API", and then use `code_execution_api` for the name.

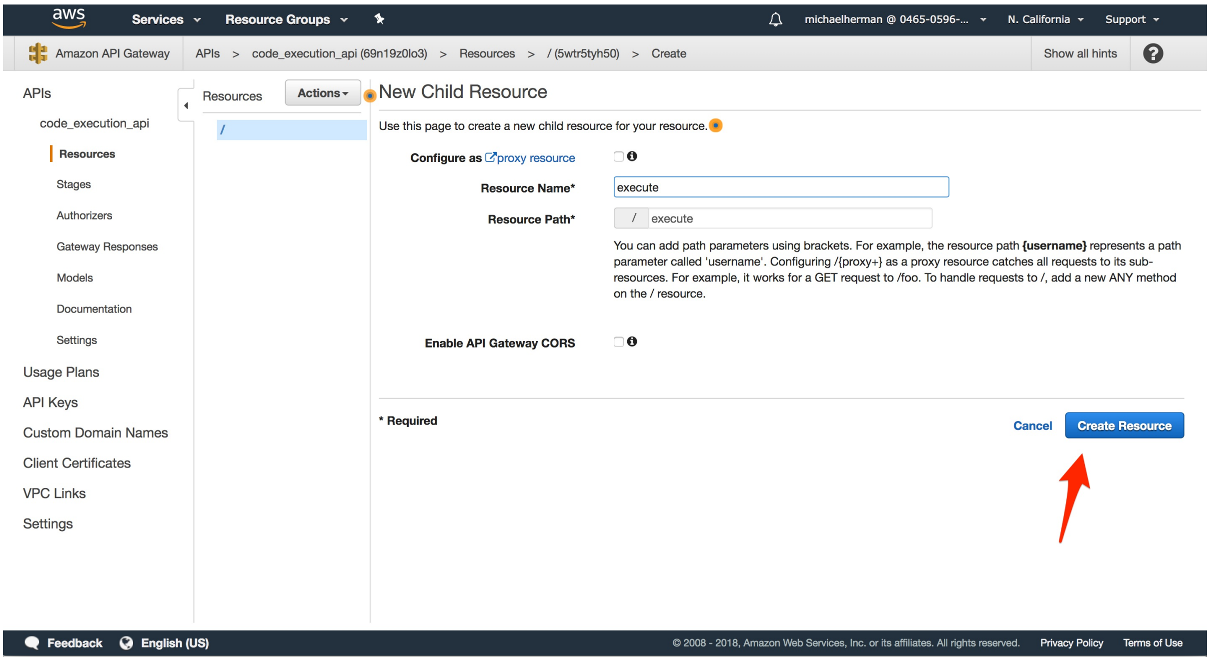



Then, create the API.

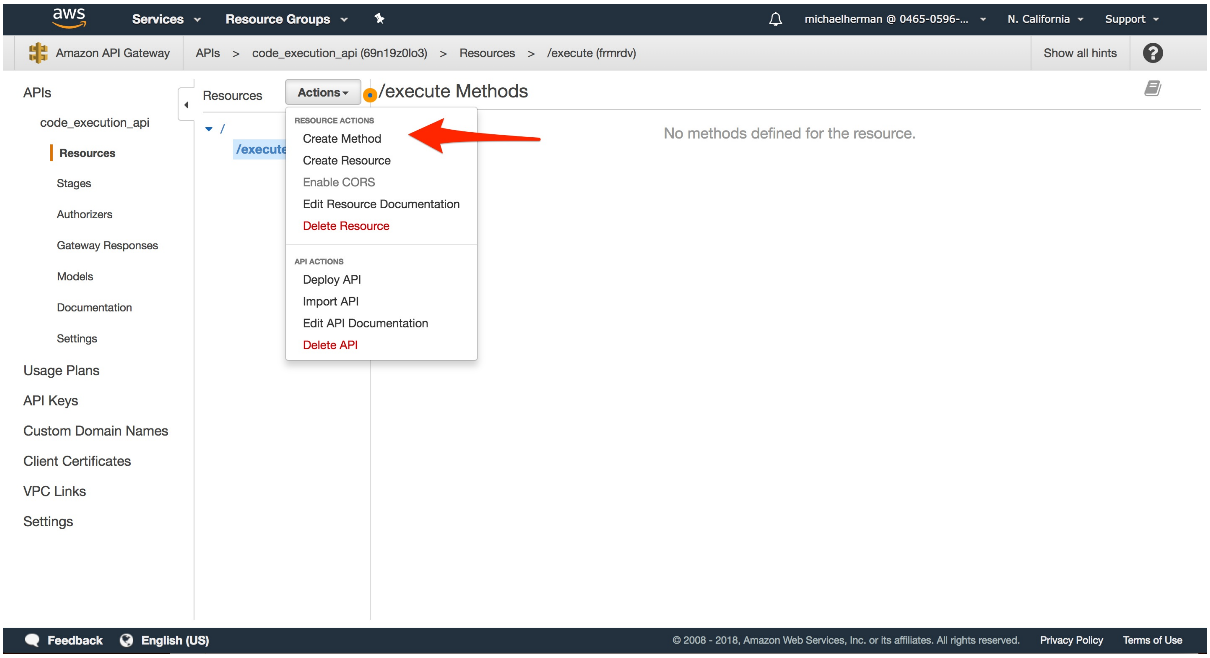
Select "Create Resource" from the "Actions" drop-down.



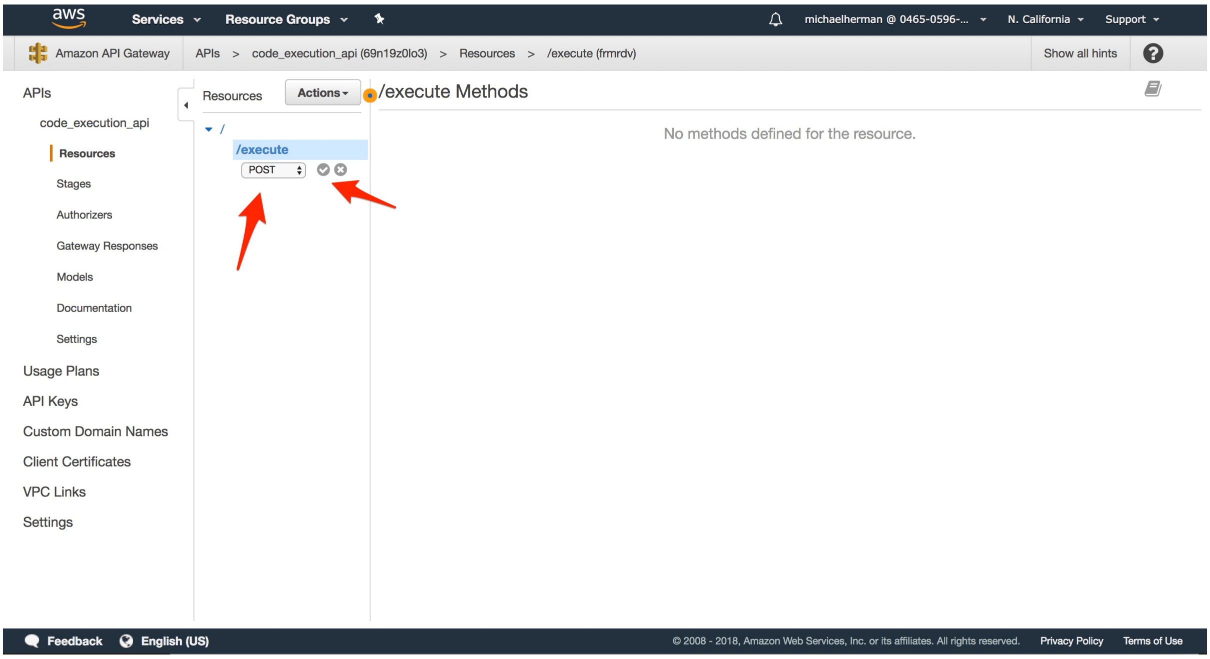

Name the resource `execute`, and then click "Create Resource".



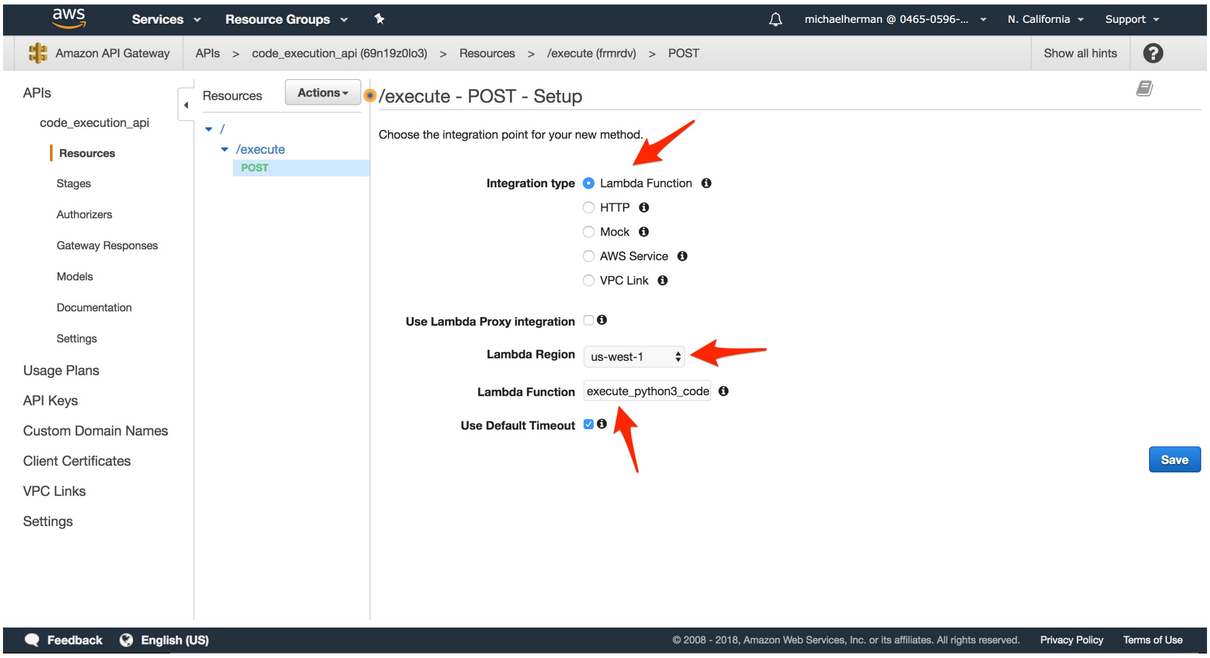

With the resource highlighted, select "Create Method" from the "Actions" drop-down.




Choose "POST" from the method drop-down. Click the checkmark next to it.



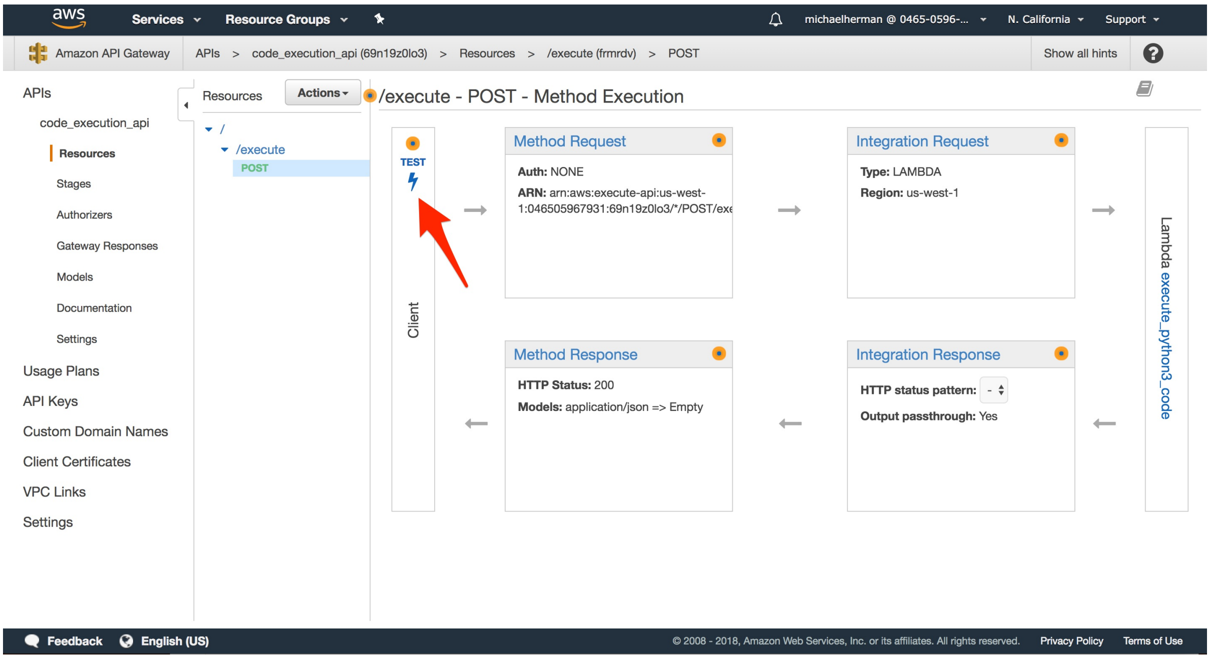

In the "Setup" step, select "Lambda Function" as the "Integration type", select the "us-west-1" region in the drop-down, and enter the name of the Lambda function that you just created.




Click "Save", and then click "OK" to give permission to the API Gateway to run your Lambda function.

## Test it manually

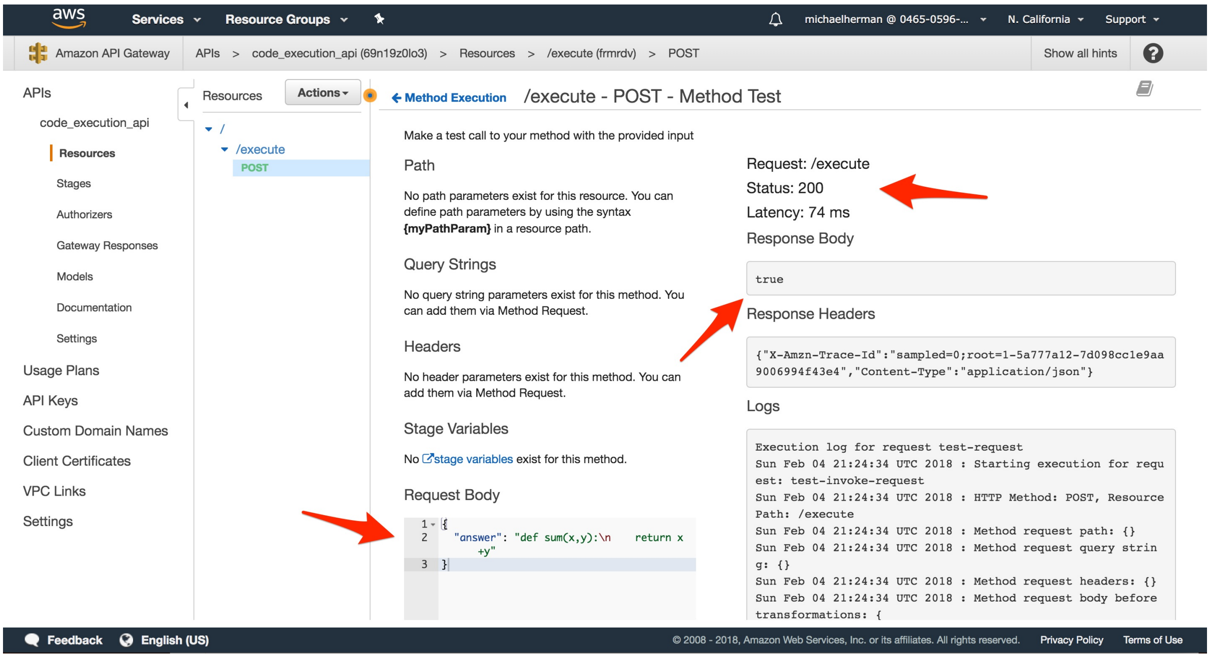
To test, click on the lightning bolt that says "Test".




Scroll down to the "Request Body" input and add the same JSON code we used with the Lambda function:

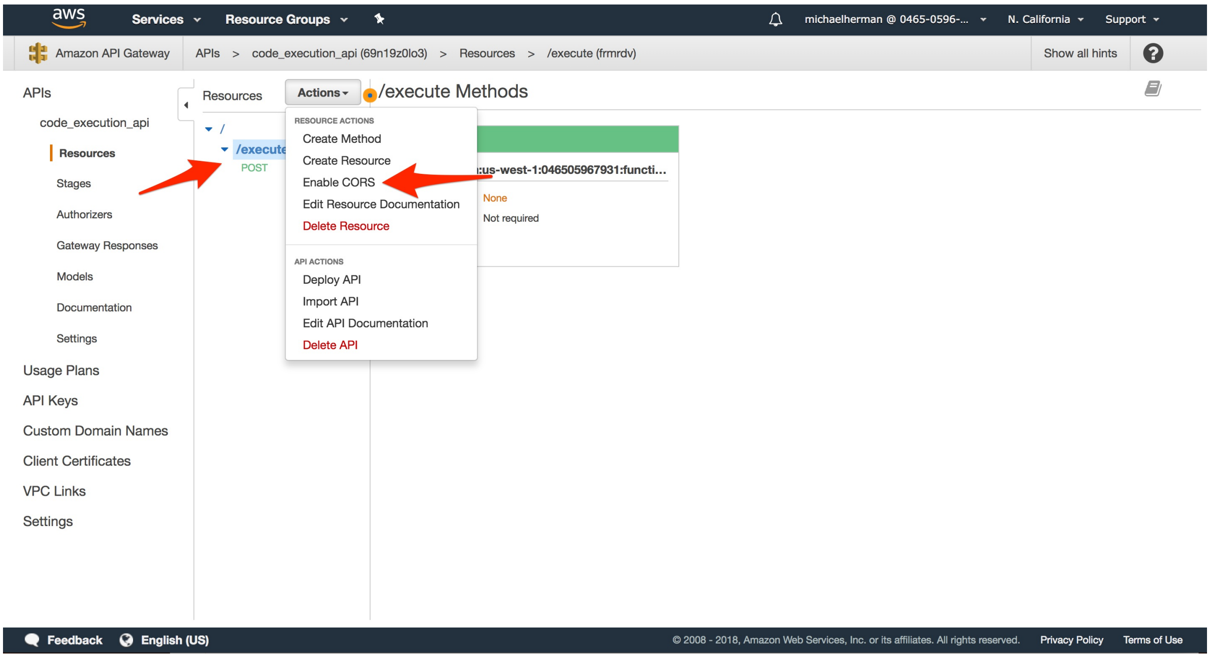
```
{
  "answer": "def sum(x,y):\n    return x+y"
}
```

Click "Test". You should see something similar to:




## Enable CORS

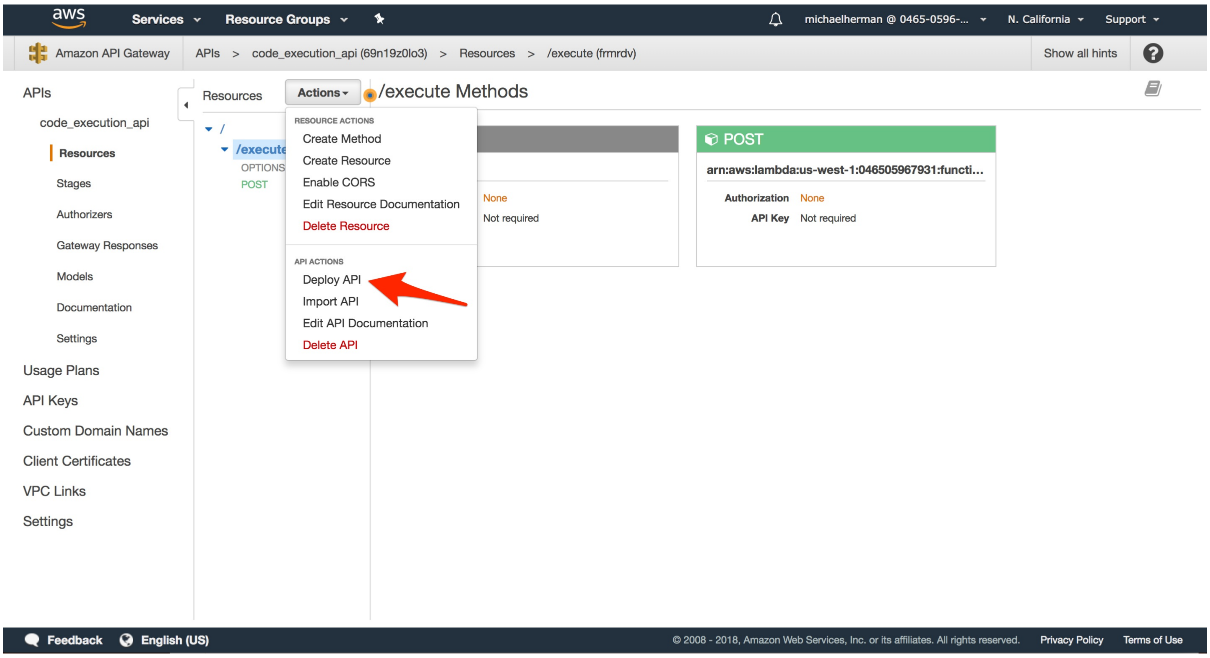
Next, we need to enable CORS so that we can POST to the API endpoint from another domain. With the resource highlighted, select "Enable CORS" from the "Actions" drop-down:



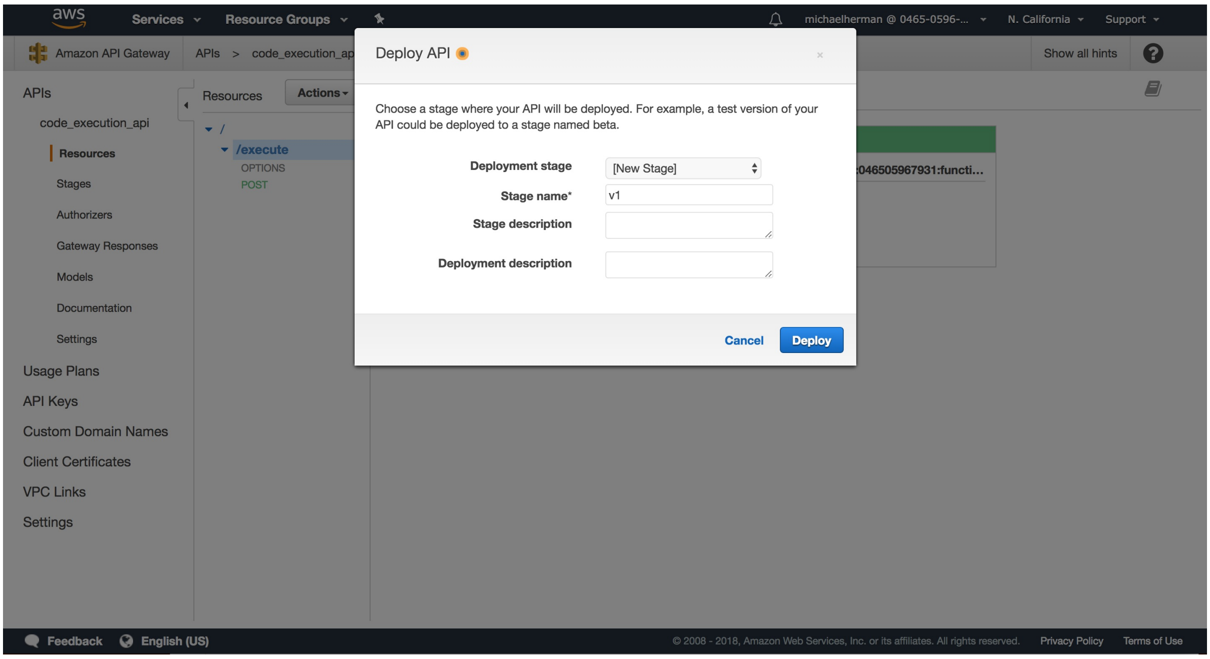

Just keep the defaults for now since we're still testing the API. Click the "Enable CORS and replace existing CORS headers" button.

## Deploy the API

Finally, to deploy, select "Deploy API" from the "Actions" drop-down:

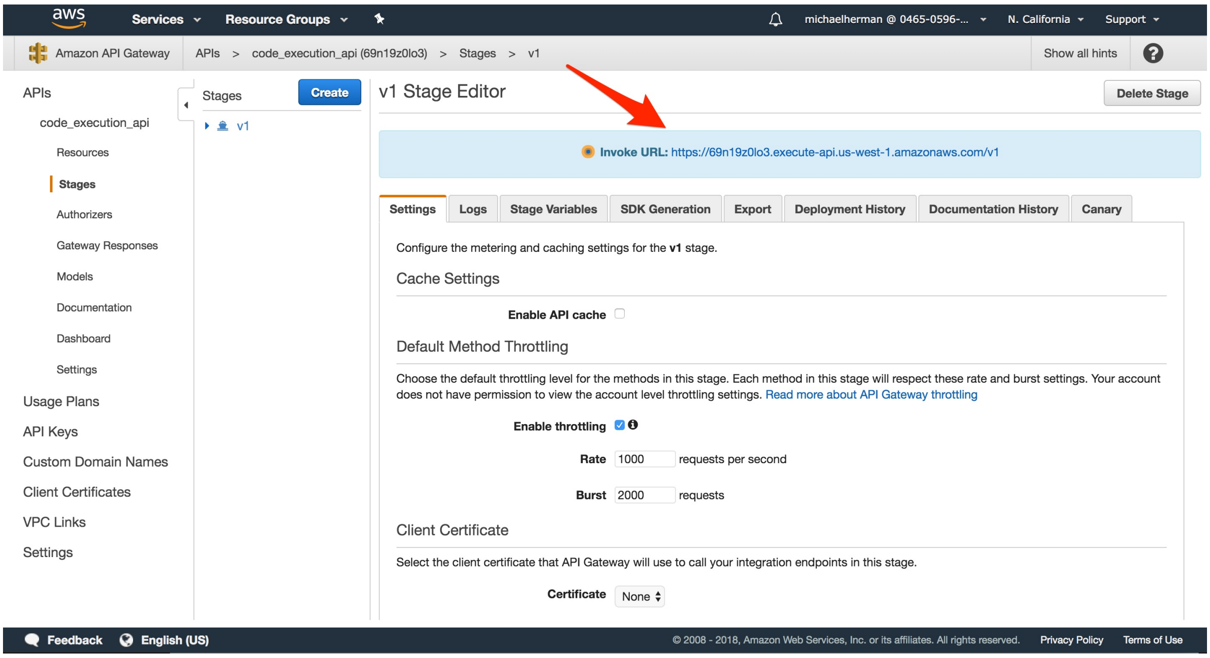



Create a new "Deployment stage" called `v1` :




API gateway will generate a random subdomain for the API endpoint URL, and the stage name will be added to the end of the URL. You should now be able to make POST requests to a similar URL:

```
https://69n19z0lo3.execute-api.us-west-1.amazonaws.com/v1/execute
```




## Test via cURL

True:

```
$ curl -H "Content-Type: application/json" -X POST \
  -d '{"answer":"def sum(x,y):\n    return x+y"}' \
  https://YOUR_INVOKE_URL
```

False:

```
$ curl -H "Content-Type: application/json" -X POST \
  -d '{"answer":"def sum(x,y):\n    return x+y+9999999999"}' \
  https://YOUR_INVOKE_URL
```

With that, let's turn our attention to the client-side...

# Update Exercises Component

In this lesson, we'll add an AJAX request to the `Exercises` component to connect with API Gateway...

## Workflow

1. User submits solution
2. AJAX request is sent to the API Gateway endpoint
3. On submit, the `Run Code` button is disabled and a grading message appears (so the user knows something is happening in case the process takes more than a few seconds)
4. Once the Lambda is complete and the response is received, the grading message disappears and either a correct or incorrect message is displayed

Before we dive in, let's add a test!

## Test

Set `testdriven-dev` as the active Docker Machine:

```
$ docker-machine env testdriven-dev
$ eval $(docker-machine env testdriven-dev)
```

Set the environment variables:

```
$ export REACT_APP_USERS_SERVICE_URL=http://DOCKER_MACHINE_DEV_IP
$ export TEST_URL=http://DOCKER_MACHINE_DEV_IP
```

Fire up the containers:

```
$ docker-compose -f docker-compose-dev.yml up -d --build
```

Create and seed the database:

```
$ docker-compose -f docker-compose-dev.yml \
  run exercises python manage.py recreate_db

$ docker-compose -f docker-compose-dev.yml \
  run users python manage.py recreate_db

$ docker-compose -f docker-compose-dev.yml \
  run users python manage.py seed_db
```

Run the full test suite to ensure all is well:

```
$ sh test.sh dev
```

Then, within "e2e", create a new file called *exercises.test.js*:

```
import { Selector } from 'testcafe';

const randomstring = require('randomstring');

const username = randomstring.generate();
const email = `${username}@test.com`;
const password = 'greaterthanten';

const TEST_URL = process.env.TEST_URL;


fixture('/').page(`${TEST_URL}/`);

test(`should display the exercises correctly if a user is not logged in`, async (t) => {
  await t
    .navigateTo(`${TEST_URL}/`)
    .expect(Selector('H1').withText('Exercises').exists).ok()
    .expect(Selector('.alert-warning').withText('Please log in to submit an exercise.').exists).ok()
    .expect(Selector('button').withText('Run Code').exists).notOk()
});

test(`should allow a user to submit an exercise if logged in`, async (t) => {
  await t
    .navigateTo(`${TEST_URL}/register`)
    .typeText('input[name="username"]', username)
    .typeText('input[name="email"]', email)
    .typeText('input[name="password"]', password)
    .click(Selector('input[type="submit"]'))
  await t
    .navigateTo(`${TEST_URL}/`)
    .expect(Selector('H1').withText('Exercises').exists).ok()
    .expect(Selector('.alert-warning').withText('Please log in to submit an exercise.').exists).notOk()
    .expect(Selector('button').withText('Run Code').exists).ok()
    .click(Selector('button').withText('Run Code'))
    .expect(Selector('h4').withText('Incorrect!').exists).ok()
});
```

Review the code on your own, and then run the tests again to ensure `should allow a user to submit an exercise if logged in` fails.

# Code

## AJAX request to API Gateway

Update `submitExercise()` in *services/client/src/components/Exercises.jsx*:

```
submitExercise(event) {
  event.preventDefault();
  const data = { answer: this.state.editor.value };
  const url = process.env.REACT_APP_API_GATEWAY_URL;
  axios.post(url, data)
  .then((res) => { console.log(res); })
  .catch((err) => { console.log(err); })
};
```

Add the import:

```
import axios from 'axios';
```

To test, first add the `REACT_APP_API_GATEWAY_URL` environment variable to *docker-compose-dev.yml*:

```
client:
  container_name: client
  build:
    context: ./services/client
    dockerfile: Dockerfile-dev
  volumes:
    - './services/client:/usr/src/app'
    - '/usr/src/app/node_modules'
  ports:
    - '3007:3000'
  environment:
    - NODE_ENV=development
    - REACT_APP_USERS_SERVICE_URL=${REACT_APP_USERS_SERVICE_URL}
    - REACT_APP_API_GATEWAY_URL=${REACT_APP_API_GATEWAY_URL}
    # - CHOKIDAR_USEPOLLING=true
  depends_on:
    - users
  links:
    - users
```

Then, set the variable:

```
$ export REACT_APP_API_GATEWAY_URL=https://API_GATEWAY_URL
```

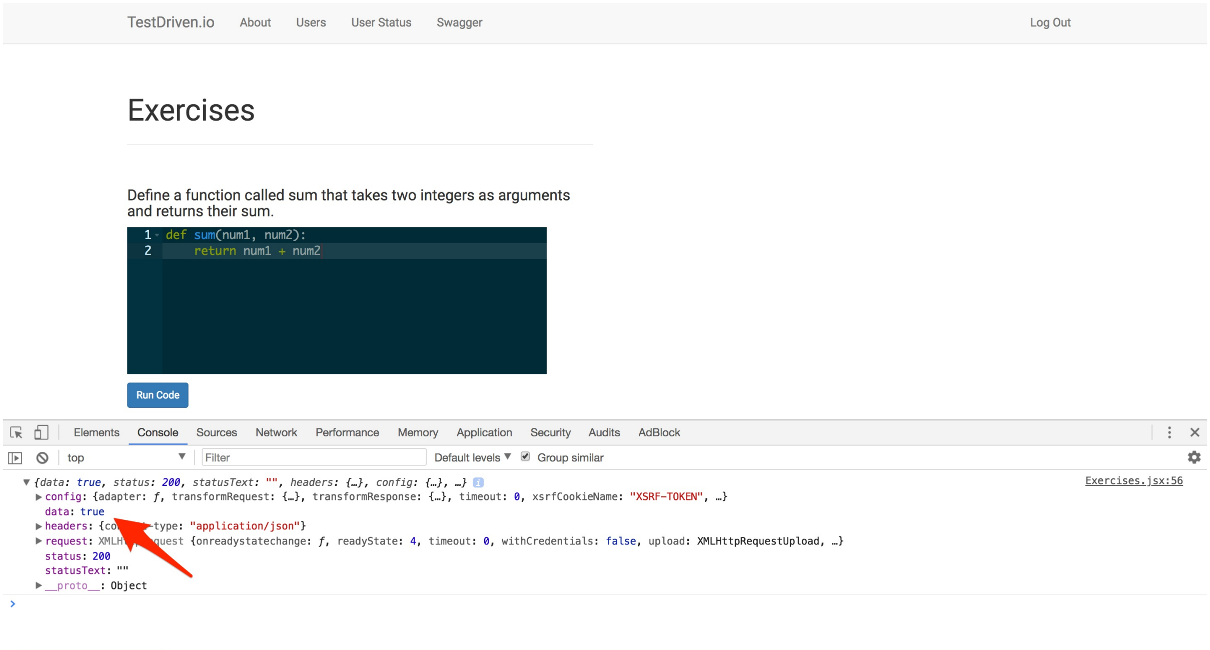
Update the containers:

```
$ docker-compose -f docker-compose-dev.yml up -d --build
```

Open the JavaScript console in your browser and enter the following code into the Ace code editor:

```
def sum(num1, num2):
    return num1 + num2
```

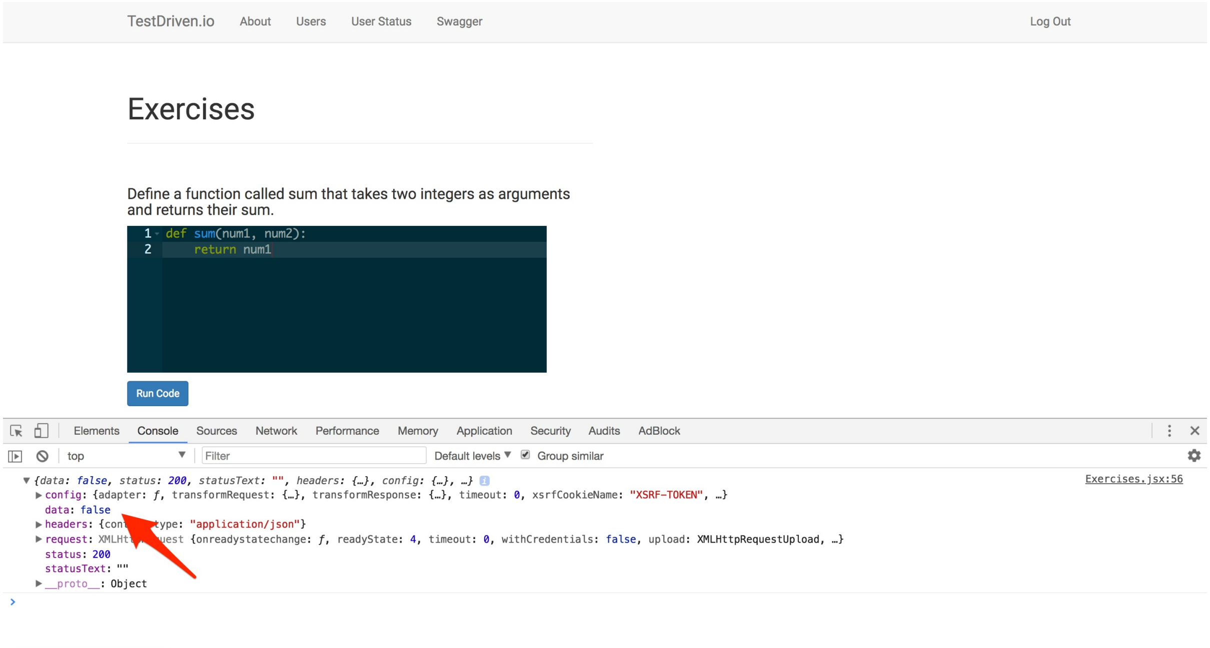
The response object should have a key of `data` with a value of `true`.




Update the code to:

```
def sum(num1, num2):
    return num1
```

Make sure the value of `data` is now `false`.




## Display grading message

Update the button and add the grading message within the `render()` :

```
{this.props.isAuthenticated &&
  <div>
    <Button
      bsStyle="primary"
      bsSize="small"
      onClick={this.submitExercise}
      disabled={this.state.editor.button.isDisabled}
    >Run Code</Button>
  {this.state.editor.showGrading &&
    <h4>
       
      <Glyphicon glyph="repeat" className="glyphicon-spin"/>
       
      Grading...
    </h4>
  }
  </div>
}
```

Update the state:

```
this.state = {
  exercises: [],
  editor: {
    value: '# Enter your code here.',
    button: {
      isDisabled: false,
```

```
    },
    showGrading: false,
  },
};
```

So, since `isDisabled` defaults to `false` the button will be clickable when the component is first rendered. The grading message will also not be displayed.

Make sure to update the import, to bring in the Glyphicon component:

```
import { Button, Glyphicon } from 'react-bootstrap';
```

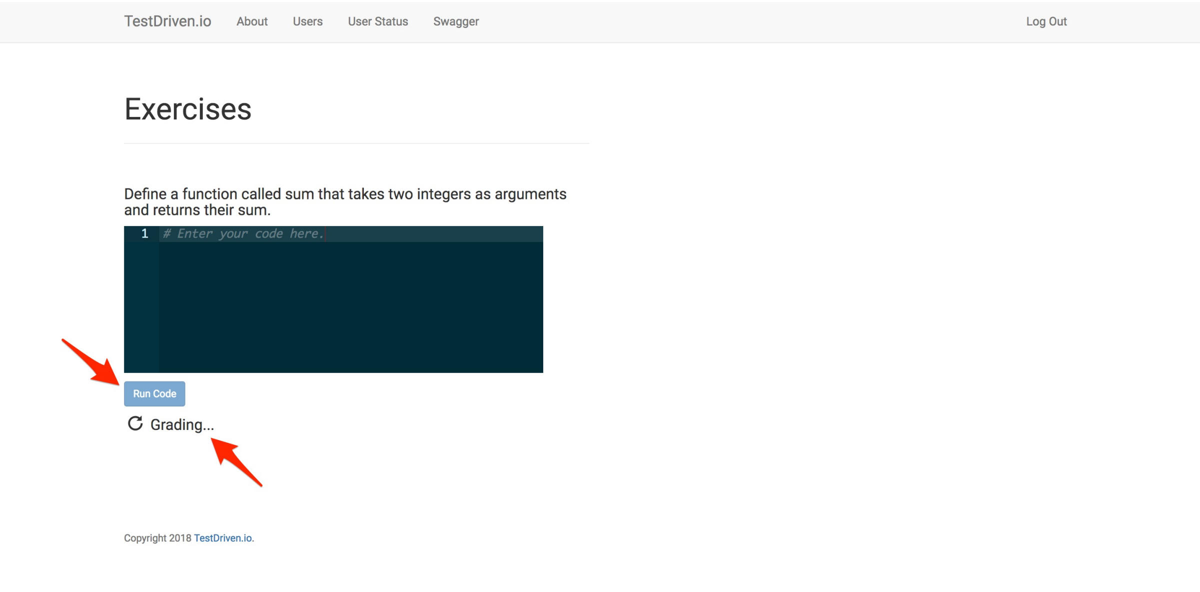
Update the `submitExercise` function to change the state of `showGrading` and `isDisabled` to `false` :

```
submitExercise(event) {
  event.preventDefault();
  const newState = this.state.editor;
  newState.showGrading = true;
  newState.button.isDisabled = true;
  this.setState(newState);
  const data = { answer: this.state.editor.value };
  const url = process.env.REACT_APP_API_GATEWAY_URL;
  axios.post(url, data)
  .then((res) => { console.log(res); })
  .catch((err) => { console.log(err); })
};
```

Also, update `onChange()` :

```
onChange(value) {
  const newState = this.state.editor;
  newState.value = value;
  this.setState(newState);
};
```

Test it out in the browser!




## Display correct or incorrect message

Start by adding a few more keys to the state object:

```
this.state = {
  exercises: [],
  editor: {
    value: '# Enter your code here.',
    button: {
      isDisabled: false,
    },
    showGrading: false,
    showCorrect: false,
    showIncorrect: false,
  },
};
```

Then, update the `submitExercise` function to change the state of the appropriate key based on the value of `data`:

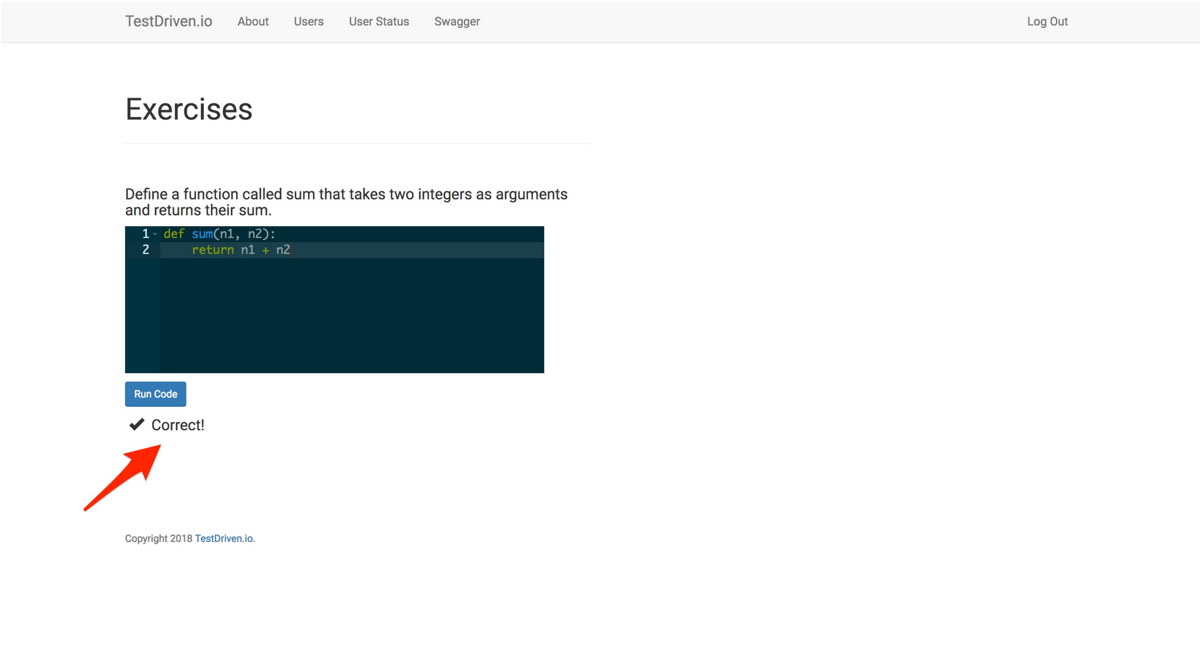
```
submitExercise(event) {
  event.preventDefault();
  const newState = this.state.editor;
  newState.showGrading = true;
  newState.showCorrect = false;
  newState.showIncorrect = false;
  newState.button.isDisabled = true;
  this.setState(newState);
  const data = { answer: this.state.editor.value };
  const url = process.env.REACT_APP_API_GATEWAY_URL;
  axios.post(url, data)
```

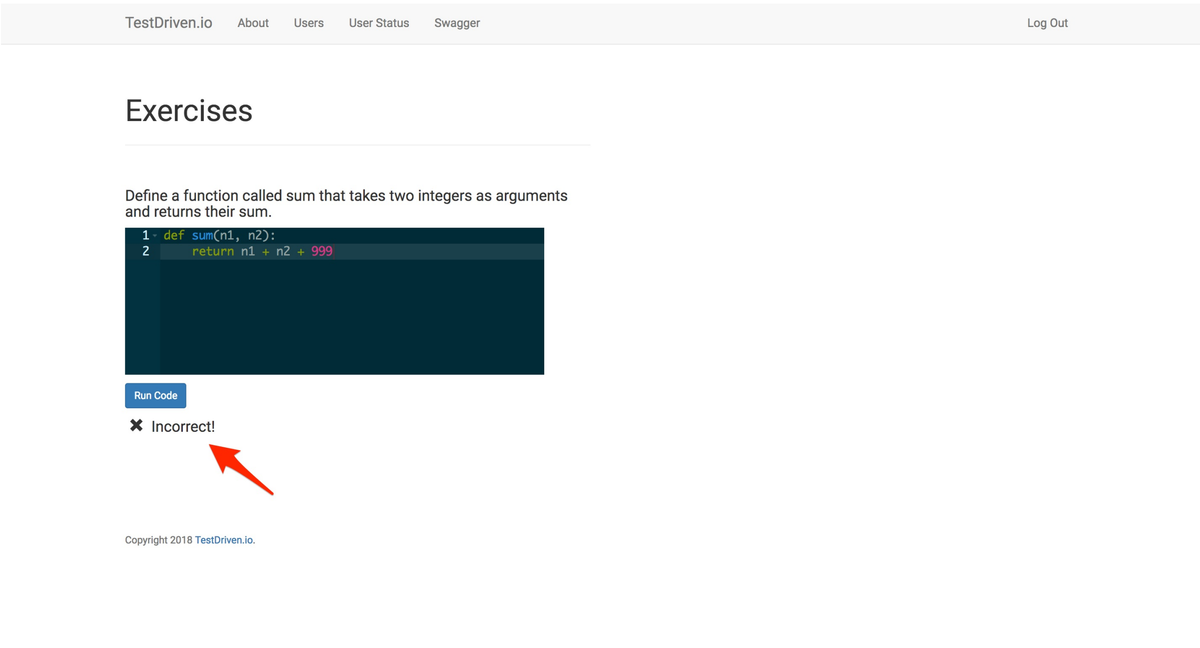
```
    .then((res) => {
      newState.showGrading = false
      newState.button.isDisabled = false
      if (res.data) {newState.showCorrect = true};
      if (!res.data) {newState.showIncorrect = true};
      this.setState(newState);
    })
    .catch((err) => {
      newState.showGrading = false
      newState.button.isDisabled = false
      console.log(err);
    })
};
```

Add a few more messages to the `render()` :

```
{this.state.editor.showCorrect &&
  <h4>
     
    <Glyphicon glyph="ok" className="glyphicon-correct"/>
     
    Correct!
  </h4>
}
{this.state.editor.showIncorrect &&
  <h4>
     
    <Glyphicon glyph="remove" className="glyphicon-incorrect"/>
     
    Incorrect!
  </h4>
}
```

Test it out again!




Make sure the end-to-end tests are now all green:

```
/
✓ should display the exercises correctly if a user is not logged in
✓ should allow a user to submit an exercise if logged in

/
✓ should display the page correctly if a user is not logged in
✓ should display the page correctly if a user is logged in

/login
✓ should display the sign in form
✓ should allow a user to sign in
✓ should throw an error if the credentials are incorrect
```

```
/register
✓ should display flash messages correctly

/register
✓ should display the registration form
✓ should allow a user to register
✓ should validate the password field
✓ should throw an error if the username is taken
✓ should throw an error if the email is taken

/status
✓ should not display user info if a user is not logged in
✓ should display user info if a user is logged in

/swagger
✓ should display the swagger docs correctly

/all-users
✓ should display the all-users page correctly if a user is not logged in


17 passed (1m 08s)
```

## Update `getExercises()`

Finally, let's update `getExercises()` so that it calls the `exercises` service:

```
getExercises() {
  axios.get(`${process.env.REACT_APP_EXERCISES_SERVICE_URL}/exercises`)
  .then((res) => { this.setState({ exercises: res.data.data.exercises }); })
  .catch((err) => { console.log(err); });
};
```

Add the `REACT_APP_EXERCISES_SERVICE_URL` environment variable to *docker-compose-dev.yml*:

```
client:
  container_name: client
  build:
    context: ./services/client
    dockerfile: Dockerfile-dev
  volumes:
    - './services/client:/usr/src/app'
    - '/usr/src/app/node_modules'
  ports:
    - '3007:3000'
  environment:
    - NODE_ENV=development
```

```
      - REACT_APP_USERS_SERVICE_URL=${REACT_APP_USERS_SERVICE_URL}
      - REACT_APP_API_GATEWAY_URL=${REACT_APP_API_GATEWAY_URL}
      - REACT_APP_EXERCISES_SERVICE_URL=${REACT_APP_EXERCISES_SERVICE_URL}
      # - CHOKIDAR_USEPOLLING=true
    depends_on:
      - users
    links:
      - users
```

Add a new `location` to the Nginx conf in "services/nginx":

```
location /exercises {
  proxy_pass          http://exercises:5000;
  proxy_redirect      default;
  proxy_set_header    Host $host;
  proxy_set_header    X-Real-IP $remote_addr;
  proxy_set_header    X-Forwarded-For $proxy_add_x_forwarded_for;
  proxy_set_header    X-Forwarded-Host $server_name;
}
```

Then, set the variable:

```
$ export REACT_APP_EXERCISES_SERVICE_URL=http://DOCKER_MACHINE_DEV_IP
```

Add a seed command to *services/exercises/manage.py*:

```
@cli.command()
def seed_db():
    """Seeds the database."""
    db.session.add(Exercise(
        body=('Define a function called sum that takes two integers as '
              'arguments and returns their sum.'),
        test_code='print(sum(2, 3))',
        test_code_solution='5'
    ))
    db.session.add(Exercise(
        body=('Define a function called reverse that takes a string as '
              'an argument and returns the string in reversed order.'),
        test_code='print(reverse(racecar))',
        test_code_solution='racecar'
    ))
    db.session.add(Exercise(
        body=('Define a function called factorial that takes a random number '
              'as an argument and then returns the factorial of that given '
              'number.'),
        test_code='print(factorial(5))',
        test_code_solution='120'
    ))
```

```
    db.session.commit()
```

Update the containers:

```
$ docker-compose -f docker-compose-dev.yml up -d --build
```

Apply the seed:

```
$ docker-compose -f docker-compose-dev.yml \
  run exercises python manage.py seed_db
```

Test it out in the browser, and then run the full test suite:

```
$ sh test.sh dev
```

You should have some failing tests from *Exercises.test.jsx*. To fix, fake the AJAX call by mocking `componentDidMount()` :

```
import React from 'react';
import { shallow, mount } from 'enzyme';
import renderer from 'react-test-renderer';

import AceEditor from 'react-ace';
jest.mock('react-ace');

import Exercises from '../Exercises';

const exercises = [
  {
    id: 0,
    body: `Define a function called sum that takes
    two integers as arguments and returns their sum.`
  },
  {
    id: 1,
    body: `Define a function called reverse that takes a string
    as an argument and returns the string in reversed order.`
  },
  {
    id: 2,
    body: `Define a function called factorial that takes a random
    number as an argument and then returns the factorial of that
    given number.`,
  }
];

test('Exercises renders properly when not authenticated', () => {
```

```
  const onDidMount = jest.fn();
  Exercises.prototype.componentDidMount = onDidMount;
  const wrapper = shallow(<Exercises isAuthenticated={false}/>);
  wrapper.setState({exercises : exercises});
  const heading = wrapper.find('h4');
  expect(heading.length).toBe(1);
  const alert = wrapper.find('.alert');
  expect(alert.length).toBe(1);
  expect(alert.get(0).props.children[1].props.children).toContain(
    'Please log in to submit an exercise.')
});

test('Exercises renders properly when authenticated', () => {
  const onDidMount = jest.fn();
  Exercises.prototype.componentDidMount = onDidMount;
  const wrapper = shallow(<Exercises isAuthenticated={true}/>);
  wrapper.setState({exercises : exercises});
  const heading = wrapper.find('h4');
  expect(heading.length).toBe(1);
  const alert = wrapper.find('.alert');
  expect(alert.length).toBe(0);
});

test('Exercises renders a snapshot properly', () => {
  const onDidMount = jest.fn();
  Exercises.prototype.componentDidMount = exercises;
  const tree = renderer.create(<Exercises/>).toJSON();
  expect(tree).toMatchSnapshot();
});

test('Exercises will call componentWillMount when mounted', () => {
  const onWillMount = jest.fn();
  Exercises.prototype.componentWillMount = onWillMount;
  const wrapper = mount(<Exercises/>);
  expect(onWillMount).toHaveBeenCalledTimes(1)
});

test('Exercises will call componentDidMount when mounted', () => {
  const onDidMount = jest.fn();
  Exercises.prototype.componentDidMount = onDidMount;
  const wrapper = mount(<Exercises/>);
  expect(onDidMount).toHaveBeenCalledTimes(1)
});
```

Test one final time:

```
$ sh test.sh dev
```

Commit and push to GitHub.

# ECS Deployment - Staging

Let's update the staging environment on ECS...

---

## Docker Machine

Set `testdriven-dev` as the active Docker Machine:

```
$ docker-machine env testdriven-dev
$ eval $(docker-machine env testdriven-dev)
```

Set the environment variables:

```
$ export REACT_APP_USERS_SERVICE_URL=http://DOCKER_MACHINE_DEV_IP
$ export TEST_URL=http://DOCKER_MACHINE_DEV_IP
$ export REACT_APP_API_GATEWAY_URL=https://API_GATEWAY_URL
$ export REACT_APP_EXERCISES_SERVICE_URL=http://DOCKER_MACHINE_DEV_IP
```

Fire up the containers:

```
$ docker-compose -f docker-compose-dev.yml up -d --build
```

Create and seed the databases:

```
$ docker-compose -f docker-compose-dev.yml \
  run exercises python manage.py recreate_db

$ docker-compose -f docker-compose-dev.yml \
  run users python manage.py recreate_db

$ docker-compose -f docker-compose-dev.yml \
  run exercises python manage.py seed_db

$ docker-compose -f docker-compose-dev.yml \
  run users python manage.py seed_db
```

Run the full test suite to ensure all is well locally:

```
$ sh test.sh dev
```

## Travis

Next, we need to update *.travis.yml* to handle the new `exercises` and `exercises-db` services by adding in the proper environment variables to the `env` and `before_script` sections:

```
language: node_js
node_js: '9'

before_install:
  - stty cols 80

dist: trusty
sudo: required

addons:
  apt:
    sources:
     - google-chrome
    packages:
     - google-chrome-stable

services:
  - docker

env:
  global:
    - DOCKER_COMPOSE_VERSION=1.18.0
    - COMMIT=${TRAVIS_COMMIT::8}
    - MAIN_REPO=https://github.com/testdrivenio/testdriven-app-2.2.git
    - USERS=test-driven-users
    - USERS_REPO=${MAIN_REPO}#${TRAVIS_BRANCH}:services/users
    - USERS_DB=test-driven-users_db
    - USERS_DB_REPO=${MAIN_REPO}#${TRAVIS_BRANCH}:services/users/project/db
    - CLIENT=test-driven-client
    - CLIENT_REPO=${MAIN_REPO}#${TRAVIS_BRANCH}:services/client
    - SWAGGER=test-driven-swagger
    - SWAGGER_REPO=${MAIN_REPO}#${TRAVIS_BRANCH}:services/swagger
    - EXERCISES=test-driven-exercises
    - EXERCISES_REPO=${MAIN_REPO}#${TRAVIS_BRANCH}:services/exercises
    - EXERCISES_DB=test-driven-exercises_db
    - EXERCISES_DB_REPO=${MAIN_REPO}#${TRAVIS_BRANCH}:services/exercises/project/db

before_install:
  - sudo rm /usr/local/bin/docker-compose
  - curl -L https://github.com/docker/compose/releases/download/${DOCKER_COMPOSE_VE
RSION}/docker-compose-`uname -s`-`uname -m` > docker-compose
  - chmod +x docker-compose
  - sudo mv docker-compose /usr/local/bin

before_script:
  - export TEST_URL=http://127.0.0.1
  - export REACT_APP_USERS_SERVICE_URL=http://127.0.0.1
```

```
  - export REACT_APP_EXERCISES_SERVICE_URL=http://127.0.0.1
  - export REACT_APP_API_GATEWAY_URL=https://API_GATEWAY_URL
  - export SECRET_KEY=my_precious
  - export DISPLAY=:99.0
  - export DOCKER_ENV=$(if [ "$TRAVIS_BRANCH" == "production" ]; then echo "prod";
else echo "stage"; fi)
  - sh -e /etc/init.d/xvfb start
  - sleep 3
  - docker-compose -f docker-compose-stage.yml up --build -d

script:
  - bash test.sh $DOCKER_ENV

after_script:
  - docker-compose -f docker-compose-stage.yml down

after_success:
  - bash ./docker-push.sh
  - bash ./docker-deploy-stage.sh
  - bash ./docker-deploy-prod.sh
```

> Make sure to replace `API_GATEWAY_URL` with the actual URL.

## Docker Compose

Add the proper `args` to the `client` service in *docker-compose-stage.yml*:

```
client:
  container_name: client
  build:
    context: ./services/client
    dockerfile: Dockerfile-stage
    args:
      - NODE_ENV=production
      - REACT_APP_USERS_SERVICE_URL=${REACT_APP_USERS_SERVICE_URL}
      - REACT_APP_API_GATEWAY_URL=${REACT_APP_API_GATEWAY_URL}
      - REACT_APP_EXERCISES_SERVICE_URL=${REACT_APP_EXERCISES_SERVICE_URL}
  expose:
    - 80
  depends_on:
    - users
  links:
    - users
```

Then, add the `exercises` and `exercises-db` services:

```
exercises:
  container_name: exercises
```

```
  build:
    context: ./services/exercises
    dockerfile: Dockerfile-stage
  expose:
    - 5000
  environment:
    - APP_SETTINGS=project.config.StagingConfig
    - DATABASE_URL=postgres://postgres:postgres@exercises-db:5432/exercises_stage
    - DATABASE_TEST_URL=postgres://postgres:postgres@exercises-db:5432/exercises_te
st
  depends_on:
    - users
    - exercises-db
  links:
    - users
    - exercises-db

exercises-db:
  container_name: exercises-db
  build:
    context: ./services/exercises/project/db
    dockerfile: Dockerfile
  expose:
    - 5432
  environment:
    - POSTGRES_USER=postgres
    - POSTGRES_PASSWORD=postgres
```

Set the environment variables in *services/client/Dockerfile-stage*:

```
ARG REACT_APP_EXERCISES_SERVICE_URL
ENV REACT_APP_EXERCISES_SERVICE_URL $REACT_APP_EXERCISES_SERVICE_URL
ARG REACT_APP_API_GATEWAY_URL
ENV REACT_APP_API_GATEWAY_URL $REACT_APP_API_GATEWAY_URL
```

Add an *entrypoint-stage.sh* file to "services/exercises":

```
#!/bin/sh

echo "Waiting for postgres..."

while ! nc -z exercises-db 5432; do
  sleep 0.1
done

echo "PostgreSQL started"

python manage.py recreate_db
python manage.py seed_db
```

```
gunicorn -b 0.0.0.0:5000 manage:app
```

Update the permissions:

```
$ chmod +x services/exercises/entrypoint-stage.sh
```

Then, update the run server command in *services/exercises/Dockerfile-stage*:

```
# run server
CMD ["./entrypoint-stage.sh"]
```

## ECR

Add the `REACT_APP_EXERCISES_SERVICE_URL` to *docker-push.sh*:

```
if [ "$TRAVIS_BRANCH" == "staging" ]
then
  export REACT_APP_USERS_SERVICE_URL="http://LOAD_BALANCER_STAGE_DNS_NAME"
  export REACT_APP_EXERCISES_SERVICE_URL="http://LOAD_BALANCER_STAGE_DNS_NAME"
fi
```

Also, update the the last `if` block to add in the appropriate built-time args to the `client` service and build, tag, and push the `exercises` and `exercises-db` services:

```
if [ "$TRAVIS_BRANCH" == "staging" ] || \
   [ "$TRAVIS_BRANCH" == "production" ]
then
  # users
  docker build $USERS_REPO -t $USERS:$COMMIT -f Dockerfile-$DOCKER_ENV
  docker tag $USERS:$COMMIT $REPO/$USERS:$TAG
  docker push $REPO/$USERS:$TAG
  # users db
  docker build $USERS_DB_REPO -t $USERS_DB:$COMMIT -f Dockerfile
  docker tag $USERS_DB:$COMMIT $REPO/$USERS_DB:$TAG
  docker push $REPO/$USERS_DB:$TAG
  # client
  docker build $CLIENT_REPO -t $CLIENT:$COMMIT -f Dockerfile-$DOCKER_ENV --build-ar
g REACT_APP_USERS_SERVICE_URL=$REACT_APP_USERS_SERVICE_URL --build-arg REACT_APP_EX
ERCISES_SERVICE_URL=$REACT_APP_EXERCISES_SERVICE_URL --build-arg REACT_APP_API_GATE
WAY_URL=$REACT_APP_API_GATEWAY_URL
  docker tag $CLIENT:$COMMIT $REPO/$CLIENT:$TAG
  docker push $REPO/$CLIENT:$TAG
  # swagger
  docker build $SWAGGER_REPO -t $SWAGGER:$COMMIT -f Dockerfile-$DOCKER_ENV $SWAGGER
_DIR
  docker tag $SWAGGER:$COMMIT $REPO/$SWAGGER:$TAG
  docker push $REPO/$SWAGGER:$TAG
```

```
    # exercises
    docker build $EXERCISES_REPO -t $EXERCISES:$COMMIT -f Dockerfile-$DOCKER_ENV
    docker tag $EXERCISES:$COMMIT $REPO/$EXERCISES:$TAG
    docker push $REPO/$EXERCISES:$TAG
    # exercises db
    docker build $EXERCISES_DB_REPO -t $EXERCISES_DB:$COMMIT -f Dockerfile
    docker tag $EXERCISES_DB:$COMMIT $REPO/$EXERCISES_DB:$TAG
    docker push $REPO/$EXERCISES_DB:$TAG
fi
```

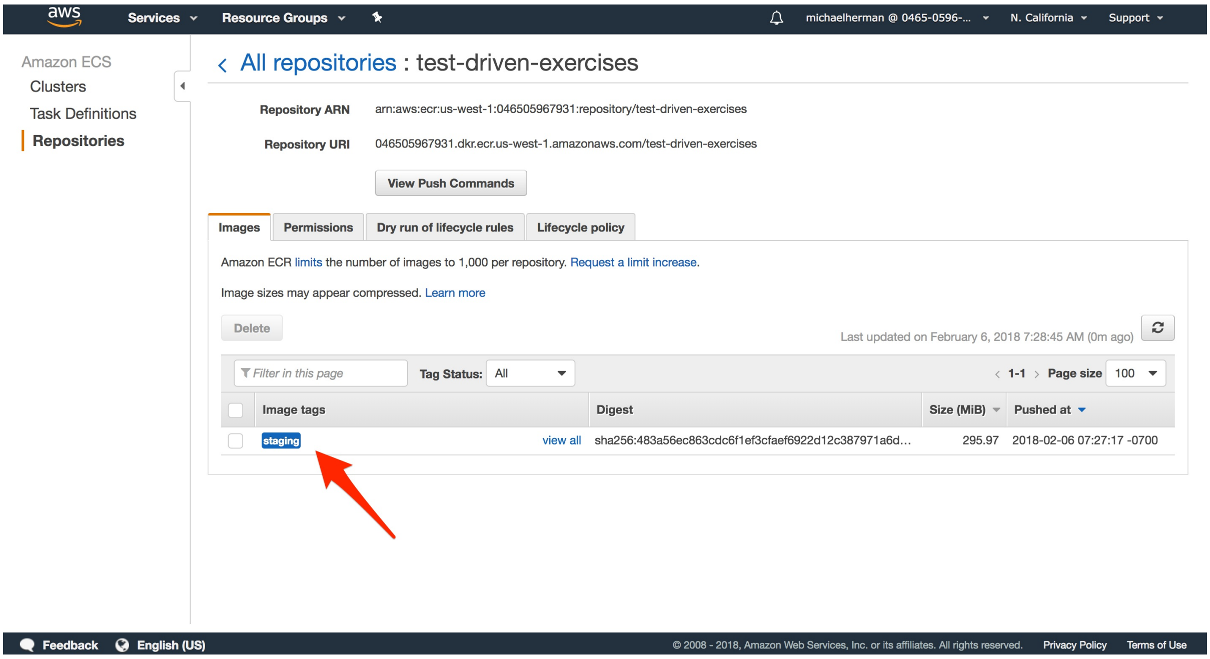
Assuming you're working from the `master` branch, commit your code, check out the `staging` branch locally, and then rebase `master` on `staging` :

```
$ git checkout staging
$ git rebase master
```

Add the Image repos to ECR:

1. `test-driven-exercises`
2. `test-driven-exercises_db`

Push to GitHub to trigger a new build on Travis. Make sure the build passes and that the images were successfully pushed to ECR.




## Task Definitions

Add `exercises` to the `deploy_cluster` function in *docker-deploy-stage.sh*:

```
# exercises
service="testdriven-exercises-stage-service"
```

```
template="ecs_exercises_stage_taskdefinition.json"
task_template=$(cat "ecs/$template")
task_def=$(printf "$task_template" $AWS_ACCOUNT_ID $AWS_ACCOUNT_ID)
echo "$task_def"
register_definition
```

Create a new Task Definition file called *ecs_exercises_stage_taskdefinition.json*:

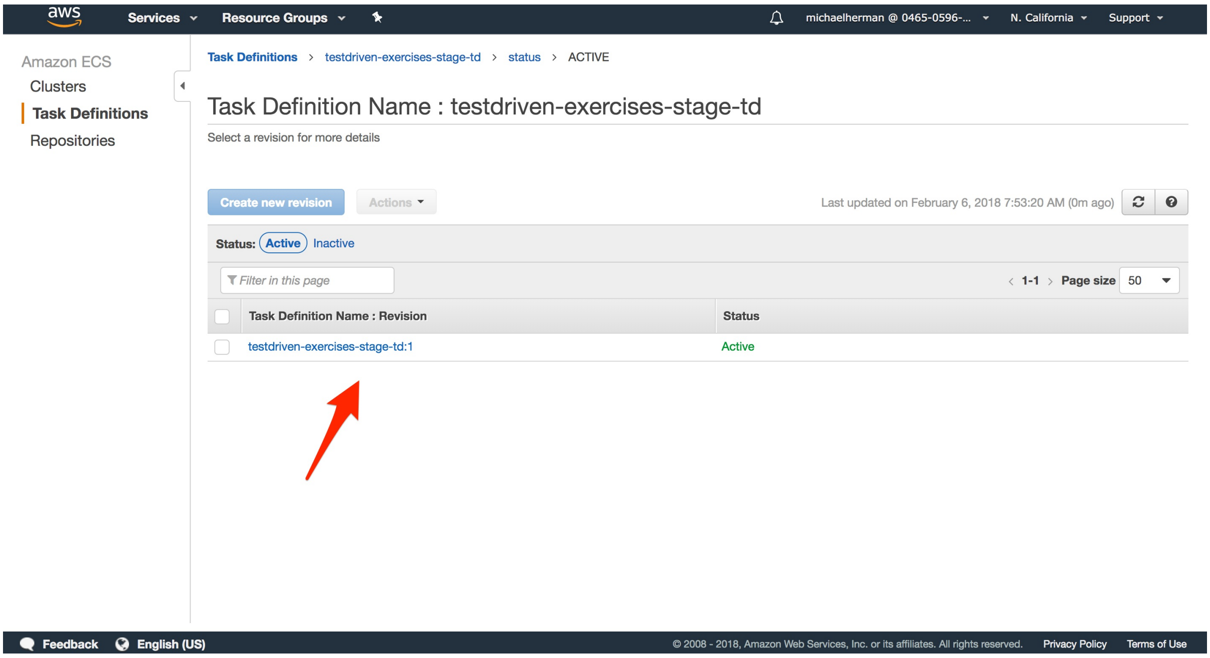
```
{
  "containerDefinitions": [
    {
      "name": "exercises",
      "image": "%s.dkr.ecr.us-west-1.amazonaws.com/test-driven-exercises:staging",
      "essential": true,
      "memoryReservation": 300,
      "portMappings": [
        {
          "hostPort": 0,
          "protocol": "tcp",
          "containerPort": 5000
        }
      ],
      "environment": [
        {
          "name": "APP_SETTINGS",
          "value": "project.config.StagingConfig"
        },
        {
          "name": "DATABASE_TEST_URL",
          "value": "postgres://postgres:postgres@exercises-db:5432/exercises_test"
        },
        {
          "name": "DATABASE_URL",
          "value": "postgres://postgres:postgres@exercises-db:5432/exercises_stage"
        }
      ],
      "links": [
        "exercises-db"
      ],
      "logConfiguration": {
        "logDriver": "awslogs",
        "options": {
          "awslogs-group": "testdriven-exercises-stage",
          "awslogs-region": "us-west-1"
        }
      }
    },
    {
      "name": "exercises-db",
      "image": "%s.dkr.ecr.us-west-1.amazonaws.com/test-driven-exercises_db:staging"
```

```
,
      "essential": true,
      "memoryReservation": 300,
      "portMappings": [
        {
          "hostPort": 0,
          "protocol": "tcp",
          "containerPort": 5432
        }
      ],
      "environment": [
        {
          "name": "POSTGRES_PASSWORD",
          "value": "postgres"
        },
        {
          "name": "POSTGRES_USER",
          "value": "postgres"
        }
      ],
      "logConfiguration": {
        "logDriver": "awslogs",
        "options": {
          "awslogs-group": "testdriven-exercises_db-stage",
          "awslogs-region": "us-west-1"
        }
      }
    }
  ],
  "family": "testdriven-exercises-stage-td"
}
```

Be sure to the add the following log groups to CloudWatch:

1. `testdriven-exercises-stage`
2. `testdriven-exercises_db-stage`

Commit and push your code. Another build should be triggered on Travis. This time ensure that the images and Task Definitions were created.




## Add Target Group

Next, let's add a new Target Group for the `exercises` service. Within Amazon EC2, click "Target Groups", and then create the following Group:

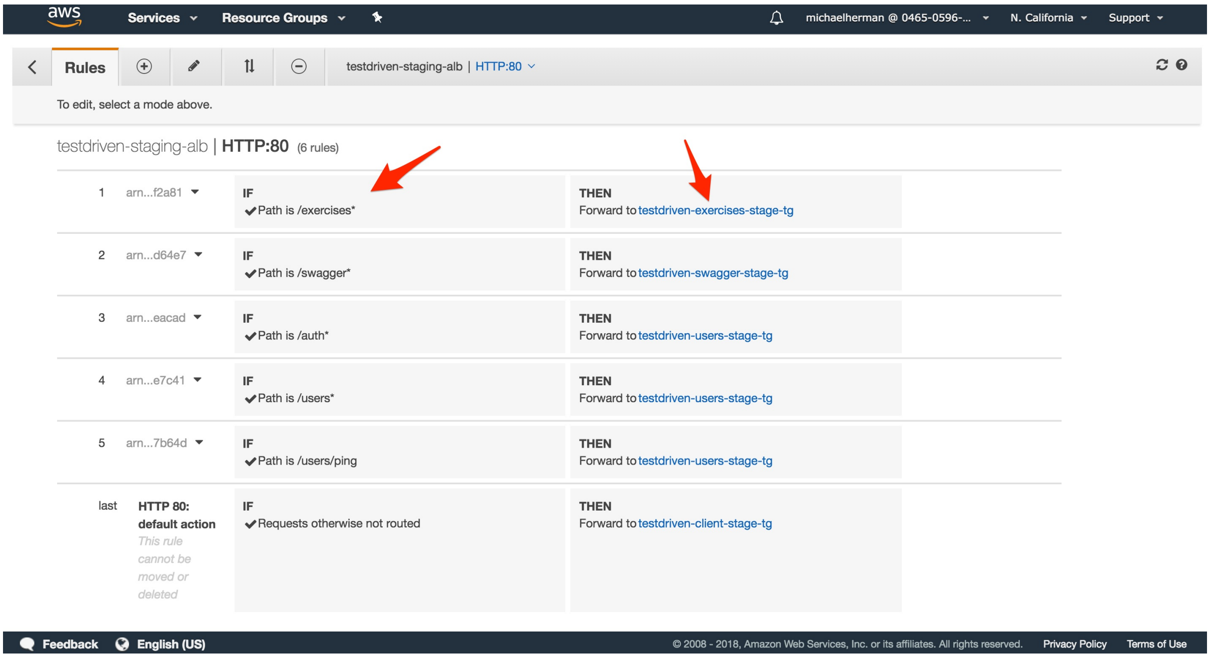
1. "Target group name": `testdriven-exercises-stage-tg`
2. "Port": `5000`
3. Then, under "Health check settings" set the "Path" to `/exercises`.

## Listener

Then, on the "Load Balancers" page, click the `testdriven-staging-alb` Load Balancer, and then select the "Listeners" tab. Here, we can add Listeners to the ALB, which are then forwarded to a specific Target Group.

Click the "View/edit rules" for "HTTP : 80", and then add one new rule:

1. If `/exercises*`, Then `testdriven-exercises-stage-tg`




## Service

Create the following ECS Service on the `test-driven-staging-cluster` Cluster...

### Exercises

*Configure service*:

1. "Task Definition": `testdriven-exercises-stage-td:LATEST_REVISION_NUMBER`
2. "Service name": `testdriven-exercises-stage-service`
3. "Number of tasks": `1`

Click "Next".

*Configure network*:

Select the "Application Load Balancer" under "Load balancer type".

1. "Load balancer name": `testdriven-staging-alb`
2. "Container name : port": `exercises:0:5000`

Click "Add to load balancer".

1. "Listener port": `80:HTTP`
2. "Target group name": `testdriven-exercises-stage-tg`

Click the next button a few times, and then "Create Service".

Wait a few minutes for the container to spin up. Navigate to the EC2 Dashboard, and click "Target Groups". Make sure `testdriven-exercises-stage-tg` has a single registered instance. The instance should be healthy.

## Sanity Check

Update the `exercises` part of the `deploy_cluster` function in *docker-deploy-stage.sh* to call `update_service`:

```
# exercises
service="testdriven-exercises-stage-service"
template="ecs_exercises_stage_taskdefinition.json"
task_template=$(cat "ecs/$template")
task_def=$(printf "$task_template" $AWS_ACCOUNT_ID $AWS_ACCOUNT_ID)
echo "$task_def"
register_definition
update_service
```

Update *swagger.json*:

```
$ python services/swagger/update-spec.py http://LOAD_BALANCER_STAGE_DNS_NAME
```

Commit and push your code to GitHub to trigger a new Travis build. Once done, you should see a new revision associated with each Task Definition and the Services should now be running a new Task based on that revision.

Grab the "DNS name" for the Load Balancer, and then test each URL in the browser:

| Endpoint | HTTP Method | Authenticated? | Result |
|---|---|---|---|
| /auth/register | POST | No | register user |
| /auth/login | POST | No | log in user |
| /auth/logout | GET | Yes | log out user |
| /auth/status | GET | Yes | check user status |
| /users | GET | No | get all users |
| /users/:id | GET | No | get single user |
| /users | POST | Yes (admin) | add a user |
| /users/ping | GET | No | sanity check |
| /exercises | GET | No | get all exercises |
| /exercises | POST | Yes (admin) | add an exercise |

> Remember: If you run into errors, you can always check the logs on CloudWatch or SSH directly into the EC2 instance to debug the containers:
>
> ```
> $ ssh -i ~/.ssh/ecs.pem ec2-user@EC2_PUBLIC_IP
> ```
>
> Be sure to double-check all environment variables!

Make sure the end-to-end tests pass as well:

```
$ export TEST_URL=http://LOAD_BALANCER_STAGE_DNS_NAME
$ export SERVER_URL=http://LOAD_BALANCER_STAGE_DNS_NAME
$ testcafe chrome e2e
```

```
/
✓ should display the exercises correctly if a user is not logged in
✓ should allow a user to submit an exercise if logged in

/
✓ should display the page correctly if a user is not logged in
✓ should display the page correctly if a user is logged in

/login
✓ should display the sign in form
✓ should allow a user to sign in
✓ should throw an error if the credentials are incorrect

/register
✓ should display flash messages correctly

/register
✓ should display the registration form
✓ should allow a user to register
✓ should validate the password field
✓ should throw an error if the username is taken
✓ should throw an error if the email is taken

/status
✓ should not display user info if a user is not logged in
✓ should display user info if a user is logged in

/swagger
✓ should display the swagger docs correctly

/all-users
✓ should display the all-users page correctly if a user is not logged in


17 passed (1m 08s)
```

# ECS Deployment - Production

Finally, let's update production on ECS...

---

## Docker Machine

Set `testdriven-dev` as the active Docker Machine:

```
$ docker-machine env testdriven-dev
$ eval $(docker-machine env testdriven-dev)
```

Set the environment variables:

```
$ export REACT_APP_USERS_SERVICE_URL=http://DOCKER_MACHINE_DEV_IP
$ export TEST_URL=http://DOCKER_MACHINE_DEV_IP
$ export REACT_APP_API_GATEWAY_URL=https://API_GATEWAY_URL
$ export REACT_APP_EXERCISES_SERVICE_URL=http://DOCKER_MACHINE_DEV_IP
```

Fire up the containers:

```
$ docker-compose -f docker-compose-dev.yml up -d --build
```

Create and seed the databases:

```
$ docker-compose -f docker-compose-dev.yml \
  run exercises python manage.py recreate_db

$ docker-compose -f docker-compose-dev.yml \
  run users python manage.py recreate_db

$ docker-compose -f docker-compose-dev.yml \
  run exercises python manage.py seed_db

$ docker-compose -f docker-compose-dev.yml \
  run users python manage.py seed_db
```

Run the full test suite to ensure all is well locally:

```
$ sh test.sh dev
```

## Docker Compose

Add the proper `args` to the `client` service in *docker-compose-prod.yml*:

```
client:
  container_name: client
  build:
    context: ./services/client
    dockerfile: Dockerfile-prod
    args:
      - NODE_ENV=production
      - REACT_APP_USERS_SERVICE_URL=${REACT_APP_USERS_SERVICE_URL}
      - REACT_APP_API_GATEWAY_URL=${REACT_APP_API_GATEWAY_URL}
      - REACT_APP_EXERCISES_SERVICE_URL=${REACT_APP_EXERCISES_SERVICE_URL}
  expose:
    - 80
  depends_on:
    - users
  links:
    - users
```

Then, add the `exercises` and `exercises-db` services:

```
exercises:
  container_name: exercises
  build:
    context: ./services/exercises
    dockerfile: Dockerfile-prod
  expose:
    - 5000
  environment:
    - APP_SETTINGS=project.config.StagingConfig
    - DATABASE_URL=postgres://postgres:postgres@exercises-db:5432/exercises_prod
    - DATABASE_TEST_URL=postgres://postgres:postgres@exercises-db:5432/exercises_test
  depends_on:
    - users
    - exercises-db
  links:
    - users
    - exercises-db

exercises-db:
  container_name: exercises-db
  build:
    context: ./services/exercises/project/db
    dockerfile: Dockerfile
  expose:
    - 5432
  environment:
    - POSTGRES_USER=postgres
```

```
    - POSTGRES_PASSWORD=postgres
```

Set the environment variables in *services/client/Dockerfile-prod*:

```
ARG REACT_APP_EXERCISES_SERVICE_URL
ENV REACT_APP_EXERCISES_SERVICE_URL $REACT_APP_EXERCISES_SERVICE_URL
ARG REACT_APP_API_GATEWAY_URL
ENV REACT_APP_API_GATEWAY_URL $REACT_APP_API_GATEWAY_URL
```

Then, update the run server command in *services/exercises/Dockerfile-prod*:

```
# run server
CMD gunicorn -b 0.0.0.0:5000 manage:app
```

## New Endpoint

To make things a bit easier, let's add a new endpoint to *services/exercises/project/api/exercises.py* that we can use for the Target Group's health check that does not rely on migrations or an authenticated user:

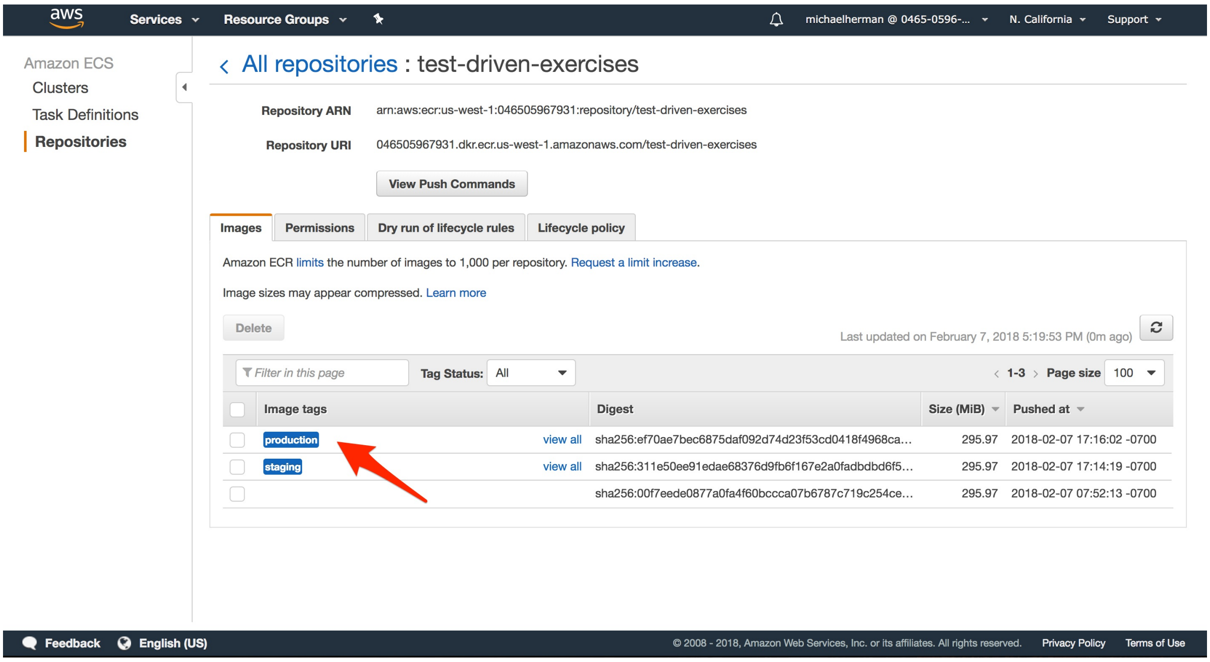
```
@exercises_blueprint.route('/exercises/ping', methods=['GET'])
def ping_pong():
    return jsonify({
        'status': 'success',
        'message': 'pong!'
    })
```

## ECR

Add the `REACT_APP_EXERCISES_SERVICE_URL` to *docker-push.sh*:

```
if [ "$TRAVIS_BRANCH" == "production" ]
then
  export REACT_APP_USERS_SERVICE_URL="http://LOAD_BALANCER_PROD_DNS_NAME"
  export REACT_APP_EXERCISES_SERVICE_URL="http://LOAD_BALANCER_PROD_DNS_NAME"
  export DATABASE_URL="$AWS_RDS_URI"
  export SECRET_KEY="$PRODUCTION_SECRET_KEY"
fi
```

Assuming you are still on the `staging` branch, commit your code and push it up to GitHub. Open a PR against the `production` branch and merge it. Make sure the `production` build passes and that the images were successfully pushed to ECR.




Check out the `production` branch locally:

```
$ git checkout production
$ git pull origin production
```

## Task Definitions

Add `exercises` to the `deploy_cluster` function in *docker-deploy-prod.sh*:

```
# exercises
service="testdriven-exercises-prod-service"
template="ecs_exercises_prod_taskdefinition.json"
task_template=$(cat "ecs/$template")
task_def=$(printf "$task_template" $AWS_ACCOUNT_ID "tbd")
echo "$task_def"
register_definition
```

> `"tbd"` is a placeholder for the RDS URI since we still need to set it up.

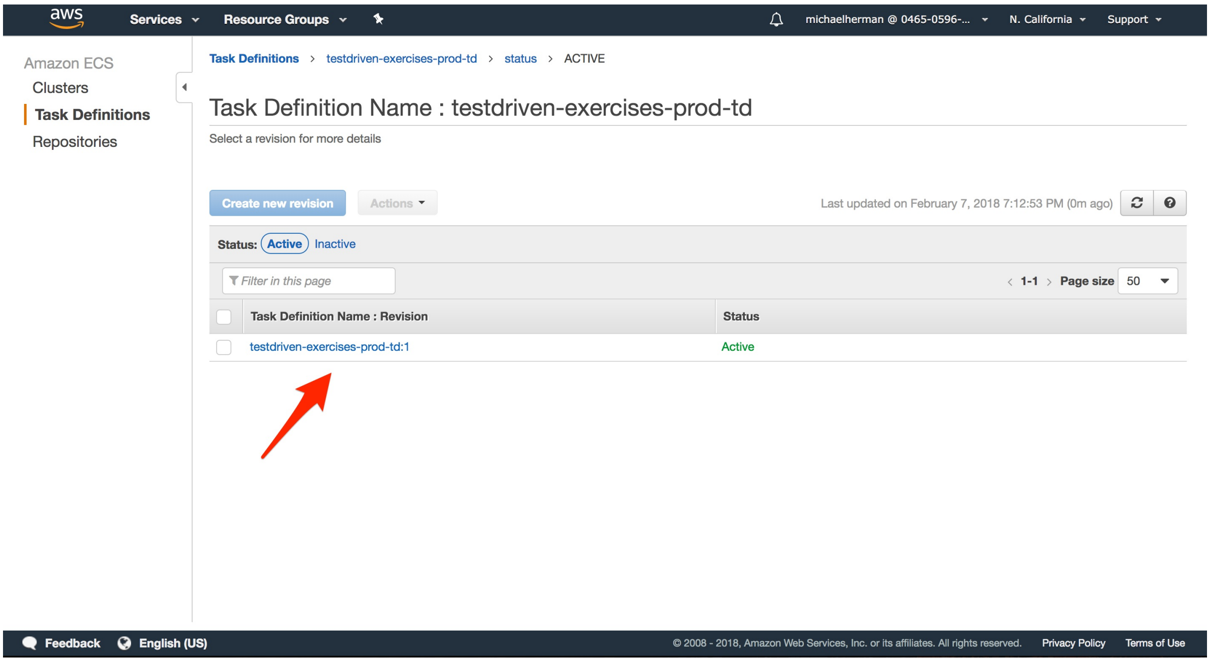
Create a new Task Definition file called *ecs_exercises_prod_taskdefinition.json*:

```
{
  "containerDefinitions": [
    {
      "name": "exercises",
      "image": "%s.dkr.ecr.us-west-1.amazonaws.com/test-driven-exercises:production"
,
      "essential": true,
      "memoryReservation": 300,
      "portMappings": [
```

```
          {
            "hostPort": 0,
            "protocol": "tcp",
            "containerPort": 5000
          }
        ],
        "environment": [
          {
            "name": "APP_SETTINGS",
            "value": "project.config.ProductionConfig"
          },
          {
            "name": "DATABASE_TEST_URL",
            "value": "postgres://postgres:postgres@exercises-db:5432/exercises_test"
          },
          {
            "name": "DATABASE_URL",
            "value": "%s"
          }
        ],
        "logConfiguration": {
          "logDriver": "awslogs",
          "options": {
            "awslogs-group": "testdriven-exercises-prod",
            "awslogs-region": "us-west-1"
          }
        }
      }
    ],
    "family": "testdriven-exercises-prod-td"
  }
```

Be sure to the add the `testdriven-exercises-prod` log group to CloudWatch.

Commit and push your code. Another build should be triggered on Travis. This time ensure that both images and Task Definitions are created.




## Add Target Group

Next, let's add a new Target Group for the `exercises` service. Within Amazon EC2, click "Target Groups", and then create the following Group:

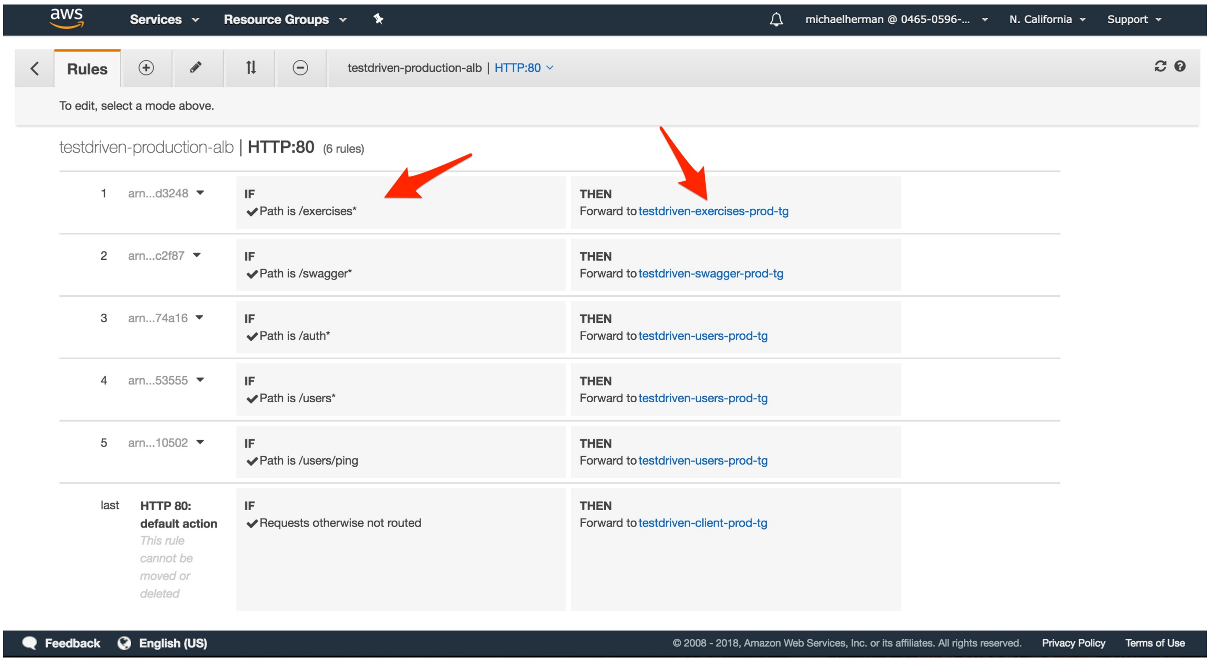
1. "Target group name": `testdriven-exercises-prod-tg`
2. "Port": `5000`
3. Then, under "Health check settings" set the "Path" to `/exercises/ping` .

## Listener

Then, on the "Load Balancers" page, click the `testdriven-production-alb` Load Balancer, and then select the "Listeners" tab. Here, we can add Listeners to the ALB, which are then forwarded to a specific Target Group.

Click the "View/edit rules" for "HTTP : 80", and then add one new rule:

1. If `/exercises*` , Then `testdriven-exercises-prod-tg`




## RDS

Within Amazon RDS, select "Instances" on the sidebar, and then click the "Launch DB Instance" button.

### Step 1: Select engine

> You *probably* want to click the "Only enable options eligible for RDS Free Usage Tier". More info.

Select the "PostgreSQL" engine and click "Next".

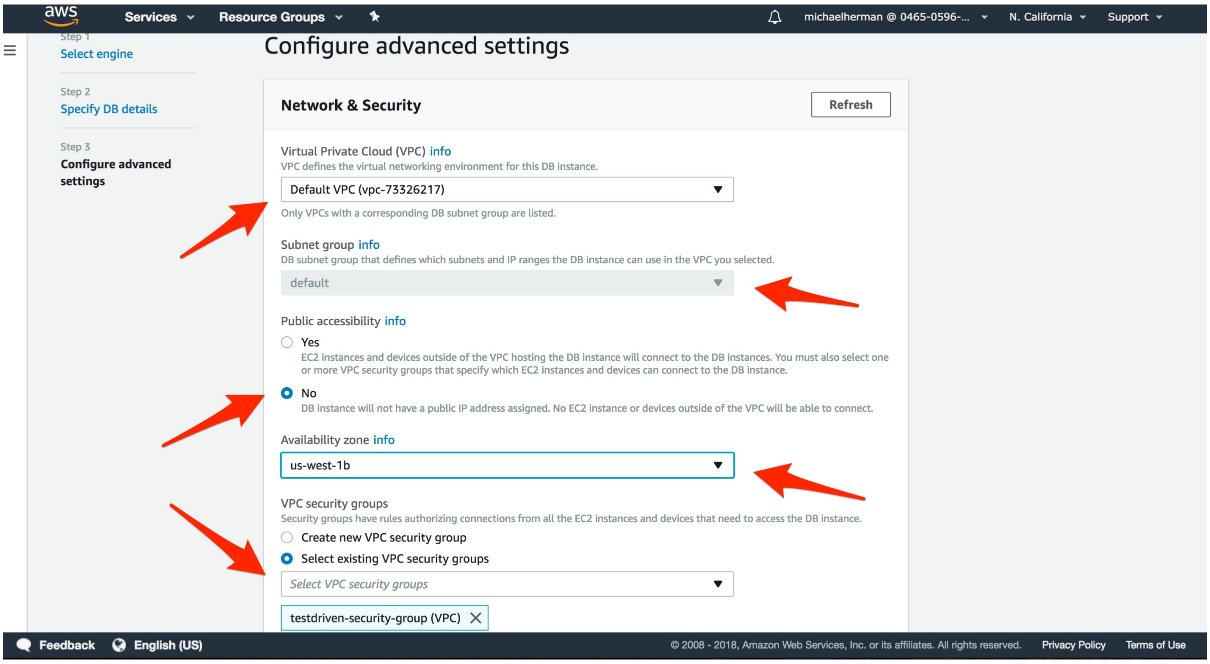
### Step 2: Specify DB details

1. "DB Engine Version": `PostgreSQL 9.6.5-R1`
2. "DB Instance Class": `db.t2.micro`
3. "Multi-AZ Deployment": `No`
4. "Storage Type": `General Purpose (SSD)`
5. "Allocated Storage": `20 GB`
6. "DB Instance Identifier": `testdriven-exercises-production`
7. "Master Username": `webapp`
8. "Master Password": `something_super_secret`

Click "Next".

### Step 3: Configure advanced settings

Under "Network & Security", make sure to pick the "VPC" and "Security group" associated with ALB. Select one of the available "Subnets" as well - either `us-west-1b` or `us-west-1c` .




Change the DB name to `exercises_prod` and then create the new database.

You can quickly check the status via:

```
$ aws rds --region us-west-1 describe-db-instances \
  --db-instance-identifier testdriven-exercises-production \
  --query 'DBInstances[].{DBInstanceStatus:DBInstanceStatus}'

[
    {
        "DBInstanceStatus": "creating"
    }
]
```

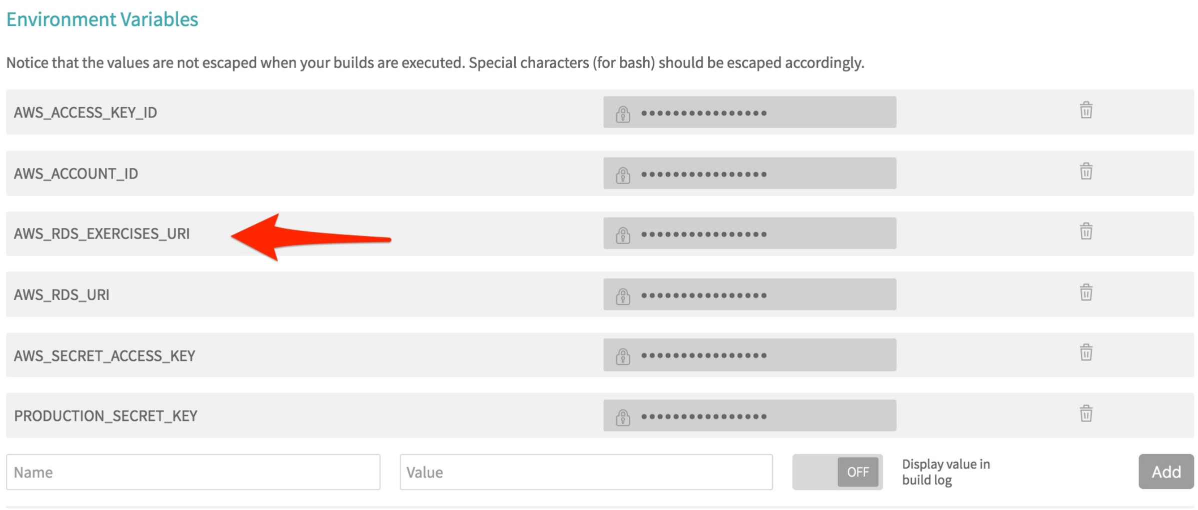
Then, once the status is "available", grab the address:

```
$ aws rds --region us-west-1 describe-db-instances \
  --db-instance-identifier testdriven-exercises-production \
  --query 'DBInstances[].{Address:Endpoint.Address}'
```

Take note of the production URI:

```
postgres://webapp:YOUR_PASSWORD@YOUR_ADDRESS:5432/exercises_prod
```

Add an environment variable called `AWS_RDS_EXERCISES_URI` to the Travis project.

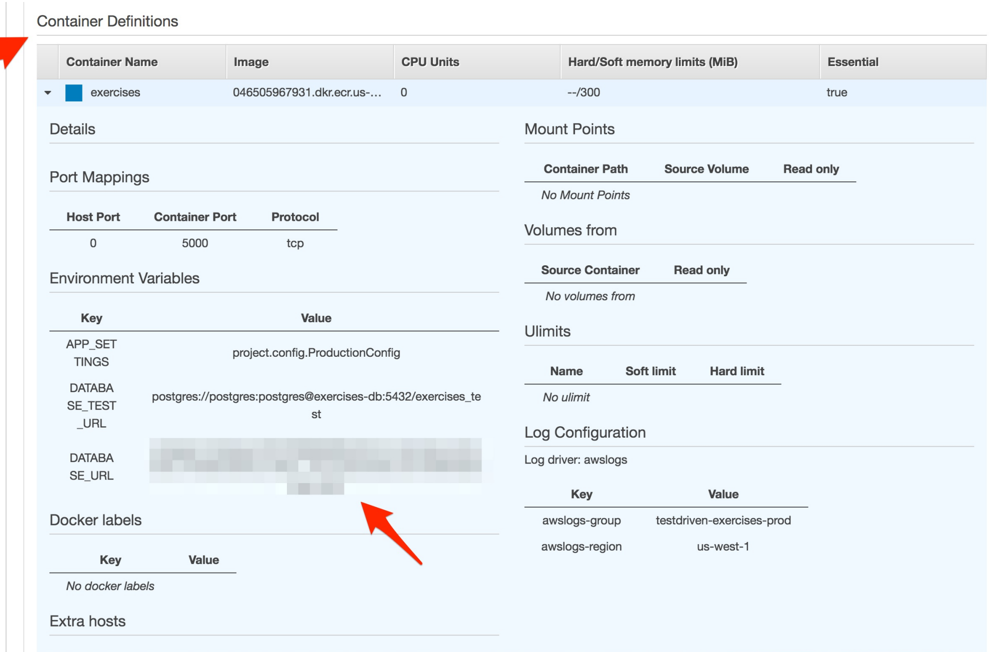



Update `exercises` in the `deploy_cluster` function again in *docker-deploy-prod.sh*, adding the `AWS_RDS_EXERCISES_URI` environment variable:

```
# exercises
service="testdriven-exercises-prod-service"
template="ecs_exercises_prod_taskdefinition.json"
task_template=$(cat "ecs/$template")
task_def=$(printf "$task_template" $AWS_ACCOUNT_ID $AWS_RDS_EXERCISES_URI)
echo "$task_def"
register_definition
```

Commit and push your code to GitHub.

After the Travis build passes, make sure new images were created and revisions to the Task Definitions were added. Also, take note of the current Task Definition revision. Under the "Container Definitions", click the drop-down next to the `exercises` container. Make sure the `DATABASE_URL` environment variable is correct:




## Service

Create the following ECS Service on the `test-driven-production-cluster` Cluster...

### Exercises

*Configure service*:

1. "Task Definition": `testdriven-exercises-prod-td:LATEST_REVISION_NUMBER`
2. "Service name": `testdriven-exercises-prod-service`
3. "Number of tasks": `1`

Click "Next".

*Configure network*:

Select the "Application Load Balancer" under "Load balancer type".

1. "Load balancer name": `testdriven-production-alb`
2. "Container name : port": `exercises:0:5000`

Click "Add to load balancer".

1. "Listener port": `80:HTTP`
2. "Target group name": `testdriven-exercises-prod-tg`

Click the next button a few times, and then "Create Service".

Wait a few minutes for the container to spin up, and then navigate to the EC2 Dashboard and click "Target Groups". Make sure `testdriven-exercises-prod-tg` has a single registered instance each. The instance should be healthy.

## Migrations

Try the `/exercises` endpoint at http://LOAD_BALANCER_PROD_DNS_NAME/exercises. You should see a 500 error since the migrations have not been ran. To do this, let's SSH into the EC2 instance associated with the `testdriven-exercises-prod-tg` Target Group:

```
$ ssh -i ~/.ssh/ecs-west.pem ec2-user@EC2_PUBLIC_IP
```

> You may need to update the permissions on the Pem file - i.e., `chmod 400 ~/.ssh/ecs-west.pem`.

Next, grab the Container ID for `users` (via `docker ps`), enter the shell within the running container, and then update the RDS database:

```
$ docker exec -it Container_ID bash
# python manage.py recreate_db
# python manage.py seed_db
```

Navigate to http://LOAD_BALANCER_PROD_DNS_NAME/exercises again and you should see the exercises.

## Sanity Check

Update `exercises` in the `deploy_cluster` function one final time to call the `update_service` function in *docker-deploy-prod.sh*:

```
# exercises
service="testdriven-exercises-prod-service"
template="ecs_exercises_prod_taskdefinition.json"
task_template=$(cat "ecs/$template")
task_def=$(printf "$task_template" $AWS_ACCOUNT_ID $AWS_RDS_EXERCISES_URI)
echo "$task_def"
register_definition
update_service
```

Update *swagger.json*:

```
$ python services/swagger/update-spec.py http://LOAD_BALANCER_PROD_DNS_NAME
```

Commit and push your code to GitHub to trigger a new Travis build. Once done, you should see a new revision associated with the each Task Definition and the Services should now be running a new Task based on that revision.

Grab the "DNS name" for the Load Balancer, and then test each URL in the browser:

| | HTTP Method | | |
|---|---|---|---|

| | HTTP Method | | |
|---|---|---|---|
| /auth/register | POST | No | register user |
| /auth/login | POST | No | log in user |
| /auth/logout | GET | Yes | log out user |
| /auth/status | GET | Yes | check user status |
| /users | GET | No | get all users |
| /users/:id | GET | No | get single user |
| /users | POST | Yes (admin) | add a user |
| /users/ping | GET | No | sanity check |
| /exercises | GET | No | get all exercises |
| /exercises | POST | Yes (admin) | add an exercise |

> Remember: If you run into errors, you can always check the logs on CloudWatch or SSH directly into the EC2 instance to debug the containers:
>
> ```
> $ ssh -i ~/.ssh/ecs-west.pem ec2-user@EC2_PUBLIC_IP
> ```
>
> Be sure to double-check all environment variables!

Make sure the end-to-end tests pass as well:

```
$ export TEST_URL=http://LOAD_BALANCER_PORD_DNS_NAME
$ export SERVER_URL=http://LOAD_BALANCER_PORD_DNS_NAME
$ testcafe chrome e2e
```

```
/
✓ should display the exercises correctly if a user is not logged in
✓ should allow a user to submit an exercise if logged in

/
✓ should display the page correctly if a user is not logged in
✓ should display the page correctly if a user is logged in

/login
✓ should display the sign in form
✓ should allow a user to sign in
✓ should throw an error if the credentials are incorrect

/register
✓ should display flash messages correctly

/register
✓ should display the registration form
```

```
✓ should allow a user to register
✓ should validate the password field
✓ should throw an error if the username is taken
✓ should throw an error if the email is taken

/status
✓ should not display user info if a user is not logged in
✓ should display user info if a user is logged in

/swagger
✓ should display the swagger docs correctly

/all-users
✓ should display the all-users page correctly if a user is not logged in


17 passed (1m 09s)
```

# Autoscaling

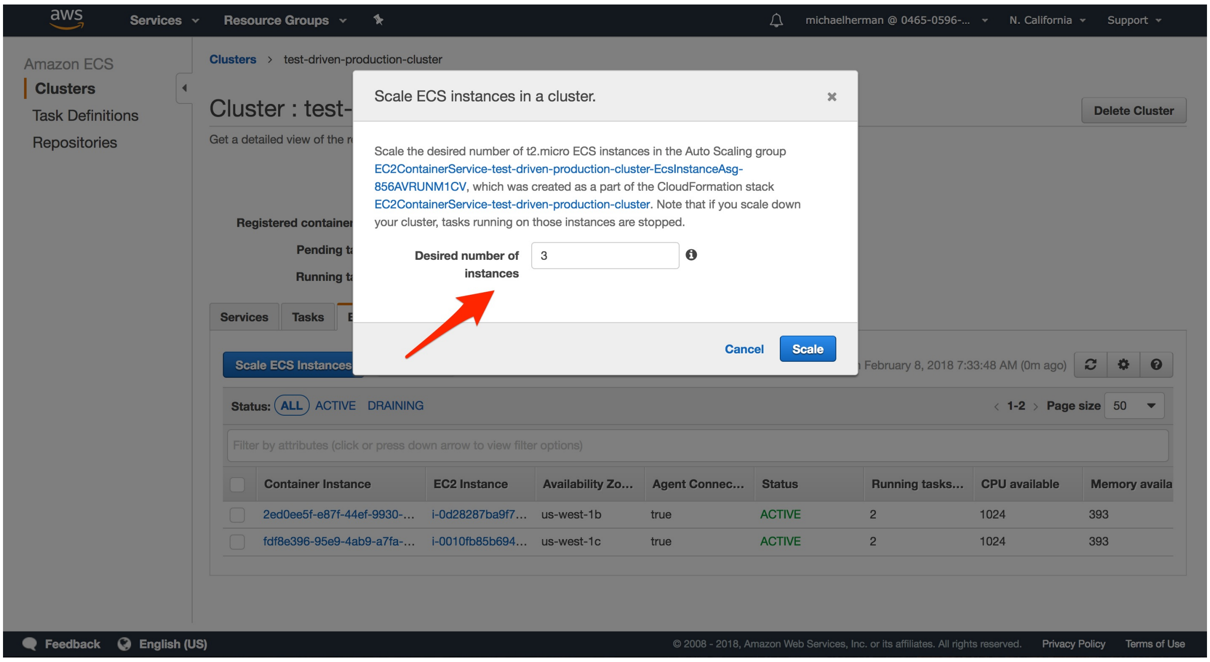
Let's take a quick look at scaling...

---

You can scale up or down at both the Cluster (adding additional EC2 instances) and Service (adding more Tasks to an existing instance) level.

## Cluster

To manually scale, navigate to the Cluster and click the "ECS Instance"" tab. Then, click the "Scale ECS Instances" button and provide the desired number of instances you'd like to scale up (or down) to.

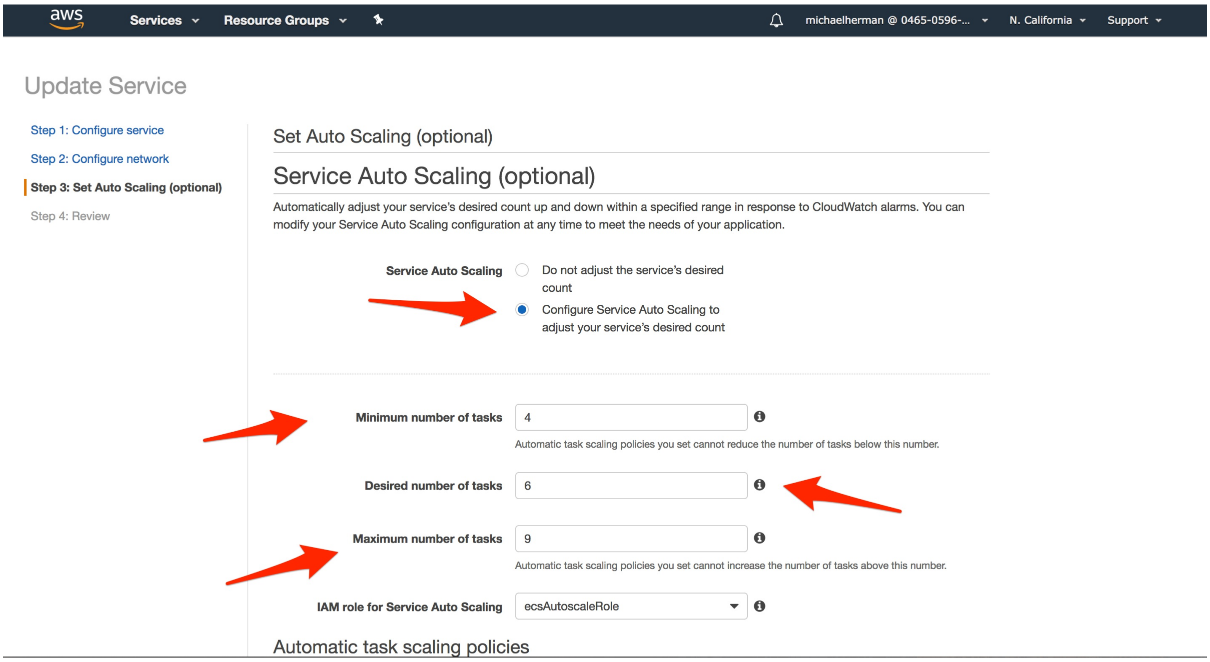


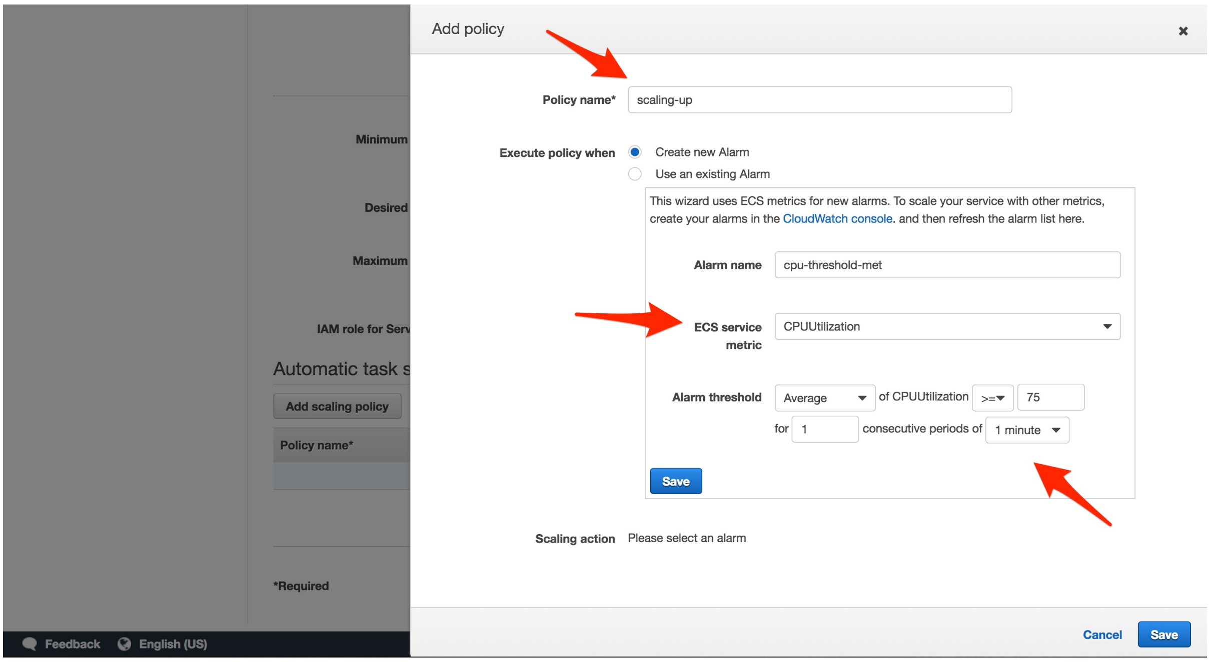

You can automate this process by setting up an Auto Scaling Group. Review the Scaling Container Instances with CloudWatch Alarms tutorial for more info.

For more on this, review the Service Auto Scaling with CloudWatch Service Utilization Metrics tutorial.

## Service

You can also scale Tasks up (or down) at the Service-level.

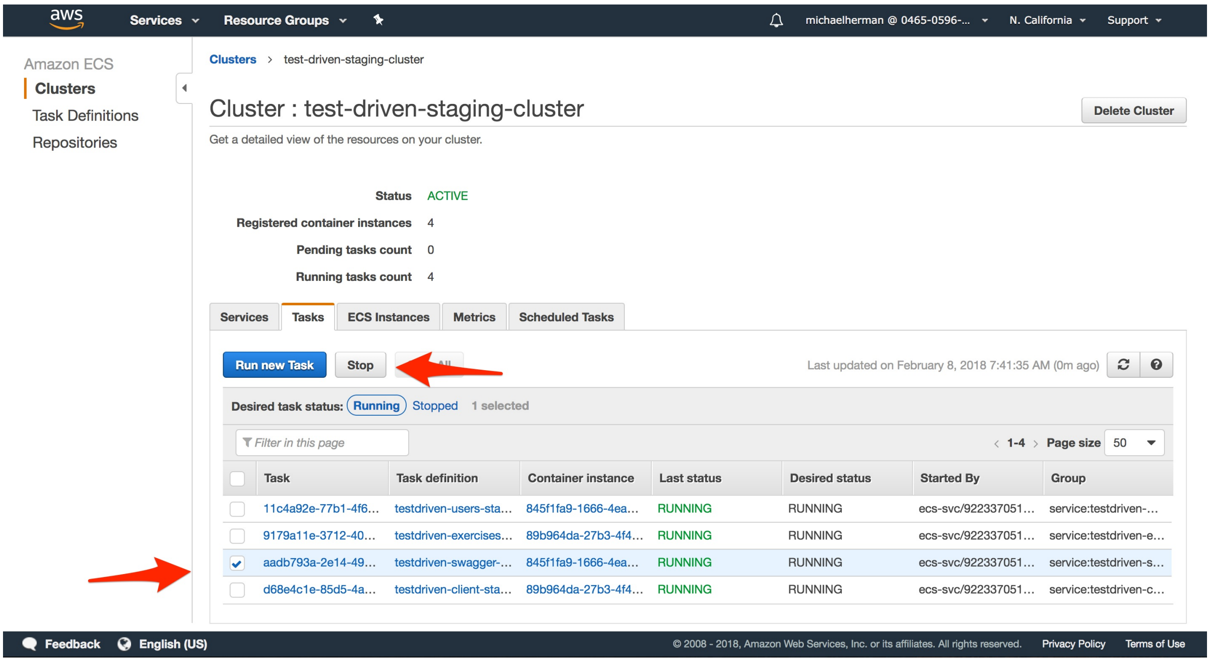


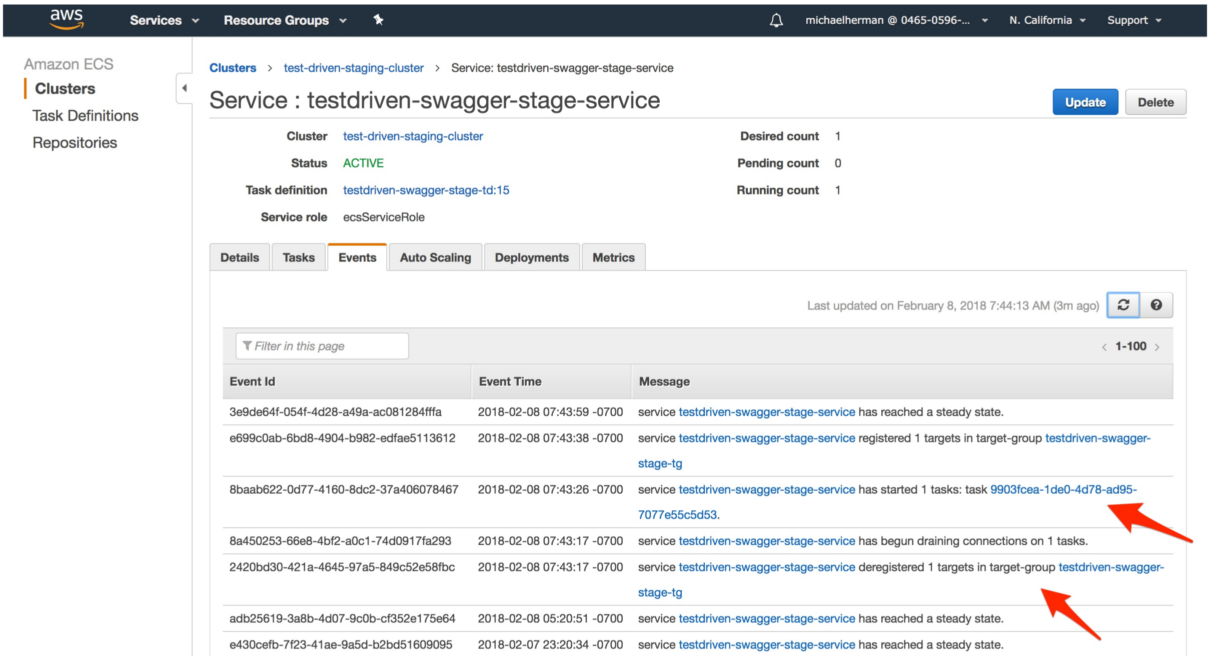

## Load Balancing

What happens if an instance goes down?

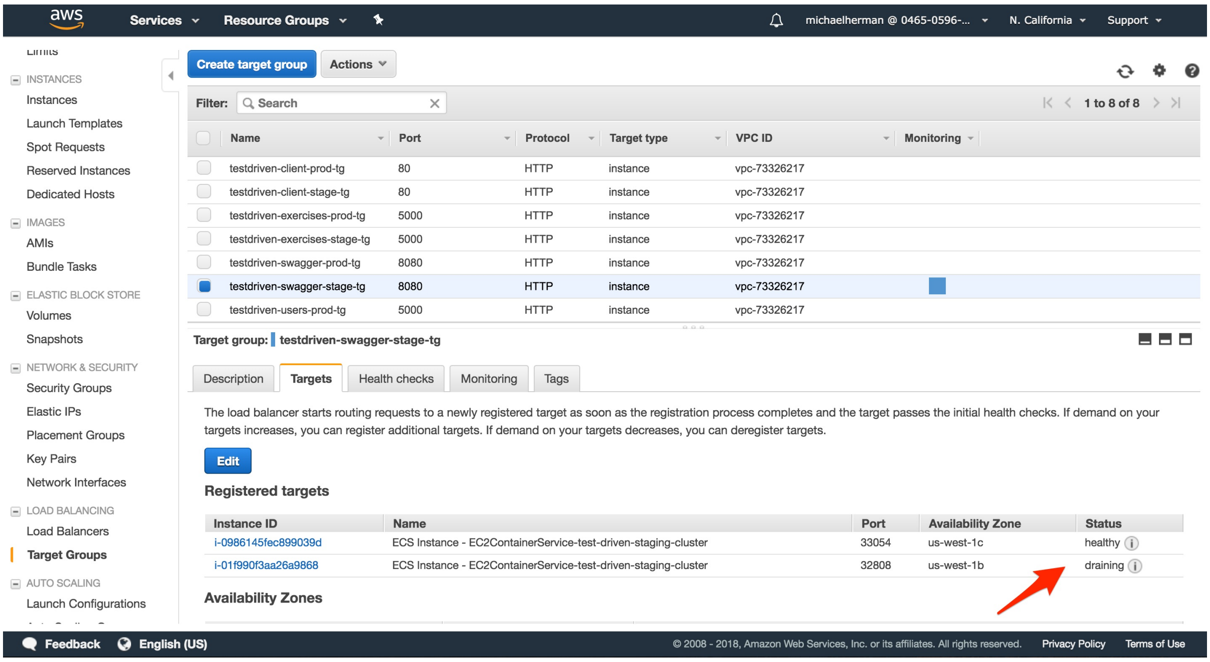
Within the "Tasks" tab on the Cluster, click the checkbox next to a currently running Task and click the "Stop" button.



Click the "Services" tab and then the link for the Service associated with the Task you just stopped. On the "Events" tab you should see an event for the Task that you stopped being drained as well as an event for a new Task starting. Perfect.



Also, if you navigate to the relevant Target Group on the EC2 Dashboard, you'll see one instance draining as well as a new instance spinning up which should be healthy.



# Workflow

Updated reference guide...

## Development Environment

The following commands are for spinning up all the containers in your `development` environment...

### Docker Machine

Set `testdriven-dev` as the active Docker Machine:

```
$ docker-machine env testdriven-dev
$ eval $(docker-machine env testdriven-dev)
```

### Environment Variables

```
$ export REACT_APP_USERS_SERVICE_URL=http://DOCKER_MACHINE_DEV_IP
$ export TEST_URL=http://DOCKER_MACHINE_DEV_IP
$ export REACT_APP_API_GATEWAY_URL=https://API_GATEWAY_URL
$ export REACT_APP_EXERCISES_SERVICE_URL=http://DOCKER_MACHINE_DEV_IP
```

### Start

Update *swagger.json*:

```
$ python services/swagger/update-spec.py http://DOCKER_MACHINE_DEV_IP
```

Build the images:

```
$ docker-compose -f docker-compose-dev.yml build
```

Run the containers:

```
$ docker-compose -f docker-compose-dev.yml up -d
```

Create and seed the databases:

```
$ docker-compose -f docker-compose-dev.yml \
  run exercises python manage.py recreate_db

$ docker-compose -f docker-compose-dev.yml \
  run users python manage.py recreate_db

$ docker-compose -f docker-compose-dev.yml \
```

```
  run exercises python manage.py seed_db

$ docker-compose -f docker-compose-dev.yml \
  run users python manage.py seed_db
```

Run the full test suite:

```
$ sh test.sh dev
```

Run the unit and integration tests:

```
$ docker-compose -f docker-compose-dev.yml \
  run users python manage.py test

$ docker-compose -f docker-compose-dev.yml \
  run exercises python manage.py test
```

Lint:

```
$ docker-compose -f docker-compose-dev.yml \
  run users flake8 project

$ docker-compose -f docker-compose-dev.yml \
  run exercises flake8 project
```

Run the client-side tests:

```
$ docker-compose -f docker-compose-dev.yml \
  run client npm test -- --verbose
```

Run the e2e tests:

```
$ testcafe chrome e2e
```

Enter psql:

```
$ docker exec -ti users-db psql -U postgres -W
```

**Stop**

Stop the containers:

```
$ docker-compose -f docker-compose-dev.yml stop
```

Bring down the containers:

```
$ docker-compose -f docker-compose-dev.yml down
```

### Aliases

To save some precious keystrokes, create aliases for both the `docker-compose` and `docker-machine` commands - `dc` and `dm`, respectively.

Simply add the following lines to your *.bashrc* file:

```
alias dc='docker-compose'
alias dm='docker-machine'
```

Save the file, then execute it:

```
$ source ~/.bashrc
```

Test out the new aliases!

> On Windows? You will first need to create a PowerShell Profile (if you don't already have one), and then you can add the aliases to it using Set-Alias - i.e., `Set-Alias dc docker-compose`.

### "Saved" State

Is the VM stuck in a "Saved" state?

```
$ docker-machine ls

NAME    ACTIVE    DRIVER        STATE      URL                          SWARM    DOCKER
      ERRORS
aws      *        amazonec2     Running    tcp://34.207.173.181:2376             v17.05.0
-ce
dev      -        virtualbox    Saved                                            Unknown
```

To break out of this, you'll need to power off the VM:

1. Start virtualbox - `virtualbox`
2. Select the VM and click "start"
3. Exit the VM and select "Power off the machine"
4. Exit virtualbox

The VM should now have a "Stopped" state:

```
$ docker-machine ls

NAME    ACTIVE    DRIVER        STATE      URL                          SWARM    DOCKER
      ERRORS
aws      *        amazonec2     Running    tcp://34.207.173.181:2376             v17.05.0
```

```
-ce
dev     -         virtualbox   Stopped                                     Unknown
```

Now you can start the machine:

```
$ docker-machine start dev
```

It should be "Running":

```
$ docker-machine ls

NAME    ACTIVE    DRIVER       STATE     URL                         SWARM    DOCKER
      ERRORS
aws      *        amazonec2    Running   tcp://34.207.173.181:2376            v17.05.0
-ce
dev      -        virtualbox   Running   tcp://192.168.99.100:2376            v17.05.0
-ce
```

## Other Commands

Want to force a build?

```
$ docker-compose build --no-cache
```

Remove exited containers:

```
$ docker rm -v $(docker ps -a -q -f status=exited)
```

Remove images:

```
$ docker rmi $(docker images -q)
```

Remove untagged images:

```
$ docker rmi $(docker images | grep “^<none>” | awk ‘{print $3}’)
```

Reset Docker environment back to localhost, unsetting all Docker environment variables:

```
$ eval $(docker-machine env -u)
```

## Development Workflow

Try out the following development workflow...

**Development**:

1. Create a new feature branch from the `master` branch
2. Make an arbitrary change; commit and push it up to GitHub
3. After the build passes, open a PR against the `development` branch to trigger a new build on Travis
4. Merge the PR after the build passes

**Staging**:

1. Open PR from the `development` branch against the `staging` branch to trigger a new build on Travis
2. Merge the PR after the build passes to trigger a new build
3. After the build passes, images are created, tagged `staging`, and pushed to ECR, revisions are added to the Task Definitions, and the Service is updated

**Production**:

1. Open PR from the `staging` branch against the `production` branch to trigger a new build on Travis
2. Merge the PR after the build passes to trigger a new build
3. After the build passes, images are created, tagged `production`, and pushed to ECR, revisions are added to the Task Definitions, and the Service is updated
4. Merge the changes into the `master` branch

# Structure

At the end of part 6, your project structure should look like this:

```
├── README.md
├── docker-compose-dev.yml
├── docker-compose-prod.yml
├── docker-compose-stage.yml
├── docker-deploy-prod.sh
├── docker-deploy-stage.sh
├── docker-push.sh
├── e2e
│   ├── exercise.js
│   ├── index.test.js
│   ├── login.test.js
│   ├── message.test.js
│   ├── register.test.js
│   ├── status.test.js
│   ├── swagger.test.js
│   └── users.test.js
├── ecs
│   ├── ecs_client_prod_taskdefinition.json
│   ├── ecs_client_stage_taskdefinition.json
│   ├── ecs_exercises_prod_taskdefinition.json
│   ├── ecs_exercises_stage_taskdefinition.json
│   ├── ecs_swagger_prod_taskdefinition.json
│   ├── ecs_swagger_stage_taskdefinition.json
│   ├── ecs_users_prod_taskdefinition.json
│   └── ecs_users_stage_taskdefinition.json
├── package.json
├── services
│   ├── client
│   │   ├── Dockerfile-dev
│   │   ├── Dockerfile-prod
│   │   ├── Dockerfile-stage
│   │   ├── README.md
│   │   ├── build
│   │   ├── conf
│   │   │   └── conf.d
│   │   │       └── default.conf
│   │   ├── coverage
│   │   ├── package.json
│   │   ├── public
│   │   │   ├── favicon.ico
│   │   │   ├── index.html
│   │   │   ├── main.css
│   │   │   └── manifest.json
│   │   └── src
│   │       ├── App.jsx
```

```
│   │       ├── components
│   │       │   ├── About.jsx
│   │       │   ├── AddUser.jsx
│   │       │   ├── Exercises.jsx
│   │       │   ├── Footer.css
│   │       │   ├── Footer.jsx
│   │       │   ├── Logout.jsx
│   │       │   ├── Message.jsx
│   │       │   ├── NavBar.css
│   │       │   ├── NavBar.jsx
│   │       │   ├── UserStatus.jsx
│   │       │   ├── UsersList.jsx
│   │       │   ├── __tests__
│   │       │   │   ├── About.test.jsx
│   │       │   │   ├── AddUser.test.jsx
│   │       │   │   ├── App.test.jsx
│   │       │   │   ├── Exercises.test.jsx
│   │       │   │   ├── Footer.test.jsx
│   │       │   │   ├── Form.test.jsx
│   │       │   │   ├── FormErrors.test.jsx
│   │       │   │   ├── Logout.test.jsx
│   │       │   │   ├── Message.test.jsx
│   │       │   │   ├── NavBar.test.jsx
│   │       │   │   ├── UsersList.test.jsx
│   │       │   │   └── __snapshots__
│   │       │   │       ├── About.test.jsx.snap
│   │       │   │       ├── AddUser.test.jsx.snap
│   │       │   │       ├── Exercises.test.jsx.snap
│   │       │   │       ├── Footer.test.jsx.snap
│   │       │   │       ├── Form.test.jsx.snap
│   │       │   │       ├── FormErrors.test.jsx.snap
│   │       │   │       ├── Logout.test.jsx.snap
│   │       │   │       ├── Message.test.jsx.snap
│   │       │   │       ├── NavBar.test.jsx.snap
│   │       │   │       └── UsersList.test.jsx.snap
│   │       │   └── forms
│   │       │       ├── Form.jsx
│   │       │       ├── FormErrors.css
│   │       │       ├── FormErrors.jsx
│   │       │       └── form-rules.js
│   │       ├── index.js
│   │       ├── logo.svg
│   │       ├── registerServiceWorker.js
│   │       └── setupTests.js
│   ├── exercises
│   │   ├── Dockerfile-dev
│   │   ├── Dockerfile-prod
│   │   ├── Dockerfile-stage
│   │   ├── entrypoint-stage.sh
│   │   ├── entrypoint.sh
│   │   ├── manage.py
```

```
|   |   ├── project
|   |   |   ├── __init__.py
|   |   |   ├── api
|   |   |   |   ├── __init__.py
|   |   |   |   ├── base.py
|   |   |   |   ├── exercises.py
|   |   |   |   ├── models.py
|   |   |   |   └── utils.py
|   |   |   ├── config.py
|   |   |   ├── db
|   |   |   |   ├── Dockerfile
|   |   |   |   └── create.sql
|   |   |   └── tests
|   |   |       ├── __init__.py
|   |   |       ├── base.py
|   |   |       ├── test_base.py
|   |   |       ├── test_config.py
|   |   |       ├── test_exercises_api.py
|   |   |       ├── test_exercises_model.py
|   |   |       └── utils.py
|   |   └── requirements.txt
|   ├── lambda
|   |   └── handler.py
|   ├── nginx
|   |   ├── Dockerfile-dev
|   |   ├── Dockerfile-prod
|   |   ├── Dockerfile-stage
|   |   ├── dev.conf
|   |   └── prod.conf
|   ├── swagger
|   |   ├── Dockerfile-dev
|   |   ├── Dockerfile-prod
|   |   ├── Dockerfile-stage
|   |   ├── nginx.conf
|   |   ├── start.sh
|   |   ├── swagger.json
|   |   └── update-spec.py
|   └── users
|       ├── Dockerfile-dev
|       ├── Dockerfile-prod
|       ├── Dockerfile-stage
|       ├── entrypoint-stage.sh
|       ├── entrypoint.sh
|       ├── htmlcov
|       ├── manage.py
|       ├── migrations
|       ├── project
|       |   ├── __init__.py
|       |   ├── api
|       |   |   ├── __init__.py
|       |   |   ├── auth.py
```

```
│       │   │   ├── models.py
│       │   │   ├── templates
│       │   │   │   └── index.html
│       │   │   ├── users.py
│       │   │   └── utils.py
│       │   ├── config.py
│       │   ├── db
│       │   │   ├── Dockerfile
│       │   │   └── create.sql
│       │   └── tests
│       │       ├── __init__.py
│       │       ├── base.py
│       │       ├── test_auth.py
│       │       ├── test_config.py
│       │       ├── test_user_model.py
│       │       ├── test_users.py
│       │       └── utils.py
│       └── requirements.txt
└── test.sh
```

Code for part 6: https://github.com/testdrivenio/testdriven-app-2.2/releases/tag/part6

# Part 7

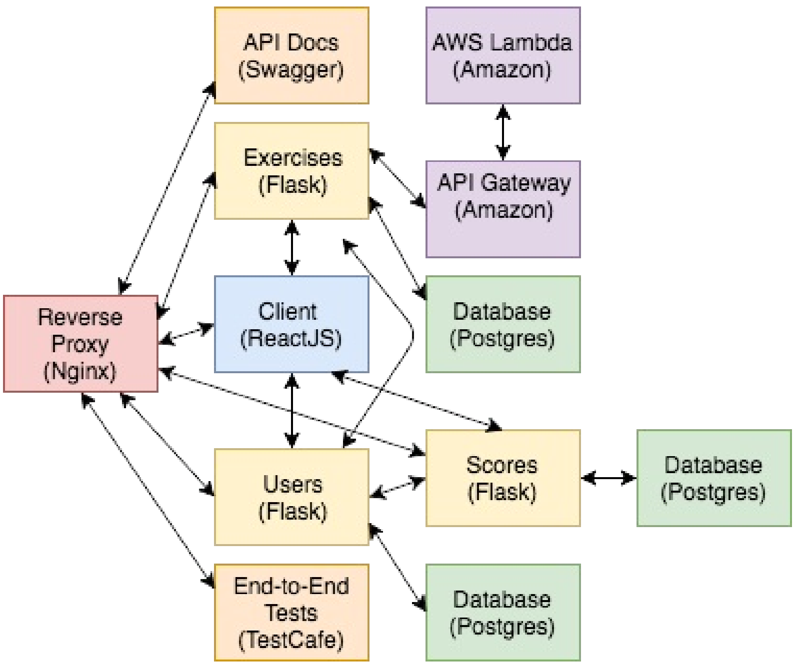
In part 7, we'll refactor the *AWS Lambda* function to make it dynamic so it can be used with more than one exercise, introduce *type checking* on the client-side with *React PropTypes*, add *Redux* to the React app, and update a number of components. We'll also introduce another new *Flask* service to manage scores. Again, we'll update the staging and production environments on ECS.

## Objectives

By the end of part 7, you will be able to...

1. Enable type checking with React PropTypes
2. Describe the difference between Redux actions, reducers, and stores as well as how each relates to state
3. Add Redux to a React app
4. Practice test driven development while refactoring code
5. Integrate a new microservice in the existing set of services
6. Refactor an AWS Lambda function and update API Gateway
7. Update the staging and production environments on Amazon ECS

## App




Check out the live apps, running on EC2 -

1. Production
2. Staging

You can also test out the following endpoints...

| Endpoint | HTTP Method | Authenticated? | Result |
|---|---|---|---|
| /auth/register | POST | No | register user |
| /auth/login | POST | No | log in user |
| /auth/logout | GET | Yes | log out user |
| /auth/status | GET | Yes | check user status |
| /users | GET | No | get all users |
| /users/:id | GET | No | get single user |
| /users | POST | Yes (admin) | add a user |
| /users/ping | GET | No | sanity check |
| /exercises | GET | No | get all exercises |
| /exercises | POST | Yes (admin) | add an exercise |
| /scores/ping | GET | No | sanity check |
| /scores | GET | No | get all scores |
| /scores/user | GET | Yes | get all scores by user id |
| /scores/user/:id | GET | Yes | get single score by user id |
| /scores | POST | Yes | add a score |
| /scores/:exercise_id | PUT | Yes | update a score by exercise id |

Grab the code: https://github.com/testdrivenio/testdriven-app-2.2/releases/tag/part7

## Dependencies

You will use the following dependencies in part 7:

1. prop-types v15.6.0
2. Redux v3.7.2
3. React Redux v5.0.6
4. Flask v0.12.2
5. Flask-SQLAlchemy v2.3.2
6. psycopg2 v2.7.3.2
7. Flask-Testing v0.6.2
8. Gunicorn v19.7.1
9. Coverage.py v4.4.2
10. flake8 v3.5.0
11. Flask Debug Toolbar v0.10.1
12. Flask-CORS v3.0.3
13. Flask-Migrate v2.1.1
14. Requests v2.18.4

# Lambda Refactor

In this lesson, we'll update the Lambda function as well as the code submission workflow...

---

## Lambda

Let's update *services/lambda/handler.py* so that it takes the test code along with the expected output, rather than hard coding them:

```
import sys
from io import StringIO


def lambda_handler(event, context):
    # get code, test, and solution from payload
    code = event['answer']
    test = event['test']
    solution = event['solution']
    test_code = code + '\nprint(' + test + ')'
    # capture stdout
    buffer = StringIO()
    sys.stdout = buffer
    # execute code
    try:
        exec(test_code)
    except Exception:
        return False
    # return stdout
    sys.stdout = sys.stdout
    # check
    if buffer.getvalue()[:-1] == solution:
        return True
    return False
```

Now, since we are using the current Lambda in production, let's spin up a new function for testing:

1. Name: `execute_python3_code_test`
2. Runtime: `Python 3.6`
3. Role: `Choose an existing role`
4. Existing role: `basic_api_gateway_access`

Add the above code to the inline code editor, and then add several tests:

1. `sum` :

   ```
   {
   ```

```
    "answer": "def sum(x,y):\n    return x+y",
    "test": "sum(1, 1)",
    "solution": "2"
  }
```

2. `diff` :

```
  {
    "answer": "def diff(x,y):\n    return x-y",
    "test": "diff(1, 1)",
    "solution": "0"
  }
```

3. `sumlist` :

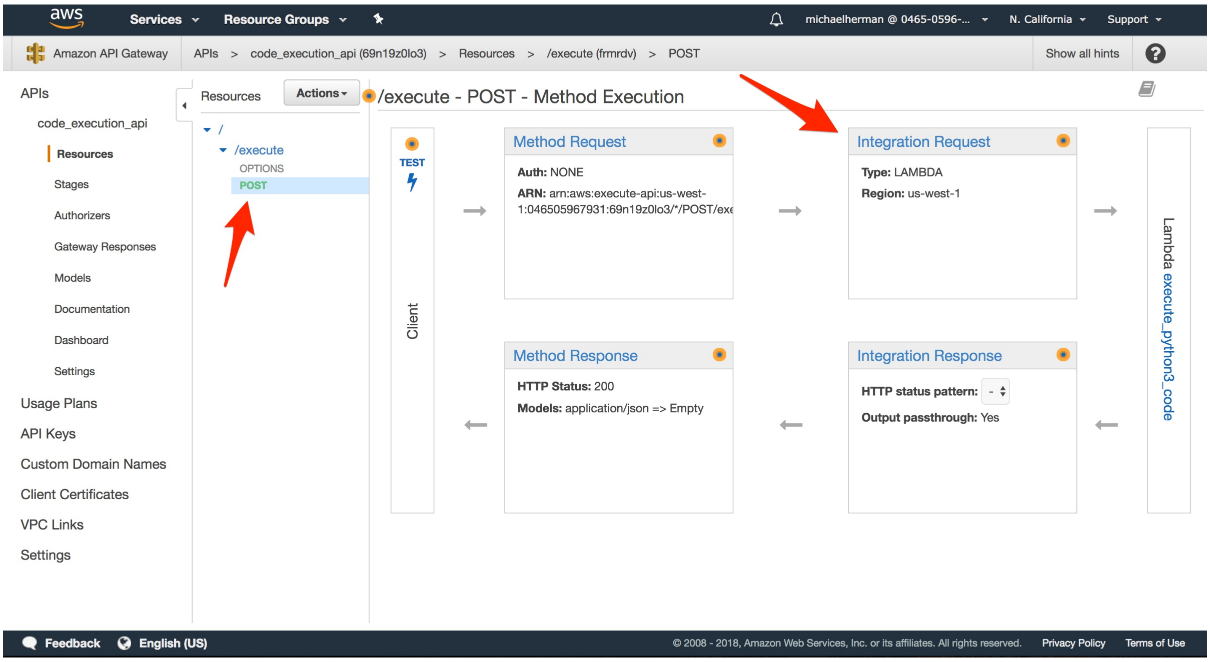
```
  {
    "answer": "def sum_list(x):\n    return sum(x)",
    "test": "sum_list([10, 11, 12, 13, 14, 15, 16])",
    "solution": "91"
  }
```

Make sure they all pass.

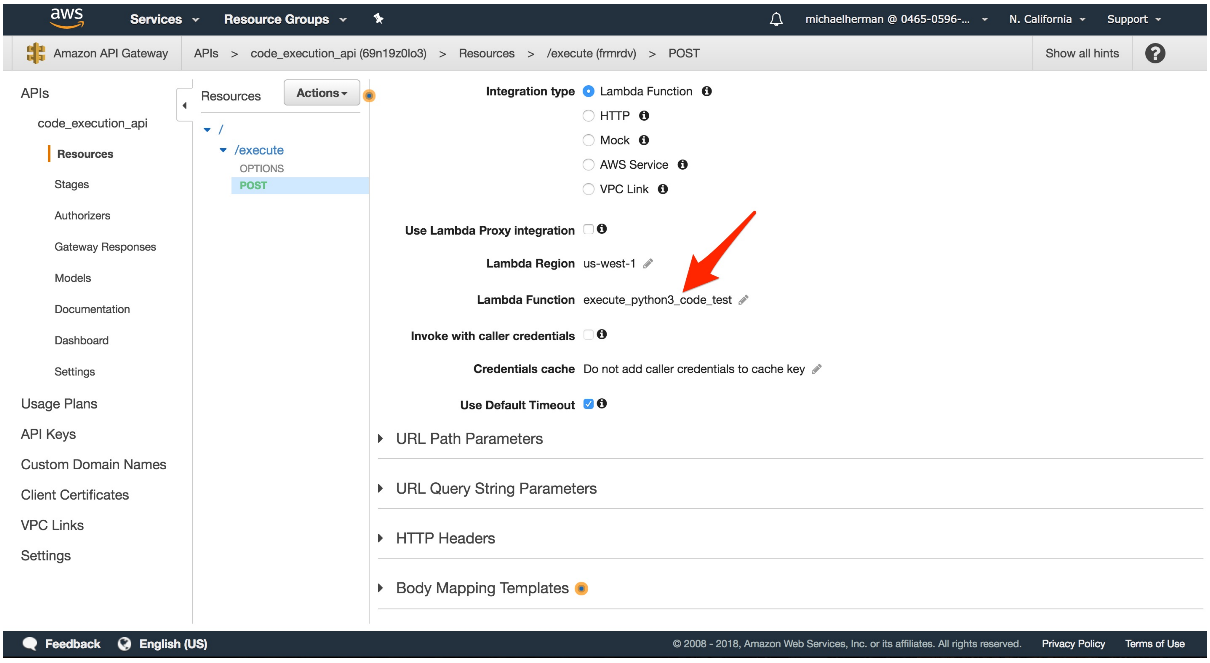
## API Gateway

Navigate to the API Gateway page and select the `code_execution_api` . We can create a new Stage to associate an end point to the new created Lambda function.

Select the "POST" method and then click the "Integration Request" link:



Update the "Lambda Function" to `execute_python3_code_test` :




Select "Deploy API" from the "Actions" drop-down, and then create a new "Deployment stage" called `v2` .

## Test via cURL

First, let's ensure that `v1` is still working:

```
$ curl -H "Content-Type: application/json" -X POST \
  -d '{"answer":"def sum(x,y):\n    return x+y"}' \
  https://API_GATEWAY_URL/v1/execute
```

Then, test `v2` ...

### Sum

True:

```
$ curl -H "Content-Type: application/json" -X POST \
https://API_GATEWAY_URL/v2/execute \
-d @- << EOF

{
    "answer": "def sum(x,y):\n    return x+y",
    "test": "sum(20, 30)",
    "solution": "50"
}
EOF
```

False:

```
$ curl -H "Content-Type: application/json" -X POST \
https://API_GATEWAY_URL/v2/execute \
-d @- << EOF

{
    "answer": "def sum(x,y):\n    return x+y",
    "test": "sum(20, 30)",
    "solution": "incorrect"
}
EOF
```

**Diff**

True:

```
$ curl -H "Content-Type: application/json" -X POST \
https://API_GATEWAY_URL/v2/execute \
-d @- << EOF

{
    "answer": "def diff(x,y):\n    return x-y",
    "test": "diff(80, 20)",
    "solution": "60"
}
EOF
```

False:

```
$ curl -H "Content-Type: application/json" -X POST \
https://API_GATEWAY_URL/v2/execute \
-d @- << EOF

{
    "answer": "def diff(x,y):\n    return x-y",
    "test": "diff(80, 20)",
    "solution": "incorrect"
}
EOF
```

## Sum List

True:

```
$ curl -H "Content-Type: application/json" -X POST \
https://API_GATEWAY_URL/v2/execute \
-d @- << EOF
```

```
{
    "answer": "def sum_list(x):\n    return sum(x)",
    "test": "sum_list([10, 11, 12, 13, 14, 15, 16])",
    "solution": "91"
}
EOF
```

False:

```
$ curl -H "Content-Type: application/json" -X POST \
https://API_GATEWAY_URL/v2/execute \
-d @- << EOF

{
    "answer": "def sum_list(x):\n    return sum(x)",
    "test": "sum_list([10, 11, 12, 13, 14, 15, 16])",
    "solution": "incorrect"
}
EOF
```

## Update Exercises

Finally, we need to remove the `print` statements from the seed command in *services/exercises/manage.py* since we're handling that in the Lamdbda function itself:

```
@cli.command()
def seed_db():
    """Seeds the database."""
    db.session.add(Exercise(
        body=('Define a function called sum that takes two integers as '
              'arguments and returns their sum.'),
        test_code='sum(2, 3)',
        test_code_solution='5'
    ))
    db.session.add(Exercise(
        body=('Define a function called reverse that takes a string as '
              'an argument and returns the string in reversed order.'),
        test_code='reverse("racecar")',
        test_code_solution='racecar'
    ))
    db.session.add(Exercise(
        body=('Define a function called factorial that takes a random number '
              'as an argument and then returns the factorial of that given '
              'number.'),
        test_code='factorial(5)',
        test_code_solution='120'
    ))
    db.session.commit()
```

# Exercise Component

In this lesson, we'll refactor the `Exercises` component and add an `Exercise` component...

## Docker Machine

Set `testdriven-dev` as the active Docker Machine:

```
$ docker-machine env testdriven-dev
$ eval $(docker-machine env testdriven-dev)
```

Set the environment variables:

```
$ export REACT_APP_USERS_SERVICE_URL=http://DOCKER_MACHINE_DEV_IP
$ export TEST_URL=http://DOCKER_MACHINE_DEV_IP
$ export REACT_APP_API_GATEWAY_URL=https://API_GATEWAY_URL
$ export REACT_APP_EXERCISES_SERVICE_URL=http://DOCKER_MACHINE_DEV_IP
```

Update *swagger.json*:

```
$ python services/swagger/update-spec.py http://DOCKER_MACHINE_DEV_IP
```

Spin up the app:

```
$ docker-compose -f docker-compose-dev.yml up -d --build
```

Create and seed the databases:

```
$ docker-compose -f docker-compose-dev.yml \
  run exercises python manage.py recreate_db

$ docker-compose -f docker-compose-dev.yml \
  run users python manage.py recreate_db

$ docker-compose -f docker-compose-dev.yml \
  run exercises python manage.py seed_db

$ docker-compose -f docker-compose-dev.yml \
  run users python manage.py seed_db
```

Ensure the app is working in the browser, and then run the tests:

```
$ sh test.sh dev
```

## Exercise Component

Let's abstract out the actual exercise to a new component to keep things clean, starting with the tests...

### Test

Update the following two test cases in *services/client/src/components/__tests__/Exercises.test.jsx*:

```
test('Exercises renders properly when not authenticated', () => {
  const onDidMount = jest.fn();
  Exercises.prototype.componentDidMount = onDidMount;
  const wrapper = shallow(<Exercises isAuthenticated={false}/>);
  wrapper.setState({exercises : exercises});
  const alert = wrapper.find('.alert');
  expect(alert.length).toBe(1);
  expect(alert.get(0).props.children[1].props.children).toContain(
    'Please log in to submit an exercise.')
});

test('Exercises renders properly when authenticated', () => {
  const onDidMount = jest.fn();
  Exercises.prototype.componentDidMount = onDidMount;
  const wrapper = shallow(<Exercises isAuthenticated={true}/>);
  wrapper.setState({exercises : exercises});
  const alert = wrapper.find('.alert');
  expect(alert.length).toBe(0);
});
```

Then, add a new file called *Exercise.test.jsx*:

```
import React from 'react';
import { shallow, mount } from 'enzyme';
import renderer from 'react-test-renderer';

import AceEditor from 'react-ace';
jest.mock('react-ace');

import Exercise from '../Exercise';

const testData = {
  exercise: {
    id: 0,
    body: `Define a function called sum that takes two integers
    as arguments and returns their sum.`
  },
  editor: {
    value: '# Enter your code here.',
```

```
    button: {
      isDisabled: false,
    },
    showGrading: false,
    showCorrect: false,
    showIncorrect: false,
  },
  isAuthenticated: false,
  onChange: jest.fn(),
  submitExercise: jest.fn(),
}

test('Exercise renders properly', () => {
  const wrapper = shallow(<Exercise {...testData}/>);
  const heading = wrapper.find('h4');
  expect(heading.length).toBe(1);
  expect(heading.text()).toBe(testData.exercise.body)
});

test('Exercises renders a snapshot properly when not authenticated', () => {
  const tree = renderer.create(<Exercise {...testData}/>).toJSON();
  expect(tree).toMatchSnapshot();
});

test('Exercises renders a snapshot properly when authenticated', () => {
  testData.isAuthenticated = true;
  const tree = renderer.create(<Exercise {...testData}/>).toJSON();
  expect(tree).toMatchSnapshot();
});
```

### Code

To get the tests to pass, first update the `render` method in *services/client/src/components/Exercises.jsx*

```
render() {
  return (
    <div>
      <h1>Exercises</h1>
      <hr/><br/>
      {!this.props.isAuthenticated &&
        <div>
          <div className="alert alert-warning">
            <span
              className="glyphicon glyphicon-exclamation-sign"
              aria-hidden="true">
            </span>
            <span> Please log in to submit an exercise.</span>
          </div>
          <br/>
```

```
          </div>
        }
        {this.state.exercises.length &&
          <Exercise
            exercise={this.state.exercises[0]}
            editor={this.state.editor}
            isAuthenticated={this.props.isAuthenticated}
            onChange={this.onChange}
            submitExercise={this.submitExercise}
          />
      }
      </div>
    )
  };
```

Then, update the imports:

```
import React, { Component } from 'react';
import axios from 'axios';

import Exercise from './Exercise';
```

For the new component, add a new file called *Exercise.jsx* to "services/client/src/components"

```
import React from 'react';
import { Button, Glyphicon } from 'react-bootstrap';
import AceEditor from 'react-ace';
import 'brace/mode/python';
import 'brace/theme/solarized_dark';

const Exercise = (props) => {
  return (
    <div key={props.exercise.id}>
      <h4>{props.exercise.body}</h4>
        <AceEditor
          mode="python"
          theme="solarized_dark"
          name={(props.exercise.id).toString()}
          fontSize={14}
          height={'175px'}
          showPrintMargin={true}
          showGutter={true}
          highlightActiveLine={true}
          value={props.editor.value}
          style={{
            marginBottom: '10px'
          }}
          onChange={props.onChange}
        />
```

```
        {props.isAuthenticated &&
          <div>
            <Button
              bsStyle="primary"
              bsSize="small"
              onClick={props.submitExercise}
              disabled={props.editor.button.isDisabled}
            >Run Code</Button>
          {props.editor.showGrading &&
            <h4>
               
              <Glyphicon glyph="repeat" className="glyphicon-spin"/>
               
              Grading...
            </h4>
          }
          {props.editor.showCorrect &&
            <h4>
               
              <Glyphicon glyph="ok" className="glyphicon-correct"/>
               
              Correct!
            </h4>
          }
          {props.editor.showIncorrect &&
            <h4>
               
              <Glyphicon glyph="remove" className="glyphicon-incorrect"/>
               
              Incorrect!
            </h4>
          }
          </div>
        }
      <br/><br/>
    </div>
  )
};

export default Exercise;
```

Run the tests:

```
$ docker-compose -f docker-compose-dev.yml \
  run client npm test -- --coverage
```

They should pass:

```
PASS  src/components/__tests__/Form.test.jsx
```

```
PASS  src/components/__tests__/Exercises.test.jsx
PASS  src/components/__tests__/Message.test.jsx
PASS  src/components/__tests__/FormErrors.test.jsx
PASS  src/components/__tests__/UsersList.test.jsx
PASS  src/components/__tests__/Exercise.test.jsx
PASS  src/components/__tests__/Logout.test.jsx
PASS  src/components/__tests__/App.test.jsx
PASS  src/components/__tests__/AddUser.test.jsx
PASS  src/components/__tests__/NavBar.test.jsx
PASS  src/components/__tests__/Footer.test.jsx
PASS  src/components/__tests__/About.test.jsx

Test Suites: 12 passed, 12 total
Tests:       40 passed, 40 total
Snapshots:   15 passed, 15 total
Time:        14.851s
Ran all test suites.
```

Commit and push your code to GitHub once done.

# AJAX Refactor

With the `Exercises` and `Exercise` components in place along with the new Lambda, we can now refactor the actual AJAX request to tie everything together...

## Update Endpoint

With `testdriven-dev` as the active Docker Machine, update the `REACT_APP_API_GATEWAY_URL` environment variable:

```
$ export REACT_APP_API_GATEWAY_URL=https://API_GATEWAY_URL/v2/execute
```

Update the containers:

```
$ docker-compose -f docker-compose-dev.yml up -d
```

Run the tests. You should see a failing end-to-end test:

```
/
✓ should display the exercises correctly if a user is not logged in
✖ should allow a user to submit an exercise if logged in

  1) AssertionError: expected false to be truthy

     Browser: Chrome 62.0.3202 / Mac OS X 10.12.0

        30 |    .navigateTo(`${TEST_URL}/`)
        31 |    .expect(Selector('H1').withText('Exercises').exists).ok()
        32 |    .expect(Selector('.alert-warning').withText('Please log in to submi
t an
     exercise.').exists).notOk()
        33 |    .expect(Selector('button').withText('Run Code').exists).ok()
        34 |    .click(Selector('button').withText('Run Code'))
      > 35 |    .expect(Selector('h4').withText('Incorrect!').exists).ok()
        36 |});
        37 |
```

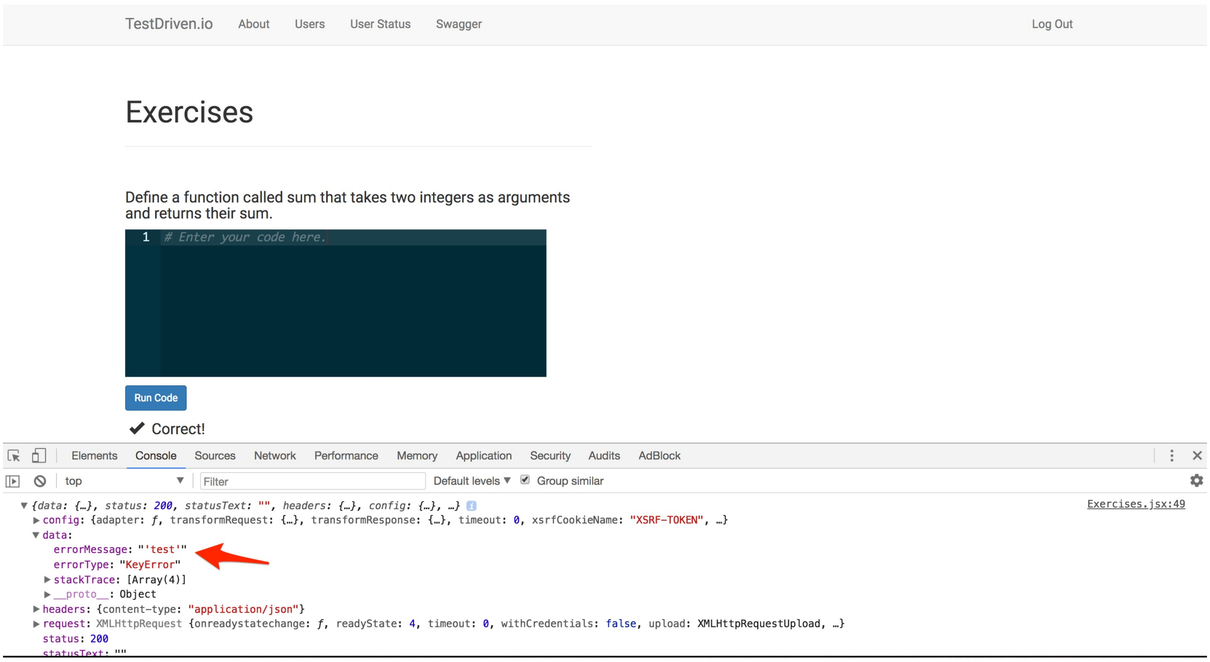
What's happening?

Head to the browser. After logging in, try running the code without entering anything in the code editor. You should see the `Correct` message displayed, which is, in fact, incorrect. To debug, log the response from the AJAX request within the `submitExercise` method:

```
submitExercise(event) {
```

```
  event.preventDefault();
  const newState = this.state.editor;
  newState.showGrading = true;
  newState.showCorrect = false;
  newState.showIncorrect = false;
  newState.button.isDisabled = true;
  this.setState(newState);
  const data = { answer: this.state.editor.value };
  const url = process.env.REACT_APP_API_GATEWAY_URL;
  axios.post(url, data)
  .then((res) => {
    console.log(res);
    newState.showGrading = false
    newState.button.isDisabled = false
    if (res.data) {newState.showCorrect = true};
    if (!res.data) {newState.showIncorrect = true};
    this.setState(newState);
  })
  .catch((err) => {
    newState.showGrading = false
    newState.button.isDisabled = false
    console.log(err);
  })
};
```

Test it out in the browser again. You should see the response object in the JavaScript console:




Make the updates to `submitExercise()`:

```
submitExercise(event) {
  event.preventDefault();
  const newState = this.state.editor;
```

```
    newState.showGrading = true;
    newState.showCorrect = false;
    newState.showIncorrect = false;
    newState.button.isDisabled = true;
    this.setState(newState);
    const data = { answer: this.state.editor.value };
    const url = process.env.REACT_APP_API_GATEWAY_URL;
    axios.post(url, data)
    .then((res) => {
      newState.showGrading = false
      newState.button.isDisabled = false
      if (res.data && !res.data.errorType) {newState.showCorrect = true};
      if (!res.data || res.data.errorType) {newState.showIncorrect = true};
      this.setState(newState);
    })
    .catch((err) => {
      newState.showGrading = false
      newState.button.isDisabled = false
      console.log(err);
    })
  };
```

## Update AJAX

To get the editor working, we need to pass along the test code and solution with the answer.

## Test

Update the `should allow a user to submit an exercise if logged in` test case in *e2e/exercises.test.js*:

```
test(`should allow a user to submit an exercise if logged in`, async (t) => {
  await t
    .navigateTo(`${TEST_URL}/register`)
    .typeText('input[name="username"]', username)
    .typeText('input[name="email"]', email)
    .typeText('input[name="password"]', password)
    .click(Selector('input[type="submit"]'))
  await t
    .navigateTo(`${TEST_URL}/`)
    .expect(Selector('H1').withText('Exercises').exists).ok()
    .expect(Selector('.alert-warning').withText('Please log in to submit an exercis
e.').exists).notOk()
    .expect(Selector('button').withText('Run Code').exists).ok()
    .click(Selector('button').withText('Run Code'))
    .expect(Selector('h4').withText('Incorrect!').exists).ok()
    .expect(Selector('h4').withText('Correct!').exists).notOk()
  await t
    .navigateTo(`${TEST_URL}/`)
```

```
    .selectText(Selector('textarea'))
    .pressKey('home')
    for (let i = 0; i < 23; i++) {
      await t.pressKey('delete')
    }
  await t
    .typeText('textarea', 'def sum(x,y):\nreturn x+y')
    .click(Selector('button').withText('Run Code'))
    .expect(Selector('h4').withText('Incorrect!').exists).notOk()
    .expect(Selector('h4').withText('Correct!').exists).ok()
});
```

## Code

Update the `onClick` handler on the button within *services/client/src/components/Exercise.jsx* to pass along the associated exercise `id` :

```
<Button
  bsStyle="primary"
  bsSize="small"
  onClick={(evt) => props.submitExercise(evt, props.exercise.id)}
  disabled={props.editor.button.isDisabled}
>Run Code
</Button>
```

Then, edit `submitExercise()` again, to obtain the exercise from the state object and update the payload:

```
submitExercise(event, id) {
  event.preventDefault();
  const newState = this.state.editor;
  const exercise = this.state.exercises.filter(el => el.id === id)[0]
  newState.showGrading = true;
  newState.showCorrect = false;
  newState.showIncorrect = false;
  newState.button.isDisabled = true;
  this.setState(newState);
  const data = {
    answer: this.state.editor.value,
    test: exercise.test_code,
    solution: exercise.test_code_solution,
  };
  const url = process.env.REACT_APP_API_GATEWAY_URL;
  axios.post(url, data)
  .then((res) => {
    newState.showGrading = false
    newState.button.isDisabled = false
    if (res.data && !res.data.errorType) {newState.showCorrect = true};
```

```
    if (!res.data || res.data.errorType) {newState.showIncorrect = true};
    this.setState(newState);
  })
  .catch((err) => {
    newState.showGrading = false
    newState.button.isDisabled = false
    console.log(err);
  })
};
```

Test the app in the browser, and then run the end-to-end tests:

```
✓ should display the exercises correctly if a user is not logged in
✓ should allow a user to submit an exercise if logged in

/
✓ should display the all-users page correctly if a user is not logged in

/login
✓ should display the sign in form
✓ should allow a user to sign in
✓ should throw an error if the credentials are incorrect

/register
✓ should display flash messages correctly

/register
✓ should display the registration form
✓ should allow a user to register
✓ should validate the password field
✓ should throw an error if the username is taken
✓ should throw an error if the email is taken

/status
✓ should not display user info if a user is not logged in
✓ should display user info if a user is logged in

/swagger
✓ should display the swagger docs correctly

/all-users
✓ should display the all-users page correctly if a user is not logged in


16 passed (1m 54s)
```

Did you notice that with the previous and current versions of the Lambda that you can cheat if you know the solution?

```
def sum(n1, n2):
    return 5
```

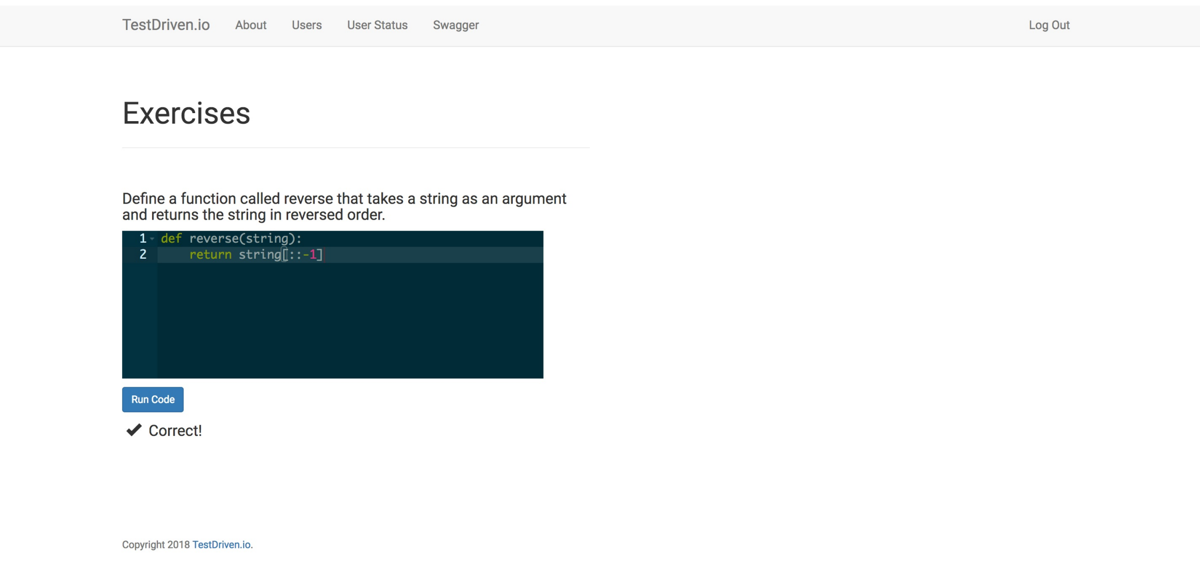
Maybe we should provide a few different solutions to test against? At the very least, it's something to think about as we move forward. Feel free to implement it on your own.

## Manually Test

We can manually test the other two exercises by updating the `exercise` prop passed down to the `Exercise` component.

### Reverse String

```
<Exercise
  exercise={this.state.exercises[1]}
  editor={this.state.editor}
  isAuthenticated={this.props.isAuthenticated}
  onChange={this.onChange}
  submitExercise={this.submitExercise}
/>
```

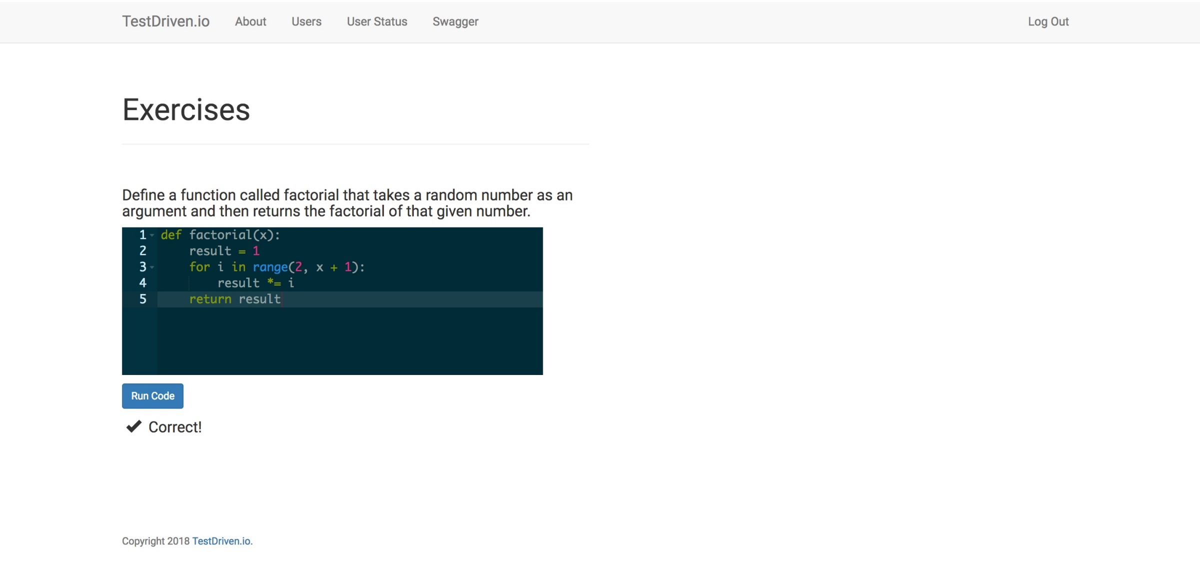



---

### Factorial

```
<Exercise
  exercise={this.state.exercises[2]}
  editor={this.state.editor}
  isAuthenticated={this.props.isAuthenticated}
  onChange={this.onChange}
```

```
  submitExercise={this.submitExercise}
/>
```




---

What happens if you change `range` to `xrange`? How would you debug this inside of the Lambda? Perhaps instead of doing the evaluation within the Lambda, and swallowing the error in the `except`, we can just execute the code against the test case and return the results? Refactor on your own.

---

Run all the tests one last time, then commit your code.

# Type Checking

In this lesson, we'll introduce type checking on the client-side with PropTypes...

---

## PropTypes

As our app continues to grow and scale, you may have ran into issues with certain props not being what you originally thought. To maintain consistency and add predicability, we can add a type-checking library called PropTypes to our application. This will help prevent unwanted and/or incorrect props from being passed to a component

PropTypes can be used for validating types - e.g., once a string, always a string - as well as for documenting out components, telling other developers (as well as our future selves) what props can be expected for a given component.

To start, add prop-types to the *services/client/package.json* file:

```
"dependencies": {
  "axios": "^0.17.1",
  "prop-types": "^15.6.0",
  "react": "^16.2.0",
  "react-ace": "^5.9.0",
  "react-bootstrap": "^0.32.1",
  "react-dom": "^16.2.0",
  "react-router-bootstrap": "^0.24.4",
  "react-router-dom": "^4.2.2",
  "react-scripts": "1.1.0"
},
```

Update the containers to install the dependency:

```
$ docker-compose -f docker-compose-dev.yml up -d --build
```

Now, we can validate each prop being passed in to the child components. Let's start with the `Exercise` component since we just added it.

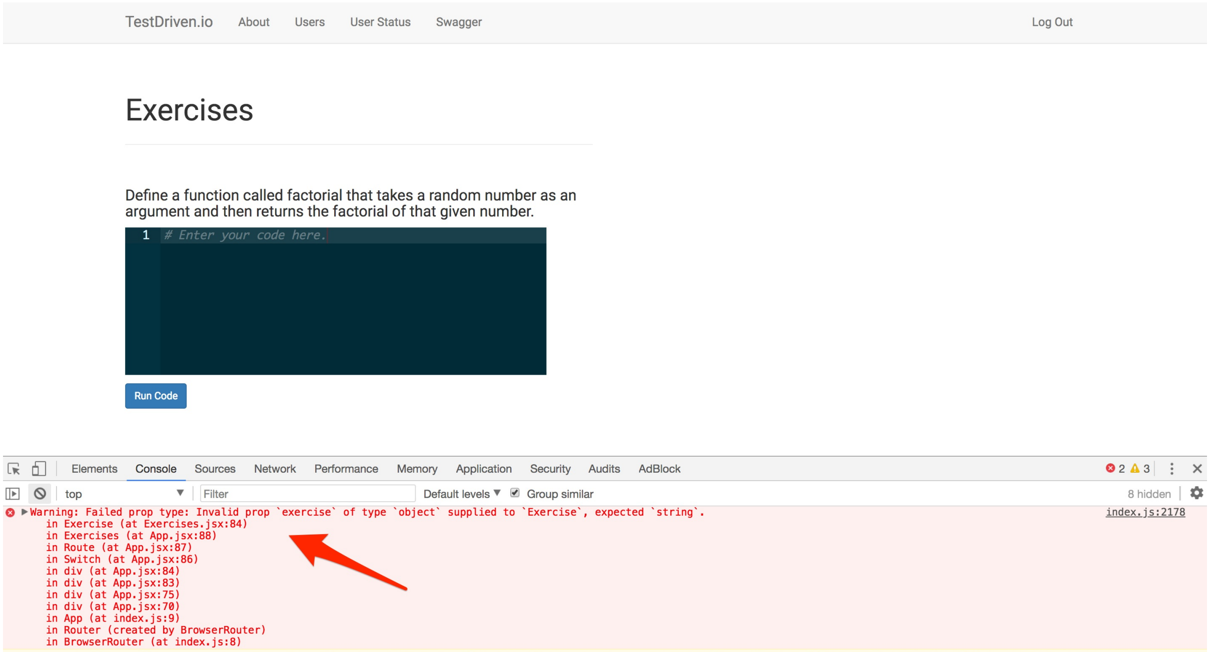
## Exercise Component

Simply import `PropTypes` at the top and then set the `propTypes` property at the bottom:

```
import React from 'react';
import PropTypes from 'prop-types';
import { Button, Glyphicon } from 'react-bootstrap';
import AceEditor from 'react-ace';
```

```
import 'brace/mode/python';
import 'brace/theme/solarized_dark';

const Exercise = (props) => {
  ...
};

Exercise.propTypes = {
  exercise: PropTypes.object.isRequired,
  editor: PropTypes.object.isRequired,
  isAuthenticated: PropTypes.bool.isRequired,
  onChange: PropTypes.func.isRequired,
  submitExercise: PropTypes.func.isRequired,
};

export default Exercise;
```

That's it. Turn to the browser. You shouldn't see anything different. Try changing the `exercise` type to `PropTypes.string.isRequired`. You should see the following error, indicating that type checking is working:




Revert the change.

Now, since the `exercise` and `editor` props are both objects, we can also validate the shape of those objects:

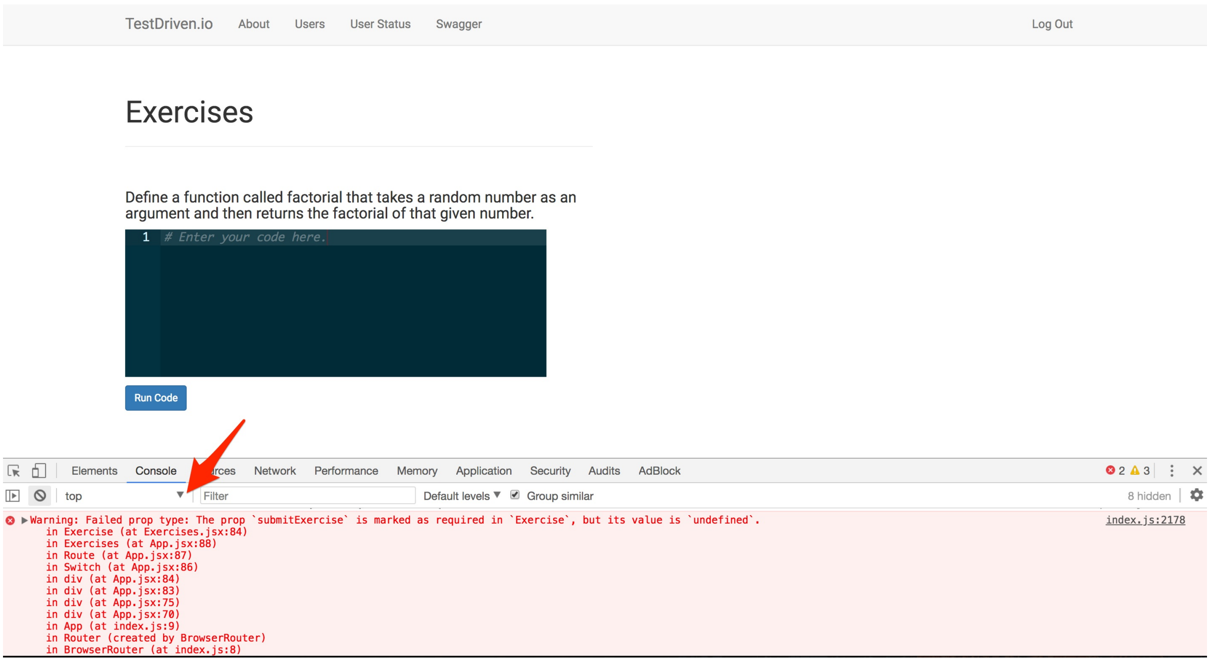
```
Exercise.propTypes = {
  exercise: PropTypes.shape({
    body: PropTypes.string.isRequired,
    id: PropTypes.number.isRequired,
    test_code: PropTypes.string.isRequired,
```

```
      test_code_solution: PropTypes.string.isRequired,
    }).isRequired,
    editor: PropTypes.shape({
      button: PropTypes.object.isRequired,
      showCorrect: PropTypes.bool.isRequired,
      showGrading: PropTypes.bool.isRequired,
      showIncorrect: PropTypes.bool.isRequired,
      value: PropTypes.string.isRequired,
    }).isRequired,
    isAuthenticated: PropTypes.bool.isRequired,
    onChange: PropTypes.func.isRequired,
    submitExercise: PropTypes.func.isRequired,
  };
```

What happens if you don't pass a required prop down? Test it out by removing the `submitExercise` prop:

```
<Exercise
  exercise={this.state.exercises[0]}
  editor={this.state.editor}
  isAuthenticated={this.props.isAuthenticated}
  onChange={this.onChange}
/>
```

You should see the following error:




Perfect. Revert the change again, and then be sure to review all the available PropTypes here before moving on.

Run the tests:

```
$ docker-compose -f docker-compose-dev.yml \
  run client npm test --watchAll
```

You should see a new warning:

```
PASS  src/components/__tests__/Form.test.jsx
PASS  src/components/__tests__/Message.js
PASS  src/components/__tests__/Exercises.test.jsx
PASS  src/components/__tests__/App.test.jsx
PASS  src/components/__tests__/Exercise.test.jsx
 ● Console

   console.error node_modules/fbjs/lib/warning.js:33
     Warning: Failed prop type:
      The prop `exercise.test_code` is marked as required in `Exercise`,
      but its value is `undefined`.
         in Exercise (at Exercise.test.jsx:31)

 PASS  src/components/__tests__/UsersList.test.jsx
 PASS  src/components/__tests__/NavBar.test.jsx
 PASS  src/components/__tests__/FormErrors.test.jsx
 PASS  src/components/__tests__/Message.test.jsx
 PASS  src/components/__tests__/Logout.test.jsx
 PASS  src/components/__tests__/AddUser.test.jsx
 PASS  src/components/__tests__/Footer.test.jsx
 PASS  src/components/__tests__/About.test.jsx
```

Let's update the tests to ensure that no errors are thrown by stubbing `console.error` and asserting that it is not being called.

Update *services/client/src/components/__tests__/Exercise.test.jsx*:

```
import React from 'react';
import { shallow, mount } from 'enzyme';
import renderer from 'react-test-renderer';

import AceEditor from 'react-ace';
jest.mock('react-ace');

import Exercise from '../Exercise';

const testData = {
  exercise: {
    id: 0,
    body: `Define a function called sum that takes two integers
    as arguments and returns their sum.`,
    test_code: 'sum(2,2)',
    test_code_solution: '4'
```

```
  },
  editor: {
    value: '# Enter your code here.',
    button: {
      isDisabled: false,
    },
    showGrading: false,
    showCorrect: false,
    showIncorrect: false,
  },
  isAuthenticated: false,
  onChange: jest.fn(),
  submitExercise: jest.fn(),
}

beforeEach(() => {
  console.error = jest.fn();
  console.error.mockClear();
});

test('Exercise renders properly', () => {
  const wrapper = shallow(<Exercise {...testData}/>);
  const heading = wrapper.find('h4');
  expect(heading.length).toBe(1);
  expect(heading.text()).toBe(testData.exercise.body)
  expect(console.error).toHaveBeenCalledTimes(0);
});

test('Exercise does not render properly when not all props are defined', () => {
  delete testData.submitExercise
  const wrapper = shallow(<Exercise {...testData}/>);
  const heading = wrapper.find('h4');
  expect(heading.length).toBe(1);
  expect(heading.text()).toBe(testData.exercise.body)
  expect(console.error).toHaveBeenCalledTimes(1);
});

test('Exercises renders a snapshot properly when not authenticated', () => {
  const tree = renderer.create(<Exercise {...testData}/>).toJSON();
  expect(tree).toMatchSnapshot();
});

test('Exercises renders a snapshot properly when authenticated', () => {
  testData.isAuthenticated = true;
  const tree = renderer.create(<Exercise {...testData}/>).toJSON();
  expect(tree).toMatchSnapshot();
});
```

Take note of the changes. What does `mockClear()` do? Look it up on your own.

## Remaining Components

Do this on your own! Then review the code below. Make sure you add tests as well. You can find the associated test specs in the code repo under the `part7` release tag.

### AddUser

```
AddUser.propTypes = {
  username: PropTypes.string.isRequired,
  email: PropTypes.string.isRequired,
  handleChange: PropTypes.func.isRequired,
  addUser: PropTypes.func.isRequired,
};
```

### Exercises

```
Exercises.propTypes = {
  isAuthenticated: PropTypes.bool.isRequired,
};
```

### Form

```
Form.propTypes = {
  formType: PropTypes.string.isRequired,
  isAuthenticated: PropTypes.bool.isRequired,
  loginUser: PropTypes.func.isRequired,
  createMessage: PropTypes.func.isRequired,
};
```

### Logout

```
Logout.propTypes = {
  logoutUser: PropTypes.func.isRequired,
};
```

### Message

```
Message.propTypes = {
  messageName: PropTypes.string,
  messageType: PropTypes.string,
  removeMessage: PropTypes.func.isRequired,
};
```

### NavBar

```
NavBar.propTypes = {
  title: PropTypes.string.isRequired,
  isAuthenticated: PropTypes.bool.isRequired,
};
```

## UserList

```
UsersList.propTypes = {
  users: PropTypes.array.isRequired,
};
```

You probably already noticed, but you will also need to check to ensure that `users` even exists before iterating through it:

```
{
  props.users && props.users.map((user) => {
    return (
      <tr key={user.id}>
        <td>{user.id}</td>
        <td>{user.username}</td>
        <td>{String(user.active)}</td>
        <td>{String(user.admin)}</td>
      </tr>
    )
  })
}
```

## UserStatus

```
UserStatus.propTypes = {
  isAuthenticated: PropTypes.bool.isRequired,
};
```

---

Ensure all the tests pass:

```
$ sh test.sh dev
```

Commit your code.

# Scores Service

In this lesson, we'll wire up a new Flask microservice that is responsible for maintaining scores...

---

Check your understanding by setting this service up on your own. Refer to the steps below as well as back to Part 6, when we set up the `exercises` services, for help.

## Model

| Name | Type | Example |
|---|---|---|
| `id` | integer | 1 |
| `user_id` | integer | 2 |
| `exercise_id` | integer | 3 |
| `correct` | boolean | `None` (if unanswered), `True` , or `False` |

## Routes

| Endpoint | HTTP Method | Authenticated? | Result |
|---|---|---|---|
| /scores/ping | GET | No | sanity check |
| /scores | GET | No | get all scores |
| /scores/user | GET | Yes | get all scores by user id |
| /scores/user/:id | GET | Yes | get single score by user id |
| /scores | POST | Yes | add a score |
| /scores/:exercise_id | PUT | Yes | update a score by exercise id |

## Steps

### Setup

1. Create new service from the "base" project directory in *base.zip*
2. Add the service to *docker-compose-dev.yml*, *docker-compose-stage.yml*, and *docker-compose-prod.yml*
3. Update *test.sh*
4. Spin up the new container (you may need to run `chmod +x services/scores/entrypoint.sh` )
5. Run the tests `sh test.sh dev`

### Database

1. Write a test
2. Configure SQLAlchemy
3. Set up a `Score` model
4. Add a seed command to *manage.py*
5. Add the service to *docker-compose-dev.yml*, *docker-compose-stage.yml*, and *docker-compose-prod.yml*
6. Update the containers and run create the database
7. Run the tests

### API

For each route:

1. write a test
2. run the test to ensure it fails (**red**)
3. write just enough code to get the test to pass (**green**)
4. **refactor** (if necessary)

Don't forget to update the Nginx conf as well in *services/nginx/dev.conf* and *services/nginx/prod.conf*!

## Sanity Check

With `testdriven-dev` as the active Docker Machine, set the environment variables:

```
$ export REACT_APP_USERS_SERVICE_URL=http://DOCKER_MACHINE_DEV_IP
$ export TEST_URL=http://DOCKER_MACHINE_DEV_IP
$ export REACT_APP_API_GATEWAY_URL=https://API_GATEWAY_URL
$ export REACT_APP_EXERCISES_SERVICE_URL=http://DOCKER_MACHINE_DEV_IP
```

Spin up the app:

```
$ docker-compose -f docker-compose-dev.yml up -d --build
```

Create the new database:

```
$ docker-compose -f docker-compose-dev.yml \
  run scores python manage.py recreate_db
```

Ensure the app is working in the browser, and then run the tests:

```
$ sh test.sh dev
```

Have you looked at the test coverage?

```
$ docker-compose -f docker-compose-dev.yml \
```

```
run scores python manage.py cov
```

It should be around 58%:

```
Coverage Summary:
Name                        Stmts   Miss Branch BrPart  Cover
-----------------------------------------------------------
project/__init__.py            23     10      0      0    57%
project/api/base.py             6      5      0      0    17%
project/api/models.py          13      9      0      0    31%
project/api/scores.py          67     22     14      1    72%
project/api/utils.py           32     18      8      2    45%
-----------------------------------------------------------
TOTAL                         141     64     22      3    58%
```

What did you learn from the Coverage summary? Can you think of any additional tests that should be written? How about routes? Add them on your own.

## Add Scores

Next, when a user registers, we need to add the appropriate scores.

Start by firing a new method called `addScores` in `handleUserFormSubmit()` in *services/client/src/components/forms/Form.jsx* if `formType` is `register`:

```
handleUserFormSubmit(event) {
  event.preventDefault();
  const formType = this.props.formType
  let data;
  if (formType === 'login') {
    data = {
      email: this.state.formData.email,
      password: this.state.formData.password
    };
  };
  if (formType === 'register') {
    data = {
      username: this.state.formData.username,
      email: this.state.formData.email,
      password: this.state.formData.password
    };
  };
  const url = `${process.env.REACT_APP_USERS_SERVICE_URL}/auth/${formType}`;
  return axios.post(url, data)
  .then((res) => {
    this.clearForm();
    this.props.loginUser(res.data.auth_token);
    if (formType === 'register') this.addScores();
```

```
  })
  .catch((err) => {
    if (formType === 'login') {
      this.props.createMessage('User does not exist.', 'danger');
    };
    if (formType === 'register') {
      this.props.createMessage('That user already exists.', 'danger');
    };
  });
};
```

Add the method:

```
addScores() {
  const exercisesURL = process.env.REACT_APP_EXERCISES_SERVICE_URL;
  return axios.get(`${exercisesURL}/exercises`)
  .then((res) => {
    const requests = res.data.data.exercises.map((el) => {
      const options = {
        url: `${process.env.REACT_APP_SCORES_SERVICE_URL}/scores`,
        method: 'post',
        headers: {
          'Content-Type': 'application/json',
          Authorization: `Bearer ${window.localStorage.authToken}`
        },
        data: {exercise_id: el.id},
      };
      return axios(options);
    });
    return axios.all(requests)
  })
  .then(axios.spread((...res) => { console.log(res); }))
  .catch((err) => { console.log(err) });
};
```

Make sure to bind it as well:

```
this.addScores = this.addScores.bind(this);
```

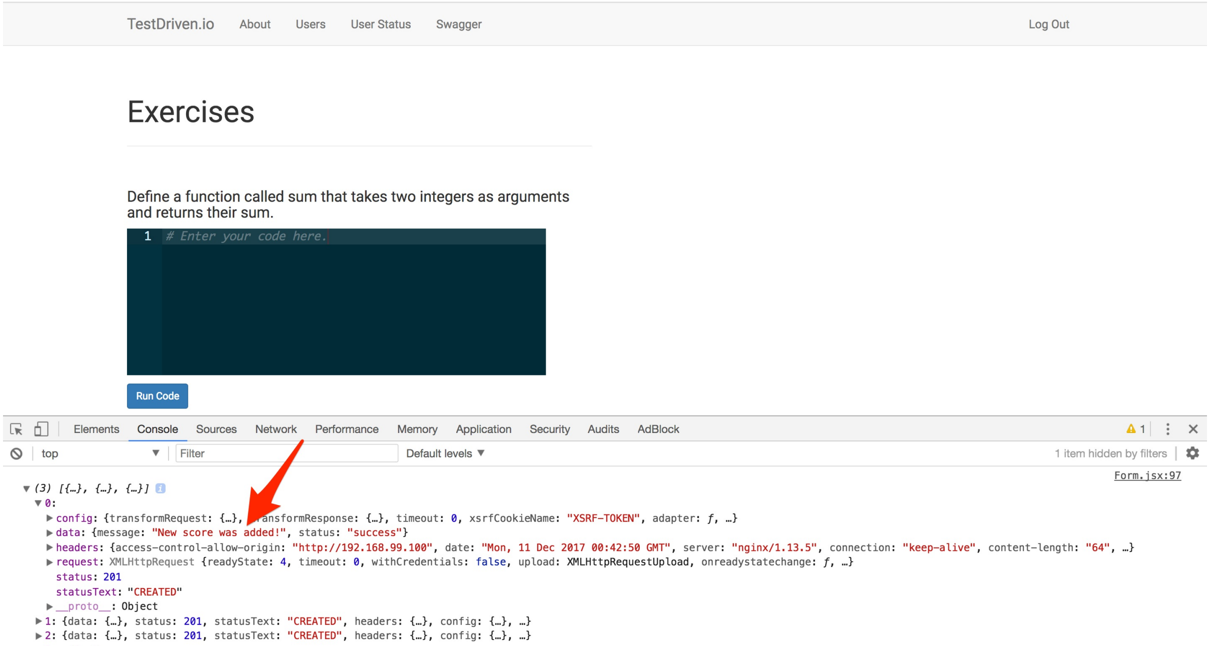
Then, add the following environment variable to the `client` service within all three *docker-compose.yml* files:

```
- REACT_APP_SCORES_SERVICE_URL=${REACT_APP_SCORES_SERVICE_URL}
```

Add the `USERS_SERVICE_URL` for the `scores` service in *docker-compose-stage.yml* and *docker-compose-prod.yml*:

```
- USERS_SERVICE_URL=${REACT_APP_USERS_SERVICE_URL}
```

Test this out in the browser. Register a new user, and then after the redirect, open the JavaScript Console:




We'll test this behavior out shortly with an end-to-end test. Don't worry!

Also, you may need to add scores for existing users in your staging or production environments. We'll handle this when we update the AWS environments later in part 7.

## Seed

Next, let's add a seed command to *services/scores/manage.py*:

```python
@cli.command()
def seed_db():
    """Seeds the database."""
    # get exercises
    url = '{0}/exercises'.format(os.environ.get('EXERCISES_SERVICE_URL'))
    response = requests.get(url)
    exercises = response.json()['data']['exercises']
    # get users
    url = '{0}/users'.format(os.environ.get('USERS_SERVICE_URL'))
    response = requests.get(url)
    users = response.json()['data']['users']
    # seed
    for user in users:
        for exercise in exercises:
            db.session.add(Score(
                user_id=user['id'],
                exercise_id=exercise['id']
```

```
            ))
    db.session.commit()
```

Add the imports:

```
import os

import requests

from project.api.models import Score
```

Then, again, add a new environment variable to the `scores` service to all three *docker-compose.yml* files...

Development:

```
- EXERCISES_SERVICE_URL=http://exercises:5000
```

Staging and production:

```
- EXERCISES_SERVICE_URL=${REACT_APP_EXERCISES_SERVICE_URL}
```

Also, since the `scores` service is dependent on the `exercises` service being up, update the `depends_on` and `links` keys in each *docker-compose.yml* file for the `score` service:

```
depends_on:
  - users
  - scores-db
  - exercises
links:
  - users
  - scores-db
  - exercises
```

## DB Script

Finally, let's create an *init_db.sh* file to create and seed the databases so we don't have to manually run the commands each time:

```
#!/bin/bash

# create
docker-compose -f docker-compose-dev.yml run exercises python manage.py recreate_db
docker-compose -f docker-compose-dev.yml run users python manage.py recreate_db
docker-compose -f docker-compose-dev.yml run scores python manage.py recreate_db
# seed
```

```
docker-compose -f docker-compose-dev.yml run exercises python manage.py seed_db
docker-compose -f docker-compose-dev.yml run users python manage.py seed_db
docker-compose -f docker-compose-dev.yml run scores python manage.py seed_db
```

Commit and push your code to Github from the `master` branch once complete.

# Exercises Component Refactor

In this lesson, we'll refactor the `Exercises` component to update the score after a user submits a solution to an exercise and add the ability to move to new questions or back to previous questions...

---

## Docker Machine

Set `testdriven-dev` as the active Docker Machine:

```
$ docker-machine env testdriven-dev
$ eval $(docker-machine env testdriven-dev)
```

Set the environment variables:

```
$ export REACT_APP_USERS_SERVICE_URL=http://DOCKER_MACHINE_DEV_IP
$ export TEST_URL=http://DOCKER_MACHINE_DEV_IP
$ export REACT_APP_API_GATEWAY_URL=https://API_GATEWAY_URL
$ export REACT_APP_EXERCISES_SERVICE_URL=http://DOCKER_MACHINE_DEV_IP
$ export REACT_APP_SCORES_SERVICE_URL=http://DOCKER_MACHINE_DEV_IP
```

Update *swagger.json*:

```
$ python services/swagger/update-spec.py http://DOCKER_MACHINE_DEV_IP
```

Spin up the app:

```
$ docker-compose -f docker-compose-dev.yml up -d --build
```

Create and seed the databases:

```
$ sh init_db.sh
```

Ensure the app is working in the browser, and then run the tests:

```
$ sh test.sh dev
```

## Update Score

So, after an exercise is graded by the Lambda, we need to update the score. Let's create a new method for this in *services/client/src/components/Exercises.jsx*:

```
updateScore(exerciseID, bool) {
  const options = {
    url: `${process.env.REACT_APP_SCORES_SERVICE_URL}/scores/${exerciseID}`,
    method: 'put',
    headers: {
      'Content-Type': 'application/json',
      Authorization: `Bearer ${window.localStorage.authToken}`
    },
    data: {correct:bool}
  };
  return axios(options)
  .then((res) => { console.log(res); })
  .catch((error) => { console.log(error); });
};
```

Add the bind:

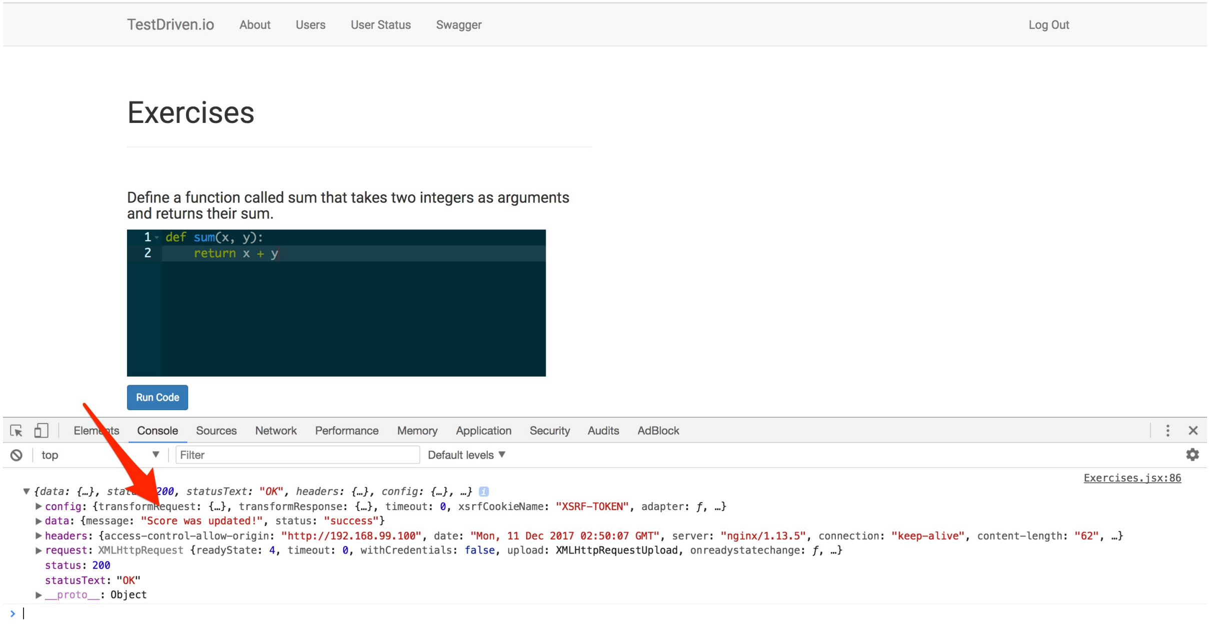
```
this.updateScore = this.updateScore.bind(this);
```

Then, update `submitExercise()` :

```
submitExercise(event, id) {
  event.preventDefault();
  const newState = this.state.editor;
  const exercise = this.state.exercises.filter(el => el.id === id)[0]
  newState.showGrading = true;
  newState.showCorrect = false;
  newState.showIncorrect = false;
  newState.button.isDisabled = true;
  this.setState(newState);
  const data = {
    answer: this.state.editor.value,
    test: exercise.test_code,
    solution: exercise.test_code_solution,
  };
  const url = process.env.REACT_APP_API_GATEWAY_URL;
  axios.post(url, data)
  .then((res) => {
    newState.showGrading = false
    newState.button.isDisabled = false
    if (res.data && !res.data.errorType) {
      newState.showCorrect = true
      this.updateScore(exercise.id, true)
    };
    if (!res.data || res.data.errorType) {
      newState.showIncorrect = true
      this.updateScore(exercise.id, false)
    };
```

```
      this.setState(newState);
    })
    .catch((err) => {
      newState.showGrading = false
      newState.button.isDisabled = false
      console.log(err);
      this.updateScore(exercise.id, false)
    })
};
```

Rebuild. Test it our in your browser:




Confirm in the database:

```
$ docker exec -ti scores-db psql -U postgres -W

# \c scores_dev
# select * from scores;

id | user_id | exercise_id | correct
----+---------+-------------+---------
 1 |       1 |           1 |
 2 |       1 |           2 |
 3 |       1 |           3 |
 4 |       2 |           1 |
 5 |       2 |           2 |
 6 |       2 |           3 |
 8 |       3 |           2 |
 9 |       3 |           3 |
 7 |       3 |           1 | t
(9 rows)
```

```
# \q
```

## Previous and Next Buttons

Let's allow the user to move on to a new question or back to a previous question if they exist, starting with some tests. One thing to think about is whether you want these buttons displayed if a user is not authenticated. We'll display them in this tutorial, but feel free to customize this on your own.

### Test

We'll focus on end-to-end tests. Write client-side tests on your own.

Let's start by asserting that the appropriate buttons are on the page in the two test specs in *e2e/exercises.test.js*:

```
test(`should display the exercises correctly if a user is not logged in`, async (t)
 => {
  await t
    .navigateTo(`${TEST_URL}/`)
    .expect(Selector('H1').withText('Exercises').exists).ok()
    .expect(Selector('.alert-warning').withText('Please log in to submit an exercis
e.').exists).ok()
    .expect(Selector('button').withText('Run Code').exists).notOk()
    .expect(Selector('.btn-group').exists).ok()
    .expect(Selector('button').withText('Next').exists).ok()
    .expect(Selector('button').withText('Prev').exists).notOk();
});

test(`should allow a user to submit an exercise if logged in`, async (t) => {
  await t
    .navigateTo(`${TEST_URL}/register`)
    .typeText('input[name="username"]', username)
    .typeText('input[name="email"]', email)
    .typeText('input[name="password"]', password)
    .click(Selector('input[type="submit"]'))
  await t
    .navigateTo(`${TEST_URL}/`)
    .expect(Selector('H1').withText('Exercises').exists).ok()
    .expect(Selector('.alert-warning').withText('Please log in to submit an exercis
e.').exists).notOk()
    .expect(Selector('button').withText('Run Code').exists).ok()
    .expect(Selector('.btn-group').exists).ok()
    .expect(Selector('button').withText('Next').exists).ok()
    .expect(Selector('button').withText('Prev').exists).notOk()
    .click(Selector('button').withText('Run Code'))
    .expect(Selector('h4').withText('Incorrect!').exists).ok()
    .expect(Selector('h4').withText('Correct!').exists).notOk()
  await t
    .navigateTo(`${TEST_URL}/`)
```

```
    .selectText(Selector('textarea'))
    .pressKey('home')
    for (let i = 0; i < 23; i++) {
      await t.pressKey('delete')
    }
  await t
    .typeText('textarea', 'def sum(x,y):\nreturn x+y')
    .click(Selector('button').withText('Run Code'))
    .expect(Selector('h4').withText('Incorrect!').exists).notOk()
    .expect(Selector('h4').withText('Correct!').exists).ok()
});
```

Ensure the tests fail.

## Code

### Update State

Add a new value to the state object in *services/client/src/components/Exercises.jsx*:

```
this.state = {
  currentExercise: 0,
  exercises: [],
  editor: {
    value: '# Enter your code here.',
    button: {
      isDisabled: false,
    },
    showGrading: false,
    showCorrect: false,
    showIncorrect: false,
  },
};
```

We can then use `currentExercise` to render the `exercise` component:

```
<Exercise
  exercise={this.state.exercises[this.state.currentExercise]}
  editor={this.state.editor}
  isAuthenticated={this.props.isAuthenticated}
  onChange={this.onChange}
  submitExercise={this.submitExercise}
/>
```

Finally, be sure to update the state in `getExercises()` :

```
getExercises() {
  return axios.get(`${process.env.REACT_APP_EXERCISES_SERVICE_URL}/exercises`)
```

```
    .then((res) => {
      this.setState({
        exercises: res.data.data.exercises,
        currentExercise: 0
      });
    })
    .catch((err) => { console.log(err); });
};
```

#### Buttons

First, wire up a function to determine if the buttons should even be displayed:

```
renderButtons() {
  const index = this.state.currentExercise;
  let nextButton = false;
  let prevButton = false;
  if (typeof this.state.exercises[index + 1] !== 'undefined') {
    nextButton = true;
  }
  if (typeof this.state.exercises[index - 1] !== 'undefined') {
    prevButton = true;
  }
  this.setState({
    showButtons: {
      next: nextButton,
      prev: prevButton
    }
  });
};
```

Add the bind:

```
this.renderButtons = this.renderButtons.bind(this);
```

Update the state:

```
this.state = {
  currentExercise: 0,
  exercises: [],
  editor: {
    value: '# Enter your code here.',
    button: {
      isDisabled: false,
    },
    showGrading: false,
    showCorrect: false,
    showIncorrect: false,
  },
```

```
    showButtons: {
      prev: false,
      next: false,
    },
  };
```

Let's fire it in the `getExercises` method:

```
getExercises() {
  return axios.get(`${process.env.REACT_APP_EXERCISES_SERVICE_URL}/exercises`)
  .then((res) => {
    this.setState({
      exercises: res.data.data.exercises,
      currentExercise: 0
    });
    this.renderButtons();
  })
  .catch((err) => { console.log(err); });
};
```

Finally, in the `render()`, just below where we render the `Exercise` component, add a button group:

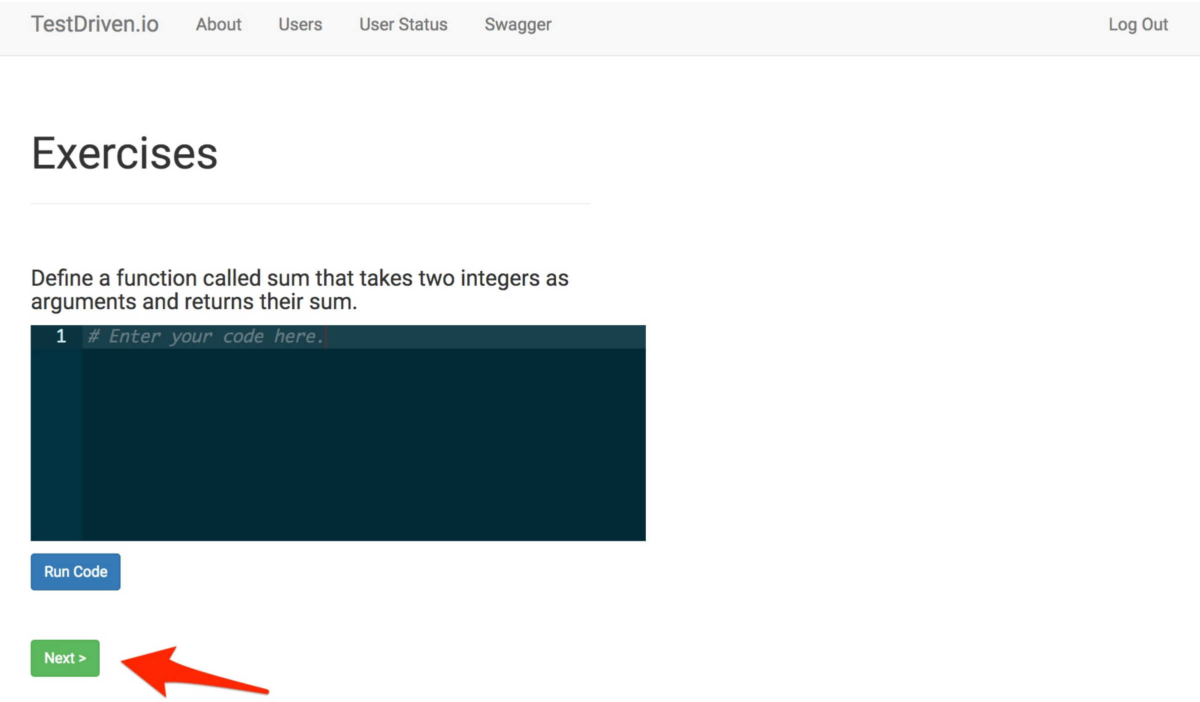
```
render() {
  return (
    <div>
      ...
      {this.state.exercises.length &&
        <Exercise
          exercise={this.state.exercises[this.state.currentExercise]}
          editor={this.state.editor}
          isAuthenticated={this.props.isAuthenticated}
          onChange={this.onChange}
          submitExercise={this.submitExercise}
        />
      }
      <ButtonGroup>
        { this.state.showButtons.prev &&
          <Button
            bsStyle="success"
            bsSize="small"
          >&lt; Prev</Button>
       }
       { this.state.showButtons.next &&
        <Button
          bsStyle="success"
          bsSize="small"
        >Next &gt;</Button>
      }
```

```
      </ButtonGroup>
    </div>
  )
};
```

Make sure to add the imports:

```
import { Button, ButtonGroup } from 'react-bootstrap';
```

Manually test this out in the browser, and then run the end-to-end tests.




## Test

Next, let's add functionality to the buttons. Add a new test case to *e2e/exercises.test.js*:

```
test(`should allow a user to move to different exercises`, async (t) => {
  await t
    .expect(Selector('.btn-group').exists).ok()
    .expect(Selector('button').withText('Next').exists).ok()
    .expect(Selector('button').withText('Prev').exists).notOk()
    .click(Selector('button').withText('Next'))
    .expect(Selector('button').withText('Next').exists).ok()
    .expect(Selector('button').withText('Prev').exists).ok()
    .click(Selector('button').withText('Next'))
    .expect(Selector('button').withText('Next').exists).notOk()
    .expect(Selector('button').withText('Prev').exists).ok()
    .click(Selector('button').withText('Prev'))
    .expect(Selector('button').withText('Next').exists).ok()
```

```
    .expect(Selector('button').withText('Prev').exists).ok()
});
```

## Code

First, add `onClick` handlers to each button:

```
<ButtonGroup>
  { this.state.showButtons.prev &&
    <Button
      bsStyle="success"
      bsSize="small"
      onClick={() => this.prevExercise()}
    >&lt; Prev</Button>
 }
 { this.state.showButtons.next &&
  <Button
    bsStyle="success"
    bsSize="small"
    onClick={() => this.nextExercise()}
  >Next &gt;</Button>
}
</ButtonGroup>
```

Then, add the functions themselves:

```
nextExercise() {
  if (this.state.showButtons.next) {
    const currentExercise = this.state.currentExercise;
    this.setState({currentExercise: currentExercise + 1}, () => {
      this.renderButtons();
    });
  }
};
prevExercise() {
  if (this.state.showButtons.prev) {
    const currentExercise = this.state.currentExercise;
    this.setState({currentExercise: currentExercise - 1}, () => {
      this.renderButtons();
    });
  }
};
```

Did you notice that we added a callback to `setState`? Essentially, `setState` is asynchronous, and, since, `renderButtons()` is dependent on `currentExercise`, we need to wait until `setState()` is done updating the state before `renderButtons()` is called.

Bind the methods:

```
this.nextExercise = this.nextExercise.bind(this);
this.prevExercise = this.prevExercise.bind(this);
```

The tests should pass, but if you test it out in the browser, you'll notice that the state of the editor value is not getting reset when we move forward or backward between exercises. Let's fix that.

## Test

Update:

```
test(`should allow a user to move to different exercises`, async (t) => {
  await t
    .expect(Selector('.btn-group').exists).ok()
    .expect(Selector('button').withText('Next').exists).ok()
    .expect(Selector('button').withText('Prev').exists).notOk()
    .expect(Selector('.ace_comment').withText(
      '# Enter your code here.').exists).ok()
    .click(Selector('button').withText('Next'))
    .expect(Selector('button').withText('Next').exists).ok()
    .expect(Selector('button').withText('Prev').exists).ok()
    .expect(Selector('.ace_comment').withText(
      '# Enter your code here.').exists).ok()
    .click(Selector('button').withText('Next'))
    .expect(Selector('button').withText('Next').exists).notOk()
    .expect(Selector('button').withText('Prev').exists).ok()
    .expect(Selector('.ace_comment').withText(
      '# Enter your code here.').exists).ok()
    .selectText(Selector('textarea'))
    .pressKey('home')
    for (let i = 0; i < 23; i++) {
      await t.pressKey('delete')
    }
  await t
    .typeText('textarea', 'def sum(x,y):\nreturn x+y')
    .click(Selector('button').withText('Prev'))
    .expect(Selector('button').withText('Next').exists).ok()
    .expect(Selector('button').withText('Prev').exists).ok()
    .expect(Selector('.ace_comment').withText(
      '# Enter your code here.').exists).ok()
});
```

## Code

Add:

```
resetEditor() {
  const editor = {
```

```
    value: '# Enter your code here.',
    button: {
      isDisabled: false,
    },
    showGrading: false,
    showCorrect: false,
    showIncorrect: false,
  }
  this.setState({editor: editor});
};
```

Bind:

```
this.resetEditor = this.resetEditor.bind(this);
```

Call:

```
nextExercise() {
  if (this.state.showButtons.next) {
    const currentExercise = this.state.currentExercise;
    this.setState({currentExercise: currentExercise + 1}, () => {
      this.resetEditor()
      this.renderButtons();
    });
  }
};
prevExercise() {
  if (this.state.showButtons.prev) {
    const currentExercise = this.state.currentExercise;
    this.setState({currentExercise: currentExercise - 1}, () => {
      this.resetEditor();
      this.renderButtons();
    });
  }
};
```

Ensure the tests pass before moving on.

# Updated Users List

Let's add the exercise scores to the `UsersList` component...

---

## Scores

First, update `getUsers()` in *services/client/src/App.jsx* to get all the scores and add them to the `users` array:

```
getUsers() {
  return axios.get(`${process.env.REACT_APP_USERS_SERVICE_URL}/users`)
  .then((res) => {
    const users = res.data.data.users;
    const updatedUsers = users.map((user) => {
      user.scores = [];
      return user;
    });
    this.setState({ users: updatedUsers });
    return axios.get(`${process.env.REACT_APP_SCORES_SERVICE_URL}/scores`)
  })
  .then((res) => {
    const scores = res.data.data.scores;
    const users = [...this.state.users]
    const updatedUsers = users.map((user) => {
      for (let score of scores) {
        if (score.user_id === user.id) {
          if (score.correct === null) score.correct = 'N/A';
          user.scores.push(score);
        }
      }
      return user;
    });
    this.setState({ users: updatedUsers });
  })
  .catch((err) => { });
};
```

Take note of `if (score.correct === null) score.correct = 'N/A';`. Essentially, if `score.correct` is `null`, we can assume that the associated exercise has not be answered.

> What happens if the second AJAX request, to the `/scores` endpoint, finishes before the state is updated in the line before it? Is this something you should worry about?

Now, if a new user registers, the `scores` array will be empty in the `users` object. To update, let's call `getUsers()` within `addScores()` in *services/client/src/components/forms/Form.jsx*:

```
addScores() {
  const exercisesURL = process.env.REACT_APP_EXERCISES_SERVICE_URL;
  return axios.get(`${exercisesURL}/exercises`)
  .then((res) => {
    const requests = res.data.data.exercises.map((el) => {
      const options = {
        url: `${process.env.REACT_APP_SCORES_SERVICE_URL}/scores`,
        method: 'post',
        headers: {
          'Content-Type': 'application/json',
          Authorization: `Bearer ${window.localStorage.authToken}`
        },
        data: {exercise_id: el.id},
      };
      return axios(options);
    });
    return axios.all(requests)
  })
  .then(axios.spread((...res) => { this.props.getUsers(); }))
  .catch((err) => { console.log(err) });
};
```

Let's pass it in via the props in *App.jsx*:

```
<Route exact path='/register' render={() => (
  <Form
    formType={'register'}
    isAuthenticated={this.state.isAuthenticated}
    loginUser={this.loginUser}
    createMessage={this.createMessage}
    getUsers={this.getUsers}
  />
)} />
<Route exact path='/login' render={() => (
  <Form
    formType={'login'}
    isAuthenticated={this.state.isAuthenticated}
    loginUser={this.loginUser}
    createMessage={this.createMessage}
    getUsers={this.getUsers}
  />
)} />
```

Don't forget the bind in `App`:

```
this.getUsers = this.getUsers.bind(this);
```

Update the PropTypes in the `Form` component:

```
Form.propTypes = {
  formType: PropTypes.string.isRequired,
  isAuthenticated: PropTypes.bool.isRequired,
  loginUser: PropTypes.func.isRequired,
  createMessage: PropTypes.func.isRequired,
  getUsers: PropTypes.func.isRequired,
};
```

You will need to update `testData` in *services/client/src/components/__tests__/Form.test.jsx* as well:

```
const testData = [
  {
    formType: 'register',
    formData: {
      username: '',
      email: '',
      password: ''
    },
    isAuthenticated: false,
    loginUser: jest.fn(),
    createMessage: jest.fn(),
    getUsers: jest.fn(),
  },
  {
    formType: 'login',
    formData: {
      email: '',
      password: ''
    },
    isAuthenticated: false,
    loginUser: jest.fn(),
    createMessage: jest.fn(),
    getUsers: jest.fn(),
  }
];
```

Your turn! Do the same for `updateScore()` in *services/client/src/components/Exercises.jsx*:

1. Call `getUsers()` within the `.then`. of `updateScore` (see below)
2. Pass in `getUsers` via the props in *App.jsx*:
3. Update the PropTypes for the `Exercises` component
4. Update the test?

```
updateScore(exerciseID, bool) {
  const options = {
    url: `${process.env.REACT_APP_SCORES_SERVICE_URL}/scores/${exerciseID}`,
    method: 'put',
```

```
    headers: {
      'Content-Type': 'application/json',
      Authorization: `Bearer ${window.localStorage.authToken}`
    },
    data: {correct:bool}
  };
  return axios(options)
  .then((res) => {
    this.props.getUsers();
  })
  .catch((error) => { console.log(error); });
};
```

## UsersList

Before updating the actual `UsersList` component, let's turn our attention to the PropTypes:

```
UsersList.propTypes = {
  users: PropTypes.array.isRequired,
};
```

Since `users` is an array of objects, let's validate the shape of the objects:

```
UsersList.propTypes = {
  users: PropTypes.arrayOf(
    PropTypes.shape({
      active: PropTypes.bool.isRequired,
      admin: PropTypes.bool.isRequired,
      email: PropTypes.string.isRequired,
      id: PropTypes.number.isRequired,
      scores: PropTypes.array.isRequired,
      username: PropTypes.string.isRequired,
    }).isRequired,
  ).isRequired,
};
```

It's worth noting that we could validate the shape of the objects within the `scores` array as well. Do this on your own.

Update the `users` props in *services/client/src/components/__tests__/UsersList.test.jsx*:

```
const users = [
  {
    active: true,
    admin: false,
    email: 'michael@notreal.com',
    id: 1,
    username: 'michael',
```

```
    scores: [],
  },
  {
    active: true,
    admin: false,
    email: 'michael@mherman.org',
    id: 2,
    username: 'michaelherman',
    scores: [],
  }
];
```

Run the tests:

```
$ docker-compose -f docker-compose-dev.yml \
  run client npm test --watchAll
```

Correct any failing specs before moving on. Update the table in the `UsersList` component to display the scores:

```
<Table striped bordered condensed hover>
  <thead>
    <tr>
      <th>User ID</th>
      <th>Username</th>
      <th>Active</th>
      <th>Admin</th>
    </tr>
  </thead>
  <tbody>
    {
      props.users && props.users.map((user) => {
        return (
          <tr key={user.id}>
            <td>{user.id}</td>
            <td>{user.username}</td>
            <td>{String(user.active)}</td>
            <td>{String(user.admin)}</td>
          </tr>
        )
      })
    }
  </tbody>
</Table>
```

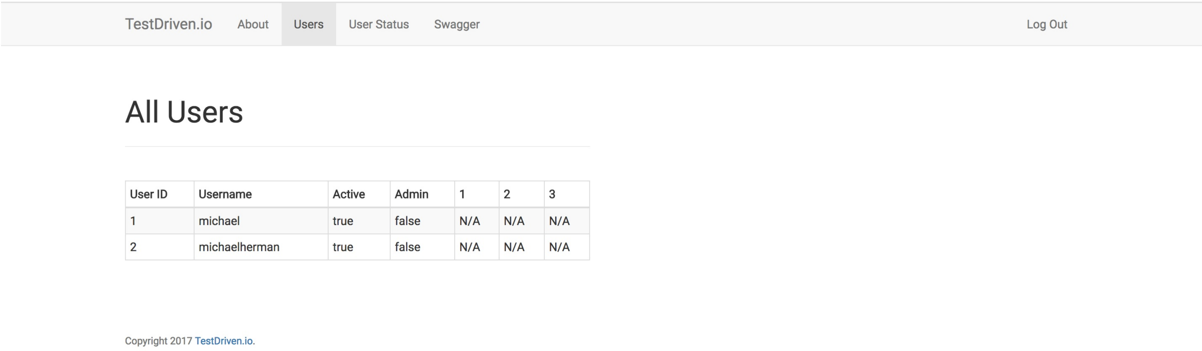
Spin up the app on the `testdriven-dev` machine:

```
$ docker-compose -f docker-compose-dev.yml up -d --build
```

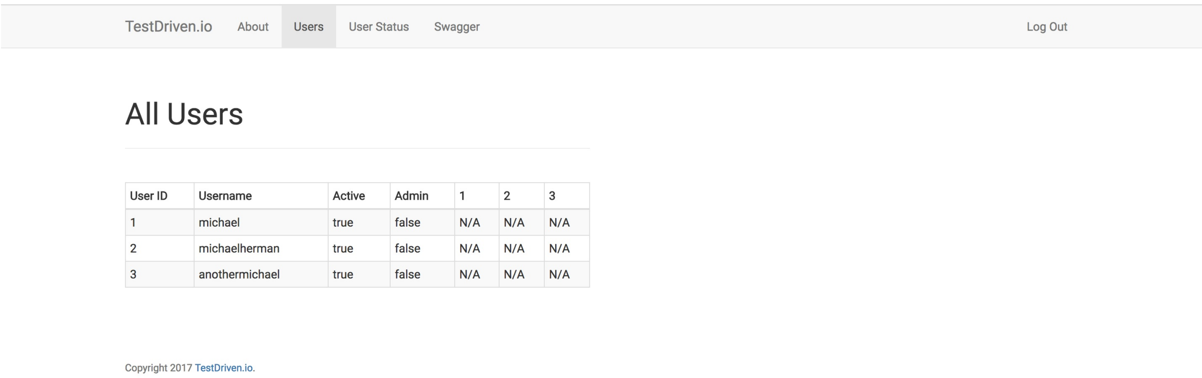
Create and seed the databases:

```
$ sh init_db.sh
```

Fire up the app in the browser. You should see:




Register a new user and ensure the scores are added:




Now that we have a some context of what we're trying to achieve, let's write some tests!

## Tests

Write both client-side and end-to-end tests on your own. In terms of end-to-end tests, you'll probably want to:

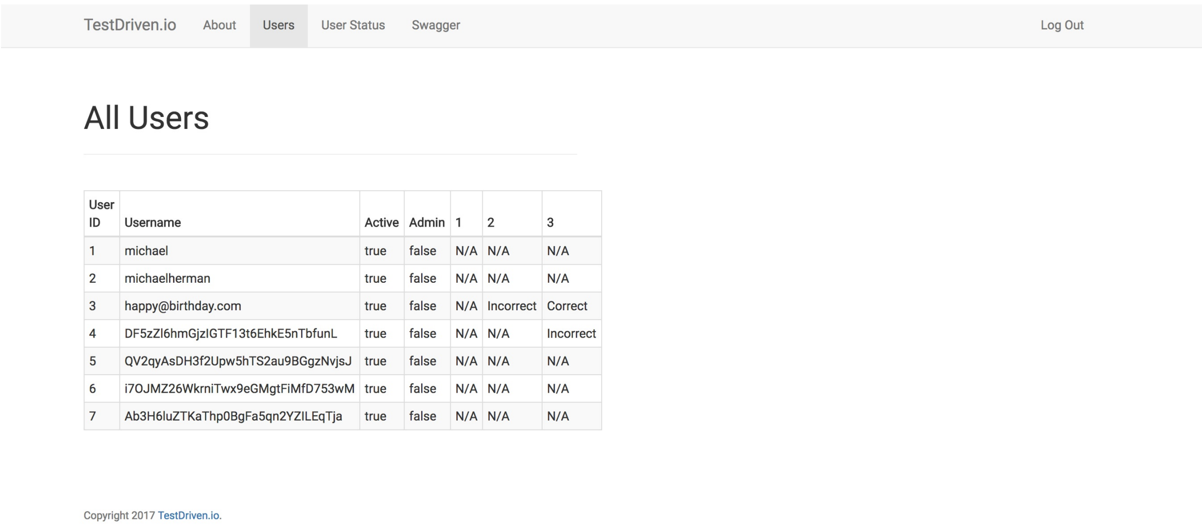
1. Register a new user
2. Navigate to `/all-users`
3. Assert that `'N/A'` is present for each of the scores
4. Navigate to `/`
5. Answer an exercise correctly
6. Answer a different exercise incorrectly
7. Navigate to `/all-users`
8. Assert that `'N/A'` is present for one of the scores
9. Assert that `'Correct'` is visible for one of the scores
10. Assert that `'Incorrect'` is visible for one of the scores
11. Navigate to `/`
12. Correct the answer that was originally answered incorrectly
13. Navigate to `/all-users`
14. Assert that `'N/A'` is present for one of the scores
15. Assert that `'Correct'` is visible for two of the scores

Which of those assertions will fail right now? Make changes as necessary.

> Want some feedback on the tests you wrote? Email a link to your code to michael@mherman.org. Cheers!

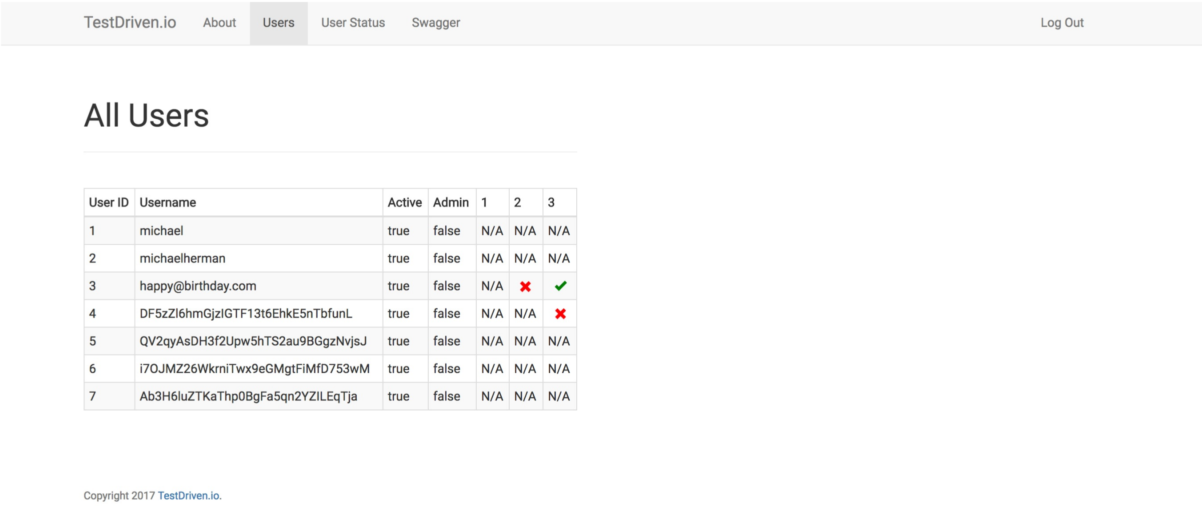
## Update UsersList

As of now, the rendered `UsersList` component should look something like:




| User ID | Username | Active | Admin | 1 | 2 | 3 |
|---|---|---|---|---|---|---|
| 1 | michael | true | false | N/A | N/A | N/A |
| 2 | michaelherman | true | false | N/A | N/A | N/A |
| 3 | happy@birthday.com | true | false | N/A | Incorrect | Correct |
| 4 | DF5zZl6hmGjzlGTF13t6EhkE5nTbfunL | true | false | N/A | N/A | Incorrect |
| 5 | QV2qyAsDH3f2Upw5hTS2au9BGgzNvjsJ | true | false | N/A | N/A | N/A |
| 6 | i7OJMZ26WkrniTwx9eGMgtFiMfD753wM | true | false | N/A | N/A | N/A |
| 7 | Ab3H6luZTKaThp0BgFa5qn2YZILEqTja | true | false | N/A | N/A | N/A |

Try refactoring so it looks more like this using the Glyphicon component:




| User ID | Username | Active | Admin | 1 | 2 | 3 |
|---|---|---|---|---|---|---|
| 1 | michael | true | false | N/A | N/A | N/A |
| 2 | michaelherman | true | false | N/A | N/A | N/A |
| 3 | happy@birthday.com | true | false | N/A | ✖ | ✔ |
| 4 | DF5zZl6hmGjzIGTF13t6EhkE5nTbfunL | true | false | N/A | N/A | ✖ |
| 5 | QV2qyAsDH3f2Upw5hTS2au9BGgzNvjsJ | true | false | N/A | N/A | N/A |
| 6 | i7OJMZ26WkrniTwx9eGMgtFiMfD753wM | true | false | N/A | N/A | N/A |
| 7 | Ab3H6luZTKaThp0BgFa5qn2YZILEqTja | true | false | N/A | N/A | N/A |

Update any tests and make sure they all pass before moving on. Commit and push your code to GitHub once done.

# ECS Staging Update

In this lesson, we'll update the staging environment on ECS...

---

## Steps

*Load Balancer*

1. Configure Target Group
2. Add Listener to the ALB

*ECS*

1. Update *Dockerfile-stage* and *Dockerfile-prod*
2. Update *.travis.yml*
3. Update *docker-push.sh*
4. Add images to ECR
5. Configure Task Definitions
6. Create Service
7. Update *docker-deploy-stage.sh*

## Load Balancer

### Target Group

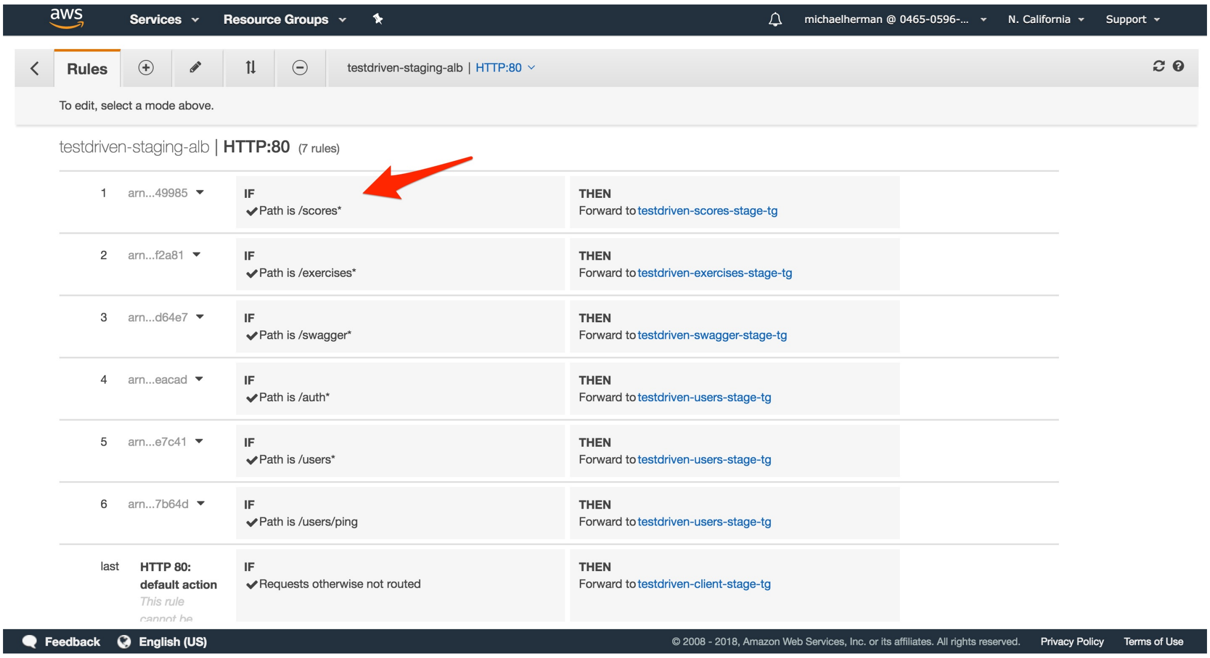
Add a new Target Group for the `scores` service. Within [Amazon EC2](), click "Target Groups", and then create the following Group:

1. "Target group name": `testdriven-scores-stage-tg`
2. "Port": `5000`
3. Then, under "Health check settings" set the "Path" to `/scores/ping` .

### Listener

On the "Load Balancers" page, click the `testdriven-staging-alb` Load Balancer, and then select the "Listeners" tab. From there, click the "View/edit rules" for "HTTP : 80", and then add one new rule:

1. If `/scores*` , Then `testdriven-scores-stage-tg`




# ECS

## Update *Dockerfile-stage* and *Dockerfile-prod*

Add the build-time args to both *Dockerfile-stage* and *Dockerfile-prod* in "services/client":

```
ARG REACT_APP_SCORES_SERVICE_URL
ENV REACT_APP_SCORES_SERVICE_URL $REACT_APP_SCORES_SERVICE_URL
```

Add an *entrypoint-stage.sh* file to "services/scores":

```
#!/bin/sh

echo "Waiting for postgres..."

while ! nc -z scores-db 5432; do
  sleep 0.1
done

echo "PostgreSQL started"

python manage.py recreate_db
python manage.py seed_db
gunicorn -b 0.0.0.0:5000 manage:app
```

Update the permissions:

```
$ chmod +x services/scores/entrypoint-stage.sh
```

## update *.travis.yml*

Update *.travis.yml* to handle the `scores` and `scores-db` services by adding the proper environment variables:

```
language: node_js
node_js: '9'

before_install:
  - stty cols 80

dist: trusty
sudo: required

addons:
  apt:
    sources:
     - google-chrome
    packages:
     - google-chrome-stable

services:
  - docker

env:
  global:
    - DOCKER_COMPOSE_VERSION=1.18.0
    - COMMIT=${TRAVIS_COMMIT::8}
    - MAIN_REPO=https://github.com/testdrivenio/testdriven-app-2.2.git
    - USERS=test-driven-users
    - USERS_REPO=${MAIN_REPO}#${TRAVIS_BRANCH}:services/users
    - USERS_DB=test-driven-users_db
    - USERS_DB_REPO=${MAIN_REPO}#${TRAVIS_BRANCH}:services/users/project/db
    - CLIENT=test-driven-client
    - CLIENT_REPO=${MAIN_REPO}#${TRAVIS_BRANCH}:services/client
    - SWAGGER=test-driven-swagger
    - SWAGGER_REPO=${MAIN_REPO}#${TRAVIS_BRANCH}:services/swagger
    - EXERCISES=test-driven-exercises
    - EXERCISES_REPO=${MAIN_REPO}#${TRAVIS_BRANCH}:services/exercises
    - EXERCISES_DB=test-driven-exercises_db
    - EXERCISES_DB_REPO=${MAIN_REPO}#${TRAVIS_BRANCH}:services/exercises/project/db
    - SCORES=test-driven-scores
    - SCORES_REPO=${MAIN_REPO}#${TRAVIS_BRANCH}:services/scores
    - SCORES_DB=test-driven-scores_db
    - SCORES_DB_REPO=${MAIN_REPO}#${TRAVIS_BRANCH}:services/scores/project/db

before_install:
  - sudo rm /usr/local/bin/docker-compose
  - curl -L https://github.com/docker/compose/releases/download/${DOCKER_COMPOSE_VE
RSION}/docker-compose-`uname -s`-`uname -m` > docker-compose
```

```
  - chmod +x docker-compose
  - sudo mv docker-compose /usr/local/bin

before_script:
  - export TEST_URL=http://127.0.0.1
  - export REACT_APP_USERS_SERVICE_URL=http://127.0.0.1
  - export REACT_APP_EXERCISES_SERVICE_URL=http://127.0.0.1
  - export REACT_APP_SCORES_SERVICE_URL=http://127.0.0.1
  - export REACT_APP_API_GATEWAY_URL=https://69n19z0lo3.execute-api.us-west-1.amazo
naws.com/v1/execute
  - export SECRET_KEY=my_precious
  - export DISPLAY=:99.0
  - export DOCKER_ENV=$(if [ "$TRAVIS_BRANCH" == "production" ]; then echo "prod";
else echo "stage"; fi)
  - sh -e /etc/init.d/xvfb start
  - sleep 3
  - docker-compose -f docker-compose-stage.yml up --build -d

script:
  - bash test.sh $DOCKER_ENV

after_script:
  - docker-compose -f docker-compose-stage.yml down

after_success:
  - bash ./docker-push.sh
  - bash ./docker-deploy-stage.sh
  - bash ./docker-deploy-prod.sh
```

## Update *docker-push.sh*

Make the following updates to *docker-push.sh*:

1. Add the `REACT_APP_SCORES_SERVICE_URL` (for both `staging` and `production`) and the build-time argument to the `client` service
2. Build, tag, and push the `scores` and `scores-db` images

Updated script:

```
#!/bin/sh

if [ -z "$TRAVIS_PULL_REQUEST" ] || [ "$TRAVIS_PULL_REQUEST" == "false" ]
then

  if [ "$TRAVIS_BRANCH" == "staging" ] || \
     [ "$TRAVIS_BRANCH" == "production" ]
  then
    curl "https://s3.amazonaws.com/aws-cli/awscli-bundle.zip" -o "awscli-bundle.zip"

    unzip awscli-bundle.zip
```

```
    ./awscli-bundle/install -b ~/bin/aws
    export PATH=~/bin:$PATH
    # add AWS_ACCOUNT_ID, AWS_ACCESS_KEY_ID, AWS_SECRET_ACCESS_KEY env vars
    eval $(aws ecr get-login --region us-west-1 --no-include-email)
    export TAG=$TRAVIS_BRANCH
    export REPO=$AWS_ACCOUNT_ID.dkr.ecr.us-west-1.amazonaws.com
  fi
  if [ "$TRAVIS_BRANCH" == "staging" ]
  then
    export REACT_APP_USERS_SERVICE_URL="http://LOAD_BALANCER_STAGE_DNS_NAME"
    export REACT_APP_EXERCISES_SERVICE_URL="http://LOAD_BALANCER_STAGE_DNS_NAME"
    export REACT_APP_SCORES_SERVICE_URL="http://LOAD_BALANCER_STAGE_DNS_NAME"
  fi
  if [ "$TRAVIS_BRANCH" == "production" ]
  then
    export REACT_APP_USERS_SERVICE_URL="LOAD_BALANCER_PROD_DNS_NAME"
    export REACT_APP_EXERCISES_SERVICE_URL="LOAD_BALANCER_PROD_DNS_NAME"
    REACT_APP_SCORES_SERVICE_URL="LOAD_BALANCER_PROD_DNS_NAME"
    export DATABASE_URL="$AWS_RDS_URI"
    export SECRET_KEY="$PRODUCTION_SECRET_KEY"
  fi
  if [ "$TRAVIS_BRANCH" == "staging" ] || \
     [ "$TRAVIS_BRANCH" == "production" ]
  then
    # users
    docker build $USERS_REPO -t $USERS:$COMMIT -f Dockerfile-$DOCKER_ENV
    docker tag $USERS:$COMMIT $REPO/$USERS:$TAG
    docker push $REPO/$USERS:$TAG
    # users db
    docker build $USERS_DB_REPO -t $USERS_DB:$COMMIT -f Dockerfile
    docker tag $USERS_DB:$COMMIT $REPO/$USERS_DB:$TAG
    docker push $REPO/$USERS_DB:$TAG
    # client
    docker build $CLIENT_REPO -t $CLIENT:$COMMIT -f Dockerfile-$DOCKER_ENV --build-arg REACT_APP_USERS_SERVICE_URL=$REACT_APP_USERS_SERVICE_URL --build-arg REACT_APP_EXERCISES_SERVICE_URL=$REACT_APP_EXERCISES_SERVICE_URL --build-arg REACT_APP_API_GATEWAY_URL=$REACT_APP_API_GATEWAY_URL
    docker tag $CLIENT:$COMMIT $REPO/$CLIENT:$TAG
    docker push $REPO/$CLIENT:$TAG
    # swagger
    docker build $SWAGGER_REPO -t $SWAGGER:$COMMIT -f Dockerfile-$DOCKER_ENV $SWAGGER_DIR
    docker tag $SWAGGER:$COMMIT $REPO/$SWAGGER:$TAG
    docker push $REPO/$SWAGGER:$TAG
    # exercises
    docker build $EXERCISES_REPO -t $EXERCISES:$COMMIT -f Dockerfile-$DOCKER_ENV
    docker tag $EXERCISES:$COMMIT $REPO/$EXERCISES:$TAG
    docker push $REPO/$EXERCISES:$TAG
    # exercises db
    docker build $EXERCISES_DB_REPO -t $EXERCISES_DB:$COMMIT -f Dockerfile
    docker tag $EXERCISES_DB:$COMMIT $REPO/$EXERCISES_DB:$TAG
```

```
      docker push $REPO/$EXERCISES_DB:$TAG
      # scores
      docker build $SCORES_REPO -t $SCORES:$COMMIT -f Dockerfile-$DOCKER_ENV
      docker tag $SCORES:$COMMIT $REPO/$SCORES:$TAG
      docker push $REPO/$SCORES:$TAG
      # scores db
      docker build $SCORES_DB_REPO -t $SCORES_DB:$COMMIT -f Dockerfile
      docker tag $SCORES_DB:$COMMIT $REPO/$SCORES_DB:$TAG
      docker push $REPO/$SCORES_DB:$TAG
    fi
  fi
```

### Add images to ECR

Commit your code, check out the `staging` branch locally, and then rebase `master` on `staging` :

```
$ git checkout staging
$ git rebase master
```

Add the following Image repos to ECR:

1. `test-driven-scores`
2. `test-driven-scores_db`

Push to GitHub to trigger a new build on Travis. Make sure the build passes and that the images were successfully pushed to ECR.

### Configure Task Definitions

Add `scores` to the `deploy_cluster` function in *docker-deploy-stage.sh*:

```
# scores
service="testdriven-scores-stage-service"
template="ecs_scores_stage_taskdefinition.json"
task_template=$(cat "ecs/$template")
task_def=$(printf "$task_template" $AWS_ACCOUNT_ID $AWS_ACCOUNT_ID)
echo "$task_def"
register_definition
```

Add a new Task Definition file called *ecs_scores_stage_taskdefinition.json*:

```
{
  "containerDefinitions": [
    {
      "name": "scores",
      "image": "%s.dkr.ecr.us-west-1.amazonaws.com/test-driven-scores:staging",
      "essential": true,
```

```
        "memoryReservation": 300,
        "portMappings": [
          {
            "hostPort": 0,
            "protocol": "tcp",
            "containerPort": 5000
          }
        ],
        "environment": [
          {
            "name": "APP_SETTINGS",
            "value": "project.config.StagingConfig"
          },
          {
            "name": "DATABASE_TEST_URL",
            "value": "postgres://postgres:postgres@scores-db:5432/scores_test"
          },
          {
            "name": "DATABASE_URL",
            "value": "postgres://postgres:postgres@scores-db:5432/scores_stage"
          },
          {
            "name": "USERS_SERVICE_URL",
            "value": "LOAD_BALANCER_STAGE_DNS_NAME"
          },
          {
            "name": "EXERCISES_SERVICE_URL",
            "value": "LOAD_BALANCER_STAGE_DNS_NAME"
          }
        ],
        "links": [
          "scores-db"
        ],
        "logConfiguration": {
          "logDriver": "awslogs",
          "options": {
            "awslogs-group": "testdriven-scores-stage",
            "awslogs-region": "us-west-1"
          }
        }
      },
      {
        "name": "scores-db",
        "image": "%s.dkr.ecr.us-west-1.amazonaws.com/test-driven-scores_db:staging",
        "essential": true,
        "memoryReservation": 300,
        "portMappings": [
          {
            "hostPort": 0,
            "protocol": "tcp",
            "containerPort": 5432
```

```
        }
      ],
      "environment": [
        {
          "name": "POSTGRES_PASSWORD",
          "value": "postgres"
        },
        {
          "name": "POSTGRES_USER",
          "value": "postgres"
        }
      ],
      "logConfiguration": {
        "logDriver": "awslogs",
        "options": {
          "awslogs-group": "testdriven-scores_db-stage",
          "awslogs-region": "us-west-1"
        }
      }
    }
  ],
  "family": "testdriven-scores-stage-td"
}
```

Add the log groups to CloudWatch:

1. `testdriven-scores-stage`
2. `testdriven-scores_db-stage`

Commit and push your code to trigger a new build on Travis. Ensure that both images and Task Definitions were created.

## Create Service

Create the following ECS Service on the `test-driven-staging-cluster` Cluster...

#### Scores

*Configure service*:

1. "Task Definition": `testdriven-scores-stage-td:LATEST_REVISION_NUMBER`
2. "Service name": `testdriven-scores-stage-service`
3. "Number of tasks": `1`

Click "Next".

*Configure network*:

Select the "Application Load Balancer" under "Load balancer type".

1. "Load balancer name": `testdriven-staging-alb`

2. "Container name : port": `scores:0:5000`

Click "Add to load balancer".

1. "Listener port": `80:HTTP`
2. "Target group name": `testdriven-scores-stage-tg`

Click the next button a few times, and then "Create Service".

Navigate to the EC2 Dashboard, and click "Target Groups". Make sure `testdriven-scores-stage-tg` has a single registered instance. The instance should be healthy.

## Update *docker-deploy-stage.sh*

Update the `scores` part of the `deploy_cluster` function in *docker-deploy-stage.sh* to call `update_service` :

```
# scores
service="testdriven-scores-stage-service"
template="ecs_scores_stage_taskdefinition.json"
task_template=$(cat "ecs/$template")
task_def=$(printf "$task_template" $AWS_ACCOUNT_ID $AWS_ACCOUNT_ID)
echo "$task_def"
register_definition
update_service
```

Update *swagger.json*:

```
$ python services/swagger/update-spec.py http://LOAD_BALANCER_STAGE_DNS_NAME
```

Commit and push your code to GitHub to trigger a new Travis build. Once done, you should see a new revision associated with the Task Definitions and the Services should now be running a new Task based on that revision.

Test each URL in the browser:

| Endpoint | HTTP Method | Authenticated? | Result |
|---|---|---|---|
| /auth/register | POST | No | register user |
| /auth/login | POST | No | log in user |
| /auth/logout | GET | Yes | log out user |
| /auth/status | GET | Yes | check user status |
| /users | GET | No | get all users |
| /users/:id | GET | No | get single user |
| /users | POST | Yes (admin) | add a user |
| /users/ping | GET | No | sanity check |

| | | | |
|---|---|---|---|
| /exercises | GET | No | get all exercises |
| /exercises | POST | Yes (admin) | add an exercise |
| /scores/ping | GET | No | sanity check |
| /scores | GET | No | get all scores |
| /scores/user | GET | Yes | get all scores by user id |
| /scores/user/:id | GET | Yes | get single score by user id |
| /scores | POST | Yes | add a score |
| /scores/:exercise_id | PUT | Yes | update a score by exercise id |

> Remember: If you run into errors, you can always check the logs on CloudWatch or SSH directly into the EC2 instance to debug the containers:
>
> ```
> $ ssh -i ~/.ssh/ecs.pem ec2-user@EC2_PUBLIC_IP
> ```
>
> Be sure to double-check all environment variables!

Make sure the end-to-end tests pass as well:

```
$ export TEST_URL=http://LOAD_BALANCER_STAGE_DNS_NAME
$ export SERVER_URL=http://LOAD_BALANCER_STAGE_DNS_NAME
$ testcafe chrome e2e
```

Results:

```
/
✓ should display the exercises correctly if a user is not logged in
✓ should allow a user to submit an exercise if logged in
✓ should allow a user to move to different exercises

/
✓ should display the all-users page correctly if a user is not logged in

/login
✓ should display the sign in form
✓ should allow a user to sign in
✓ should throw an error if the credentials are incorrect

/register
✓ should display flash messages correctly

/register
✓ should display the registration form
✓ should allow a user to register
✓ should validate the password field
✓ should throw an error if the username is taken
```

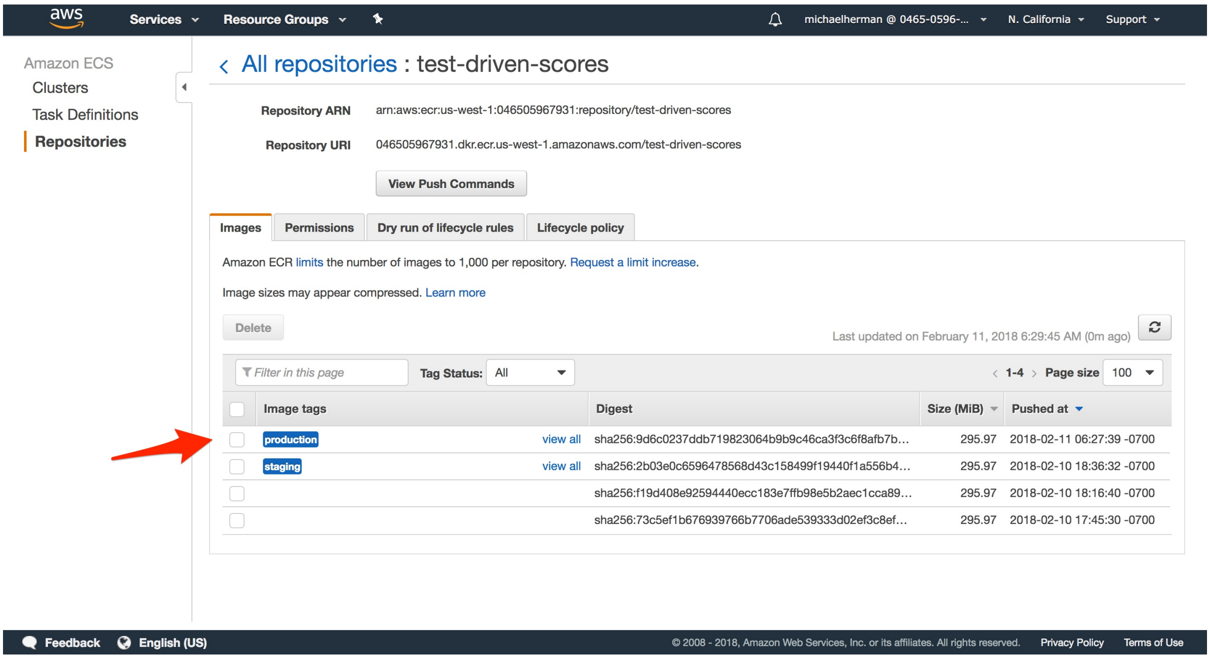
```
✓ should throw an error if the email is taken

/status
✓ should not display user info if a user is not logged in
✓ should display user info if a user is logged in

/swagger
✓ should display the swagger docs correctly

/all-users
✓ should display the all-users page correctly if a user is not logged in


17 passed (1m 28s)
```

# ECS Prod Update

In this lesson, we'll update the production environment on ECS...

---

Assuming you are still on the `staging` branch, commit your code and push it up to GitHub. Open a PR against the `production` branch and merge it. Make sure the `production` build passes and that the images were successfully pushed to ECR:




Check out the `production` branch locally:

```
$ git checkout production
$ git pull origin production
```

Review the production deployment lessons in parts 5 and 6 along with the staging deployment lesson in part 7. Check your understanding and update the remainder of the production environment on your own.

Think about how to handle the adding of scores for existing users. You will probably want to run a script, before the `scores` services fires up, to check to see if the `scores` RDS database is empty; and, if so, run the migrations and seed the database.

That's it! You're on your own. Good luck.

## Steps

*RDS*

1. Add the a new RDS instance for the `scores` database
2. Apply the migrations and seed the db

> Turn back to the RDS section in the deployment lesson of part 6 if you need help.

*Load Balancer*

1. Configure Target Group
2. Add Listener to the ALB

*ECS*

1. Configure Task Definitions
2. Create Service
3. Update *docker-deploy-stage.sh*

# Next Steps

Well, that's it. It's your turn! Spend some time refactoring and dealing with tech debt on your own...

1. **More tests**: Increase the overall test coverage of each service.
2. **Test the Lambda function**: Try testing with AWS Serverless Application Model (AWS SAM)
3. **Task queue**: Add a simple task queue - like Redis Queue, RabbitMQ, or Amazon Simple Queue Service (SQS)
4. **Swagger**: Add API documentation for the `exercises` and `scores` services as well as expected responses for errors for all services.
5. **Exercise component state**: What happens if a user submits an exercise and then closes the browser before it's complete? Also, how would you indicate to the end user that they have already submitted an exercise?
6. **DRY out the code**: There's plenty of places in the code base that could be refactored. Did you notice that we could clean up the exercise status message (grading, incorrect, correct) logic by organizing it into a single method? Try this on your own.
7. **Scores**: Refactor. You probably already noticed, but things are a bit finicky. With tests in place, refactor as necessary. Write new tests. Maybe individual users could just view their own scores while an admin can view all user scores.
8. **Redux**: Managing state on the client-side is starting to get difficult. Redux can help. It's a major refactor, but it's well worth it if you continue to add new features.
9. **Multistage Docker builds**: Using the multistage build pattern, refactor the remaining production Dockerfiles.
10. **Flask CORS**: Instead of allowing requests from any domain, lock down the Flask services by only allowing requests that originate from one of the services running on AWS.

It's also a great time to pause, review the code, and write more unit, integration, and end-to-end tests. Do this on your own to check your understanding.

> Want feedback on your code? Shoot an email to `michael@mherman.org` with a link to the GitHub repo. Cheers!

# Structure

At the end of part 7, your project structure should look like this:

```
├── README.md
├── docker-compose-dev.yml
├── docker-compose-prod.yml
├── docker-compose-stage.yml
├── docker-deploy-prod.sh
├── docker-deploy-stage.sh
├── docker-push.sh
├── e2e
│   ├── exercise.js
│   ├── index.test.js
│   ├── login.test.js
│   ├── message.test.js
│   ├── register.test.js
│   ├── status.test.js
│   ├── swagger.test.js
│   └── users.test.js
├── ecs
│   ├── ecs_client_prod_taskdefinition.json
│   ├── ecs_client_stage_taskdefinition.json
│   ├── ecs_exercises_prod_taskdefinition.json
│   ├── ecs_exercises_stage_taskdefinition.json
│   ├── ecs_scores_prod_taskdefinition.json
│   ├── ecs_scores_stage_taskdefinition.json
│   ├── ecs_swagger_prod_taskdefinition.json
│   ├── ecs_swagger_stage_taskdefinition.json
│   ├── ecs_users_prod_taskdefinition.json
│   └── ecs_users_stage_taskdefinition.json
├── init_db.sh
├── package.json
├── services
│   ├── client
│   │   ├── Dockerfile-dev
│   │   ├── Dockerfile-prod
│   │   ├── Dockerfile-stage
│   │   ├── README.md
│   │   ├── build
│   │   ├── conf
│   │   │   └── conf.d
│   │   │       └── default.conf
│   │   ├── coverage
│   │   ├── package.json
│   │   ├── public
│   │   │   ├── favicon.ico
│   │   │   ├── index.html
│   │   │   ├── main.css
```

```
│   │   │   └── manifest.json
│   │   └── src
│   │       ├── App.jsx
│   │       ├── components
│   │       │   ├── About.jsx
│   │       │   ├── AddUser.jsx
│   │       │   ├── Exercise.jsx
│   │       │   ├── Exercises.jsx
│   │       │   ├── Footer.css
│   │       │   ├── Footer.jsx
│   │       │   ├── Logout.jsx
│   │       │   ├── Message.jsx
│   │       │   ├── NavBar.css
│   │       │   ├── NavBar.jsx
│   │       │   ├── UserStatus.jsx
│   │       │   ├── UsersList.jsx
│   │       │   ├── __tests__
│   │       │   │   ├── About.test.jsx
│   │       │   │   ├── AddUser.test.jsx
│   │       │   │   ├── App.test.jsx
│   │       │   │   ├── Exercise.test.jsx
│   │       │   │   ├── Exercises.test.jsx
│   │       │   │   ├── Footer.test.jsx
│   │       │   │   ├── Form.test.jsx
│   │       │   │   ├── FormErrors.test.jsx
│   │       │   │   ├── Logout.test.jsx
│   │       │   │   ├── Message.test.jsx
│   │       │   │   ├── NavBar.test.jsx
│   │       │   │   ├── UsersList.test.jsx
│   │       │   │   ├── UsersStatus.test.jsx
│   │       │   │   └── __snapshots__
│   │       │   │       ├── About.test.jsx.snap
│   │       │   │       ├── AddUser.test.jsx.snap
│   │       │   │       ├── Exercise.test.jsx.snap
│   │       │   │       ├── Exercises.test.jsx.snap
│   │       │   │       ├── Footer.test.jsx.snap
│   │       │   │       ├── Form.test.jsx.snap
│   │       │   │       ├── FormErrors.test.jsx.snap
│   │       │   │       ├── Logout.test.jsx.snap
│   │       │   │       ├── Message.test.jsx.snap
│   │       │   │       ├── NavBar.test.jsx.snap
│   │       │   │       ├── UsersList.test.jsx.snap
│   │       │   │       └── UsersStatus.test.jsx.snap
│   │       │   └── forms
│   │       │       ├── Form.jsx
│   │       │       ├── FormErrors.css
│   │       │       ├── FormErrors.jsx
│   │       │       └── form-rules.js
│   │       ├── index.js
│   │       ├── logo.svg
│   │       ├── registerServiceWorker.js
```

```
│   │       └── setupTests.js
│   ├── exercises
│   │   ├── Dockerfile-dev
│   │   ├── Dockerfile-prod
│   │   ├── Dockerfile-stage
│   │   ├── entrypoint-stage.sh
│   │   ├── entrypoint.sh
│   │   ├── htmlcov
│   │   ├── manage.py
│   │   ├── project
│   │   │   ├── __init__.py
│   │   │   ├── api
│   │   │   │   ├── __init__.py
│   │   │   │   ├── base.py
│   │   │   │   ├── exercises.py
│   │   │   │   ├── models.py
│   │   │   │   └── utils.py
│   │   │   ├── config.py
│   │   │   ├── db
│   │   │   │   ├── Dockerfile
│   │   │   │   └── create.sql
│   │   │   └── tests
│   │   │       ├── __init__.py
│   │   │       ├── base.py
│   │   │       ├── test_base.py
│   │   │       ├── test_config.py
│   │   │       ├── test_exercises_api.py
│   │   │       ├── test_exercises_model.py
│   │   │       └── utils.py
│   │   └── requirements.txt
│   ├── lambda
│   │   └── handler.py
│   ├── nginx
│   │   ├── Dockerfile-dev
│   │   ├── Dockerfile-prod
│   │   ├── Dockerfile-stage
│   │   ├── dev.conf
│   │   └── prod.conf
│   ├── scores
│   │   ├── Dockerfile-dev
│   │   ├── Dockerfile-prod
│   │   ├── Dockerfile-stage
│   │   ├── entrypoint-stage.sh
│   │   ├── entrypoint.sh
│   │   ├── htmlcov
│   │   ├── manage.py
│   │   ├── project
│   │   │   ├── __init__.py
│   │   │   ├── api
│   │   │   │   ├── __init__.py
│   │   │   │   ├── base.py
```

```
│   │   │   │   ├── models.py
│   │   │   │   ├── scores.py
│   │   │   │   └── utils.py
│   │   │   ├── config.py
│   │   │   ├── db
│   │   │   │   ├── Dockerfile
│   │   │   │   └── create.sql
│   │   │   └── tests
│   │   │       ├── __init__.py
│   │   │       ├── base.py
│   │   │       ├── test_base.py
│   │   │       ├── test_config.py
│   │   │       ├── test_scores_api.py
│   │   │       ├── test_scores_model.py
│   │   │       └── utils.py
│   │   └── requirements.txt
│   ├── swagger
│   │   ├── Dockerfile-dev
│   │   ├── Dockerfile-prod
│   │   ├── Dockerfile-stage
│   │   ├── nginx.conf
│   │   ├── start.sh
│   │   ├── swagger.json
│   │   └── update-spec.py
│   └── users
│       ├── Dockerfile-dev
│       ├── Dockerfile-prod
│       ├── Dockerfile-stage
│       ├── entrypoint-stage.sh
│       ├── entrypoint.sh
│       ├── htmlcov
│       ├── manage.py
│       ├── migrations
│       ├── project
│       │   ├── __init__.py
│       │   ├── api
│       │   │   ├── __init__.py
│       │   │   ├── auth.py
│       │   │   ├── models.py
│       │   │   ├── templates
│       │   │   │   └── index.html
│       │   │   ├── users.py
│       │   │   └── utils.py
│       │   ├── config.py
│       │   ├── db
│       │   │   ├── Dockerfile
│       │   │   └── create.sql
│       │   └── tests
│       │       ├── __init__.py
│       │       ├── base.py
│       │       ├── test_auth.py
```

```
│       │       ├── test_config.py
│       │       ├── test_user_model.py
│       │       ├── test_users.py
│       │       └── utils.py
│       └── requirements.txt
└── test.sh
```

Code for part 7: https://github.com/testdrivenio/testdriven-app-2.2/releases/tag/part7