



Microservices

with Docker, Flask, and React



Michael Herman

Powered By
testdriven.io

Table of Contents

Introduction	1.1
Part 1	1.2
Microservices	1.2.1
App Overview	1.2.2
Getting Started	1.2.3
Docker Config	1.2.4
Postgres Setup	1.2.5
Test Setup	1.2.6
Flask Blueprints	1.2.7
RESTful Routes	1.2.8
Deployment	1.2.9
Jinja Templates	1.2.10
Workflow	1.2.11
Part 2	1.3
Code Coverage and Quality	1.3.1
Continuous Integration	1.3.2
React Setup	1.3.3
Testing React	1.3.4
React Forms	1.3.5
React and Docker	1.3.6
Next Steps	1.3.7
Part 3	1.4
Flask Migrate	1.4.1
Flask Bcrypt	1.4.2
JWT Setup	1.4.3
Auth Routes	1.4.4
React Router	1.4.5
React Bootstrap	1.4.6
React Authentication - part 1	1.4.7
Mocking User Interaction	1.4.8
React Authentication - part 2	1.4.9
Authorization	1.4.10
Update Components	1.4.11

Update Docker	1.4.12
Part 4	1.5
End-to-End Test Setup	1.5.1
End-to-End Test Specs	1.5.2
React Component Refactor	1.5.3
React Form Validation	1.5.4
React Flash Messaging	1.5.5
Swagger Setup	1.5.6
Staging Environment	1.5.7
Production Environment	1.5.8
Workflow	1.5.9
Part 5	1.6
Container Orchestration	1.6.1
EC2 Container Registry	1.6.2
Elastic Load Balancer	1.6.3
EC2 Container Service	1.6.4
ECS Staging	1.6.5
Setting up RDS	1.6.6
ECS Production Setup	1.6.7
ECS Production Automation	1.6.8
Workflow	1.6.9
Part 6	1.7
React Refactor	1.7.1
React Ace Code Editor	1.7.2
Exercises Service Setup	1.7.3
Exercises Database	1.7.4
Exercises API	1.7.5
Code Evaluation with AWS Lambda	1.7.6
Update Exercises Component	1.7.7
ECS Staging	1.7.8
ECS Production	1.7.9
Scaling	1.7.10
Workflow	1.7.11
Next Steps	1.7.12

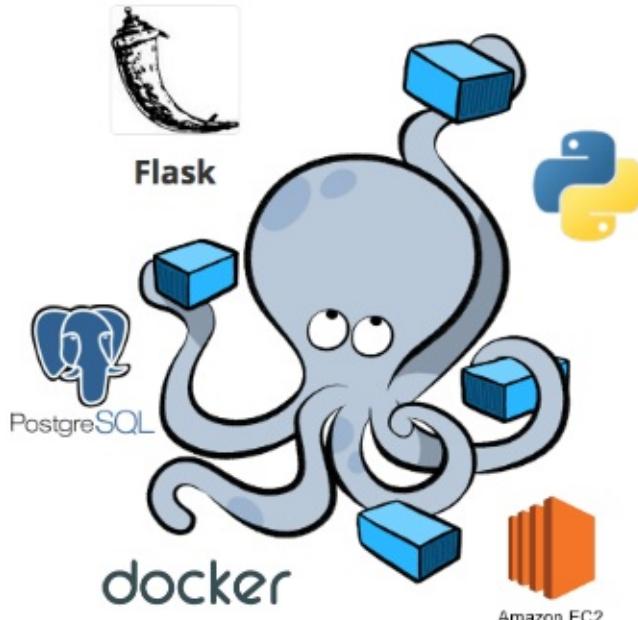
Microservices with Docker, Flask, and React

```
{  
  "version": "2.0.0",  
  "author": "Michael Herman",  
  "email": "michael@mherman.org",  
  "website": "https://testdriven.io",  
  "copyright": "Copyright 2017 Michael Herman. All rights reserved."  
}
```

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

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, Nginx, 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/realpython/testdriven-app/releases/tag/part1>

Dependencies

You will use the following dependencies in Part 1:

1. Python v3.6.3
2. Flask v0.12.2
3. Flask-Script v2.0.5
4. Docker v17.09.0-ce
5. Docker Compose v1.16.1
6. Docker Machine v0.12.2
7. Docker Compose file v3.3
8. Flask-SQLAlchemy v2.3.2
9. psycopg2 v2.7.3.1
10. Flask-Testing v0.6.2
11. Gunicorn v19.7.1
12. Nginx v1.13.5
13. Bootstrap 4

Microservices

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

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, and tools.

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.

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.

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 easier to work on, test, refactor, and scale.

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, and JavaScript, 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.

[TestDriven.io](#) [About](#) [Users](#) [Swagger](#)

[Register](#) [Log In](#)

Exercises

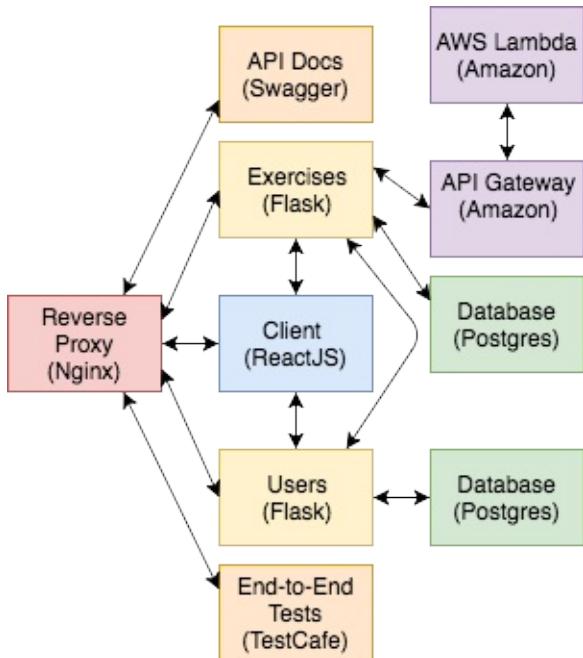
 Please log in to submit an exercise.

Define a function called sum that takes two integers as arguments and returns their sum.

```
1 # Enter your code here.
```

Copyright 2017 [TestDriven.io](#).

We'll also be practicing 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 users-service && cd users-service
$ 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:

```
# users-service/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 [Flask Script](#) to run and manage the app from the command line:

```
(env)$ pip install flask-script==2.0.5
```

Since Flask Script is [deprecated](#), you are more than welcome to use the [Flask CLI](#) in its place. Review the following [article](#) for more info.

Add a `manage.py` file to the "users-service" directory:

```
# users-service/manage.py

from flask_script import Manager
```

```
from project import app

manager = Manager(app)

if __name__ == '__main__':
    manager.run()
```

Here, we created a new `Manager` instance to handle all of the manager commands from the command line.

Run the server:

```
(env)$ python manage.py runserver
```

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:

```
# users-service/project/config.py

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

class DevelopmentConfig(BaseConfig):
    """Development configuration"""
    DEBUG = True

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

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

```
DEBUG = False
```

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

```
# users-service/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, `debug mode` should be on:

```
$ python manage.py runserver
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
* Restarting with stat
* Debugger is active!
* Debugger PIN: 107-952-069
```

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-service" directory:

```
Flask==0.12.1
Flask-Script==2.0.5
```

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 17.09.0-ce, build afdb6d4
$ docker-compose -v
docker-compose version 1.16.1, build 6d1ac21
$ docker-machine -v
docker-machine version 0.12.2, build 9371605
```

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

Learn more about the above command [here](#).

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

```
FROM python:3.6.3

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

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

# install requirements
RUN pip install -r requirements.txt

# add app
ADD . /usr/src/app

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

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

```
version: '3.3'

services:
```

```
users-service:  
  container_name: users-service  
  build:  
    context: ./users-service  
    dockerfile: Dockerfile-dev  
  volumes:  
    - './users-service:/usr/src/app'  
  ports:  
    - 5001:5000
```

This config will create a container called `users-service`, 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 - `3.3`. 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: '3.3'  
  
services:  
  
  users-service:  
    container_name: users-service
```

```
build:
  context: ./users-service
  dockerfile: Dockerfile-dev
volumes:
  - './users-service:/usr/src/app'
ports:
  - 5001:5000
environment:
  - APP_SETTINGS=project.config.DevelopmentConfig
```

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

```
# users-service/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:

```
print(app.config)
```

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-service` container...

Add [Flask-SQLAlchemy](#) and psycopg2 to the `requirements.txt` file:

```
Flask-SQLAlchemy==2.3.2
psycopg2==2.7.3.1
```

Update `config.py`:

```
# users-service/project/config.py

import os

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

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

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

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

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

```
# users-service/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: '3.3'

services:

  users-service:
    container_name: users-service
    build:
      context: ./users-service
      dockerfile: Dockerfile-dev
    volumes:
      - './users-service:/usr/src/app'
    ports:
      - 5001:5000
    environment:
      - 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: ./users-service/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-service":

```
#!/bin/sh

echo "Waiting for postgres..."

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

echo "PostgreSQL started"

python manage.py runserver -h 0.0.0.0
```

Update *Dockerfile-dev*:

```
FROM python:3.6.3

# 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
ADD ./requirements.txt /usr/src/app/requirements.txt

# install requirements
RUN pip install -r requirements.txt

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

# add app
ADD . /usr/src/app

# run server
CMD ["./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`:

```
# users-service/manage.py

from flask_script import Manager

from project import app, db

manager = Manager(app)

@manager.command
def recreate_db():
    """Recreates a database."""
    db.drop_all()
    db.create_all()
    db.session.commit()

if __name__ == '__main__':
    manager.run()
```

This registers a new command, `recreate_db`, to the manager 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-service 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`

```
# users-service/project/tests/__init__.py
```

`base.py`

```
# users-service/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`:

```
# users-service/project/tests/test_config.py
```

```
import os
```

```

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.assertTrue(app.config['DEBUG'])
        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['DEBUG'])
        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['DEBUG'])
        self.assertFalse(app.config['TESTING'])

if __name__ == '__main__':

```

```
unittest.main()
```

test_users.py

```
# users-service/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:

```
@manager.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-service 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"""
    DEBUG = False
    TESTING = False
    SQLALCHEMY_TRACK_MODIFICATIONS = False
    SECRET_KEY = 'my_precious'
```

Then re-test!

```
Ran 4 tests in 0.118s
```

```
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:

```
# users-service/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.

The, add the following code to `models.py`:

```
# users-service/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:

```
# users-service/project/__init__.py

import os
from flask import Flask, jsonify
from flask_sqlalchemy import SQLAlchemy

# instantiate the db
db = SQLAlchemy()

def create_app():

    # 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)

    return app
```

Update `manage.py`:

```
# users-service/manage.py

import unittest

from flask_script import Manager

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

app = create_app()
manager = Manager(app)
```

```

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

@manager.command
def recreate_db():
    """Recreates a database."""
    db.drop_all()
    db.create_all()
    db.session.commit()

if __name__ == '__main__':
    manager.run()

```

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)

```

Test!

```

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

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

$ docker-compose -f docker-compose-dev.yml \
    run users-service 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's 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@realpython.com'
            }),
            content_type='application/json',
        )
        data = json.loads(response.data.decode())
        self.assertEqual(response.status_code, 201)
        self.assertIn('michael@realpython.com 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-service 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 0.201s

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@realpython.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'])

def test_add_user_duplicate_user(self):
    """Ensure error is thrown if the email already exists."""
    with self.client:
        self.client.post(
            '/users',
            data=json.dumps({
                'username': 'michael',
                'email': 'michael@realpython.com'
            }),
            content_type='application/json',
        )
        response = self.client.post(
            '/users',
            data=json.dumps({
                'username': 'michael',
                'email': 'michael@realpython.com'
            }),
            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@realpython.com')
    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@realpython.com', 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@realpython.com')
    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@realpython.com', 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@realpython.com')
    add_user('fletcher', 'fletcher@realpython.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@realpython.com', data['data']['users'][0]['email'])
self.assertIn('fletcher', data['data']['users'][1]['username'])
self.assertIn(
    'fletcher@realpython.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 pass?

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:

```
@manager.command
def seed_db():
    """Seeds the database."""
    db.session.add(User(username='michael', email="michael@realpython.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-service python manage.py seed_db
```

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

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

$ docker-compose -f docker-compose-prod.yml \
  run users-service 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-service" 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
```

Add a new Dockerfile called *Dockerfile-prod*:

```
FROM python:3.6.3  
  
# 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  
ADD ./requirements.txt /usr/src/app/requirements.txt  
  
# install requirements  
RUN pip install -r requirements.txt  
  
# add entrypoint.sh  
ADD ./entrypoint-prod.sh /usr/src/app/entrypoint-prod.sh  
  
# add app  
ADD . /usr/src/app  
  
# run server  
CMD ["./entrypoint-prod.sh"]
```

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

```
build:  
  context: ./users-service  
  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 project root, and then add a *Dockerfile*:

```
FROM nginx:1.13.5

RUN rm /etc/nginx/conf.d/default.conf
ADD /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-service: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: ./nginx/
  restart: always
  ports:
    - 80:80
  depends_on:
    - users-service
  links:
    - users-service
```

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: ./nginx/  
  restart: always  
  ports:  
    - 80:80  
  depends_on:  
    - users-service  
  links:  
    - users-service
```

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 `users-service/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')
```

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-beta/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>
<!-- 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-beta/js/bootstrap.min.js"
>
</script>
{% block js %}{% endblock %}
</body>
</html>

```

Be sure to update the imports:

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

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

All Users

Enter a username

Enter an email address

Submit

No users!

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'

# All Users

', response.data)
    self.assertIn(b'

No users!

', response.data)
```

Do they pass?

```
$ docker-compose -f docker-compose-dev.yml \
  run users-service 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@realpython.com')
add_user('fletcher', 'fletcher@realpython.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 the 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@realpython.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					

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-service python manage.py recreate_db
```

Seed the database:

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

Run the tests:

```
$ docker-compose -f docker-compose-dev.yml \
  run users-service 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;
```

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 current project structure:

```

├── docker-compose-dev.yml
├── docker-compose-prod.yml
├── nginx
|   ├── Dockerfile
|   └── flask.conf
└── users-service
    ├── 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
    |       └── init.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:

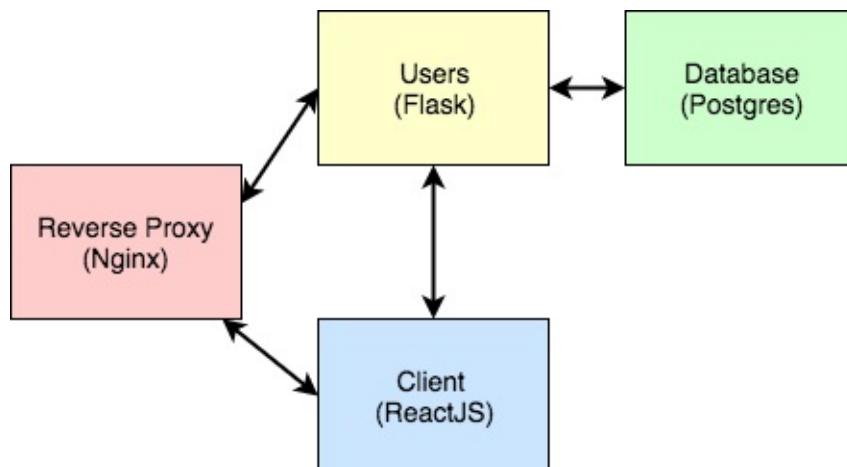
1. [flask-microservices-main](#) - Docker Compose files, Nginx, admin scripts
2. [flask-microservices-users](#) - Flask app for managing users and auth
3. [flask-microservices-client](#) - client-side, React app
4. [flask-microservices-swagger](#) - Swagger API docs
5. [flask-microservices-eval](#) - 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

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/realpython/testdriven-app/releases/tag/part2>

Dependencies

You will use the following dependencies in Part 2:

1. Coverage.py v4.4.1
2. flake8 v3.4.1
3. Node v8.7.0
4. npm v5.4.2
5. Create React App v1.4.2
6. React v16.0.0
7. React Scripts v1.0.15
8. Axios v0.16.2
9. Flask-CORS v3.0.3
10. jest-cli v20.0.4
11. Enzyme v3.1.0
12. enzyme-adapter-react-16 v1.0.2

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-service 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-service" directory:

```
coverage==4.4.1
```

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/*'  
    ]  
)  
COV.start()
```

Then add the new manager command:

```
@manager.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-service python manage.py cov
```

You should see something like:

Coverage Summary:					
Name	Stmts	Miss	Branch	BrPart	Cover
project/__init__.py	12	5	0	0	58%
project/api/models.py	12	9	0	0	25%
project/api/users.py	48	0	10	0	100%
project/config.py	16	0	0	0	100%
TOTAL	88	14	10	0	86%

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` file.

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-service" directory:

```
flake8==3.4.1
```

Update the containers:

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

Run flake8:

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

Were any errors found?

```
project/__init__.py:7:1: F401 'flask.jsonify' imported but unused
project/tests/test_users.py:60:80: E501 line too long (85 > 79 characters)
```

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](#) guid 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: 1.14.0

before_install:
  - sudo rm /usr/local/bin/docker-compose
  - curl -L https://github.com/docker/compose/releases/download/${DOCKER_COMPOSE_VERSION}/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-service python manage.py test
  - docker-compose -f docker-compose-dev.yml run users-service 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, be sure to 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?branch=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

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  
v8.7.0  
$ npm -v  
5.4.2
```

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@1.4.2 --global
```

Add a new directory to the project root called "client" 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, 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.

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 use 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` in the `head`:

```
<link
```

```
    href="//maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css"
    rel="stylesheet"
>
```

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 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@0.16.2 --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-service 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 browsers "same origin policy". Fortunately, we can use the [Flask-CORS](#) extension to handle this.

Within the "users-service" 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 `users-service/project/_init_.py` like so:

```
def create_app():

    # 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)

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

    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-service python manage.py recreate_db

$ docker-compose -f docker-compose-dev.yml \
  run users-service 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": "michael@realpython.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 was mounted before it was complete? 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 calls [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>
    )
  }
};

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

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="well"
                >{ 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>
      {
        this.state.users.map((user) => {
          return (
            <h4
              key={user.id}
              className="well"
            >{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. You will need to install the [jest-cli](#) - in the "client" folder - to run the actual tests, though:

```
$ npm install jest-cli@20.0.4 --save-dev
```

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@3.1.0 enzyme-adapter-react-16@1.0.2
```

To configure the 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

Create a new file in "components" called *UsersList.test.js*:

```
const users = [
{
  'active': true,
  'email': 'michael@realpython.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 use the `shallow` helper method to create the `UsersList` component and then we can retrieve the output and make assertions on 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.

Snapshot Testing

Once the test is green, we'll then add a `Snapshot` test to ensure the UI does not change.

Add the following test to `UsersList.test.js`:

```
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. So, on the first test run, a snapshot is saved of the component output (to the `"__snapshots__"` folder). Then, during subsequent test runs, the new output is compared to the saved output. The test fails if they differ.

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. Now, if this change is intentional, you need to [update the snapshot](#). To do so, you 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.

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
```

Testing Interactions

Enzyme can also be used to test user interactions in terms of events. We can [simulate](#) such 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 poly
    fill 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 the 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 to the "client/src/components" directory:

1. *AddUser.jsx*
2. *AddUser.test.js*

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.

All Users

Enter a username

Enter an email address

Submit

michael

michaelherman

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

```
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 like so:

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

Then, update the constructor:

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

Update the `form` element again as well:

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

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 have the correct context. 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.

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 gets updated 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>
```

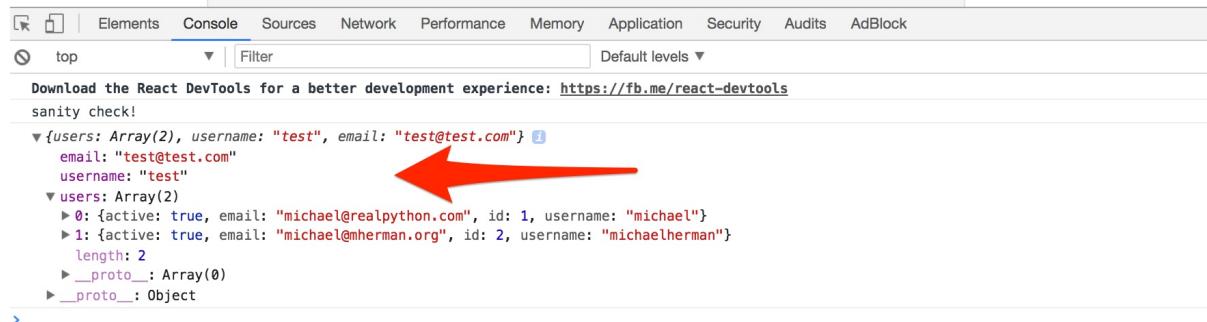
Test the form out now. It should be working. If curious, you can see the value of the state by logging it to the console in the `addUser` method:

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

All Users

A screenshot of a user input form. It has two input fields. The first field contains the text "test". The second field contains the text "test@test.com", which is highlighted with a yellow background. Below the fields is a blue "Submit" button.

A screenshot of a user input form. It has two input fields. The first field contains the text "michael". The second field contains the text "michaelherman".



Now that we have the values, let's send the AJAX request so the data can be added to the database and then update the DOM...

Send AJAX request

Turn 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))
```

Let's 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. It should work as long as the email address is unique.

If you have problems, analyze the response object from the "Network" tab in 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. Test it out. Then, run the tests. Update the snapshot test (by pressing `u` on the keyboard). Review and then commit your code.

React and Docker

Let's containerize the React app...

Refactor

Before we start, let's refactor the project structure.

Add a new folder to the project root called "services", and then add the "client", "nginx", and "users-service" directory to that folder. Then, rename the "users-service" directory to just "users".

Update `docker-compose-dev.yml`:

```
version: '3.3'

services:

  users-service:
    container_name: users-service
    build:
      context: ./services/users
      dockerfile: Dockerfile-dev
    volumes:
      - './services/users:/usr/src/app'
    ports:
      - 5001:5000
    environment:
      - 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

  nginx:
```

```

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

```

Update *docker-compose-prod.yml* as well:

```

version: '3.3'

services:

  users-service:
    container_name: users-service
    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
    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

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

```

```
- users-service  
links:  
- users-service
```

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-service python manage.py test
```

Local Development

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

```
FROM node:latest  
  
# 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  
ADD package.json /usr/src/app/package.json  
RUN npm install --silent  
RUN npm install react-scripts@1.0.15 -g --silent  
  
# start app  
CMD ["npm", "start"]
```

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'
ports:
  - '3007:3000'
environment:
  - NODE_ENV=development
  - REACT_APP_USERS_SERVICE_URL=${REACT_APP_USERS_SERVICE_URL}
depends_on:
  - users-service
links:
  - users-service

```

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-service: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-service
    - client
  links:
    - users-service

```

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
```

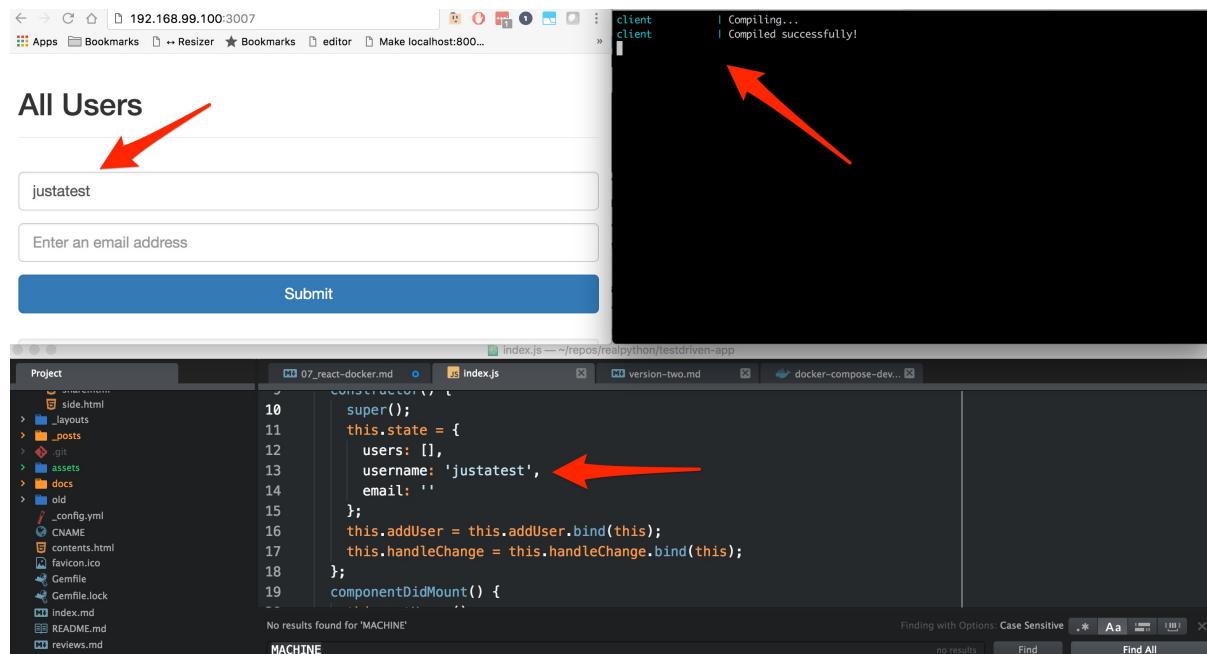
Clear the terminal screen, and then change the state object in the `App` component:

```

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? If you're using Docker Machine with VirtualBox, try [resetting](#) 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, 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, with "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 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:

```
FROM node:latest

# 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

# add environment variables
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

# install and cache app dependencies
ADD package.json /usr/src/app/package.json
RUN npm install --silent
RUN npm install pushstate-server -g --silent

# add app
ADD . /usr/src/app

# build react app
RUN npm run build

# start app
CMD ["pushstate-server", "build", "3000"]
```

When the image is built, we can pass arguments 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 3000 via the [pushstate-server](#).

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 3000 in the container to port 9000 outside the container:

```
$ docker run -d -p 9000:3000 test
```

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

Once done, grab the container ID by running `docker ps`, and then view the container's environment variables:

```
$ docker exec CONTAINER_ID bash -c 'env'
```

Stop and remove the container:

```
$ 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=development
    - REACT_APP_USERS_SERVICE_URL=${REACT_APP_USERS_SERVICE_URL}
  ports:
    - '3007:3000'
  depends_on:
    - users-service
  links:
    - users-service
```

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, `client` 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-service  
    - client  
  links:  
    - users-service
```

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 build time, if you update the variables, you *will* have to rebuild the Docker image.

Check the environment variables:

```
$ docker-compose -f docker-compose-prod.yml \  
run client env
```

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
```

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@realpython.com` with a link to the GitHub repo. Cheers!

Part 3

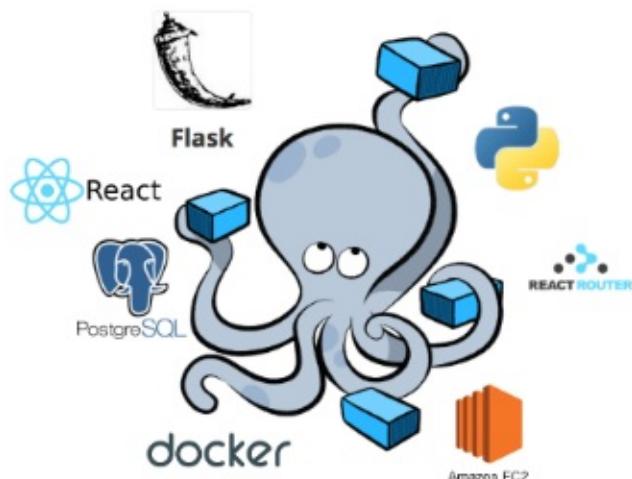
In Part 3, we'll add *database migrations* along with *password hashing* in order to implement *token-based authentication* to the users service with JSON Web Tokens (JWTs). We'll then turn our attention to the client and add *React Router* to the React app to enable client-side routing along with client-side 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/realpython/testdriven-app/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 v0.31.5
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-service 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:

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

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='justanother-test',
            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))

```

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

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:

```
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()*:

```
# users/project/__init__.py

import os

from flask_cors import CORS
from flask import Flask
from flask_sqlalchemy import SQLAlchemy
from flask_migrate import Migrate

# instantiate the db
db = SQLAlchemy()
# instantiate flask migrate
migrate = Migrate()

def create_app():

    # 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)
migrate.init_app(app, db)

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

return app
```

Then, add a new manager command to `services/users/manage.py`, just below `manager = Manager(app)`:

```
manager.add_command('db', MigrateCommand)
```

Add the import as well:

```
from flask_migrate import MigrateCommand
```

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-service python manage.py db init

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

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

$ docker-compose -f docker-compose-dev.yml \
  run users-service 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 db and start over.

Run the tests again. They should pass!

Refactor

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

First, in `services/users/project/tests/test_users.py`, rename `test_add_user_duplicate_user` to `test_add_user_duplicate_email`.

Also, 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":

```
# 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:

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

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))
```

Run the tests again to ensure nothing broke from the refactor. What about flake8?

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

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

```
# users/project/__init__.py

import os

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

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

def create_app():

    # 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)
    migrate.init_app(app, db)
```

```
migrate.init_app(app, db)

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

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', 'test')
    user_two = add_user('justatest2', 'test@test2.com', 'test')
    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.

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-service 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-service python manage.py db migrate

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

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 pass. Turning 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@realpython.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 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.assertTrue(app.config['DEBUG'])
        self.assertFalse(current_app is None)
        self.assertTrue(
            app.config['SQLALCHEMY_DATABASE_URI'] ==
            os.environ.get('DATABASE_URL'))
        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['DEBUG'])
        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.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['DEBUG'])
        self.assertFalse(app.config['TESTING'])
        self.assertTrue(app.config['BCRYPT_LOG_ROUNDS'] == 13)

```

Make sure the tests fail, then update `services/users/project/config.py`:

```

# 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

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:

```
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:

```
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 JWT 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
```

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"""
    DEBUG = False
```

```
TESTING = False
SQLALCHEMY_TRACK_MODIFICATIONS = False
SECRET_KEY = os.environ.get('SECRET_KEY')
BCRYPT_LOG_ROUNDS = 13
```

Add the `SECRET_KEY` environment variable to `environment` within `docker-compose-dev.yml`:

```
environment:
  - SECRET_KEY=my_precious
  - 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
```

Let's also add the token expiration to the config:

```
class BaseConfig:
    """Base configuration"""
    DEBUG = False
    TESTING = False
    SQLALCHEMY_TRACK_MODIFICATIONS = False
    SECRET_KEY = os.environ.get('SECRET_KEY')
    BCRYPT_LOG_ROUNDS = 13
    TOKEN_EXPIRATION_DAYS = 30
    TOKEN_EXPIRATION_SECONDS = 0

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
    TOKEN_EXPIRATION_DAYS = 0
    TOKEN_EXPIRATION_SECONDS = 3

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

Update the tests:

```

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.assertTrue(app.config['DEBUG'])
        self.assertFalse(current_app is None)
        self.assertTrue(
            app.config['SQLALCHEMY_DATABASE_URI'] ==
            os.environ.get('DATABASE_URL'))
        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['DEBUG'])
        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.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['DEBUG'])
self.assertFalse(app.config['TESTING'])
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.

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'))
    except:
```

```
    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 verify 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	20	10	0	0	50%
project/api/models.py	31	21	2	0	30%
project/api/users.py	53	0	12	0	100%
project/config.py	23	0	0	0	100%
TOTAL	127	31	14	0	77%

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 9.513s

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 5.513s
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:

```
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 5.885s

OK
```

Then, check coverage:

Coverage Summary:					
Name	Stmts	Miss	Branch	BrPart	Cover
<hr/>					
project/__init__.py	22	10	0	0	55%
project/api/auth.py	78	6	18	4	90%
project/api/models.py	31	17	2	0	48%
project/api/users.py	53	0	12	0	100%
project/config.py	23	0	0	0	100%
<hr/>					
TOTAL	207	33	32	4	85%

Finally, update `seed_db()` in `manage.py`:

```
@manager.command
def seed_db():
    """Seeds the database."""
    db.session.add(User(
        username='michael',
        email='michael@realpython.com',
        password='test'
    ))
    db.session.add(User(
        username='michaelherman',
        email='michael@mherman.org',
        password='test'
    ))
    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 our 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.js* within the "services/client/src" 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!

Router Setup

Add `react-router-dom` to the `dependencies` within *services/client/package.json* file:

```
"dependencies": {
  "axios": "^0.16.2",
  "react": "^16.0.0",
  "react-dom": "^16.0.0",
  "react-router-dom": "^4.2.2",
  "react-scripts": "1.0.14"
```

```
 },
```

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" called `*About.test.js`. And then run the tests to ensure they fail:

```
FAIL  src/components/About.test.js
● About renders properly

  TypeError: element.type is not a function
```

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}/>
<About/>
...
```

Make sure you can view the new component in the browser:

All Users

Enter a username

Enter an email address

Submit

michael

michaelherman

About

Add something relevant here.

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}>
              </div>
            )} />
          </Switch>
        </div>
      </div>
    </div>
  )
}
```

```
        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.js`:

```
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 Bootstrap](#) to the `package.json` file:

```
"dependencies": {  
  "axios": "^0.16.2",  
  "react": "^16.0.0",  
  "react-bootstrap": "^0.31.5",  
  "react-dom": "^16.0.0",  
  "react-router-bootstrap": "^0.24.4",  
  "react-router-dom": "^4.2.2",  
  "react-scripts": "1.0.14"  
},
```

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 in "src/components":

1. `NavBar.test.js`
2. `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()` :

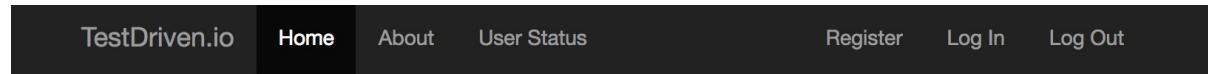
```
render() {
  return (
    <div>
      <NavBar
        title={this.state.title}
      />
      <div className="container">
        <div className="row">
          <div className="col-md-6">
            <br/>
            <Switch>
              <Route exact path='/' render={() => (
                <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>
    
```

```
)  
};
```

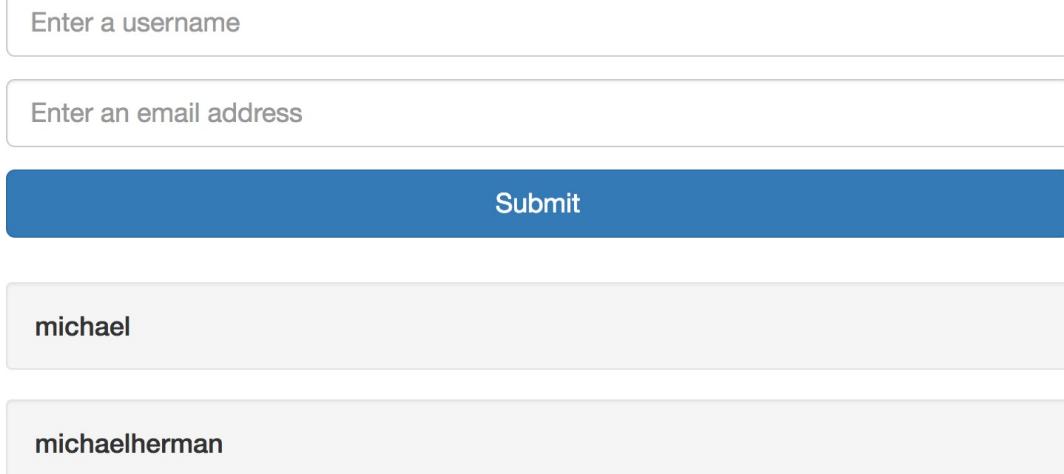
Update the container:

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

Test it out in the browser.



All Users



Enter a username
michael

Enter an email address
michaelherman

Submit

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.js*
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>
  );
}
```

```

        />
      </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/Form.test.js
PASS  src/components/NavBar.test.js
PASS  src/App.test.js
PASS  src/components/About.test.js
PASS  src/components/UsersList.test.js
PASS  src/components/AddUser.test.js

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/Form.test.js
PASS  src/components/NavBar.test.js
PASS  src/App.test.js
PASS  src/components/About.test.js
PASS  src/components/UsersList.test.js
PASS  src/components/AddUser.test.js

Test Suites: 6 passed, 6 total
Tests:       13 passed, 13 total
Snapshots:   6 passed, 6 total
Time:        3.118s, estimated 7s
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.

Test directory

Next, it's getting crowded in the "components" directory. Create a new directory within it called "`__tests__`", and move all of the `*.test.js` files into it:

1. `About.test.js`
2. `AddUser.test.js`
3. `Form.test.js`
4. `NavBar.test.js`
5. `UsersList.test.js`
6. `App.test.js`

Update the imports. Ensure the tests still pass. 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 on 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.email;
  }
  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 the Nginx config to handle requests to `/auth`:

```

location /auth {
  proxy_pass      http://users-service: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.eyJleHAiOjE00Tc3NTM2ODMsImlhdCI6MTQ5Nzc1MzY3OCwic3ViIjo0fQ.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` within `handleUserFormSubmit()` :

```
.then((res) => {
  this.setState({
    formData: {username: '', email: '', password: ''},
    username: '',
    email: ''
  });
})
```

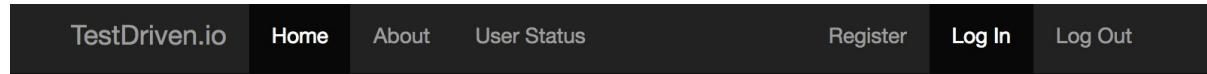
Try this out. After you register or log in, the field inputs should be cleared since we set the properties in the `formData` object to empty strings.

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 before `LocalStorage` and select `http://localhost:3000`. You should see a key of `authToken` with a value of the actual token in the pane.



Login

Key	Value
authToken	eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJleHAiOjE1...

Instead of always checking LocalStorage for the auth token, let's add a boolean to the state so we can quickly tell if there is a user 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 users 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.js* 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';
```

Commit your code.

Mocking User Interaction

Let's look at how to test user interactions with Enzyme...

When testing components, especially users 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.js`:

```
describe('When not authenticated', () => {
  testData.forEach((el) => {
    const component = <Form
      formType={el.formType}
      formData={el.formData}
      isAuthenticated={false}
    />;
    it(`#${el.formType} Form renders properly`, () => {
      const wrapper = shallow(component);
      const h1 = wrapper.find('h1');
      expect(h1.length).toBe(1);
      expect(h1.get(0).props.children).toBe(el.formType);
      const formGroup = wrapper.find('.form-group');
      expect(formGroup.length).toBe(Object.keys(el.formData).length);
      expect(formGroup.get(0).props.children.props.name).toBe(Object.keys(el.formData)[0]);
      expect(formGroup.get(0).props.children.props.value).toBe('');
    });
    it(`#${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__/App.test.js
  ✓ App renders without crashing (8ms)

PASS  src/components/__tests__/NavBar.test.js
  ✓ NavBar renders properly (8ms)
  ✓ NavBar renders a snapshot properly (14ms)

PASS  src/components/__tests__/UsersList.test.js
  ✓ UsersList renders properly (4ms)
  ✓ UsersList renders a snapshot properly (11ms)

PASS  src/components/__tests__/Form.test.js
when not authenticated
  ✓ Register Form renders properly (11ms)
  ✓ Register Form renders a snapshot properly (3ms)
  ✓ Login Form renders properly (2ms)
  ✓ Login Form renders a snapshot properly (1ms)

PASS  src/components/__tests__/Logout.test.js
  ✓ Logout renders properly (4ms)
  ✓ Logout renders a snapshot properly (4ms)

PASS  src/components/__tests__/AddUser.test.js
  ✓ AddUser renders properly (5ms)
  ✓ AddUser renders a snapshot properly (3ms)

PASS  src/components/__tests__/About.test.js
  ✓ About renders properly (3ms)
  ✓ About renders a snapshot properly (2ms)

Test Suites: 7 passed, 7 total
Tests:       15 passed, 15 total
Snapshots:   7 passed, 7 total
Time:        3.662s, 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 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:        8.226s
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.js`:

```
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);
});

```

```

    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(
        el.formData.username
      );
    });
  });
})

```

```

        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.handleSubmit).toHaveBeenCalledTimes(0);
      expect(el.handleFormChange).toHaveBeenCalledTimes(0);
      input.simulate('change');
      expect(el.handleFormChange).toHaveBeenCalledTimes(1);
      wrapper.find('form').simulate('submit', el.formData)
      expect(el.handleSubmit).toHaveBeenCalledWith(el.formData);
      expect(el.handleSubmit).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.js
✓ NavBar renders properly (5ms)
✓ NavBar renders a snapshot properly (17ms)

PASS  src/components/__tests__/App.test.js
✓ App renders without crashing (8ms)

PASS  src/components/__tests__/Form.test.js
When not authenticated
  ✓ Register Form renders properly (5ms)
  ✓ 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 (2ms)
  ✓ Login redirects properly (1ms)

PASS  src/components/__tests__/Logout.test.js
  ✓ Logout renders properly (3ms)
  ✓ Logout renders a snapshot properly (18ms)

PASS  src/components/__tests__/AddUser.test.js
  ✓ AddUser renders properly (14ms)
  ✓ AddUser renders a snapshot properly (5ms)

PASS  src/components/__tests__/About.test.js
  ✓ About renders properly (3ms)
  ✓ About renders a snapshot properly (3ms)

PASS  src/components/__tests__/UsersList.test.js
  ✓ UsersList renders properly (3ms)
  ✓ UsersList renders a snapshot properly (4ms)

Test Suites: 7 passed, 7 total
Tests:       19 passed, 19 total
Snapshots:   7 passed, 7 total
Time:        4.003s, estimated 10s
Ran all test suites.
```

Commit and push your code.

React Authentication - part 2

Moving right along, let's add some methods to handle a user signing up, logging in, and logging out...

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 endpoint 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` .

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>
  )
};
```

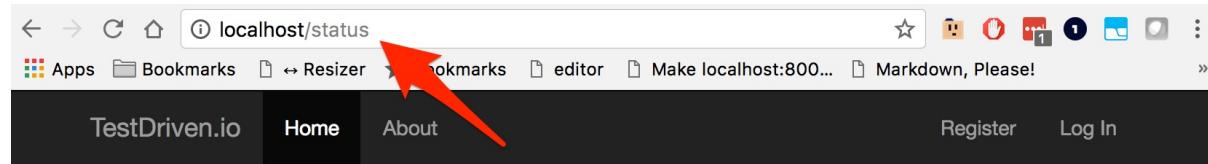
Add the import:

```
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?



You must be logged in to view this. Click [here](#) to log back in.

Name	Status	Type	Initiator	Size	Time	Waterfall	40.00 s
status	304	document	Other	1.9 KB	9.90 s		
bootstrap.min.css	200	stylesheet	status	(from di...)	121 ms		
bundle.js	304	script	status	203 B	121 ms		
status	200	xhr	VM4329:1	370 B	78 ms		
users	200	xhr	VM4329:1	839 B	92 ms		
status	401	xhr	Other	301 B	116 ms		

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-service python manage.py test

$ docker-compose -f docker-compose-dev.yml \
  run users-service 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*:

```

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 6.404s

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@realpython.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@realpython.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 6.864s

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-service python manage.py db migrate

```

Drop the `users_dev` database, and then re-create it:

```

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

```

We have to drop the database here because the existing users do not have the `admin` property. Since the property is required, we could not apply the migration. If you want to maintain the existing users, you could set `nullable` as `True` in the model. Apply the migration, update the existing users, and then create a new migration to set `nullable` as `False`.

Apply the migrations:

```
docker-compose -f docker-compose-dev.yml \
    run users-service 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@realpython.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 6.440s
OK
```

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@realpython.com', 'test')
    add_user('fletcher', 'fletcher@realpython.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@realpython.com', 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@realpython.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 6.721s  
  
OK
```

How about coverage?

Coverage Summary:					
Name	Stmts	Miss	Branch	BrPart	Cover
project/__init__.py	22	10	0	0	55%
project/api/auth.py	72	5	16	3	91%
project/api/models.py	32	18	2	0	47%
project/api/users.py	58	0	14	0	100%
project/api/utils.py	22	1	6	1	93%
project/config.py	23	0	0	0	100%
TOTAL	229	34	38	4	86%

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-service python manage.py test

$ docker-compose -f docker-compose-dev.yml \
  run users-service 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;
```

```
        )
      })
    }
  </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:



All Users

michael

michaelherman

What about the tests?

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

You should see the snapshot test fail:

```
x 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 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:



All Users

User ID	Email	Username	Active	Admin
1	michael@realpython.com	michael	true	false
2	michael@mherman.org	michaelherman	true	false
3	happy@birthday.com	happy@birthday.com	true	false

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('michael@realpython.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': 'michael@realpython.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.

```

Test Suites: 7 passed, 7 total
Tests:       19 passed, 19 total
Snapshots:  7 passed, 7 total
Time:        9.111s

```

Ran all **test** suites.

Commit your 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 to the `users-service` in `docker-compose-prod.yml`:

```
users-service:
  container_name: users-service
  build: https://github.com/realpython/flask-microservices-users.git
  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:
      condition: service_healthy
  links:
    - users-db
  command: gunicorn -b 0.0.0.0:5000 manage:app
```

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 *services/client/Dockerfile-prod*, to install react-scripts:

```
FROM node:latest

# 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

# add environment variables
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

# install and cache app dependencies
ADD package.json /usr/src/app/package.json
RUN npm install --silent
RUN npm install pushstate-server -g --silent
RUN npm install react-scripts@1.0.15 -g --silent

# add app
ADD . /usr/src/app

# build react app
RUN npm run build

# start app
CMD ["pushstate-server", "build", "3000"]
```

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-service python manage.py recreate_db

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

And run the tests:

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

$ docker-compose -f docker-compose-prod.yml \
  run users-service flake8 project

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

Test it in the browser one last time. Commit and push your code.

Part 4

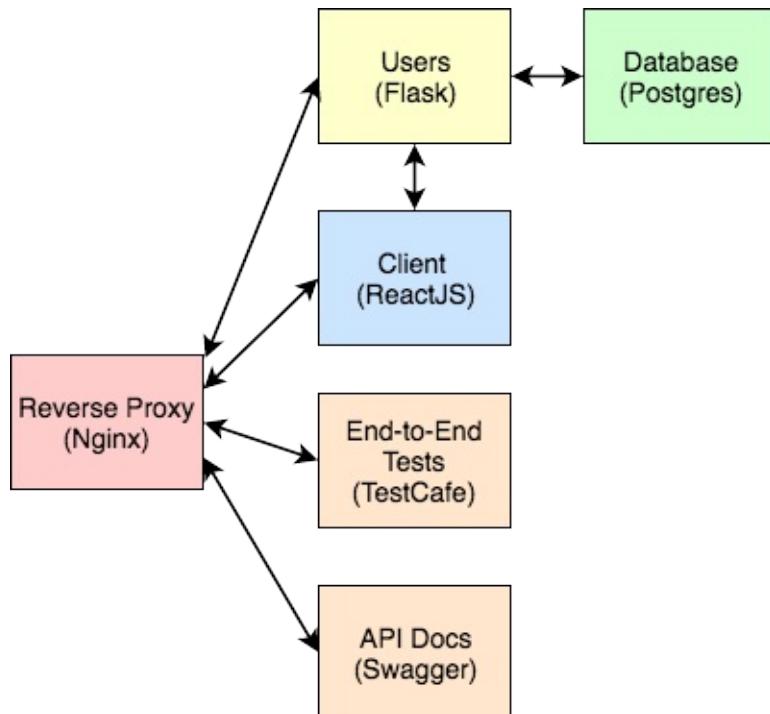
In Part 4, we'll add an *end-to-end* (e2e) testing solution, *form validation* to the React app, a *Swagger* service to document the API, and deal with some tech debt. We'll also set up a staging environment to test on before the app goes into production.

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/realpython/testdriven-app/releases/tag/part4>

Dependencies

You will use the following dependencies in Part 4:

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 v0.31.5
6. React Router Bootstrap v0.24.4

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-service python manage.py test

$ docker-compose -f docker-compose-dev.yml \
  run users-service 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-service python manage.py test
inspect $? users
docker-compose -f $file run users-service 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@0.18.2 --save-dev
```

The `--save-dev` flag adds the dependency info to the `package.json` as a [development dependency](#):

```
{
  "name": "test-driven",
  "devDependencies": {
    "testcafe": "^0.18.2"
  }
}
```

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 61.0.3163 / Mac OS X 10.12.0

/
✓ users should be able to view the '/'

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-service python manage.py test
inspect $? users
docker-compose -f $file run users-service 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: '8'

before_install:
- stty cols 80

dist: trusty
sudo: required

addons:
apt:
sources:
- google-chrome
packages:
- google-chrome-stable

```

```
services:
  - docker

env:
  DOCKER_COMPOSE_VERSION: 1.14.0

before_install:
  - sudo rm /usr/local/bin/docker-compose
  - curl -L https://github.com/docker/compose/releases/download/${DOCKER_COMPOSE_VERSION}/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-dev.yml up --build -d

script:
  - bash test.sh stage

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

Here, we added the Node version along with some basic Chrome settings. Also, we have to use `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 61.0.3163 / 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) Assertion Error: expected false to be truthy

      Browser: Chrome 61.0.3163 / 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 changed 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 61.0.3163 / 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)

```

Want to run a single test or fixture to debug? Use the [only](#) method.

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

```

Since 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__/About.test.js
  ✓ About renders properly (3ms)
  ✓ About renders a snapshot properly (2ms)

PASS  src/components/__tests__/App.test.js
  ✓ App renders without crashing (5ms)
  ✓ App will call componentWillMount when mounted (22ms)

PASS  src/components/__tests__/Form.test.js
  When not authenticated
    ✓ Register Form renders properly (4ms)
    ✓ Register Form submits the form properly (6ms)
    ✓ 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 (2ms)
  When authenticated
    ✓ Register redirects properly (1ms)
    ✓ Login redirects properly (1ms)

PASS  src/components/__tests__/Logout.test.js
  ✓ Logout renders properly (2ms)
  ✓ Logout renders a snapshot properly (4ms)

PASS  src/components/__tests__/NavBar.test.js
  ✓ NavBar renders properly (8ms)
  ✓ NavBar renders a snapshot properly (10ms)

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

PASS  src/components/__tests__/AddUser.test.js
  ✓ AddUser renders properly (10ms)
  ✓ AddUser renders a snapshot properly (3ms)
```

```
Test Suites: 7 passed, 7 total
Tests:       20 passed, 20 total
Snapshots:   7 passed, 7 total
Time:        3.384s, estimated 9s
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>
        </form>
      </div>
    );
  }
}
```

```
        />
      </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:

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

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

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

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>
    </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.

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 }); })
  }
}

export default App;
```

```
.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 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` to 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:       6.67s, estimated 9s
Ran all test suites.

```

Fortunately, we can fix this by making to small changes to the form tests in `services/client/src/components/__tests__/Form.test.js..`

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.formData);
  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:        5.277s
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 since the input values are on the state.

Let's test-drive the updates!

Validation 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` 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 of the `forEach` `services/client/src/components/_tests_/Form.test.js`:

```
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
```

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.js*.

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.js`:

```
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__/Form.test.js
  When not authenticated
    ✓ register Form renders properly (5ms)
    ✓ register Form should be disabled by default (5ms)
    ✓ register Form submits the form properly (4ms)
    ✓ register Form renders a snapshot properly (3ms)
    ✓ login Form renders properly (3ms)
    ✓ login Form should be disabled by default (2ms)
    ✓ login Form submits the form properly (2ms)
    ✓ login Form renders a snapshot properly (2ms)
  When authenticated
    ✓ register redirects properly (1ms)
    ✓ login redirects properly (1ms)

PASS  src/components/__tests__/App.test.js
  ✓ App renders without crashing (6ms)
  ✓ App will call componentWillMount when mounted (29ms)

PASS  src/components/__tests__/NavBar.test.js
  ✓ NavBar renders properly (4ms)
  ✓ NavBar renders a snapshot properly (10ms)

PASS  src/components/__tests__/UsersList.test.js
  ✓ UsersList renders properly (30ms)
  ✓ UsersList renders a snapshot properly (5ms)

PASS  src/components/__tests__/FormErrors.test.js
  ✓ FormErrors (with register form) renders properly (6ms)
  ✓ FormErrors (with register form) renders a snapshot properly (4ms)
  ✓ FormErrors (with login form) renders properly (2ms)
  ✓ FormErrors (with login form) renders a snapshot properly (2ms)

PASS  src/components/__tests__/About.test.js

```

```

✓ About renders properly (4ms)
✓ About renders a snapshot properly (4ms)

PASS  src/components/__tests__/Logout.test.js
  ✓ Logout renders properly (3ms)
  ✓ Logout renders a snapshot properly (3ms)

PASS  src/components/__tests__/AddUser.test.js
  ✓ AddUser renders properly (7ms)
  ✓ AddUser renders a snapshot properly (2ms)

Snapshot Summary
> 2 snapshots updated in 1 test suite.

Test Suites: 8 passed, 8 total
Tests:       26 passed, 26 total
Snapshots:   2 updated, 7 passed, 9 total
Time:        6.727s, estimated 10s
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"]', 'greaterthanen')
    .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"]', 'greaterthanen')
    .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 = 'greaterthanen';
```

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.

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>&nbsp;{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.



All Users

User ID	Email	Username	Active	Admin
1	michael@realpython.com	michael	true	false
2	michael@mherman.org	michaelherman	true	false

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;
```

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>&nbsp;{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 Test

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.js`:

```

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();
  });
});
```

Ensure the tests pass:

```
PASS  src/components/__tests__/UsersList.test.js
  ✓ UsersList renders properly (17ms)
  ✓ UsersList renders a snapshot properly (7ms)

PASS  src/components/__tests__/App.test.js
  ✓ App renders without crashing (12ms)
  ✓ App will call componentWillMount when mounted (37ms)

PASS  src/components/__tests__/NavBar.test.js
  ✓ NavBar renders properly (27ms)
  ✓ NavBar renders a snapshot properly (20ms)

PASS  src/components/__tests__/FormErrors.test.js
  ✓ FormErrors (with register form) renders properly (7ms)
  ✓ FormErrors (with register form) renders a snapshot properly (6ms)
  ✓ FormErrors (with login form) renders properly (2ms)
  ✓ FormErrors (with login form) renders a snapshot properly (7ms)

PASS  src/components/__tests__/Form.test.js
  When not authenticated
    ✓ register Form renders properly (8ms)
    ✓ register Form should be disabled by default (3ms)
    ✓ register Form submits the form properly (4ms)
    ✓ register Form renders a snapshot properly (9ms)
    ✓ login Form renders properly (4ms)
    ✓ login Form should be disabled by default (2ms)
    ✓ login Form submits the form properly (4ms)
    ✓ login Form renders a snapshot properly (4ms)
  When authenticated
    ✓ register redirects properly (2ms)
    ✓ login redirects properly (1ms)

PASS  src/components/__tests__/Message.js
  When given a success message
    ✓ Message renders properly (9ms)
    ✓ Message renders a snapshot properly (6ms)
  When given a danger message
    ✓ Message renders properly (5ms)
    ✓ Message renders a snapshot properly (3ms)

PASS  src/components/__tests__/Logout.test.js
  ✓ Logout renders properly (5ms)
  ✓ Logout renders a snapshot properly (7ms)

PASS  src/components/__tests__/AddUser.test.js
  ✓ AddUser renders properly (8ms)
  ✓ AddUser renders a snapshot properly (7ms)
```

```
PASS  src/components/__tests__/About.test.js
  ✓ About renders properly (4ms)
  ✓ About renders a snapshot properly (4ms)

Test Suites: 9 passed, 9 total
Tests:       30 passed, 30 total
Snapshots:   11 passed, 11 total
Time:        9.753s
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:1.13.5

ENV SWAGGER_UI_VERSION 3.4.5
ENV URL **None**

RUN apt-get update \
    && apt-get install -y curl \
    && curl -L https://github.com/swagger-api/swagger-ui/archive/v${SWAGGER_UI_VERSION}.tar.gz | tar -z xv -C /tmp \
    && cp -R /tmp/swagger-ui-${SWAGGER_UI_VERSION}/dist/* /usr/share/nginx/html \
    && rm -rf /tmp/*

RUN rm /etc/nginx/conf.d/default.conf
ADD /nginx.conf /etc/nginx/conf.d

ADD swagger.json /usr/share/nginx/html/swagger.json
ADD start.sh /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/nginx/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-service

```

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/flask.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-service 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-service
```

Update the container. Test it out in the browser.

The screenshot shows the Swagger UI interface. At the top, there's a green header bar with the 'swagger' logo, a 'swagger.json' input field containing 'swagger.json', and a 'Explore' button. Below the header, the main content area has a title 'Users Service' with version '0.0.1' and 'OAS3' badge. It says 'Swagger spec for documenting the users service'. A 'Servers' dropdown menu is open, showing 'http://localhost' as the selected option. At the bottom of the main content area, a message box displays 'No operations defined in spec!'

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"
      }
    }
}
```

```
        "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.

The screenshot shows the Swagger UI interface for a service named "Users Service". At the top, there's a green header bar with the "swagger" logo, a "swagger.json" link, and an "Explore" button. Below the header, the title "Users Service" is displayed with version "0.0.1" and "OAS3" badges. A "swagger.json" link is also present. The main content area is titled "default" and contains a list of API endpoints:

- GET /users/ping** Just a sanity check
- GET /users** Returns all users
- GET /users/{id}** Returns a user based on a single user ID
- POST /auth/register** Creates a new user
- POST /auth/login** Logs a user in

A "Servers" dropdown at the top left is set to "http://localhost".

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"
    }
  }
}
}
```

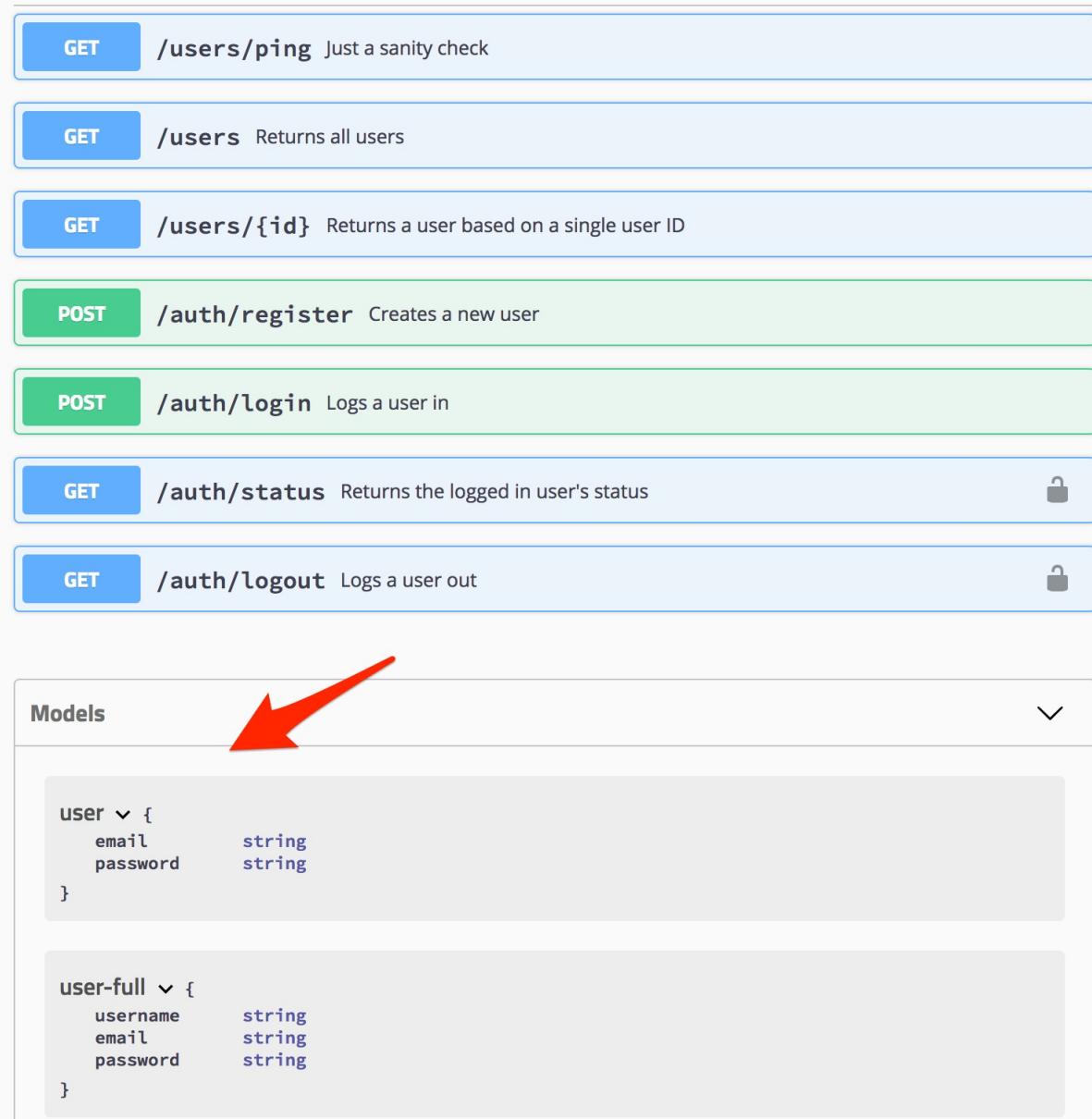
Now, turn back to the `/auth/login` route and update the `schema` like so:

```
"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.



The screenshot shows a Swagger UI interface with the following sections:

- API Routes:**
 - GET /users/ping** Just a sanity check
 - GET /users** Returns all users
 - GET /users/{id}** Returns a user based on a single user ID
 - POST /auth/register** Creates a new user
 - POST /auth/login** Logs a user in
 - GET /auth/status** Returns the logged in user's status (with a lock icon)
 - GET /auth/logout** Logs a user out (with a lock icon)
- Models:**
 - user**:
 - email string
 - password string
 - user-full**:
 - username string
 - email string
 - password string

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": {
      "type": "object",
      "properties": {
        "email": {
          "type": "string"
        },
        "password": {
          "type": "string"
        }
      }
    }
  }
}
```

```
  "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.

Servers **http://localhost** ▾

default ▾

GET	/users/ping Just a sanity check
GET	/users Returns all users
GET	/users/{id} Returns a user based on a single user ID
POST	/auth/register Creates a new user
POST	/auth/login Logs a user in
GET	/auth/status Returns the logged in user's status
GET	/auth/logout Logs a user out



To test, first log a user in and grab the provided token.

Responses

Curl

```
curl -X POST "http://localhost/auth/login" -H "accept: */*" -H "Content-Type: application/json" -d "{\"email\":\"happy@birthday.com\", \"password\":\"happy@birthday.com\"}"
```

Request URL

```
http://localhost/auth/login
```

Server response

Code	Details
200	Response body <pre>{ "auth_token": "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJleHAiOiE1MTM3MjE5NDEs-ImlhDCI6MTUxMTEyOTk0MSwic3ViIjo2fQ.3D3vWXpT-6TgXxGYokIX4MmLWQyB3QSxfXhlsrei-6M", "message": "Successfully logged in.", "status": "success" }</pre>

Response headers

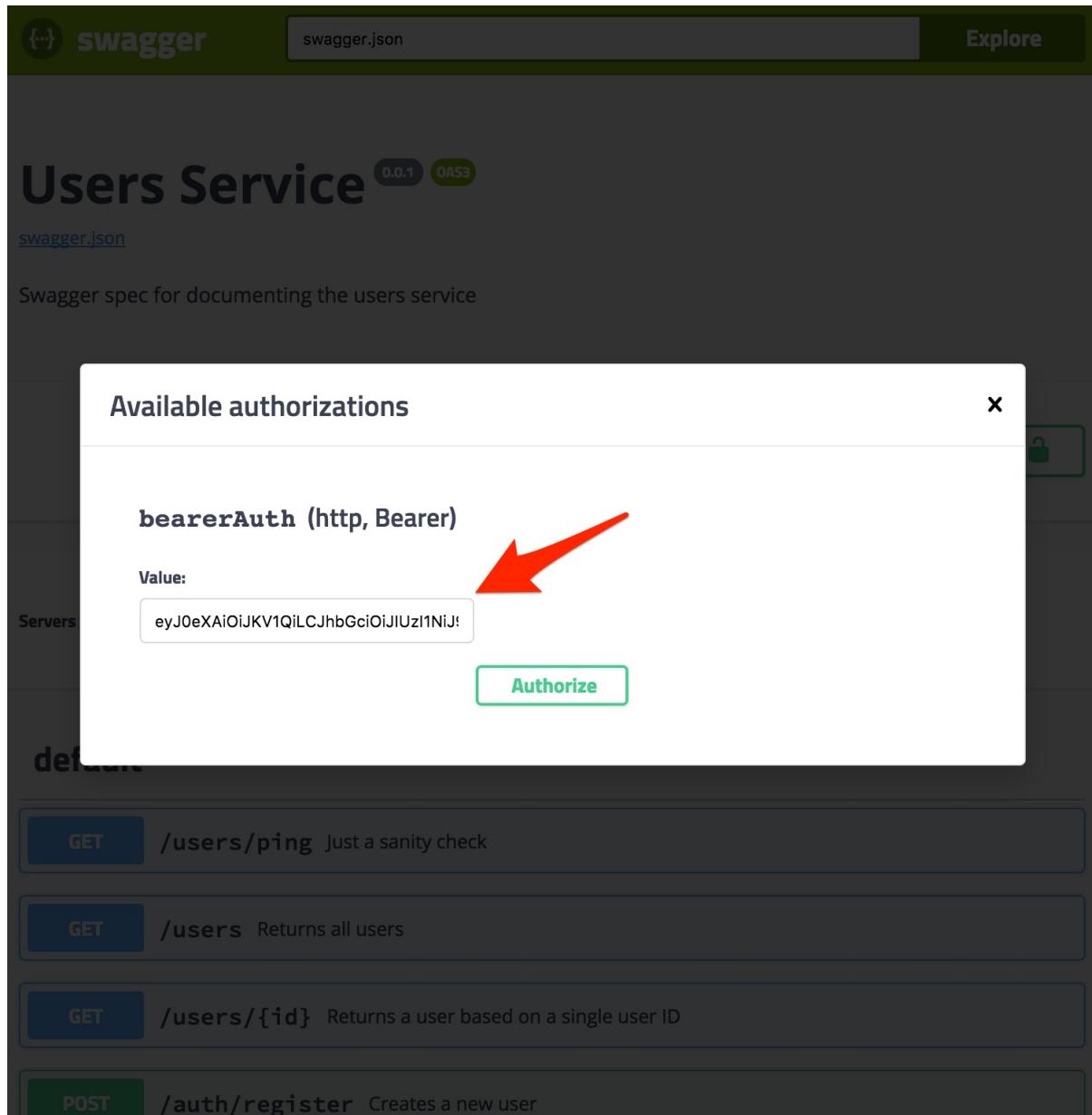
```
access-control-allow-origin: http://localhost
connection: keep-alive
content-length: 227
content-type: application/json
date: Sun, 19 Nov 2017 22:19:01 GMT
server: nginx/1.13.5
vary: Origin
```

Responses

Code	Description	Links
200	<i>Successfully logged in</i>	No links



Then click the "Authorize" button at the top of the page and add the token to the input box.



The screenshot shows the Swagger UI interface for a 'Users Service'. At the top, there's a navigation bar with 'swagger.json', '0.0.1', 'OAS3', and 'Explore' buttons. Below the navigation, the title 'Users Service' is displayed along with its version '0.0.1' and specification 'OAS3'. A link to 'swagger.json' is also present. The main content area is titled 'Available authorizations' and contains a section for 'bearerAuth (http, Bearer)'. It shows a 'Value:' input field containing a JWT token: 'eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9'. A large red arrow points from the left towards this input field. Below the input field is a green 'Authorize' button. In the background, there are several API endpoint definitions listed under 'definitions':

- GET /users/ping**: Just a sanity check.
- GET /users**: Returns all users.
- GET /users/{id}**: Returns a user based on a single user ID.
- POST /auth/register**: Creates a new user.

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.

The end-to-end tests will fail on Travis. Why? Turn to the `.travis.yml` file. We're using `docker-compose-prod.yml` to build and spin up the services, which does not include Swagger. Meanwhile, the `services/nginx/flask.conf` references the Swagger service. Since it does not exist, the service will fail and the reverse proxy will not be set up correctly. We'll correct this in an upcoming lesson.

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. 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: '3.3'

services:

  users-service:
    container_name: users-service
    build:
      context: ./services/users
      dockerfile: Dockerfile-stage
    expose:
      - '5000'
    environment:
      - SECRET_KEY=my_precious
      - 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
    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

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

client:
  container_name: client
  build:
    context: ./services/client
    dockerfile: Dockerfile-stage
    args:
      - NODE_ENV=development
      - REACT_APP_USERS_SERVICE_URL=${REACT_APP_USERS_SERVICE_URL}
  ports:
    - '3007:3000'
  depends_on:
    - users-service
  links:
    - users-service

swagger:
  container_name: swagger
  build:
    context: ./services/swagger
    dockerfile: Dockerfile-stage
  ports:
    - '3008:8080'
  environment:
    - URL=swagger.json
  depends_on:
    - users-service

```

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"""
    DEBUG = False
    SQLALCHEMY_DATABASE_URI = os.environ.get('DATABASE_URL')
```

Dockerfiles

Add three new *Dockerfiles*:

1. `services/users/Dockerfile-stage`
2. `services/client/Dockerfile-stage`
3. `services/swagger/Dockerfile-stage`

services/users/Dockerfile-stage

```
FROM python:3.6.3

# 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
ADD ./requirements.txt /usr/src/app/requirements.txt

# install requirements
RUN pip install -r requirements.txt

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

# add app
ADD . /usr/src/app

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

services/client/Dockerfile-stage

```

FROM node:latest

# 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

# add environment variables
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

# install and cache app dependencies
ADD package.json /usr/src/app/package.json
RUN npm install --silent
RUN npm install pushstate-server -g --silent
RUN npm install react-scripts@1.0.15 -g --silent

# add app
ADD . /usr/src/app

# build react app
RUN npm run build

# start app
CMD ["pushstate-server", "build", "3000"]

```

services/swagger/Dockerfile-stage

```

FROM nginx:1.13.5

ENV SWAGGER_UI_VERSION 3.4.5
ENV URL **None**

RUN apt-get update \
    && apt-get install -y curl \
    && curl -L https://github.com/swagger-api/swagger-ui/archive/v${SWAGGER_UI_VERSION}.tar.gz | tar -z xv -C /tmp \
    && cp -R /tmp/swagger-ui-${SWAGGER_UI_VERSION}/dist/* /usr/share/nginx/html \
    && rm -rf /tmp/*

RUN rm /etc/nginx/conf.d/default.conf
ADD /nginx.conf /etc/nginx/conf.d

ADD ./swagger.json /usr/share/nginx/html/swagger.json
ADD start.sh /start.sh

```

```
CMD ["/start.sh"]
```

What's the difference between `services/swagger/Dockerfile-stage` and `services/swagger/Dockerfile-dev` - and why?

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
```

Does this command hang at either the `users-service` or `client`? You may be out of space. Try adding a `.dockerignore` to both "services/users" and "services/client" to ignore `env` and `node_modules`, respectively.

Create and seed the database and run the following tests:

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

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

$ docker-compose -f docker-compose-stage.yml \
  run users-service python manage.py test

$ docker-compose -f docker-compose-stage.yml \
  run users-service flake8 project

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

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.

The screenshot shows a web browser window with the URL `34.201.217.245/swagger` in the address bar. The page title is "swagger". A red arrow points to the "Resizer" icon in the browser's toolbar. The main content area displays the "Users Service" documentation, version 0.0.1 (OAS3). It includes a link to `swagger.json` and a description: "Swagger spec for documenting the users service". On the right, there is an "Authorize" button with a lock icon. Below the main content, a dropdown menu labeled "Servers" shows the selected URL `http://34.201.217.245`. A large red arrow points to this URL field. The bottom section shows a "default" dropdown and a list of API endpoints, including a GET method for `/auth/logout` which logs a user out.

← → C ⌘ ⌘ 34.201.217.245/swagger

Bookmarks Bookmarks editor Make localhost:800... Markdown, Please! »

swagger.json Explore

Users Service 0.0.1 OAS3

[swagger.json](#)

Swagger spec for documenting the users service

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*

```
swagger:  
  container_name: swagger  
  build:  
    context: ./services/swagger  
    dockerfile: Dockerfile-prod  
  ports:  
    - '3008:8080'  
  environment:  
    - URL=swagger.json  
  depends_on:  
    - users-service
```

Dockerfile

Create a new file called *services/swagger/Dockerfile-prod*:

```
FROM nginx:1.13.5  
  
ENV SWAGGER_UI_VERSION 3.4.5  
ENV URL **None**  
  
RUN apt-get update \  
  && apt-get install -y curl \  
  && curl -L https://github.com/swagger-api/swagger-ui/archive/v${SWAGGER_UI_VERS  
ION}.tar.gz | tar -z xv -C /tmp \  
  && cp -R /tmp/swagger-ui-${SWAGGER_UI_VERSION}/dist/* /usr/share/nginx/html \  
  && rm -rf /tmp/*
```

```
RUN rm /etc/nginx/conf.d/default.conf  
ADD /nginx.conf /etc/nginx/conf.d  
  
ADD ./swagger.json /usr/share/nginx/html/swagger.json  
ADD start.sh /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-service python manage.py recreate_db
```

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

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

$ docker-compose -f docker-compose-prod.yml \
  run users-service flake8 project

$ docker-compose -f docker-compose-prod.yml \
  run client npm test -- --verbose

$ testcafe chrome e2e
```

Make sure the Swagger docs are up and running. Commit and push your code! The build should now pass.

The screenshot shows a web browser displaying the Swagger UI for a 'Users Service'. The address bar shows the URL `34.204.79.179/swagger`. A red arrow points from the address bar to the 'swagger.json' link in the top navigation bar. Another red arrow points from the bottom left to the 'Servers' dropdown menu, which is set to `http://34.204.79.179`. The main content area displays the 'Users Service' documentation, including the version `0.0.1` and `OAS3`. The 'default' section lists a single endpoint: `GET /auth/logout`, which is described as 'Logs a user out'. There is also an 'Authorize' button with a lock icon.

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

Run the unit and integration tests:

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

Lint:

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

Create and seed the database:

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

Run the unit and integration tests:

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

Lint:

```
$ docker-compose -f docker-compose-dev.yml \
  run users-service 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?

NAME	ACTIVE	DRIVER	STATE	URL	SWARM	DOCKER
aws	*	amazonec2	Running	tcp://34.207.173.181:2376		v17.05.0

-ce						Unknown
dev	-	virtualbox	Saved			

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)
```

Part 5

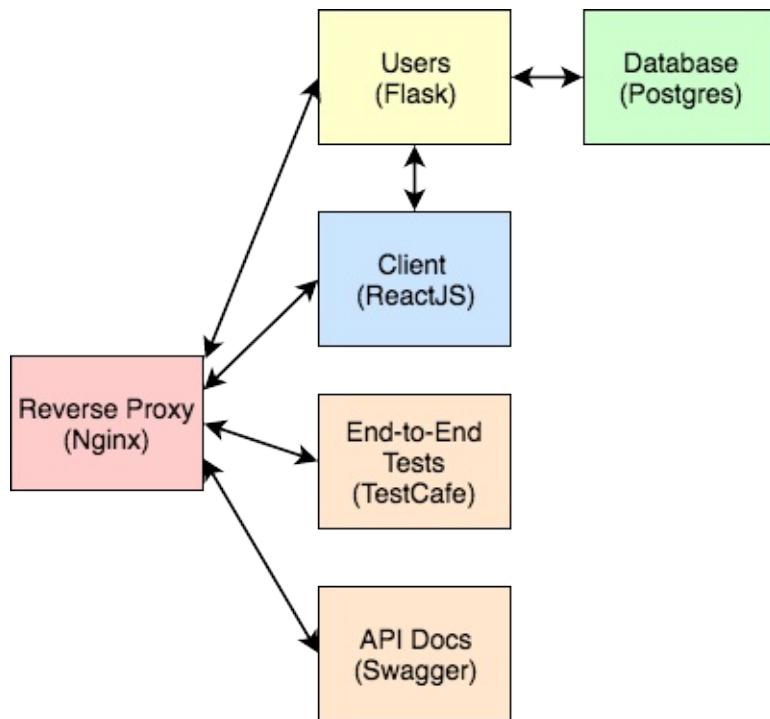
In Part 5, we'll dive into *container orchestration* with [Amazon ECS](#) as we move our staging and production environments to a more scaleable infrastructure. We'll also add [Amazon EC2 Container Registry](#) along with Amazon's [Elastic Load Balancing](#) for *load balancing* and Amazon's [Relational 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 EC2 Container Service (ECS) over other orchestration tools like Kubernetes, Mesos, and Docker Swarm
3. Configure an Application Load Balancer (ALB) along with ECS to run a set of microservices
4. Integrate Amazon EC2 Container Registry (ECR) into the deployment process
5. Send container logs to CloudWatch
6. Update a running container via a zero-downtime deployment strategy to not disrupt the current users or your application
7. Explain the types of scaling that are available to you
8. 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/realpython/testdriven-app/releases/tag/part5>

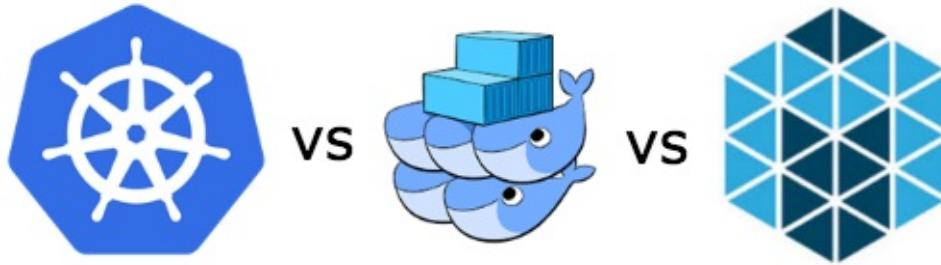
Dependencies

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.

EC2 Container Registry

In this lesson, we'll add the EC2 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.

IAM

Although not required, it's a good idea to set up a new IAM User and Role specifically for container instances. For more, review the following [guide](#) from Amazon.

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 [EC2 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 [EC2 Container Service](#) (which we'll be setting up shortly)

Navigate to [Amazon ECS](#), click "Repositories", and then add five 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.

The screenshot shows the AWS ECR (Amazon Elastic Container Registry) interface. In the left sidebar, 'Repositories' is selected. The main area is titled 'Repositories' and contains a table with the following data:

Repository name	Repository URI	Created at
test-driven-users	046505967931.dkr.ecr.us-east-1.amazonaws.com/test-driven-users	2017-11-22 06:43:45 -0700
test-driven-swagger	046505967931.dkr.ecr.us-east-1.amazonaws.com/test-driven-swagger	2017-11-22 06:54:07 -0700
test-driven-client	046505967931.dkr.ecr.us-east-1.amazonaws.com/test-driven-client	2017-11-22 06:53:54 -0700
test-driven-users_db	046505967931.dkr.ecr.us-east-1.amazonaws.com/test-driven-users_db	2017-11-22 06:50:27 -0700

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 $(aws ecr get-login --region us-east-1)
        export TAG=$TRAVIS_BRANCH
        export REPO=$AWS_ACCOUNT_ID.dkr.ecr.us-east-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
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

realpython / testdriven-app



build passing

[Current](#) [Branches](#) [Build History](#) [Pull Requests](#) [Settings](#) [More options](#)
General

- | | |
|---|---|
| <input type="button" value="OFF"/> Build only if .travis.yml is present | <input checked="" type="button" value="ON"/> Build branch updates |
| <input type="button" value="OFF"/> Limit concurrent jobs <small>(?)</small> | <input checked="" type="button" value="ON"/> Build pull request updates |

Auto Cancellation

Auto Cancellation allows you to only run builds for the latest commits in the queue. This setting can be applied to builds for Branch builds and Pull Request builds separately. Builds will only be canceled if they are waiting to run, allowing for any running jobs to finish.

- | | |
|--|--|
| <input checked="" type="button" value="ON"/> Auto cancel branch builds | <input checked="" type="button" value="ON"/> Auto cancel pull request builds |
|--|--|

Environment Variables

Notice that the values are not escaped when your builds are executed. Special characters (for bash) should be escaped accordingly.

AWS_ACCESS_KEY_ID <input type="text"/>	
AWS_ACCOUNT_ID <input type="text"/>	
AWS_SECRET_ACCESS_KEY <input type="text"/>	

Update `.travis.yml`, adding in the necessary environment variables and an `after_success` step:

```

language: node_js
node_js: '8'

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=1.14.0
    - COMMIT=${TRAVIS_COMMIT::8}
    - MAIN_REPO=https://github.com/realpython/testdriven-app.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_VERSION}/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.

Ran 41 tests in 1.391s

OK

Starting users-db ...

- Chrome 62.0.3202 / Linux 0.0.0

/

✓ should display the page correctly if a user is not logged in

1 passed (2s)

Tests passed!

The command "bash test.sh stage" exited with 0.

```
$ bash ./docker_push.sh
$ docker-compose -f docker-compose-prod.yml down
```

after_success 291.71s
after_script 21.85s

Done. Your build exited with 0.

AWS Management Console Screenshot:

Services > Resource Groups > All repositories : test-driven-client

Repository ARN: arn:aws:ecr:us-east-1:046505967931:repository/test-driven-client

Repository URI: 046505967931.dkr.ecr.us-east-1.amazonaws.com/test-driven-client

View Push Commands

Images Permissions Dry run of lifecycle rules Lifecycle policy

Amazon ECR limits the number of images to 1,000 per repository. Request a limit increase.

Image sizes may appear compressed. Learn more

Last updated on November 23, 2017 6:01:07 AM (0m ago)

Image tags	Digest	Size (MiB)	Pushed at
staging	view all sha256:e5968f1843dc853848b168f1f8167506497a60f2122f194a113eb6be6662a9b9	364.12	2017-11-22 14:12:38 -0700
	sha256:2a9ef990a183059a37c2244bee6acf5ef4a56bc03b5d8868541bdea835b47b4	364.05	2017-11-22 13:23:41 -0700

Filter In this page Tag Status: All

< 1-2 > Page size: 100

View Details

© 2006 - 2017, Amazon Web Services, Inc. or its affiliates. All rights reserved. Privacy Policy Terms of Use

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: '8'

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=1.14.0
    - COMMIT=${TRAVIS_COMMIT::8}
    - MAIN_REPO=https://github.com/realpython/testdriven-app.git
    - USERS=test-driven-users
    - USERS_REPO=${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_VERSION}/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 $(aws ecr get-login --region us-east-1)
        export TAG=$TRAVIS_BRANCH
        export REPO=$AWS_ACCOUNT_ID.dkr.ecr.us-east-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 $SWAGGER_DIR
        docker tag $SWAGGER:$COMMIT $REPO/$SWAGGER:$TAG
    fi
fi

```

```
fi
```



Sanity Check

Assuming you're own 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

The screenshot shows the AWS ECR console with the repository `test-driven-client`. The `Images` tab is selected. The table lists several image tags, with `staging` highlighted by a red arrow. The table columns include `Image tags`, `Digest`, `Size (MiB)`, and `Pushed at`.

Image tags	Digest	Size (MiB)	Pushed at
<code>production</code>	sha256:ad198ba257b66de6941e94d69a51386fc2b2f1530a750301907...	364.25	2017-11-23 14:32:38 -0700
<code>staging</code>	sha256:e902380dcfd4bd12a3046557b5b1cad7086319660c2be1bf1...	364.25	2017-11-23 13:39:29 -0700
	sha256:c13ee8711d6c7882b36091f704d93a9ef4835a03476358a78d0...	364.29	2017-11-23 08:22:31 -0700
	sha256:b01d8b76bb960404c1dea97465f256f75e439e24aa8b9d8d26...	364.25	2017-11-23 07:44:48 -0700
	sha256:8585f5aad50041e7bb9719c67ad67df7cd837ddc0c8148669...	364.19	2017-11-23 07:07:54 -0700
	sha256:e5968f1843dc853848b168ff18167506497a60f2122f194a113e...	364.12	2017-11-22 14:12:38 -0700
	sha256:2a9ef990a183059a37c2244bee6acf5e4a56bc0c3b5d8868541...	364.05	2017-11-22 13:23:41 -0700

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 to 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 applications 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

The screenshot shows the AWS Step 1: Configure Load Balancer interface. At the top, there are tabs for 1. Configure Load Balancer (which is active), 2. Configure Security Settings, 3. Configure Security Groups, 4. Configure Routing, 5. Register Targets, and 6. Review. Below the tabs, the title is "Step 1: Configure Load Balancer" and the sub-section is "Listeners". A note says "A listener is a process that checks for connection requests, using the protocol and port that you configured." Under "Load Balancer Protocol", "HTTP" is selected. Under "Load Balancer Port", "80" is entered. There is a "Add listener" button. The "Availability Zones" section has a note: "Specify the Availability Zones to enable for your load balancer. The load balancer routes traffic to the targets in these Availability Zones only. You can specify only one subnet per Availability Zone. You must specify subnets from at least two Availability Zones to increase the availability of your load balancer." Two subnets are selected: "us-east-1a" and "us-east-1b". A red arrow points to the "us-east-1b" selection. At the bottom right, there are "Cancel" and "Next: Configure Security Settings" buttons.

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 3: Configure Security Groups

A security group is a set of firewall rules that control the traffic to your load balancer. On this page, you can add rules to allow specific traffic to reach your load balancer. First, decide whether to create a new security group or select an existing one.

Assign a security group:

- Create a new security group
- Select an existing security group

Security group name: `testdriven-security-group`

Description: Security Group for Testdriven

Type	Protocol	Port Range	Source
HTTP	TCP	80	Anywhere 0.0.0.0/0
SSH	TCP	22	Anywhere 0.0.0.0/0

Add Rule

Cancel Previous Next: Configure Routing

Step 4: Configure Routing

1. "Name": `testdriven-client-stage-tg`
2. "Port": `3000`
3. "Path": `/`

Step 4: Configure Routing

Your load balancer routes requests to the targets in this target group using the protocol and port that you specify, and performs health checks on the targets using these health check settings. Note that each target group can be associated with only one load balancer.

Target group

Target group: `New target group`

Name: `testdriven-client-stage-tg`

Protocol: `HTTP`

Port: `3000`

Target type: `instance`

Health checks

Protocol: `HTTP`

Path: `/`

Advanced health check settings

Cancel Previous Next: Register Targets

Step 5: Register Targets

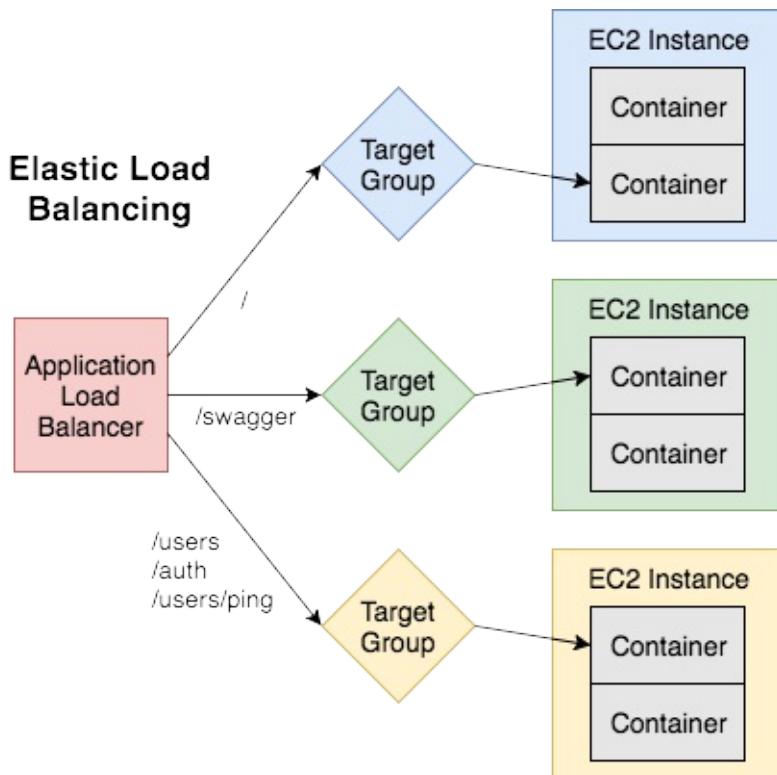
Do not assign any instances manually since this will be managed by ECS. Review and then create the new load balancer.

Once created, take note of the new Security Group:

The screenshot shows the AWS Elastic Load Balancer console. A new load balancer named 'testdriven-staging-alb' has been created. Key configuration details highlighted with red arrows include:

- VPC ID:** vpc-46e1103f (highlighted in blue)
- IP address type:** ipv4 (highlighted in blue)
- Availability Zones:** subnet-80c0e8e5 - us-east-1b, subnet-dcf4fb186 - us-east-1a (highlighted in blue)
- Security groups:** sg-a9fa4ddc, testdriven-security-group (highlighted in blue)

With that, we also need to set up [Target Groups](#) and [Listeners](#):



Target Groups

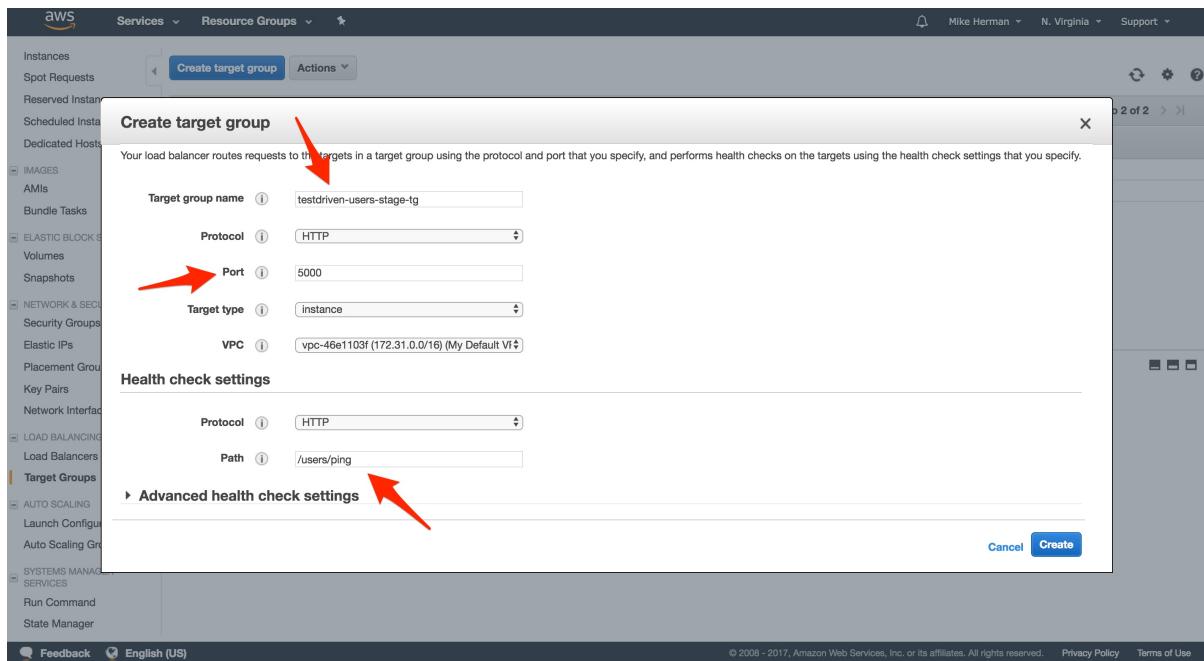
Target Groups are attached to the Application Load Balancer and are used to route traffic to the containers found in the ECS Service.

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...

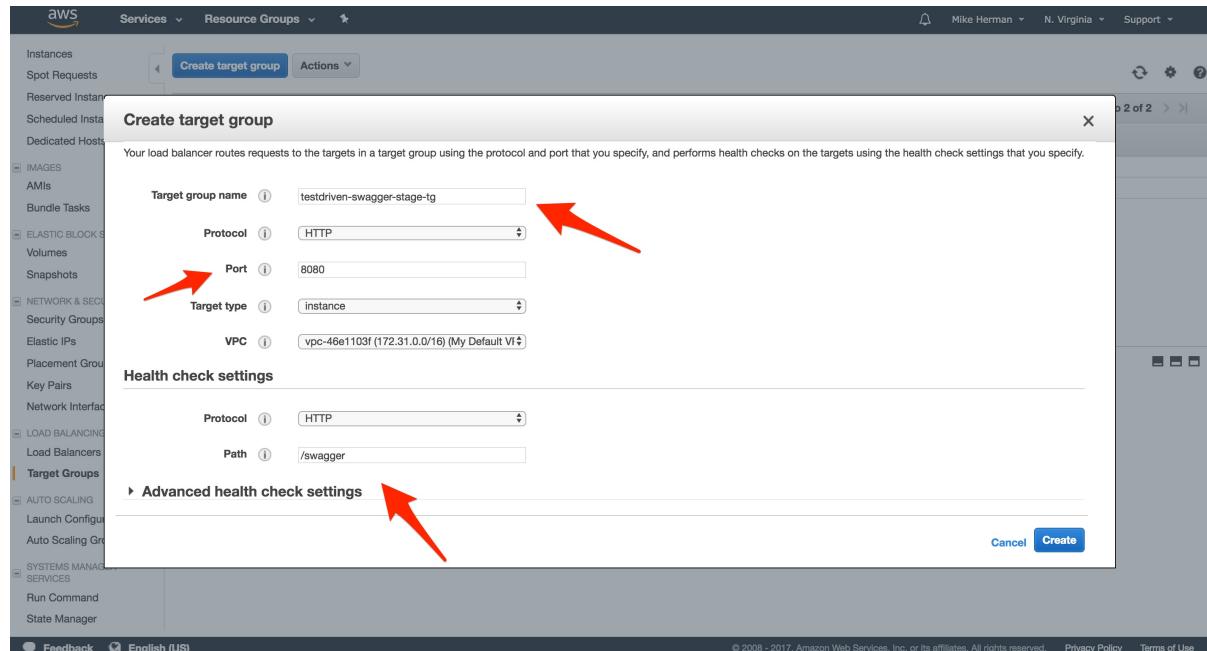
Target Group 1: `users`

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:

Name	Port	Protocol	Target type	VPC ID
testdriven-client-stage-tg	3000	HTTP	instance	vpc-46e1103f
testdriven-swagger-stage-tg	8080	HTTP	instance	vpc-46e1103f
testdriven-users-stage-tg	5000	HTTP	instance	vpc-46e1103f

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

Rules for: testdriven-staging-alb HTTP 80

Index	ARN	IF	THEN
1		✓ Path is /swagger*	Forward to testdriven-swagger-stage-tg
2		✓ Path is /auth*	Forward to testdriven-users-stage-tg
3		✓ Path is /users*	Forward to testdriven-users-stage-tg
4		✓ Path is /users/ping	Forward to testdriven-users-stage-tg
last	HTTP 80: default action <small>This rule cannot be moved or deleted</small>	✓ Requests otherwise not routed	Forward to testdriven-client-stage-tg

Update Travis

Finally, navigate back to the Load Balancer and grab the "DNS name" from the "Description" tab:

Create Load Balancer Actions

Name	DNS name	State	VPC ID	Availability Zones	Type	Created At
testdriven-staging-alb	testdriven-staging-alb-137894177.us-east-1.elb.amazonaws.com (A Record)	active	vpc-46e1103f	us-east-1b, us-east-1a	application	November 24, 2017 at 7:06:18 AM UTC-7

Load balancer: testdriven-staging-alb

Description Listeners Monitoring Tags

Basic Configuration

Name:	testdriven-staging-alb
ARN:	arn:aws:elasticloadbalancing:us-east-1:046505967931:loadbalancer/app/testdriven-staging-alb/9193f3107284bfab
DNS name:	testdriven-staging-alb-137894177.us-east-1.elb.amazonaws.com (A Record)
Scheme:	internet-facing
Type:	application
Availability Zones:	subnet-80c0e8e5 - us-east-1b, subnet-dcf1fb186 - us-east-1a

Security

Security groups:	sg-a9fa4ddc, testdriven-security-group
------------------	--

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 $(aws ecr get-login --region us-east-1)
        export TAG=$TRAVIS_BRANCH
        export REPO=$AWS_ACCOUNT_ID.dkr.ecr.us-east-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
    fi
fi
```

```
# 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 $SWAGGER_DIR
docker tag $SWAGGER:$COMMIT $REPO/$SWAGGER:$TAG
fi
fi
```

With that, we can turn our attention to ECS...

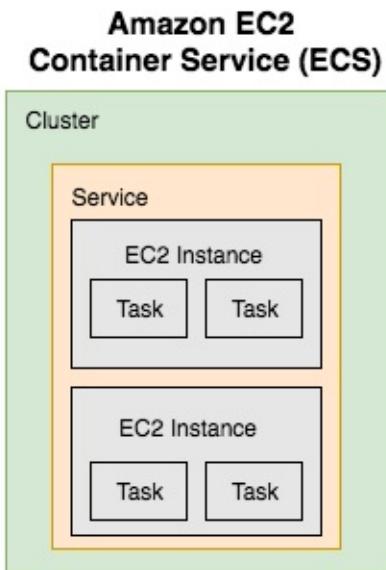
EC2 Container Service

Let's configure a Task Definition along with a Cluster and a Service within EC2 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-east-1.amazonaws.com/test-driven-client:staging
3. "Memory Limits (MB)": 300 soft limit
4. "Port mappings": 0 host, 3000 container

We set the host port for the service to 0 so that a port is dynamically assigned when the Task is spun up.

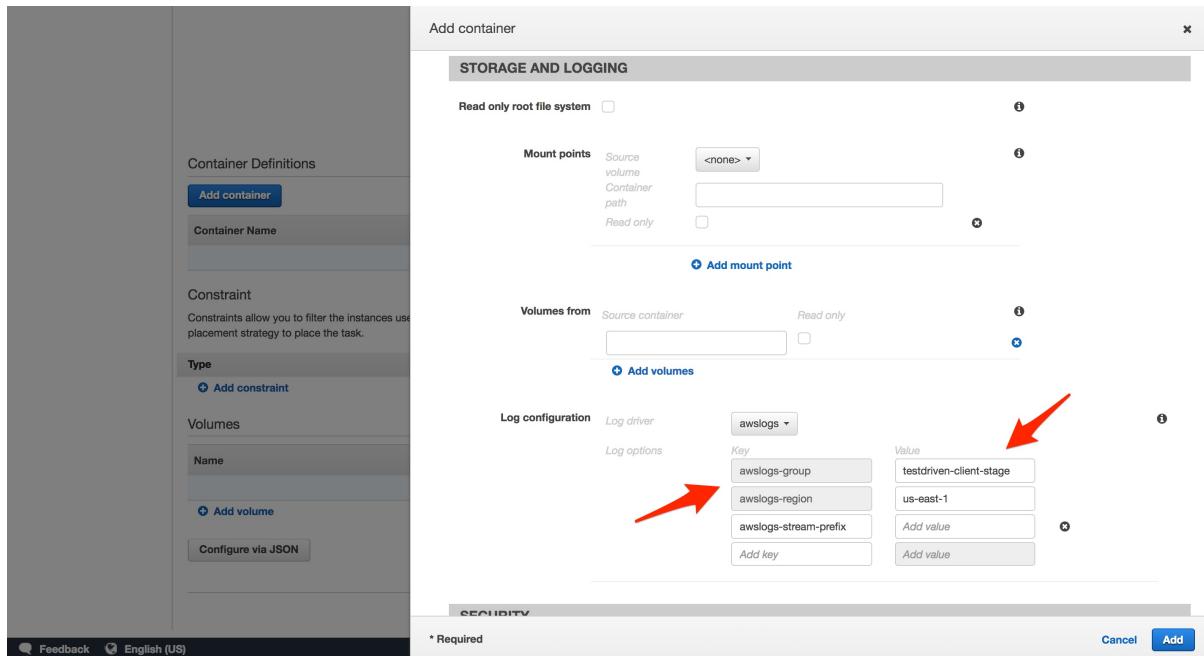
The screenshot shows the 'Edit container' dialog for a task named 'client'. The 'Image' field contains the URL '046505967931.dkr.ecr.us-east-1.amazonaws.com/test-driven-client:staging'. The 'Memory Limits (MB)' field has '300' entered. In the 'Port mappings' section, the 'Host port' is set to '0' and the 'Container port' is set to '3000'. A red arrow points to the 'Image' field, another to the 'Memory Limits' field, and a third to the 'Host port' field in the 'Port mappings' section.

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

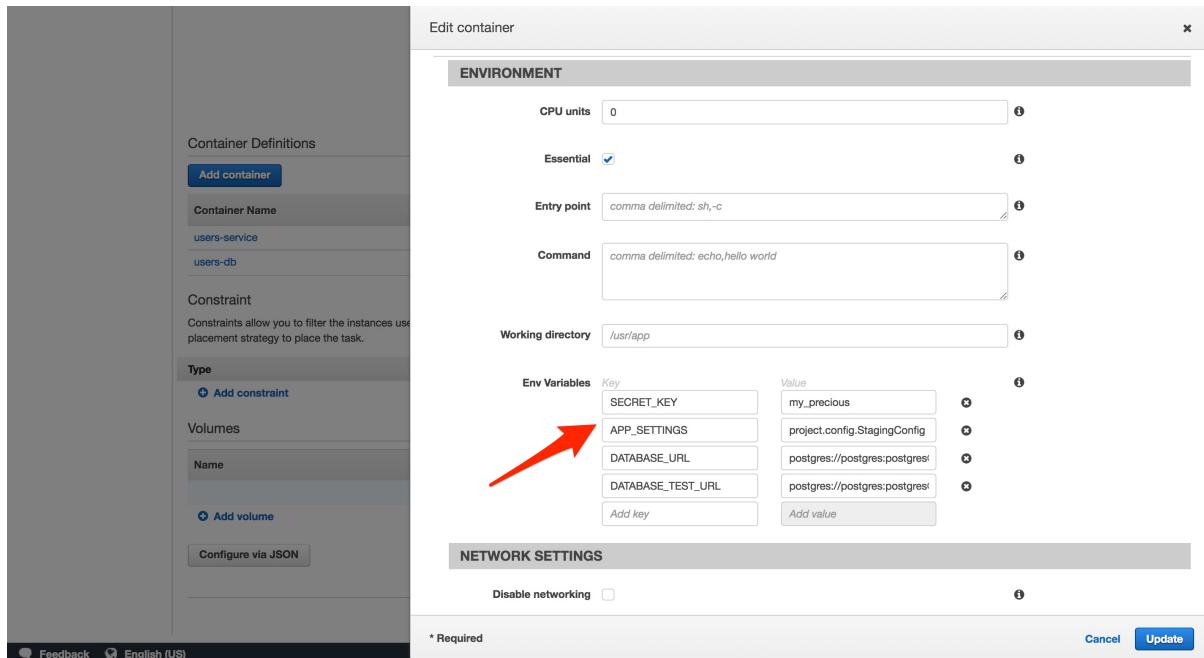
Set up a single Task Definition for the `users` service with the name `testdriven-users-stage-td`. Then, add two containers:

Container 1:

1. "Container name": `users-service`
2. "Image": `YOUR_AWS_ACCOUNT_ID.dkr.ecr.us-east-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_ACCOUNT_ID.dkr.ecr.us-east-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-east-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

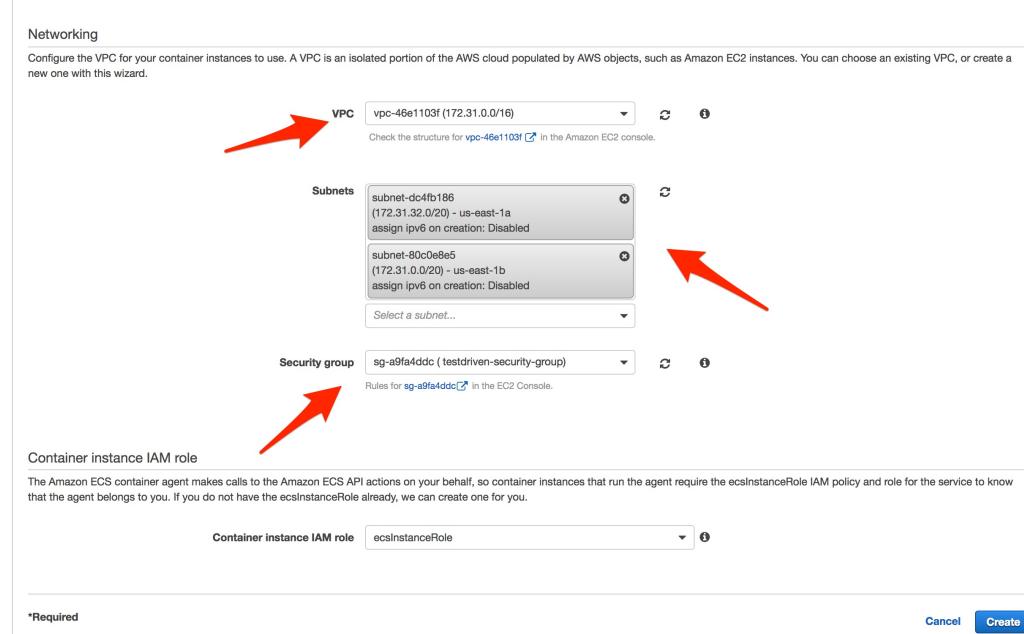
Clusters are where the actual containers run. They are just groups of EC2 instances that run Docker containers managed by ECS. To create a Cluster, click "Clusters" on the [ECS Console](#) sidebar, and then click the "Create Cluster" button.

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)

The screenshot shows the 'Create Cluster' wizard in the AWS ECS console. The left sidebar shows 'Clusters' selected. The main area has a title 'Create Cluster' with a sub-instruction: 'When you run tasks using Amazon ECS, you place them on a cluster, which is a logical grouping of EC2 instances. This wizard will guide you through the process to [create a cluster](#). You will name your cluster, and then configure the container instances that your tasks can be placed on, the security group for your container instances to use, and the IAM role to associate with your container instances so that they can make calls to the AWS APIs on your behalf.' Below this, there's a 'Cluster name*' field containing 'test-driven-staging-cluster', a checked checkbox for 'Create an empty cluster', and a 'Instance configuration' section. In the 'Instance configuration' section, there are four fields with red arrows pointing to them: 'EC2 instance type*' (set to 't2.micro'), 'Number of instances*' (set to '2'), 'EBS storage (GiB)*' (set to '22'), and 'Key pair' (set to 'ecs'). A note at the bottom of the configuration section states: 'You will not be able to SSH into your EC2 instances without a key pair. You can create a new key pair in the [EC2 console](#)'.

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`

Screenshot of the AWS EC2 Container Service "Create Service" wizard, Step 1: Configure service.

Task Definition: testdriven-client-stage-td;1

Cluster: test-driven-staging-cluster

Service name: testdriven-client-stage-service

Number of tasks: 1

Minimum healthy percent: 50

Maximum percent: 200

Task Placement: AZ Balanced Spread

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:0:3000

Load balancing

An Elastic Load Balancing load balancer distributes incoming traffic across the tasks running in your service. Choose an existing load balancer, or create a new one in the [Amazon EC2 console](#).

Load balancer type:

- None: Your service will not use a load balancer.
- Application Load Balancer: Allows containers to use dynamic host port mapping (multiple tasks allowed per container instance). Multiple services can use the same listener port on a single load balancer with rule-based routing and paths.
- Network Load Balancer: A Network Load Balancer functions at the fourth layer of the Open Systems Interconnection (OSI) model. After the load balancer receives a request, it selects a target from the target group for the default rule using a flow hash routing algorithm.
- Classic Load Balancer: Requires static host port mappings (only one task allowed per container instance); rule-based routing and paths are not supported.

Service IAM role: ecsServiceRole

Load balancer name: testdriven-staging-alb

Container to load balance:

Container name : port: client:0:3000

Add to load balancer

*Required

Cancel Previous Next step

Click "Add to load balancer".

1. "Listener port": 80:HTTP
2. "Target group name": testdriven-client-stage-tg

Service IAM role Task definitions that use the awsvpc network mode use the AWSRoleForECS service-linked role, which is created for you automatically. [Learn more](#).

Service IAM role ecsServiceRole

Load balancer name testdriven-staging-alb

Container to load balance

client : 3000 Remove X

Listener port 80:HTTP

Listener protocol HTTP

Target group name testdriven-client... ▾

Target group protocol HTTP

Target type instance

Path pattern / Evaluation order default

Health check path / Additional health check options can be configured in the ELB console after you create your service.

*Required Cancel Previous Next step

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:

The screenshot shows the AWS ECS Cluster details page for 'test-driven-staging-cluster'. The cluster status is ACTIVE with 2 registered container instances and 3 running tasks. The 'Services' tab is selected, displaying a table with three rows:

Service Name	Status	Task Definition	Desired tasks	Running tasks
testdriven-swagger-stage-service	ACTIVE	testdriven-swagger-stage-td:1	1	1
testdriven-client-stage-service	ACTIVE	testdriven-client-stage-td:2	1	1
testdriven-users-stage-service	ACTIVE	testdriven-users-stage-td:1	1	1

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.

The screenshot shows the AWS EC2 Container Service Target Groups interface. On the left, a sidebar lists various AWS services like Instances, Security Groups, and Load Balancers. The main area displays a table of target groups, with one named 'testdriven-users-stage-tg' selected. Below the table, there are tabs for 'Description', 'Targets' (which is active), 'Health checks', 'Monitoring', and 'Tags'. A note below the table explains how the load balancer routes requests to registered targets. An 'Edit' button is available for the target group. The 'Registered targets' section shows a table with columns: Instance ID, Name, Port, Availability Zone, and Status. The first target listed is 'i-058302138390d8ed6' with the name 'ECS Instance - EC2ContainerService-test-driven-staging-cluster', port 32791, availability zone 'us-east-1b', and status 'unhealthy'. The 'Availability Zones' section shows a table with columns: Availability Zone, Target count, and Healthy?. The 'us-east-1b' zone has a target count of 1 and is marked as 'No (Availability Zone contains no healthy targets)'. At the bottom, there are links for Feedback, English (US), and legal notices.

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:

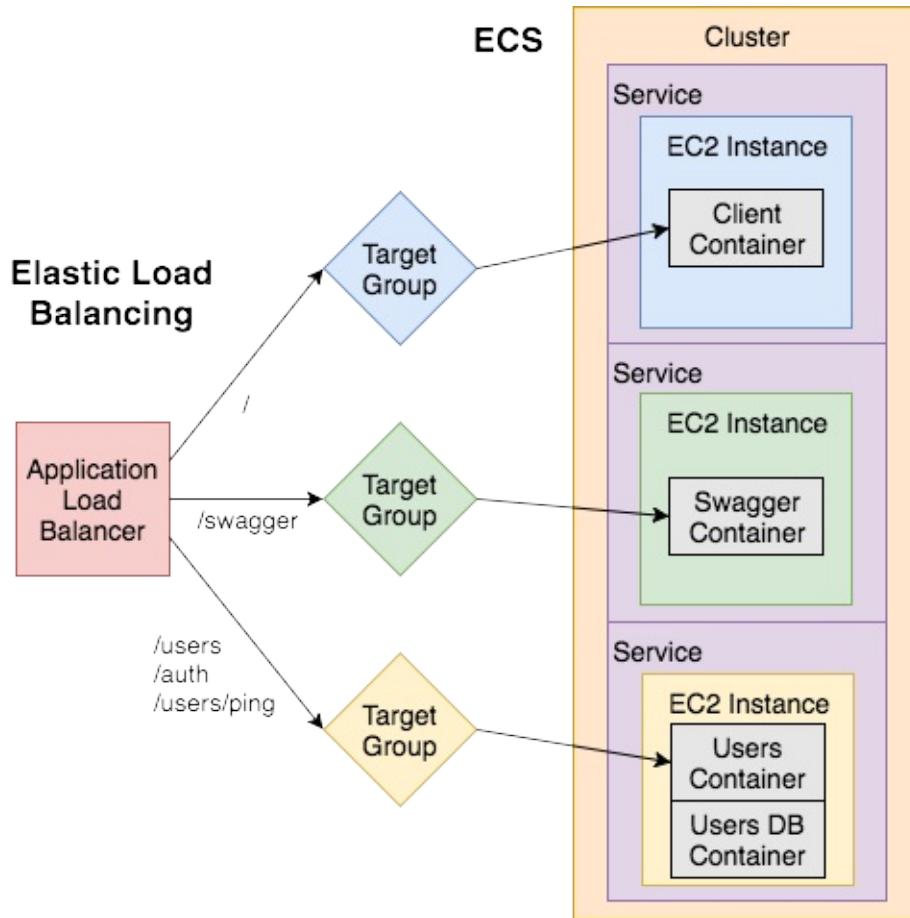
1. "Type": All traffic
2. "Port Range": 0 - 65535
3. "Source": Choose Custom, then add the Security Group ID

The screenshot shows the AWS Management Console with the EC2 Container Service selected. In the left sidebar, under 'SECURITY GROUPS', 'Security Groups' is selected. The main area displays a table of security groups, with one row selected: 'sg-a9fa4ddc' (testdriven-security-group). A modal window titled 'Edit inbound rules' is open over the table. The modal contains a table with columns: Type, Protocol, Port Range, Source, and Description. Three rows are present: 'HTTP' to port 80 from 'Custom 0.0.0.0/0' with the description 'e.g. SSH for Admin Desktop'; 'SSH' to port 22 from 'Custom 0.0.0.0/0' with the same description; and 'All traffic' to port 0-65535 from 'Custom sg-a9fa4ddc' with the same description. Below the table is a note: 'NOTE: Any edits made on existing rules will result in the edited rule being deleted and a new rule created with the new details. This will cause traffic that depends on that rule to be dropped for a very brief period of time until the new rule can be created.' At the bottom of the modal are 'Cancel' and 'Save' buttons. Red arrows point to the 'Add Rule' button, the 'Port Range' column, and the 'Source' column.

Once added, the next time a container is added to each of the Target Groups, the instance should be *healthy*:

The screenshot shows the AWS Management Console with the EC2 Container Service selected. In the left sidebar, under 'TARGET GROUPS', 'Target Groups' is selected. The main area displays a table of target groups, with one row selected: 'testdriven-users-stage-tg'. Below the table, a section titled 'Target group: testdriven-users-stage-tg' shows tabs for 'Description', 'Targets', 'Health checks', 'Monitoring', and 'Tags'. The 'Targets' tab is selected, showing a table of registered targets. One target is listed: 'i-0ed67875f461f74d7' (ECS Instance - EC2ContainerService-test-driven-staging-cluster) on port 32789 in the 'us-east-1a' availability zone, with a status of 'healthy'. A red arrow points to the 'Status' column. At the bottom of the page are 'Feedback', 'English (US)', and navigation links.

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/ping` 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.

The screenshot shows the AWS ECS service configuration page. The service name is 'testdriven-users-stage-service'. It is associated with the cluster 'test-driven-staging-cluster'. The status is ACTIVE, and there is one running task. The task definition is 'testdriven-users-stage-td:3' and it uses the 'ecsServiceRole'. The 'Details' tab is selected, showing a table for 'Load Balancing' with one entry: Target Group Name 'testdriven-users-stage-tg', Container Name 'users-service', and Container Port '5000'.

From there, click on the "Tasks" tab and click the link for the associated Task.

The screenshot shows the AWS ECS task details page for task ID '78cf57e2-5400-402d-a322-7cd24a9e1df6'. The task is part of the cluster 'test-driven-staging-cluster'. It has an EC2 instance ID 'i-0ed67675f461f74d7' and a container instance ID '39a259d0-61d7-4758-9173-30d926fb434'. The task definition is 'testdriven-users-stage-td:3' and it is assigned to the group 'service:testdriven-users-stage-service'. The task role is 'None' and the last status was 'RUNNING'. The desired status is also 'RUNNING'. The task was created at '2017-11-24 19:46:45 -0700'. The 'Network' section shows 'Network Mode' set to 'bridge'. The 'Containers' section lists two containers: 'users-db' and 'users-service', both of which are currently running.

Then, click the link for the EC2 Instance ID and grab the public IP:

EC2 Dashboard Services ▾ Resource Groups ▾ Mike Herman ▾ N. Virginia ▾ Support ▾

Instances

- Instances
- Spot Requests
- Reserved Instances
- Scheduled Instances
- Dedicated Hosts

IMAGES

- AMIs
- Bundle Tasks

ELASTIC BLOCK STORE

- Volumes
- Snapshots

NETWORK & SECURITY

- Security Groups
- Elastic IPs
- Placement Groups
- Key Pairs
- Network Interfaces

LOAD BALANCING

Actions

Launch Instance Connect Actions ▾

Instance ID: i-0ed67875f461f74d7 Add filter

Name	Instance ID	Instance Type	Availability Zone	Instance State	Status Checks	Alarm Status	Public DNS (IPv4)	IPv4 Public IP
ECS Instanc...	i-0ed67875f461f74d7	t2.micro	us-east-1a	running	1/2 checks ...	None	ec2-34-203-207-7.com...	34.203.207.7

Instance: i-0ed67875f461f74d7 (ECS Instance - EC2ContainerService-test-driven-staging-cluster) Public DNS: ec2-34-203-207-7.compute-1.amazonaws.com

Description Status Checks Monitoring Tags

Instance ID	i-0ed67875f461f74d7	Public DNS (IPv4)	ec2-34-203-207-7.compute-1.amazonaws.com
Instance state	running	IPv4 Public IP	34.203.207.7
Instance type	t2.micro	IPv6 IPs	-
Elastic IPs		Private DNS	ip-172-31-45-252.ec2.internal
Availability zone	us-east-1a	Private IPs	172.31.45.252
Security groups	testdriven-security-group. view inbound rules	Secondary private IPs	
Scheduled events	No scheduled events	VPC ID	vpc-46e1103f
AMI ID	amzn-ami-2017.09.4-amazon-ecs-optimized (ami-20ff515a)	Subnet ID	subnet-dc4fb186
Platform	-	Network interfaces	eth0
IAM role	ecsinstanceRole	Source/dst. check	True
Keypair name	ecs		

Feedback English (US) © 2008 - 2017, Amazon Web Services, Inc. or its affiliates. All rights reserved. Privacy Policy Terms of Use

SSH into the instance:

```
$ 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 database:

```
$ docker exec -it Container_ID bash
# python manage.py recreate_db
# python manage.py seed_db
```

```
2. michael.herman@gMH8c2cb7-911: ~/repos/realpython/testdriven-app (zsh)
gitbook (node) ⌘1 ..estdriven-app (zsh) ⌘2 ~ (zsh) ⌘3
[ec2-user@ip-172-31-45-252 ~]$ docker exec -it 5a20b8d6a4f5 bash
root@5a20b8d6a4f5:/usr/src/app# python manage.py recreate_db
root@5a20b8d6a4f5:/usr/src/app# python manage.py seed_db
root@5a20b8d6a4f5:/usr/src/app# exit
exit
[ec2-user@ip-172-31-45-252 ~]$ exit
logout
Connection to 34.203.207.7 closed.
→ testdriven-app git:(staging) |
```

Navigate to http://EC2_PUBLIC_IP/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 the 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-east-1.amazonaws.com/test-driven-client:staging",  
      "essential": true,  
      "memoryReservation": 300,
```

```

"portMappings": [
  {
    "hostPort": 0,
    "protocol": "tcp",
    "containerPort": 3000
  }
],
"logConfiguration": {
  "logDriver": "awslogs",
  "options": {
    "awslogs-group": "testdriven-client-stage",
    "awslogs-region": "us-east-1"
  }
}
],
"family": "testdriven-client-stage-td"
}

```

Users

```

{
  "containerDefinitions": [
    {
      "name": "users-service",
      "image": "%s.dkr.ecr.us-east-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-east-1"
    }
},
{
    "name": "users-db",
    "image": "%s.dkr.ecr.us-east-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-east-1"
        }
    }
},
],
"family": "testdriven-users-stage-td"
}
```

Swagger

```
{
  "containerDefinitions": [
    {
      "name": "swagger",
      "image": "%s.dkr.ecr.us-east-1.amazonaws.com/test-driven-swagger:staging",
      "essential": true,

      "logConfiguration": {
        "logDriver": "awslogs",
        "options": {
          "awslogs-group": "testdriven-swagger-stage",
          "awslogs-region": "us-east-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-east-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

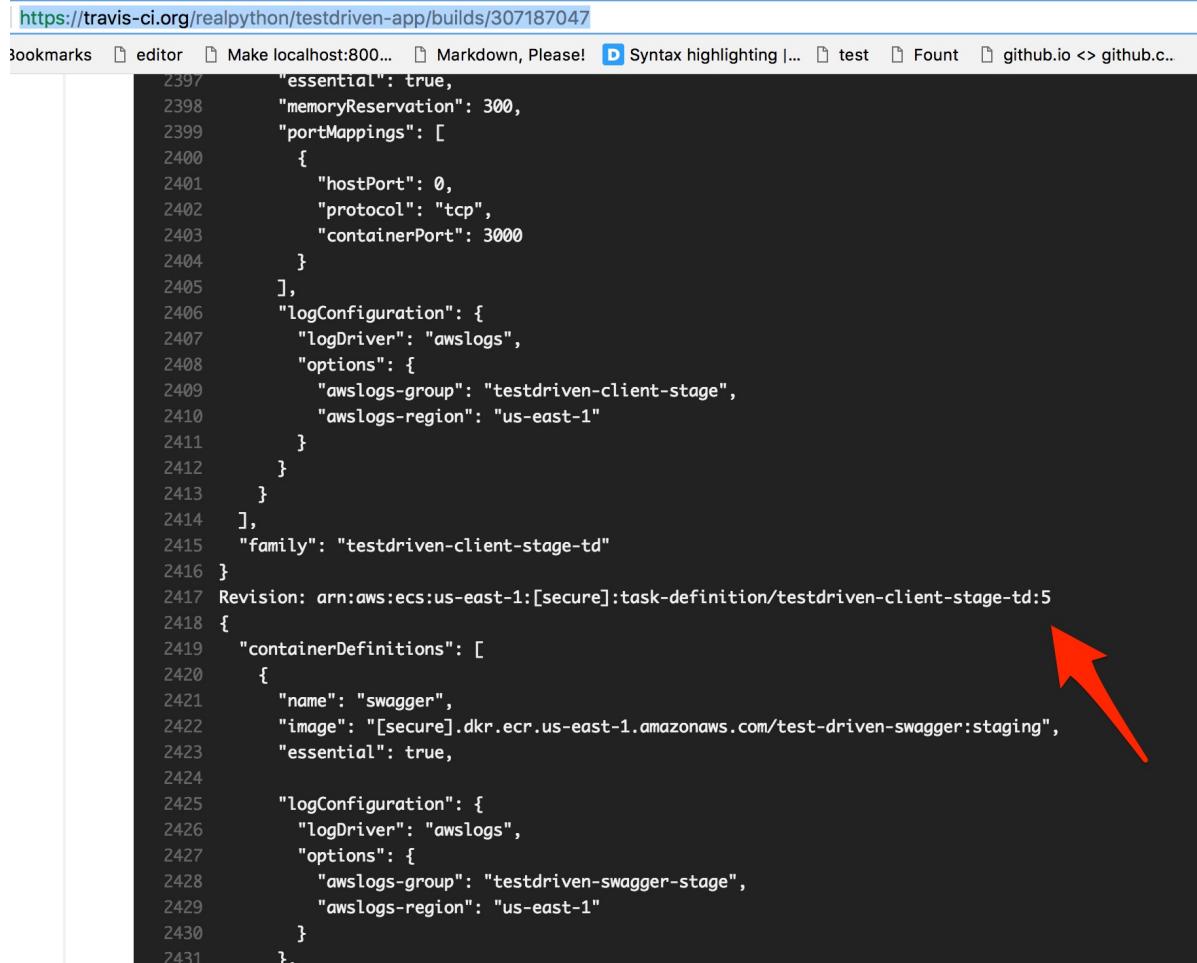
```

Here, if the branch is `staging` and it's not a 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.



The screenshot shows a JSON configuration for an AWS Task Definition. The code is as follows:

```

2397   "essential": true,
2398   "memoryReservation": 300,
2399   "portMappings": [
2400     {
2401       "hostPort": 0,
2402       "protocol": "tcp",
2403       "containerPort": 3000
2404     }
2405   ],
2406   "logConfiguration": {
2407     "logDriver": "awslogs",
2408     "options": {
2409       "awslogs-group": "testdriven-client-stage",
2410       "awslogs-region": "us-east-1"
2411     }
2412   }
2413 ],
2414 ],
2415 "family": "testdriven-client-stage-td"
2416 }
2417 Revision: arn:aws:ecs:us-east-1:[secure]:task-definition/testdriven-client-stage-td:5
2418 {
2419   "containerDefinitions": [
2420     {
2421       "name": "swagger",
2422       "image": "[secure].dkr.ecr.us-east-1.amazonaws.com/test-driven-swagger:staging",
2423       "essential": true,
2424
2425       "logConfiguration": {
2426         "logDriver": "awslogs",
2427         "options": {
2428           "awslogs-group": "testdriven-swagger-stage",
2429           "awslogs-region": "us-east-1"
2430         }
2431       },

```

The screenshot shows the AWS ECS Task Definitions page. The URL is [Task Definitions > testdriven-client-stage-td > status > ACTIVE](#). The page title is "Task Definition Name : testdriven-client-stage-td". Below it says "Select a revision for more details". There are two buttons: "Create new revision" and "Actions". The status is set to "Active". A red arrow points to the first item in the list, "testdriven-client-stage-td:5", which is also highlighted with a blue border.

Task Definition Name : Revision	Status
<input type="checkbox"/> testdriven-client-stage-td:5	Active
<input type="checkbox"/> testdriven-client-stage-td:4	Active
<input type="checkbox"/> testdriven-client-stage-td:3	Active
<input type="checkbox"/> testdriven-client-stage-td:2	Active
<input type="checkbox"/> testdriven-client-stage-td:1	Active

Then, navigate to the Cluster. Update each of the Services so that they use the new Task Definitions.

The screenshot shows the "Update Service" configuration page. The URL is [Services > Resource Groups > \[Cluster\] > Services > \[Service\] > Update](#). The title is "Configure service". It shows the "Step 1: Configure service" section. The "Task Definition" dropdown is set to "testdriven-client-stage-td:5". The "Cluster" dropdown is set to "test-driven-staging-cluster". Other fields include "Service name: testdriven-client-stage-service", "Number of tasks: 1", "Minimum healthy percent: 50", and "Maximum percent: 200". A red arrow points to the "Cluster" dropdown.

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:

The screenshot shows the AWS CloudWatch interface with the path: CloudWatch > Log Groups > testdriven-users-stage > 3ec54e05f768cc137edf188893d3f011a97230435af17851c66bd31c2bc12571. The log entries are filtered by Time (UTC +00:00) and Message. The last entry, timestamped 2017-11-25 at 17:23:47, contains the text 'LINE 2: FROM users'.

We could run the migrations manually, but we can just do this automatically by updating the `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 again, 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-east-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-definition $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
```

```
[ ]
```

```
[ ]
```

Let's also 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 9.6.3-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-east-1a or us-east-1b .

The screenshot shows the 'Configure advanced settings' step of the RDS setup wizard. It includes sections for 'Network & Security' (with a dropdown for 'Default VPC'), 'Subnet group info' (with a dropdown for 'default'), 'Public accessibility info' (radio buttons for 'Yes' and 'No'), 'Availability zone info' (dropdown set to 'us-east-1a'), and 'VPC security groups' (radio buttons for 'Create new VPC security group' and 'Select existing VPC security groups', with the latter selected and a dropdown showing 'testdriven-security-group (VPC)'). Red arrows highlight the 'Default VPC' dropdown, the 'Availability zone info' dropdown, and the 'Select VPC security groups' dropdown.

Change the DB name to users_prod and then create the new database.

You can quickly check the status via:

```
$ aws rds describe-db-instances \
--db-instance-identifier testdriven-production \
--query 'DBInstances[].{DBInstanceStatus:DBInstanceState}'
```

```
[{"DBInstanceStatus": "creating"}]
```

Then, once the status is "available", grab the address:

```
$ aws 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": `3000`
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.

The screenshot shows the AWS Load Balancers console with a list of existing load balancers. A new load balancer named 'testdriven-production-alb' is being created. The 'Basic Configuration' section is displayed, showing the load balancer's name, ARN, DNS name, scheme, type, and availability zones. The 'Security' section shows the selected security group 'sg-a9fa4ddc'. Several red arrows highlight these fields: one arrow points to the 'Name' field, another to the 'Availability Zones' dropdown, and two others point to the 'Security groups' section.

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:

Name	Port	Protocol	Target type	VPC ID
testdriven-users-stage-tg	5000	HTTP	instance	vpc-46e1103f
testdriven-users-prod-tg	5000	HTTP	instance	vpc-46e1103f
testdriven-swagger-stage-tg	8080	HTTP	instance	vpc-46e1103f
testdriven-swagger-prod-tg	8080	HTTP	instance	vpc-46e1103f
testdriven-client-stage-tg	3000	HTTP	instance	vpc-46e1103f
testdriven-client-prod-tg	3000	HTTP	instance	vpc-46e1103f

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

Feedback English (US) © 2008 - 2017, Amazon Web Services, Inc. or its affiliates. All rights reserved. Privacy Policy Terms of Use

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 $(aws ecr get-login --region us-east-1)
    export TAG=$TRAVIS_BRANCH
    export REPO=$AWS_ACCOUNT_ID.dkr.ecr.us-east-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
fi
fi

```

ECS

Task Definitions

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-prod-td` and then add a new container:

1. "Container name": `client`
2. "Image": `YOUR_AWS_ACCOUNT_ID.dkr.ecr.us-east-1.amazonaws.com/test-driven-`
`client:production`
3. "Memory Limits (MB)": `300` soft limit
4. "Port mappings": `0 host, 3000 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-service`
2. "Image": `YOUR_AWS_ACCOUNT_ID.dkr.ecr.us-east-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-east-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

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.

The screenshot shows the AWS ECR console for the repository 'test-driven-swagger'. It lists several images with their digests, sizes, and push times. A red arrow points to the 'production' tag in the list of tags.

Image tags	Digest	Size (MiB)	Pushed at
<input type="checkbox"/> production	sha256:ca02ce5cc73de0f5846df6b4055d4c0f00d759d638e4d5e8e0a4...	63.46	2017-11-25 20:09:36 -0700
<input type="checkbox"/> staging	sha256:29ffa50d4c3d2ae70357f6a527215318c0809c1cbc0b2302362a...	63.46	2017-11-25 19:57:44 -0700
<input type="checkbox"/>	sha256:252749125d5a015bb75f2f3b979041742f8681904b516a294c7f...	63.46	2017-11-25 16:17:52 -0700
<input type="checkbox"/>	sha256:90464faabb2b2b133464acd3f0ecaf5147328184f659c48bf5e50...	63.46	2017-11-25 13:29:26 -0700
<input type="checkbox"/>	sha256:5ea21699b4a49acfcd39a96768640ad82b91c783b38c5911d...	63.46	2017-11-25 12:11:01 -0700
<input type="checkbox"/>	sha256:5338977feb9716297d16f833b8991f7736c20a7b19b0a1434456...	63.46	2017-11-25 11:45:43 -0700
<input type="checkbox"/>	sha256:4874911f189089d1312bb640bddb904b826afb6d1be82770651...	63.46	2017-11-25 11:26:49 -0700
<input type="checkbox"/>	sha256:7a469a1f514a2a31812a2350c68506c3ada2076ab7453f5d3...	63.46	2017-11-25 11:13:58 -0700
<input type="checkbox"/>	sha256:ba756b6efb2012e8ff7b05276b0203e042dab580468cef9...	63.46	2017-11-25 10:39:58 -0700
<input type="checkbox"/>	sha256:5ab507d0086ff4c32a67bab37354288e6707501741b4a686eb6f...	63.46	2017-11-25 10:07:31 -0700
<input type="checkbox"/>	sha256:44de696f42aa74812ce958133047bcbebdd3234fb8a32503f641c...	63.46	2017-11-25 09:55:32 -0700

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:

- "Cluster name": `test-driven-production-cluster`
- "EC2 instance type": `t2.micro`
- "Number of instances": `2`
- "Key pair": Select an existing [Key Pair](#), like `ecs`, or create a new one
- Make sure to pick the "VPC" and "Security group" associated with ALB along with the appropriate "Subnets" - `us-east-1a` and `us-east-1b`.

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.sh
```

```
ADD ./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
ADD ./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:0:3000

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-service: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:

The screenshot shows the AWS ECS Cluster details page for 'test-driven-production-cluster'. The cluster status is ACTIVE. It lists three registered container instances: testdriven-swagger-prod-service, testdriven-users-prod-service, and testdriven-client-prod-service. Each service has one active task defined by the 'testdriven-' prefix in their task definitions.

Service Name	Status	Task Definition	Desired tasks	Running tasks
testdriven-swagger-prod-service	ACTIVE	testdriven-swagger-prod-td:2	1	1
testdriven-users-prod-service	ACTIVE	testdriven-users-prod-td:2	1	1
testdriven-client-prod-service	ACTIVE	testdriven-client-prod-td:2	1	1

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`

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".

Client

```
{  
  "containerDefinitions": [  
    {  
      "name": "client",  
      "image": "%s.dkr.ecr.us-east-1.amazonaws.com/test-driven-client:production",  
      "essential": true,  
      "memoryReservation": 300,  
      "portMappings": [  
        {  
          "hostPort": 0,  
          "protocol": "tcp",  
          "containerPort": 3000  
        }  
      ],  
      "logConfiguration": {  
        "logDriver": "awslogs",  
        "options": {  
          "awslogs-group": "testdriven-client-prod",  
          "awslogs-region": "us-east-1"  
        }  
      }  
    }  
  ]  
}
```

```

        }
    }
],
"family": "testdriven-client-prod-td"
}

```

Users

```

{
  "containerDefinitions": [
    {
      "name": "users-service",
      "image": "%s.dkr.ecr.us-east-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-east-1"
        }
      }
    ]
  ],
}

```

```

    "family": "testdriven-users-prod-td"
}

```

Swagger

```

{
  "containerDefinitions": [
    {
      "name": "swagger",
      "image": "%s.dkr.ecr.us-east-1.amazonaws.com/test-driven-swagger:production",
      "essential": true,

      "logConfiguration": {
        "logDriver": "awslogs",
        "options": {
          "awslogs-group": "testdriven-swagger-prod",
          "awslogs-region": "us-east-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.

Environment Variables

Notice that the values are not escaped when your builds are executed. Special characters (for bash) should be escaped accordingly.

AWS_ACCESS_KEY_ID	•••••••••••••••••••••••••	
AWS_ACCOUNT_ID	••••••••••••••••••••••••	
AWS_RDS_URI	••••••••••••••••••••••••	
AWS_SECRET_ACCESS_KEY	••••••••••••••••••••••••	
PRODUCTION_SECRET_KEY	••••••••••••••••••••••••	
Name	Value	<input type="checkbox"/> OFF <small>Display value in build log</small>
		Add

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-east-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
        }
    fi
fi
```

```

    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 $PRODUCTION_SECRET_KEY)
    echo "$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.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-service` container. Make sure the `DATABASE_URL` and `SECRET_KEY` environment variables are correct:

Container Name	Image	CPU Units	Hard/Soft memory limits (MiB)	Essential
users-service	046505967931.dkr.ecr.us-east-1....	0	--/300	true

Environment Variables

Key	Value
APP_SETTINGS	project.config.ProductionConfig
DATABASE_TEST_URL	postgres://postgres:postgres@users-db:5432/users_test
DATABASE_URL	[REDACTED]
SECRET_KEY	[REDACTED]

Docker labels

Key	Value
No docker labels	

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-east-1
        }

        configure_aws_cli
        aws ecs update-service --cluster $CLUSTER_NAME --service $SERVICE_NAME --task-definition $TASK_DEFINITION_NAME --force-new-deployment
```

```

        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 set up is simple. Review [Routing Traffic to an ELB Load Balancer](#) for more details.

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

Run the unit and integration tests:

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

Lint:

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

Part 6

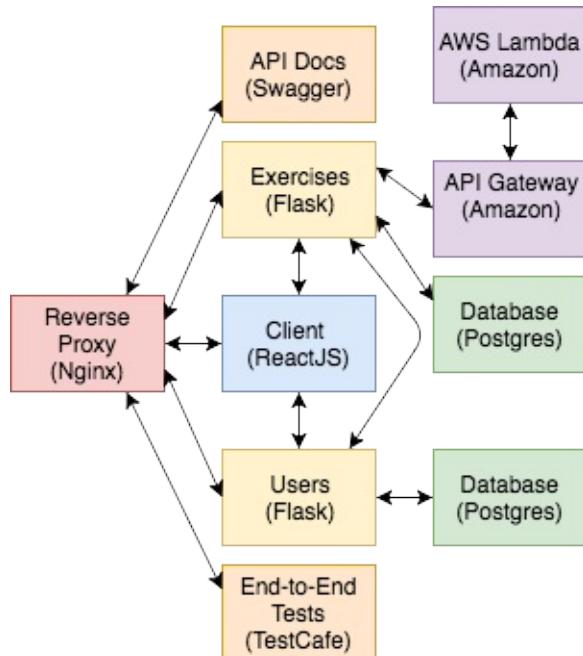
In the final part, we'll focus our attention on adding a new *Flask* service, with two RESTful-resources, to evaluate user-submitted code. Along the way, we'll tie in *AWS Lambda* and *API Gateway* and spend a bit of time refactoring *React* and the *end-to-end* test suite. Finally, we'll update the 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 you 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

Grab the code: <https://github.com/realpython/testdriven-app/>

Dependencies

You will use the following dependencies in Part 6:

1. React-Ace v5.7.0
2. Flask v0.12.2
3. Flask-Cors v3.0.3
4. Flask-Script v2.0.5
5. Flask-Testing v0.6.2
6. Gunicorn v19.7.1
7. Coverage.py v4.4.1
8. Requests v2.18.4
9. Flask-SQLAlchemy v2.3.2
10. Flask-Migrate v2.1.1
11. psycopg2 v2.7.3.1
12. flake8 v3.4.1

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)
```

Spin up the app:

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

Update the database:

```
$ docker-compose -f docker-compose-stage.yml \  
  run users-service python manage.py recreate_db  
  
$ docker-compose -f docker-compose-stage.yml \  
  run users-service 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-service python manage.py test  
  
$ docker-compose -f docker-compose-dev.yml \  
  run users-service 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 the 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.REACT_APP_USERS_SERVICE_URL;

fixture('/swagger').page(` ${TEST_URL}/`);

test.only(`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()
});

```

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;
```

So, `LinkContainer` does not play nice with external links, so we had to use a regular `<a>` tag with custom styles.

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 2017 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 2017 <a href="http://testdriven.io">TestDriven.io</a>.</span>
      </small>
    </div>
  </footer>
);
```

```

        </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`:

```

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">
```

[TestDriven.io](#)[About](#)[Users](#)[Swagger](#)[Register](#)[Log In](#)

Register

- ✗ Username must be greater than 5 characters.
- ✗ Email must be greater than 5 characters.
- ✗ Email must be a valid email address.
- ✗ Password must be greater than 10 characters.

 Enter a username Enter an email address Enter a password

Copyright 2017 [TestDriven.io](#).

Add more tests as needed. Ensure they are all green before moving on.

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.js*:

```
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 write 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": "^0.16.2",
  "react": "^16.0.0",
  "react-ace": "^5.7.0",
  "react-bootstrap": "^0.31.5",
  "react-dom": "^16.0.0",
  "react-router-bootstrap": "^0.24.4",
  "react-router-dom": "^4.2.2",
  "react-scripts": "1.0.15",
```

```

    "testcafe": "^0.16.1"
},

```

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-dev.yml \
  run client npm test -- --verbose
```

Jump back to the browser. You should see something similar to:

TestDriven.io About Users Swagger Register Log In

Exercises

Define a function called sum that takes two integers as arguments and returns their sum.

```
1 # Enter your code here.
```

Run Code

Copyright 2017 TestDriven.io.

Run the tests:

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

You should see a few failures since the `AceEditor` is not being rendered properly in the tests. Let's [mock](#) the entire module.

Exercises.test.js:

```
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.js:

```

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.js*:

```

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>&ampnbspPlease 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 to get the value in 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
    }
  });
};
```

Finally, 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.

TestDriven.io About Users User Status Swagger

Log Out

Exercises

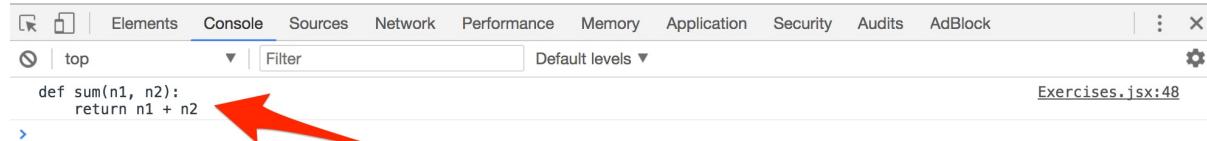
Define a function called sum that takes two integers as arguments and returns their sum.

```

1 def sum(n1, n2):
2     return n1 + n2

```

Run Code



Run the end-to-end tests. You should have a single failure in `e2e/index.test.js` since the page displays an alert on the page load if the user is not authenticated. To fix, remove the following expect:

```
.expect(Selector('.alert').exists).notOk()
```

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/realpython/testdriven-app/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:
    - APP_SETTINGS=project.config.DevelopmentConfig
    - USERS_SERVICE_URL=http://users-service:5000
  depends_on:
    - users-service
  links:
    - users-service
```

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
```

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-service python manage.py recreate_db

$ docker-compose -f docker-compose-dev.yml \
    run users-service 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-service python manage.py test
inspect $? users
docker-compose -f $file run users-service 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 do this all on your own. Put your skills to test!

Model

Set up an `Exercise` model with the following columns:

Name	Type	Example
<code>id</code>	integer	1
<code>body</code>	string	Define a function called sum that takes two integers as arguments and returns their sum.
<code>test_code</code>	string	<code>sum(2, 2)</code>
<code>test_code_solution</code>	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-service python manage.py recreate_db

$ docker-compose -f docker-compose-dev.yml \
  run users-service 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"""
    DEBUG = False
    TESTING = False
    USERS_SERVICE_URL = os.environ.get('USERS_SERVICE_URL')
    SQLALCHEMY_TRACK_MODIFICATIONS = False

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

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

```

```

class StagingConfig(BaseConfig):
    """Staging configuration"""
    DEBUG = False
    SQLALCHEMY_DATABASE_URI = os.environ.get('DATABASE_URL')

class ProductionConfig(BaseConfig):
    """Production configuration"""
    DEBUG = False
    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_sqlalchemy import SQLAlchemy
from flask_migrate import Migrate

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

def create_app():

    # 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)
    migrate.init_app(app, db)

    # register blueprints
    from project.api.base import base_blueprint

```

```
    app.register_blueprint(base_blueprint)

    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.assertTrue(app.config['DEBUG'] is True)
    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['DEBUG'])
        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['DEBUG'])
        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_staging;
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` to link the `eval-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:
    - APP_SETTINGS=project.config.DevelopmentConfig
    - 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-service
    - exercises-db
  links:
    - users-service
    - 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 runserver -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_script import Manager
from flask_migrate import MigrateCommand
```

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

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

app = create_app()
manager = Manager(app)
manager.add_command('db', MigrateCommand)

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

@manager.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

@manager.command
def recreate_db():
    """Recreates a database."""
    db.drop_all()
```

```
db.create_all()
db.session.commit()

if __name__ == '__main__':
    manager.run()
```

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
    exercise_body = post_data.get('exercise_body')
    test_code = post_data.get('test_code')

```

```

test_code_solution = post_data.get('test_code_solution')
try:
    db.session.add(Exercise(
        exercise_body=exercise_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-service python manage.py recreate_db  
  
$ docker-compose -f docker-compose-dev.yml \  
run users-service 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
```

Coverage Summary:

Name	Stmts	Miss	Branch	BrPart	Cover
project/api/base.py	6	0	0	0	100%
project/api/exercises.py	33	5	8	2	83%
project/api/models.py	13	9	0	0	31%
project/api/utils.py	33	11	8	2	63%

TOTAL	85	25	16	4	69%
-------	----	----	----	---	-----

Write any additional routes and tests. 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, 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":

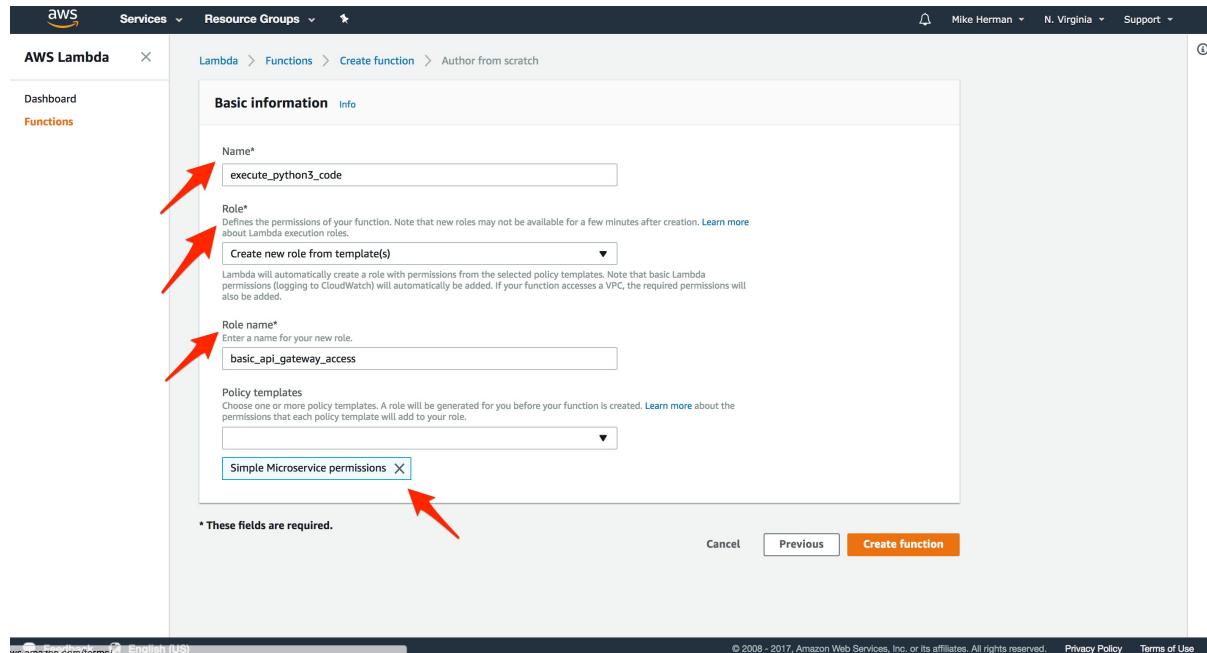
The screenshot shows the AWS Lambda homepage. At the top right, there is a 'Get started' section with a 'Create a function' button. A large red arrow points to this button. Below the 'Get started' section, there is a 'How it works' section and a 'Related services' section. The 'Related services' section includes links for S3 and DynamoDB. To the right, there is a 'More resources' sidebar with links for Documentation, API reference, Serverless Application Model (SAM), SAM Local, and Forums. At the bottom, there is a 'Pricing & costs (US)' section and a footer with links for Feedback, English (US), Privacy Policy, and Terms of Use.

Create function

Click "Author from scratch" to start with a blank function:

The screenshot shows the 'Select blueprint' step in the AWS Lambda function creation wizard. On the left, there is a navigation bar with steps: Step 1 Select blueprint, Step 2 Configure triggers, Step 3 Configure function, and Step 4 Review. The 'Select blueprint' step is currently active. In the main area, there is a message: 'Welcome to AWS Lambda! You can get started on creating your first Lambda function by choosing one of the blueprints below.' Below this, there is a 'Blueprints' table with two rows. The first row contains 'kinesis-firehose-syslog-to-json' and 'splunk-elb-application-access-logs-processor'. The second row contains 'alexa-skill-kit-sdk-factskill' and 'batch-get-job-python27'. A red arrow points to the 'Author from scratch' button at the top right of the blueprint list.

Name the function `execute_python_code`, and for the Role select "Create a new Role from template(s)" from the drop-down. 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

Select "Python 3.6" in the "Runtime" drop-down. 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
```

```

lambda_function.py
1 import sys
2 from io import StringIO
3
4
5 def lambda_handler(event, context):
6     # get code from payload
7     code = event['answer']
8     test_code = code + '\nprint(sum(1,1))'
9     # capture stdout
10    buffer = StringIO()
11    sys.stdout = buffer
12    # execute code
13    try:
14        exec(test_code)
15    except:
16        return False
17    # return stdout
18    sys.stdout = sys.stdout
19    # check

```

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.

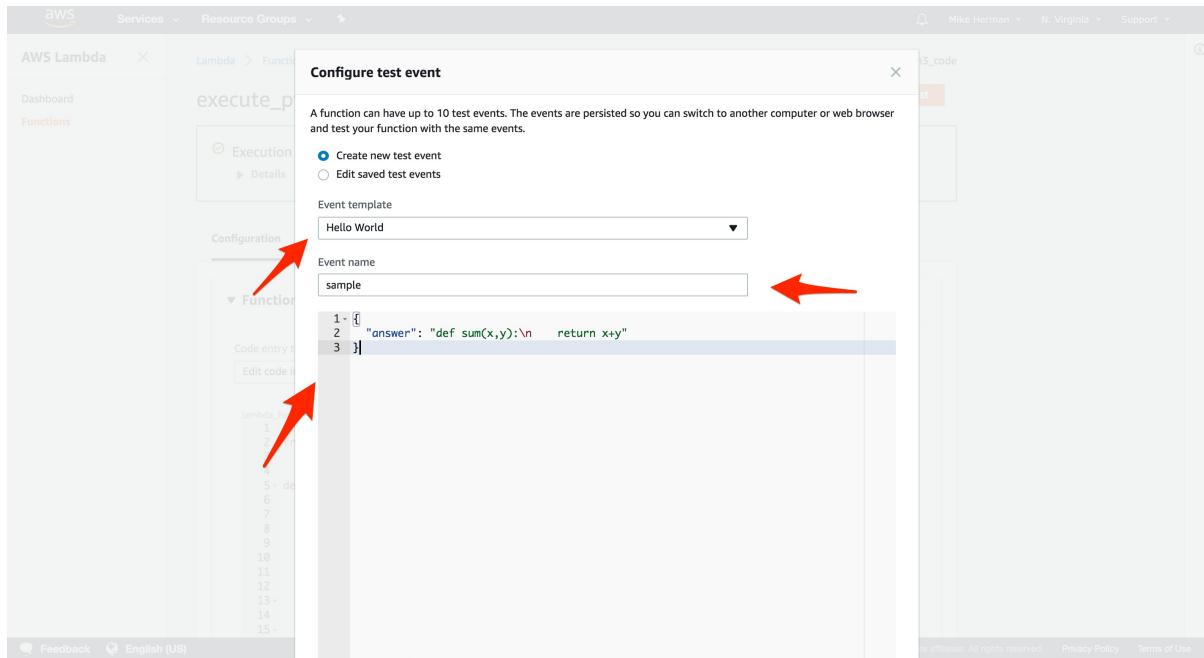
You can store the `lambda_handler` function in `services/lambda/handler.py`.

Test

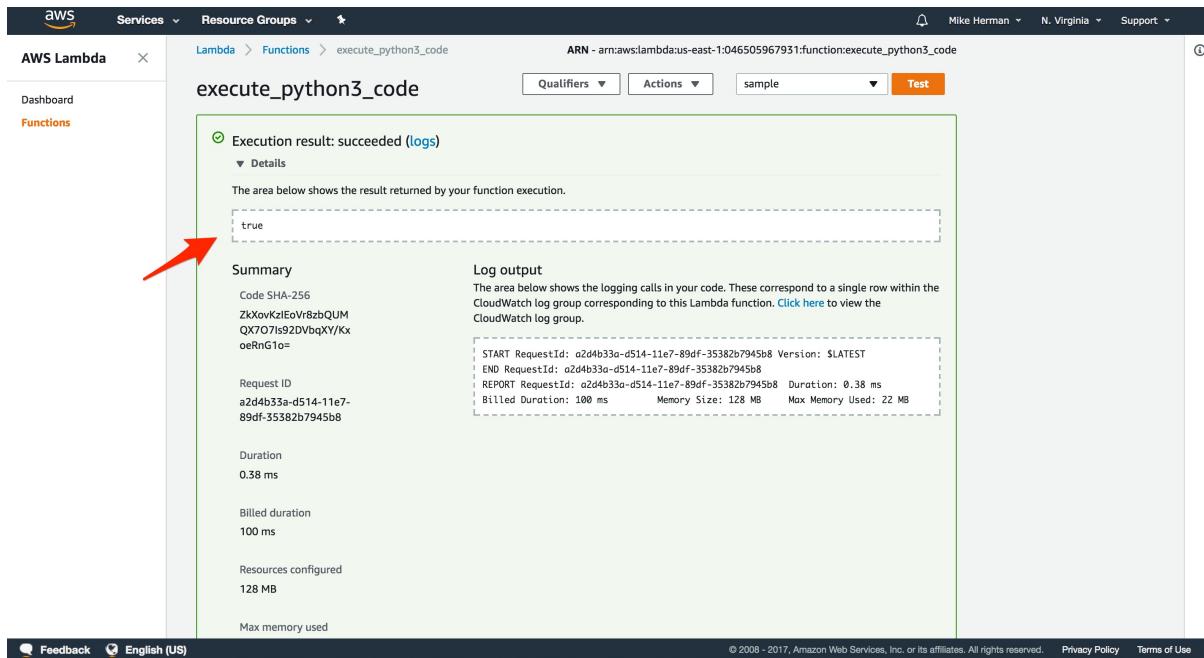
Next click on the "Test" button to execute the newly created Lambda. Using the "Hello World" event template, replace the sample with:

```
{
  "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:



The screenshot shows the Amazon API Gateway homepage. At the top, there's a navigation bar with 'Services', 'Resource Groups', and user information ('Mike Herman', 'N. Virginia', 'Support'). Below the navigation is the 'Amazon API Gateway' logo and a brief description: 'Amazon API Gateway helps developers to create and manage APIs to back-end systems running on Amazon EC2, AWS Lambda, or any publicly addressable web service. With Amazon API Gateway, you can generate custom client SDKs for your APIs, to connect your back-end systems to mobile, web, and server applications or services.' A prominent blue 'Get Started' button is centered below the text, with a large red arrow pointing to it. Below the button is a link to the 'Getting Started Guide'.



Streamline API development
Amazon API Gateway lets you simultaneously run multiple versions and release stages of the same API, allowing you to quickly iterate, test, and release new versions.



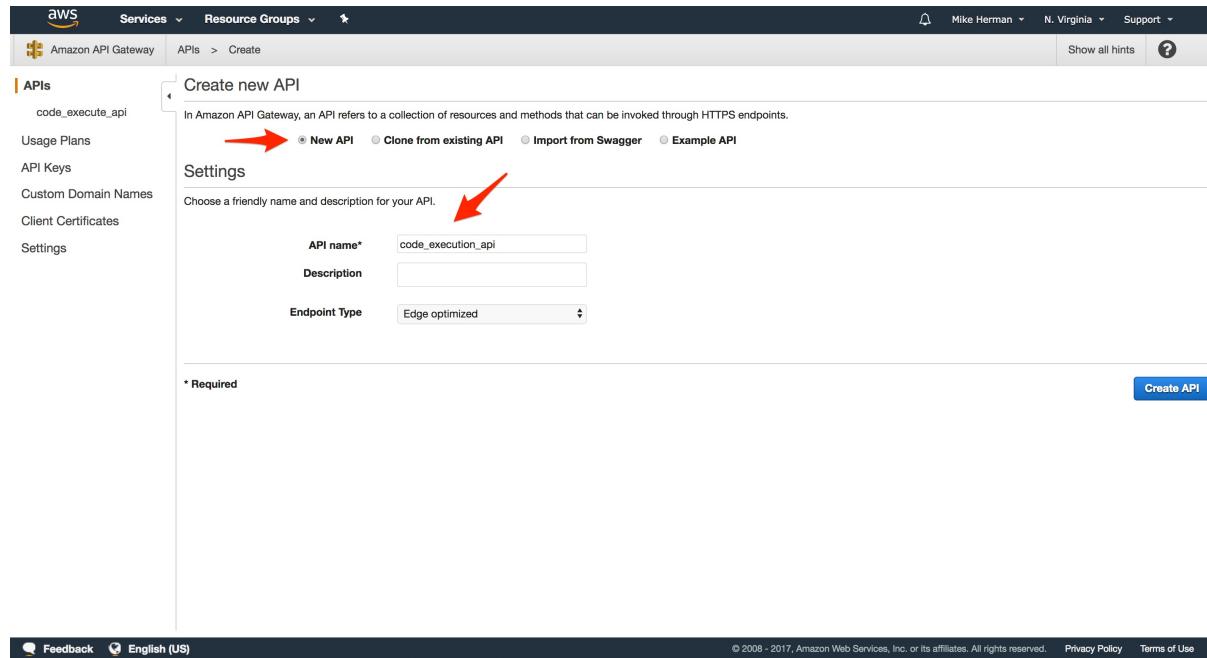
Performance at scale
Amazon API Gateway helps you improve performance by managing traffic to your existing back-end systems, throttling API call spikes, and enabling result caching.



SDK generation
Amazon API Gateway can generate client SDKs for JavaScript, iOS, and Android, which you can use to quickly test new APIs from your applications and distribute SDKs to third-party developers.

[Feedback](#) [English \(US\)](#) © 2008 - 2017, Amazon Web Services, Inc. or its affiliates. All rights reserved. [Privacy Policy](#) [Terms of Use](#)

Select "New API", and then use `code_execution_api` for the name.

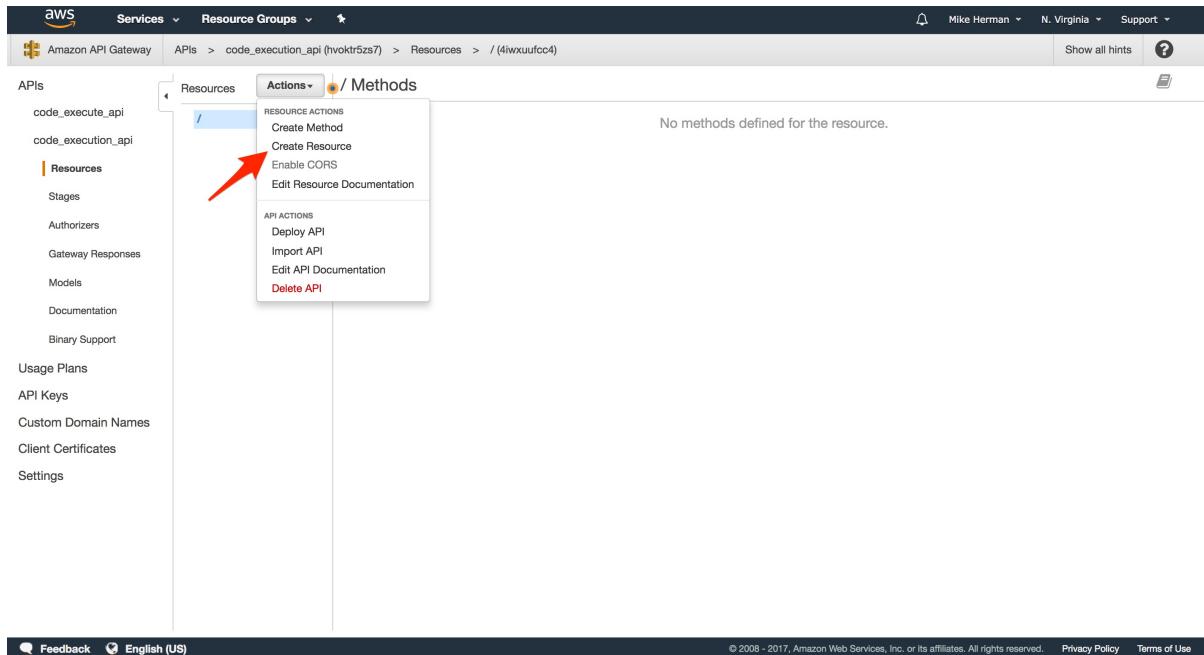


The screenshot shows the 'Create new API' form. On the left, there's a sidebar with 'APIs' and sub-options: 'code_execute_api', 'Usage Plans', 'API Keys', 'Custom Domain Names', 'Client Certificates', and 'Settings'. The main area has a title 'Create new API' with a sub-instruction: 'In Amazon API Gateway, an API refers to a collection of resources and methods that can be invoked through HTTPS endpoints.' Below this are three radio buttons: 'New API' (selected), 'Clone from existing API', 'Import from Swagger', and 'Example API'. A red arrow points to the 'New API' button. The next section is 'Settings' with the instruction 'Choose a friendly name and description for your API.' It contains fields: 'API name*' (with 'code_execution_api' entered), 'Description' (empty), and 'Endpoint Type' (set to 'Edge optimized'). A red arrow points to the 'API name' input field. At the bottom right is a blue 'Create API' button.

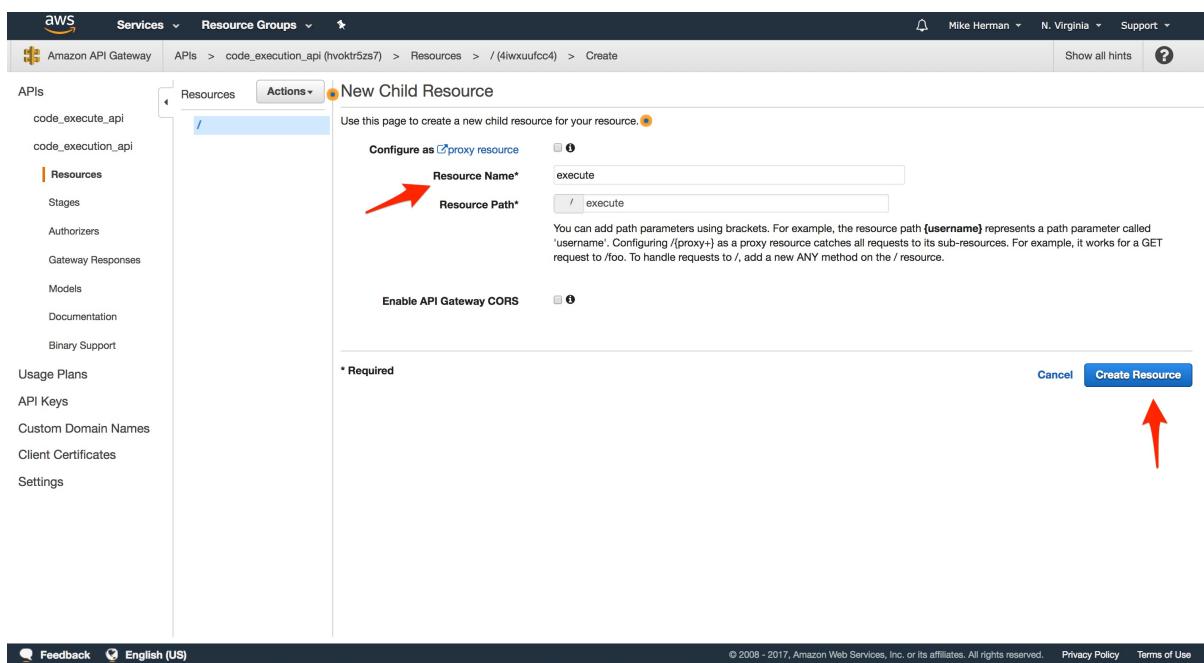
[Feedback](#) [English \(US\)](#) © 2008 - 2017, Amazon Web Services, Inc. or its affiliates. All rights reserved. [Privacy Policy](#) [Terms of Use](#)

Then, create the API.

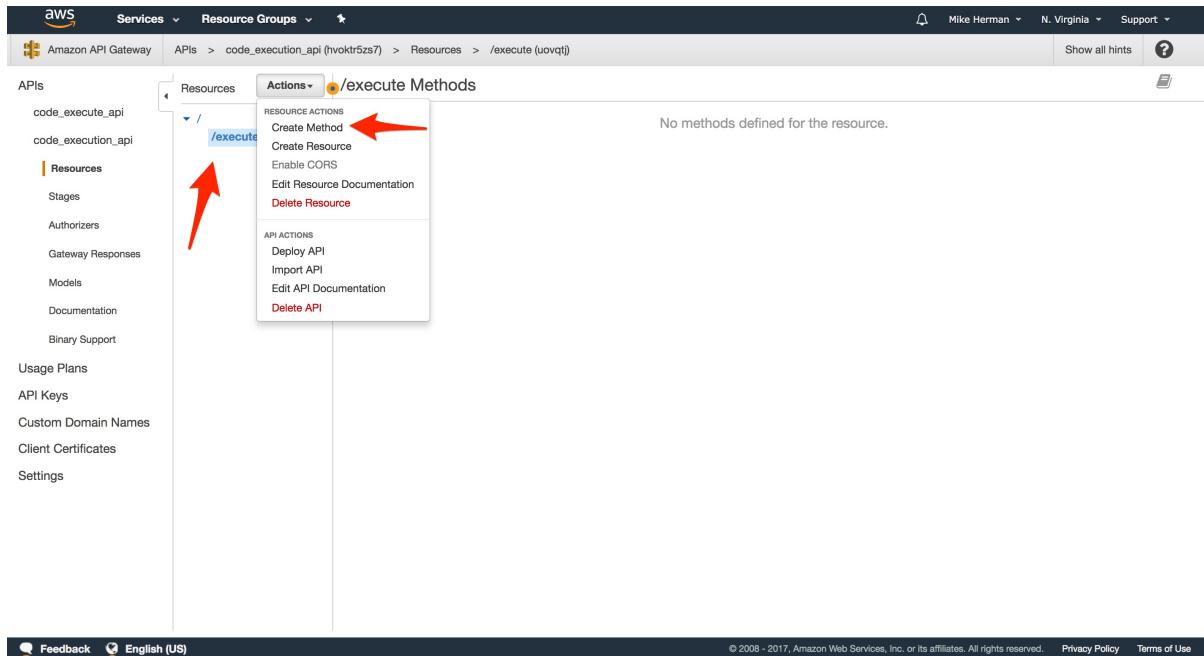
Select "Create Resource" from the "Actions" drop-down.



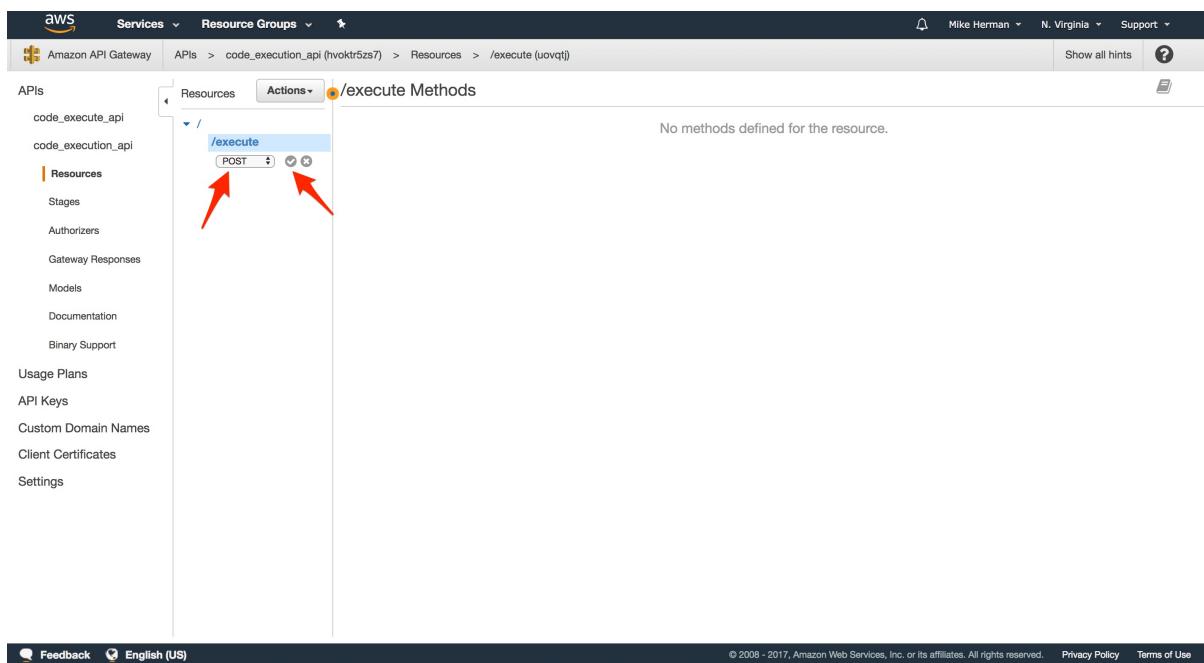
Name the resource `execute`, and then click "Create Resource".



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-east-1" region in the drop-down, and enter the name of the Lambda function that you just created.

APIs > code_execute_api > Resources > /execute (uvqtj) > POST

Choose the integration point for your new method.

Integration type: Lambda Function (selected)

Use Lambda Proxy integration:

Lambda Region: us-east-1

Lambda Function: execute_python3_code

Use Default Timeout:

Save

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".

APIs > code_execute_api > Resources > /execute (uvqtj) > POST

Method Request

Auth: NONE
ARN: arn:aws:execute-api:us-east-1:046505967931:hwoktr5zs7/*POST/execute

Integration Request

Type: LAMBDA
Region: us-east-1

Method Response

HTTP Status: 200
Models: application/json => Empty

Integration Response

HTTP status pattern: -
Output passthrough: Yes

Lambda execute_python3_code

TEST

Client

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:

The screenshot shows the AWS Lambda Test interface. On the left, the navigation pane includes 'APIs', 'Resources' (selected), 'Stages', 'Authorizers', 'Gateway Responses', 'Models', 'Documentation', 'Binary Support', 'Usage Plans', 'API Keys', 'Custom Domain Names', 'Client Certificates', and 'Settings'. The main area is titled 'Method Execution /execute - POST - Method Test'. It contains sections for 'Path', 'Query Strings' (with a red arrow pointing to a text input field containing 'true'), 'Headers', 'Stage Variables' (with a red arrow pointing to a note about stage variables), 'Request Body' (containing code: 1 - [{ 2 - "answer": "def sum(x,y):\n return x+y"\n 3 - }]), and 'Logs' (showing execution logs). The status bar at the bottom indicates '© 2008 - 2017, Amazon Web Services, Inc. or its affiliates. All rights reserved. Privacy Policy Terms of Use'.

Enable CORS

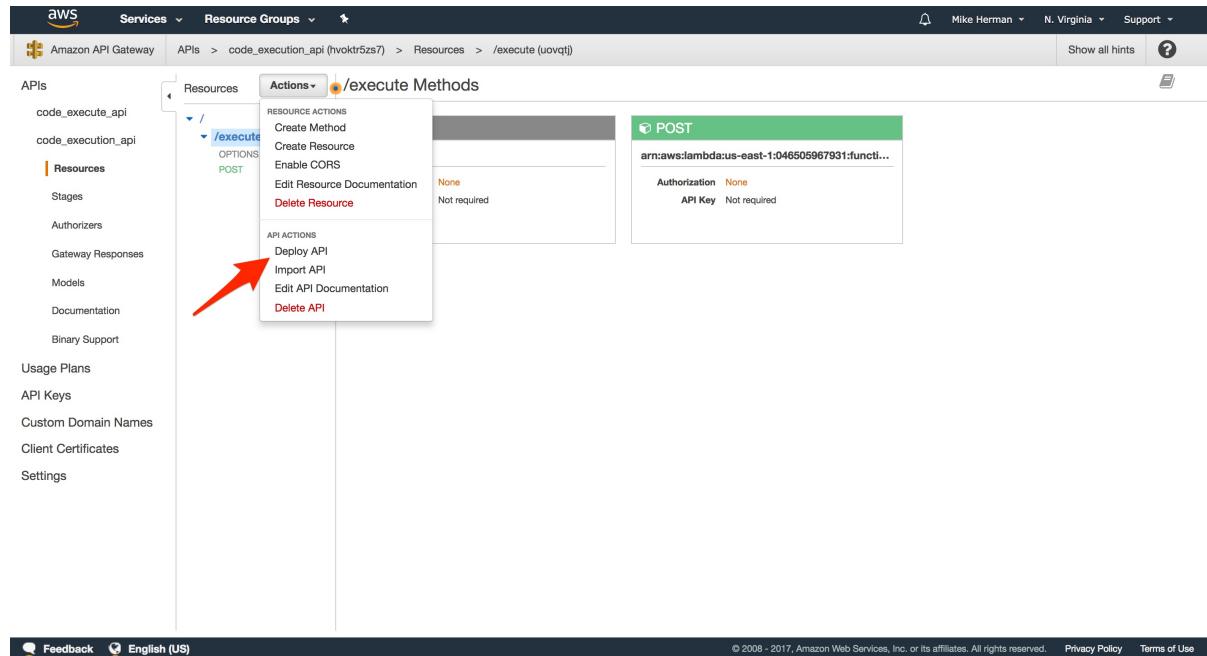
Next, we need to enable [CORS](#) so that we can POST to the API endpoint from another domain. With the resource highlighted, select "Enable CORS" from the "Actions" drop-down:

The screenshot shows the AWS Lambda Actions menu for the '/execute POST' method. The 'Actions' dropdown is open, showing options: 'Create Method', 'Create Resource', 'Enable CORS' (which is highlighted with a red arrow), 'Edit Resource Documentation', and 'Delete Resource'. Below this, under 'API ACTIONS', are 'Deploy API', 'Import API', 'Edit API Documentation', and 'Delete API'. The status bar at the bottom indicates '© 2008 - 2017, Amazon Web Services, Inc. or its affiliates. All rights reserved. Privacy Policy Terms of Use'.

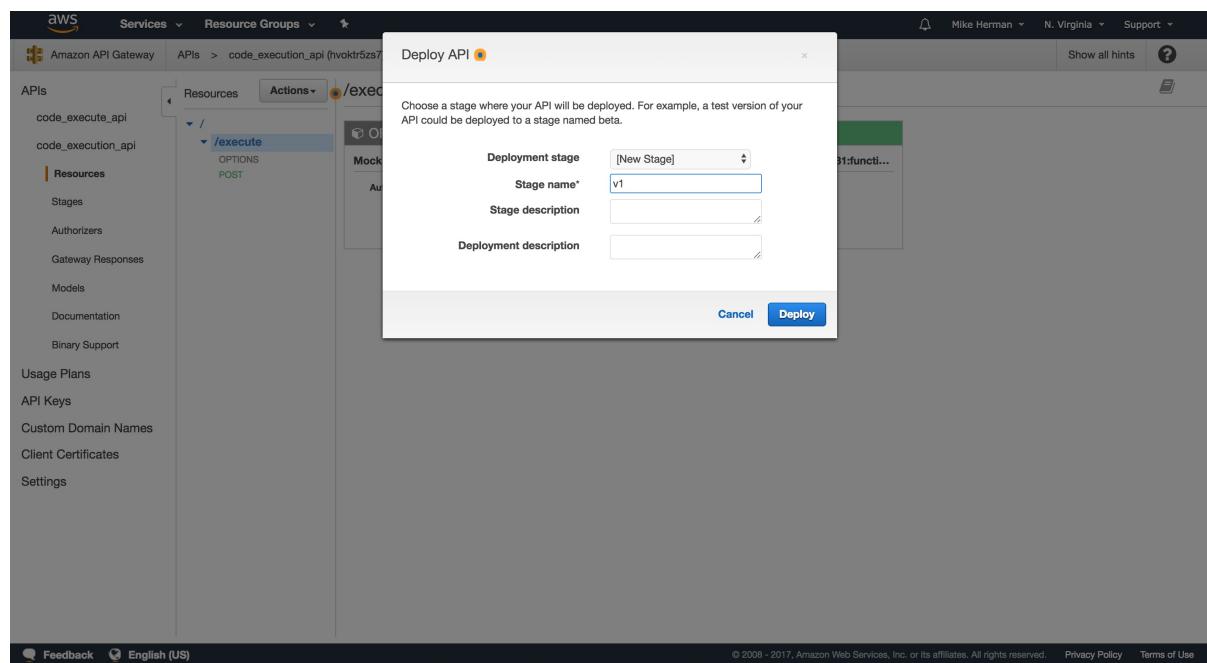
Just keep the defaults for now since we're still testing the API. Click the "Enable CORS and replace existing CORS headers" button.

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://hvoktr5zs7.execute-api.us-east-1.amazonaws.com/v1/execute
```

The screenshot shows the AWS API Gateway Stage Editor for the 'v1' stage. On the left sidebar, under 'Stages', 'v1' is selected. The main area is titled 'v1 Stage Editor' and contains tabs for Settings, Logs, Stage Variables, SDK Generation, Export, Deployment History, Documentation History, and Canary. The 'Settings' tab is active. It includes sections for Cache Settings, Default Method Throttling (with Rate set to 10000 requests per second and Burst set to 5000 requests), and Client Certificate (set to None). At the bottom right of the editor is a 'Save Changes' button. Above the editor, the 'Invoke URL' is displayed as a blue link: <https://hvoktr5zs7.execute-api.us-east-1.amazonaws.com/v1>.

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+999999999999"}' \
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-service python manage.py recreate_db  
  
$ docker-compose -f docker-compose-dev.yml \  
run users-service 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 environment variable to `docker-compose-dev.yml`:

```
client:
  container_name: client
  build:
    context: ./services/client
    dockerfile: Dockerfile-dev
  volumes:
    - './services/client:/usr/src/app'
  ports:
    - '3007:3000' # expose ports - HOST:CONTAINER
  environment:
    - NODE_ENV=development
    - REACT_APP_USERS_SERVICE_URL=${REACT_APP_USERS_SERVICE_URL}
    - REACT_APP_API_GATEWAY_URL=${REACT_APP_API_GATEWAY_URL}
  depends_on:
    - users-service
  links:
    - users-service
```

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`.

The screenshot shows the TestDriven.io platform. At the top, there's a navigation bar with links for 'TestDriven.io', 'About', 'Users', 'User Status', 'Swagger', and 'Log Out'. Below the navigation is a section titled 'Exercises' with a sub-instruction: 'Define a function called sum that takes two integers as arguments and returns their sum.' A code editor window contains the Python code shown above. A 'Run Code' button is at the bottom of the editor. Below the editor is a browser developer tools Network tab, with a red arrow pointing to the 'top' entry in the list. The Network tab shows a response object with the following details:

```
Exercises.jsx:52
{
  "data": true,
  "status": 200,
  "statusText": "",
  "headers": {},
  "config": {
    "transformRequest: (...), transformResponse: (...), timeout: 0, xsrfCookieName: "XSRF-TOKEN", adapter: f, ...},
  "data": true
}
headers: {content-type: "application/json"}
request: XMLHttpRequest {readyState: 4, timeout: 0, withCredentials: false, upload: XMLHttpRequestUpload, onreadystatechange: f, ...}
status: 200
statusText: ""
__proto__: Object
```

Update the code to:

```
def sum(num1, num2):
    return num1
```

Make sure the value of `data` is now `false`.

TestDriven.io About Users User Status Swagger Log Out

Exercises

Define a function called sum that takes two integers as arguments and returns their sum.

```
1+ def sum(num1, num2):
2     return num1
```

Run Code

Elements Console Sources Network Performance Memory Application Security Audits AdBlock

top Filter Default levels Exercises.jsx:52

```
▼ {data: false, status: 200, statusText: "", headers: {}, config: {}, ...}
  ► config: {transformRequest: {}, transformResponse: {}, timeout: 0, xsrfCookieName: "XSRF-TOKEN", adapter: f, ...}
    data: false
  ► headers: {content-type: "application/json"}
  ► request: XMLHttpRequest {readyState: 4, timeout: 0, withCredentials: false, upload: XMLHttpRequestUpload, onreadystatechange: f, ...}
    status: 200
    statusText: ""
  ► __proto__: Object
```

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>
      &nbsp;
      <Glyphicon glyph="repeat" className="glyphicon-spin"/>
      &nbsp;
      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 = false;
  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!

TestDriven.io About Users User Status Swagger Log Out

Exercises

Define a function called sum that takes two integers as arguments and returns their sum.

```
1- def sum(n1, n2):
2-     return n1 + n2
```

[Run Code](#)

 Grading...

Copyright 2017 TestDriven.io.

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>
  &nbsp;
  <Glyphicon glyph="ok" className="glyphicon-correct"/>
  &nbsp;
  Correct!
</h4>
}
{this.state.editor.showIncorrect &&
<h4>
  &nbsp;
  <Glyphicon glyph="remove" className="glyphicon-incorrect"/>
  &nbsp;
  Incorrect!
</h4>
}
```

Test it out again!

TestDriven.io About Users User Status Swagger Log Out

Exercises

Define a function called sum that takes two integers as arguments and returns their sum.

```
1- def sum(num1, num2):
2     return num1 + num2
```

Run Code

✓ Correct!



Copyright 2017 TestDriven.io.

TestDriven.io About Users User Status Swagger Log Out

Exercises

Define a function called sum that takes two integers as arguments and returns their sum.

```
1- def sum(num1, num2):
2     return num1 + num2 + 99999999999
```

Run Code

✗ Incorrect!



Copyright 2017 TestDriven.io.

Make sure the end-to-end tests work as well:

```
$ 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

/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 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 environment variable to `docker-compose-dev.yml`:

```

client:
  container_name: client
  build:
    context: ./services/client
    dockerfile: Dockerfile-dev
  volumes:
    - './services/client:/usr/src/app'
  ports:
    - '3007:3000' # expose ports - HOST:CONTAINER
  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}

depends_on:
- users-service
links:
- users-service

```

Add a new `location` to `services/nginx/flask.conf`:

```

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`:

```

@manager.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.js*. To fix, we'll need to mock 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 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-service 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-service 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:

```

language: node_js
node_js: '8'

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=1.14.0
    - COMMIT=${TRAVIS_COMMIT::8}
    - MAIN_REPO=https://github.com/realpython/testdriven-app.git
    - USERS=test-driven-users
    - USERS_REPO=${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_VERSION}/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=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}
  ports:
    - '3007:3000'
  depends_on:
    - users-service
  links:
    - users-service

```

Then, add the `exercises` and `exercises-db` services:

```

exercises:
  container_name: exercises

```

```

build:
  context: ./services/exercises
  dockerfile: Dockerfile-stage
ports:
  - 5002:5000
environment:
  - APP_SETTINGS=project.config.StagingConfig
  - DATABASE_URL=postgres://postgres:postgres@exercises-db:5432/exercises_staging
  - DATABASE_TEST_URL=postgres://postgres:postgres@exercises-db:5432/exercises_te
st
depends_on:
  - users-service
  - exercises-db
links:
  - users-service
  - exercises-db

exercises-db:
  container_name: exercises-db
  build:
    context: ./services/exercises/project/db
    dockerfile: Dockerfile
  ports:
    - 5436:5432
  environment:
    - POSTGRES_USER=postgres
    - POSTGRES_PASSWORD=postgres

```

Set the environment variables in *services/client/Dockerfile-stage*:

```

# add environment variables
ARG NODE_ENV
ARG REACT_APP_USERS_SERVICE_URL
ARG REACT_APP_EXERCISES_SERVICE_URL
ARG REACT_APP_API_GATEWAY_URL
ENV NODE_ENV $NODE_ENV
ENV REACT_APP_USERS_SERVICE_URL $REACT_APP_USERS_SERVICE_URL
ENV REACT_APP_EXERCISES_SERVICE_URL $REACT_APP_EXERCISES_SERVICE_URL
ENV REACT_APP_API_GATEWAY_URL $REACT_APP_API_GATEWAY_URL

```

Add an *entrypoint-stage.sh* file to "services/exercises/project":

```

#!/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

```

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://testdriven-staging-alb-1378944177.us-east-1.elb.amazonaws.com"
    export REACT_APP_EXERCISES_SERVICE_URL="http://testdriven-staging-alb-1378944177.us-east-1.elb.amazonaws.com"
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-$DOCKER_ENV --build-arg REACT_APP_USERS_SERVICE_URL=$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
fi
```

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.

The screenshot shows the AWS ECR console interface. On the left, there's a sidebar with 'Amazon ECS' selected, followed by 'Clusters', 'Task Definitions', and 'Repositories'. Under 'Repositories', 'test-driven-exercises' is selected. The main area shows the repository details: 'Repository ARN' is `arn:aws:ecr:us-east-1:046505967931:repository/test-driven-exercises` and 'Repository URI' is `046505967931.dkr.ecr.us-east-1.amazonaws.com/test-driven-exercises`. Below this is a 'View Push Commands' button. A navigation bar at the top includes 'Services', 'Resource Groups', and user information 'Mike Herman'.

Images tab is selected. The table below lists images:

Image tags	Digest	Size (MiB)	Pushed at
<input type="checkbox"/> staging	view all sha256:10447669089cc326538cbe00bcb5e0af7d6efd743ac8e660f18f...	291.26	2017-11-29 19:55:45 -0700

At the bottom, there are links for 'Feedback', 'English (US)', '© 2008 - 2017, Amazon Web Services, Inc. or its affiliates. All rights reserved.', 'Privacy Policy', and 'Terms of Use'.

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-east-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_staging"
        }
      ],
      "links": [
        "exercises-db"
      ],
      "logConfiguration": {
        "logDriver": "awslogs",
        "options": {
          "awslogs-group": "testdriven-exercises-stage",
          "awslogs-region": "us-east-1"
        }
      }
    }
  ]
}
```

```
 },
{
  "name": "exercises-db",
  "image": "%s.dkr.ecr.us-east-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-east-1"
    }
  }
],
"family": "testdriven-exercises-stage-td"
}
```

Be sure to 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 both images and Task Definitions are created.

The screenshot shows the AWS ECS Task Definitions page. At the top, there's a navigation bar with the AWS logo, Services dropdown, Resource Groups dropdown, and user info (Mike Herman, N. Virginia, Support). A prominent banner at the top right announces Fargate, a new ECS launch type. Below the banner, the main content area has a heading 'Task Definitions' and a sub-instruction: 'Task definitions specify the container information for your application, such as how many containers are part of your task, what resources they will use, how they are linked together, and which host ports they will use.' A 'Create new Task Definition' button is visible. The main table lists one task definition:

Task Definition	Latest revision status
testdriven-exercises-stage-td	ACTIVE

At the bottom of the page, there are links for Feedback, English (US), and footer links: © 2008 - 2017, Amazon Web Services, Inc. or its affiliates. All rights reserved., Privacy Policy, Terms of Use.

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`

The screenshot shows the AWS Lambda Rules editor for an Application Load Balancer (ALB) named 'testdriven-staging-alb'. The interface includes a toolbar with icons for back, forward, search, and delete, and a top bar showing 'Rules for: testdriven-staging-alb' and 'HTTP 80'. Below is a table of rules:

Index	ARN	IF	THEN
1	ARN	IF ✓ Path is /exercises*	THEN Forward to testdriven-exercises-stage-tg
2	ARN	IF ✓ Path is /swagger*	THEN Forward to testdriven-swagger-stage-tg
3	ARN	IF ✓ Path is /auth*	THEN Forward to testdriven-users-stage-tg
4	ARN	IF ✓ Path is /users*	THEN Forward to testdriven-users-stage-tg
5	ARN	IF ✓ Path is /users/ping	THEN Forward to testdriven-users-stage-tg
Last	HTTP 80: default action <small>This rule cannot be moved or deleted</small>	IF ✓ Requests otherwise not routed	THEN Forward to testdriven-client-stage-tg

At the bottom, there are links for 'Feedback', 'English (US)', and copyright information: '© 2008 - 2017, Amazon Web Services, Inc. or its affiliates. All rights reserved.' followed by 'Privacy Policy' and 'Terms of Use'.

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".

Navigate to the [EC2 Dashboard](#), and click "Target Groups". Make sure `testdriven-exercises-stage-tg` has a single registered instance each. 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
```

```
$ 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

/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 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-service 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-service 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-stage
  args:
    - 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}
  ports:
    - '3007:3000'
  depends_on:
    - users-service
  links:
    - users-service

```

Then, add the `exercises` and `exercises-db` services:

```

swagger:
  container_name: swagger
  build:
    context: ./services/swagger
    dockerfile: Dockerfile-prod
  ports:
    - '3008:8080'
  environment:
    - URL=swagger.json
  depends_on:
    - users-service

exercises:
  container_name: exercises
  build:
    context: ./services/exercises
    dockerfile: Dockerfile-prod
  ports:
    - 5002:5000
  environment:
    - APP_SETTINGS=project.config.ProductionConfig
    - DATABASE_URL=postgres://postgres:postgres@exercises-db:5432/exercises_prod
    - DATABASE_TEST_URL=postgres://postgres:postgres@exercises-db:5432/exercises_te
st
  depends_on:
    - users-service
    - exercises-db
  links:

```

```

- users-service
- exercises-db

exercises-db:
  container_name: exercises-db
  build:
    context: ./services/exercises/project/db
    dockerfile: Dockerfile
  ports:
    - 5436:5432
  environment:
    - POSTGRES_USER=postgres
    - POSTGRES_PASSWORD=postgres

```

Set the environment variables in *services/client/Dockerfile-prod*:

```

# add environment variables
ARG NODE_ENV
ARG REACT_APP_USERS_SERVICE_URL
ARG REACT_APP_EXERCISES_SERVICE_URL
ARG REACT_APP_API_GATEWAY_URL
ENV NODE_ENV $NODE_ENV
ENV REACT_APP_USERS_SERVICE_URL $REACT_APP_USERS_SERVICE_URL
ENV REACT_APP_EXERCISES_SERVICE_URL $REACT_APP_EXERCISES_SERVICE_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://testdriven-production-alb-484275327.us-east-1.elb.amazonaws.com"
    export REACT_APP_EXERCISES_SERVICE_URL="http://testdriven-production-alb-484275327.us-east-1.elb.amazonaws.com"
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.

Image tags	Digest	Size (MiB)	Pushed at
<input type="checkbox"/> production	sha256:00053a965cf39c2776ef2ccce69780402658076627519c6472f7...	291.26	2017-11-30 07:35:25 -0700
<input type="checkbox"/> staging	sha256:29dabbff0c1008389da409bb80d97eff9bd2a288c6b21d3dd81...	291.26	2017-11-30 07:33:46 -0700
<input type="checkbox"/>	sha256:eb3822628e300fb945066a010aeb4b4a39e927d6956b235ad47...	291.26	2017-11-30 07:11:24 -0700
<input type="checkbox"/>	sha256:ccc40c76416c3efd55b12689661427451bbb0574224d0cb92c...	291.26	2017-11-29 22:15:23 -0700
<input type="checkbox"/>	sha256:e87d82c53ae2a6353f0fde84b0ad14fa15ddbc68b689187000e...	291.26	2017-11-29 21:59:47 -0700
<input type="checkbox"/>	sha256:e16c56d6b0634e2dc07af0386a44ba42c546f85810e2fe50b584...	291.26	2017-11-29 21:31:40 -0700
<input type="checkbox"/>	sha256:497c29dfaf2c3b0b6491d4f6fb21c035015c870487374294682...	291.26	2017-11-29 21:13:45 -0700
<input type="checkbox"/>	sha256:10447669089cc326538cbc00ccb5e0af7d6ef743ac8e660f18f...	291.26	2017-11-29 19:55:45 -0700

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_stage_taskdefinition.json`:

```
{
  "containerDefinitions": [
    {
      "name": "exercises",
      "image": "%s.dkr.ecr.us-east-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-east-1"
        }
      }
    ],
    "family": "testdriven-exercises-prod-td"
  }
```

Be sure to 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.

The screenshot shows the AWS ECS Task Definitions page. The left sidebar has links for Amazon ECS Clusters, Task Definitions (which is selected and highlighted in orange), and Repositories. The main content area shows a breadcrumb trail: Task Definitions > testdriven-exercises-prod-td > status > ACTIVE. Below this, the title 'Task Definition Name : testdriven-exercises-prod-td' is displayed. A note says 'Select a revision for more details'. There are buttons for 'Create new revision' and 'Actions'. The status is set to 'Active'. A table lists one task definition:

Task Definition Name : Revision	Status
testdriven-exercises-prod-td:1	Active

At the bottom, there are links for 'Feedback' and 'English (US)', and a footer with copyright information: '© 2008 - 2017, Amazon Web Services, Inc. or its affiliates. All rights reserved.' and links for 'Privacy Policy' and 'Terms of Use'.

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`

The screenshot shows the AWS CloudFront Rules configuration for the 'testdriven-production-alb' distribution. It displays five rules:

- Rule 1:** ARN: IF Path is /exercises* THEN Forward to testdriven-exercises-prod-tg
- Rule 2:** ARN: IF Path is /swagger* THEN Forward to testdriven-swagger-prod-tg
- Rule 3:** ARN: IF Path is /auth* THEN Forward to testdriven-users-prod-tg
- Rule 4:** ARN: IF Path is /users* THEN Forward to testdriven-users-prod-tg
- Rule 5:** ARN: IF Path is /users/ping THEN Forward to testdriven-users-prod-tg

Below these rules is a **default action** rule:

- Last Rule:** HTTP 80: default action
IF Requests otherwise not routed THEN Forward to testdriven-client-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

- "DB Engine Version": PostgreSQL 9.6.3-R1
- "DB Instance Class": db.t2.micro
- "Multi-AZ Deployment": No
- "Storage Type": General Purpose (SSD)
- "Allocated Storage": 20 GB
- "DB Instance Identifier": testdriven-exercises-production
- "Master Username": webapp
- "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-east-1a or us-east-1b .

The screenshot shows the 'Configure advanced settings' step in the AWS RDS console. The left sidebar shows the 'Amazon RDS' dashboard with various options like Instances, Clusters, and Performance Insights. The main panel is titled 'Configure advanced settings' and is divided into sections: 'Network & Security', 'Subnet group info', 'Public accessibility info', 'Availability zone info', and 'VPC security groups'. Red arrows highlight three specific fields: 'Default VPC (vpc-46e1103f)' in the Network & Security section, 'us-east-1a' in the Availability zone info section, and 'testdriven-security-group (VPC)' in the VPC security groups section.

Change the DB name to `exercises_prod` and then create the new database.

You can quickly check the status via:

```
$ aws rds describe-db-instances \
--db-instance-identifier testdriven-exercises-production \
--query 'DBInstances[].[DBInstanceStatus:DBInstanceState]'

[
  {
    "DBInstanceState": "creating"
  }
]
```

Then, once the status is "available", grab the address:

```
$ aws rds 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.

Environment Variables

Notice that the values are not escaped when your builds are executed. Special characters (for bash) should be escaped accordingly.

AWS_ACCESS_KEY_ID		
AWS_ACCOUNT_ID		
AWS_RDS_EXERCISES_URI		
AWS_RDS_URI		
AWS_SECRET_ACCESS_KEY		
PRODUCTION_SECRET_KEY		
Name	Value	OFF	Display value in build log
			Add

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:

Container Definitions 

Container Name	Image	CPU Units	Hard/Soft memory limits (MiB)	Essential
exercises	046505967931.dkr.ecr.us-east-1....	0	--300	true

Details

Port Mappings

Host Port	Container Port	Protocol
0	5000	tcp

Environment Variables

Key	Value
APP_SETTINGS	project.config.ProductionConfig
DATABASE_TEST_URL	postgres://postgres:postgres@exercises-db:5432/exercises_test
DATABASE_URL	[REDACTED]

Docker labels

Key	Value
No docker labels	

Extra hosts

Hostname	IP address
No host entries	

Mount Points

Container Path	Source Volume	Read only
No Mount Points		

Volumes from

Source Container	Read only
No volumes from	

Ulimits

Name	Soft limit	Hard limit
No ulimit		

Log Configuration

Log driver: awslogs

Key	Value
awslogs-group	testdriven-exercises-prod
awslogs-region	us-east-1

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".

Navigate to the [EC2 Dashboard](#), and click "Target Groups". Make sure `testdriven-exercises-stage-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.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/exercises again and you should see the users.

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 the `exercises` part of `deploy_cluster` function in `docker-deploy-stage.sh` to call `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 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
```

```
$ 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

/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 09s)
```

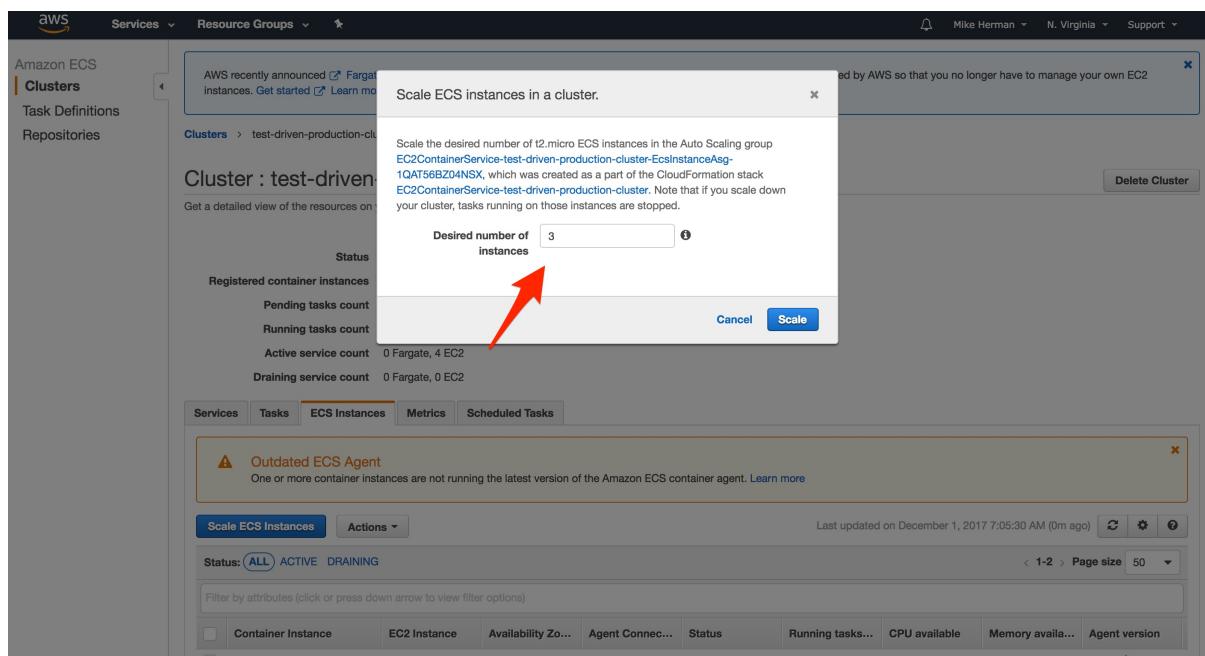
Autoscaling

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 Instances" 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.

Screenshot of the AWS CloudFormation 'Update Service' step 3: Set Auto Scaling (optional) configuration.

The configuration includes:

- Service Auto Scaling:** Configure Service Auto Scaling to adjust your service's desired count.
- Minimum number of tasks:** 4
- Desired number of tasks:** 6 (highlighted by a red arrow)
- Maximum number of tasks:** 9 (highlighted by a red arrow)
- IAM role for Service Auto Scaling:** ecsAutoscaleRole

A modal window titled 'Add policy' is open, showing the configuration for a scaling policy:

- Policy name:** scaling-up (highlighted by a red arrow)
- Execute policy when:** Create new Alarm (selected)
- Alarm name:** cpu-threshold-met (highlighted by a red arrow)
- ECS service metric:** CPUUtilization
- Alarm threshold:** Average of CPUUtilization >= 75 for 1 consecutive periods of 1 minute (highlighted by a red arrow)
- Save button:** (highlighted by a red arrow)

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.

Cluster : test-driven-staging-cluster

[Delete Cluster](#)

Get a detailed view of the resources on your cluster.

Status ACTIVE

Registered container instances 4

- Pending tasks count 0 Fargate, 0 EC2
- Running tasks count 0 Fargate, 4 EC2
- Active service count 0 Fargate, 4 EC2
- Draining service count 0 Fargate, 0 EC2

Services	Tasks	ECS Instances	Metrics	Scheduled Tasks
Run new Task	Stop			
Last updated on December 1, 2017 7:24:24 AM (1m ago) Edit ?				
Desired task status: Running Stopped 1 selected				
Filter in this page Launch type ALL ▼ < 1-4 > Page size 50 ▼				
Task	Task definition	Container inst...	Last status	Desired status
<input type="checkbox"/> 69707fa6-6c6d...	testdriven-exercises-stage-td:9	e7272a2d-451...	RUNNING	RUNNING
<input type="checkbox"/> 9d78e392-40c...	testdriven-users-stage-td:26	39a259d0-61d...	RUNNING	RUNNING
<input type="checkbox"/> b9d80c2a-dcc...	testdriven-client-stage-td:26	39a259d0-61d...	RUNNING	RUNNING
<input checked="" type="checkbox"/> f55e4446-c4d5...	testdriven-swagger-stage-td:25	d1c78ea6-b22...	RUNNING	RUNNING

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.

AWS Services Resource Groups [Mike Herman](#) N. Virginia Support

Amazon ECS Clusters Task Definitions Repositories

Clusters > test-driven-staging-cluster > Service: testdriven-swagger-stage-service

Service : testdriven-swagger-stage-service

Cluster test-driven-staging-cluster Status ACTIVE Desired count 1 Pending count 0 Running count 1

Task definition testdriven-swagger-stage-td:25 Launch type EC2 Service role ecsServiceRole

Details Tasks Events Auto Scaling Deployments Metrics Logs

Last updated on December 1, 2017 7:33:17 AM (0m ago) < 1-100 >

Event Id	Event Time	Message
202681bc-44a7-4951-b81a-d4693cac0a50	2017-12-01 07:33:12 -0700	service testdriven-swagger-stage-service registered 1 targets in target-group testdriven-swagger-stage-tg
acdfe652-9ca8-43b7-84bb-3c858b0b935	2017-12-01 07:33:02 -0700	service testdriven-swagger-stage-service has started 1 tasks: task e83533d-b17b-42a4-99c8-2dc7a333bf3.
d8f1686d-9ed4-4882-981e-3bf4be592dc9	2017-12-01 07:32:20 -0700	service testdriven-swagger-stage-service has begun draining connections on 1 tasks.
582c9a2c-d452-42d7-99fd-6d3805fc3e9a	2017-12-01 07:32:20 -0700	service testdriven-swagger-stage-service deregistered 1 targets in target-group testdriven-swagger-stage-tg
781a61ac-bffd-49a0-8520-542a3cbe7d5c	2017-12-01 03:54:27 -0700	service testdriven-swagger-stage-service has reached a steady state.
1341cbbb-5287-46da-904e-0f25d99764d8	2017-11-30 21:53:56 -0700	service testdriven-swagger-stage-service has reached a steady state.
96f6490c-9582-4512-914c-b68990e8ee43	2017-11-30 21:53:28 -0700	service testdriven-swagger-stage-service has stopped 1 running tasks: task e2ba11e-b509-48c5-8df1-930f9b0dbb5c.
b25009d0-676d-4323-960a-6b68abafad8	2017-11-30 21:48:22 -0700	service testdriven-swagger-stage-service has begun draining connections on 1 tasks.
485f3583-66a0-45ca-a6ad-f62da3bf3f5	2017-11-30 21:48:22 -0700	service testdriven-swagger-stage-service deregistered 1 targets in target-group testdriven-swagger-stage-tg
3b27d314-e5f7-4e01-ab4f-eaa97ae6dd34	2017-11-30 21:48:08 -0700	service testdriven-swagger-stage-service registered 1 targets in target-group testdriven-swagger-stage-tg
d2c55212-dc62-43e8-9df3-2df28a935684	2017-11-30 21:47:50 -0700	service testdriven-swagger-stage-service has started 1 tasks: task f55e4446-c4d5-43d0-85f7-1c249843352f.
760c9e71-3fc3-4a09-bfb8-87162da95159	2017-11-30 20:29:58 -0700	service testdriven-swagger-stage-service has reached a steady state.

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.

Screenshot of the AWS CloudFront console showing the configuration of a target group.

Services: Resource Groups

Target Group: testdriven-swagger-stage-tg

Registered Targets:

Instance ID	Name	Port	Availability Zone	Status
i-058302138390d8ed6	ECS Instance - EC2ContainerService-test-driven-staging-cluster	32839	us-east-1b	healthy ⓘ
i-0f5287c50d010d5cb	ECS Instance - EC2ContainerService-test-driven-staging-cluster	32802	us-east-1b	draining ⓘ

Availability Zones:

Availability Zone	Target count	Healthy?
us-east-1b	2	Yes

A red arrow points to the "draining" status of the second target in the "Registered Targets" table.

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

```
run exercises python manage.py seed_db

$ docker-compose -f docker-compose-dev.yml \
  run users-service 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-service 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-service 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
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
aws	*	amazonec2	Running	tcp://34.207.173.181:2376		v17.05.0

-ce					
dev	-	virtualbox	Stopped		Unknown
-ce					

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

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. **Message queue:** Add a simple message queue - like [Redis Queue](#) or [RabbitMQ](#)
4. **Swagger:** Add API documentation for the `exercises` service.
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:** We should probably keep track of user scores. You could spin up a new service to handle this. Maybe try a different language or framework.
8. **Summary table:** Did you set up the scores service? How about adding a summary table for individual user scores? Maybe individual users could just view their own scores while an admin can view all user scores.

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!