Design Document

## Workflow

Coordination was done entirely through a set up project on Jira. Sprint meetings would happen at the start of each sprint to determine and assign individual tasks. The aimed sprint length was two days, however during the initial setup this was shorter in order to accommodate that initially things were quite interdependent and required more frequent communication.

As the deploy tasks were mostly independent of the actual application, they were done in parallel to development.

The timeline for sprints was as follows :

### Sprint 1: 04 - 05 Sep

Initial setup tasks including setting up a git repo, creating the models and the intial flask app. Jenkins was set up on an aws instance with a webhook pointing towards it.

### Sprint 2: 05 - 06 Sep

Added basic template routes that would be filled out later. Set up a Jenkins pipeline job that runs unit tests and populated the database with initial records

### Sprint 3: 06 - 08 Sep

Populated the routes. As this was going to take more time than before routes were split between team members with deployment jobs reserved for the next sprint.

### Sprint 4: 08 - 12 Sep (Break over weekend)

Began implementing unit tests for the routes, as well as fixing bugs with some of the remaining ones. A docker compose file was created and implemented in the jenkins pipeline. Created a risk assessment to identify any potential risks and mitigate them.

### Sprint 5: 12 - 14 Sep

Bug fixes, validation and some desirable tasks in the functionality were implemented.

Payments page was also created, with forms, validators and tests. Doccumentation was also implemented.

# Risk Assessment

## Sensitive information

* Names from the person model.
* Card information.
* Address information.

It was determined the Database Server should be kept in a standalone instance without any publicly accessible ports. The Router to access the internal instances responsible for this should be equipped with a firewall to block malicious traffic.

## Passwords

* Passwords should be hashed in the database so in the event the database gets compromised a malicious action wouldn't be able to use it to log into that database
* Passwords should use a salt so that duplicate passwords will not be identifiable across the database.

Access Credentials

Any sort of authentication details should be kept as environment details in the jenkins credentials.

## Copyright

Because we are using this for educational use, we are permitted to use copyright protected images in the project. This is to verify the images are correctly loading on the page.

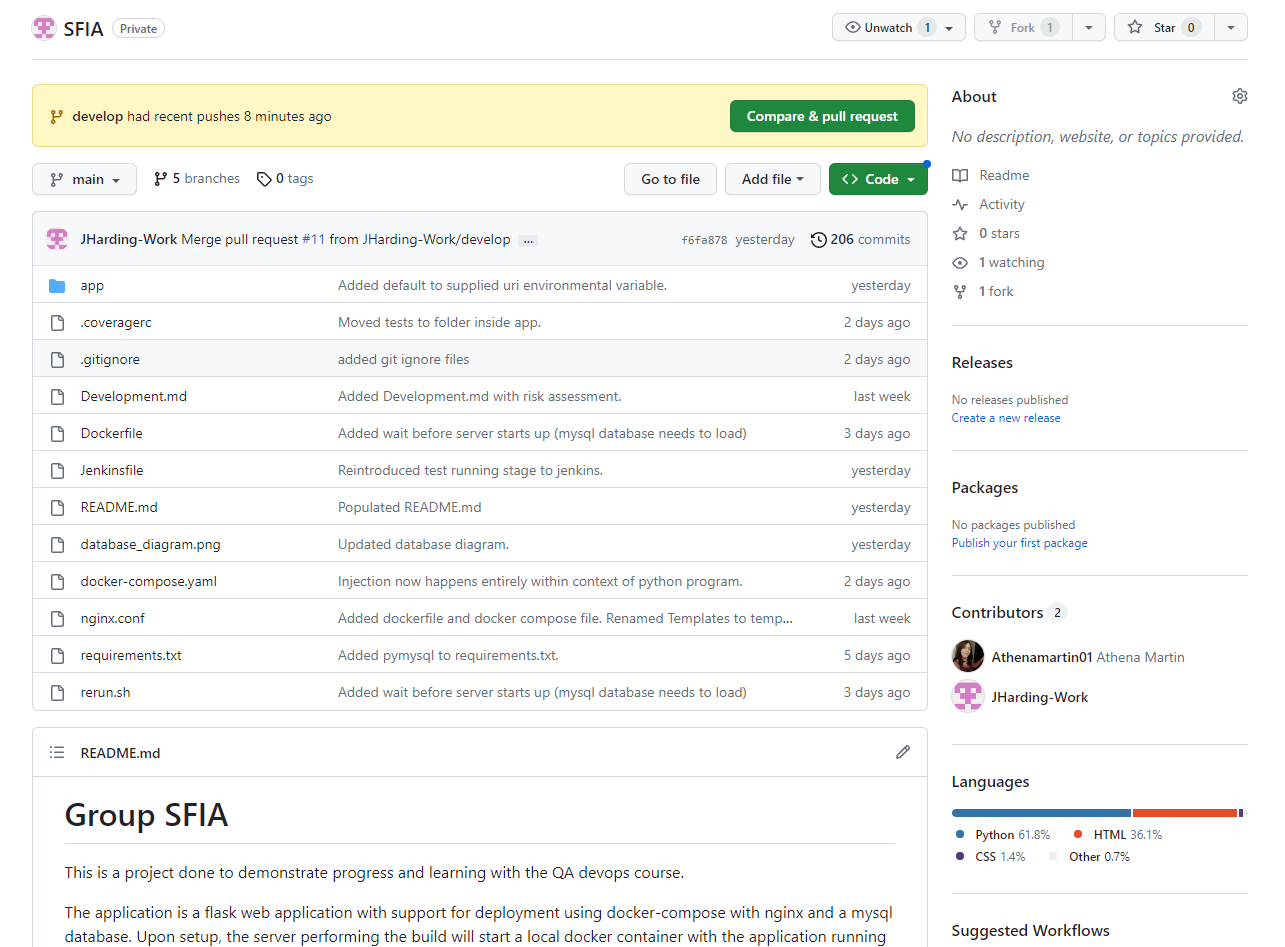
# Build Process

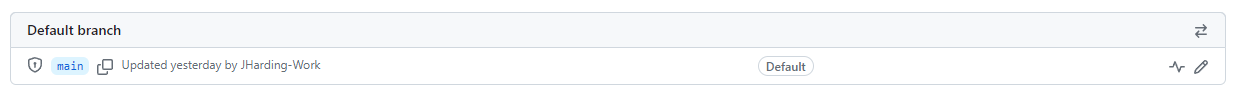
## Build Tools:

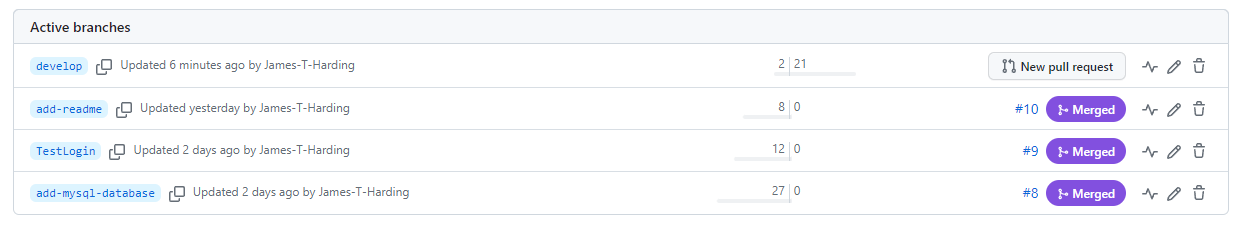
* Source Control: Git
* Infrastructure: AWS
* Continuous Integration: Jenkins
* Containerization: Docker, Docker Compose

## Source Control

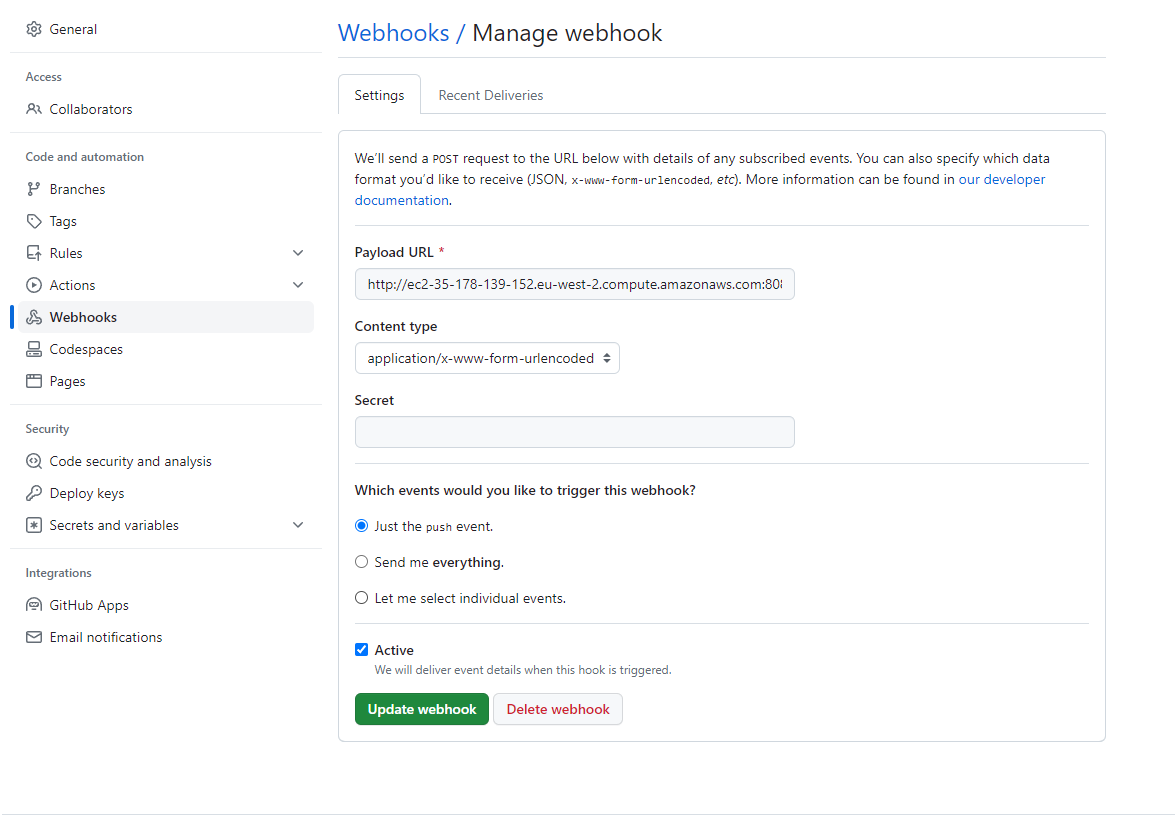
A central git repository was set up with a main and develop branch. Depending on the complexity of the job, changes would either be made by committing to the develop branch or branching off develop.





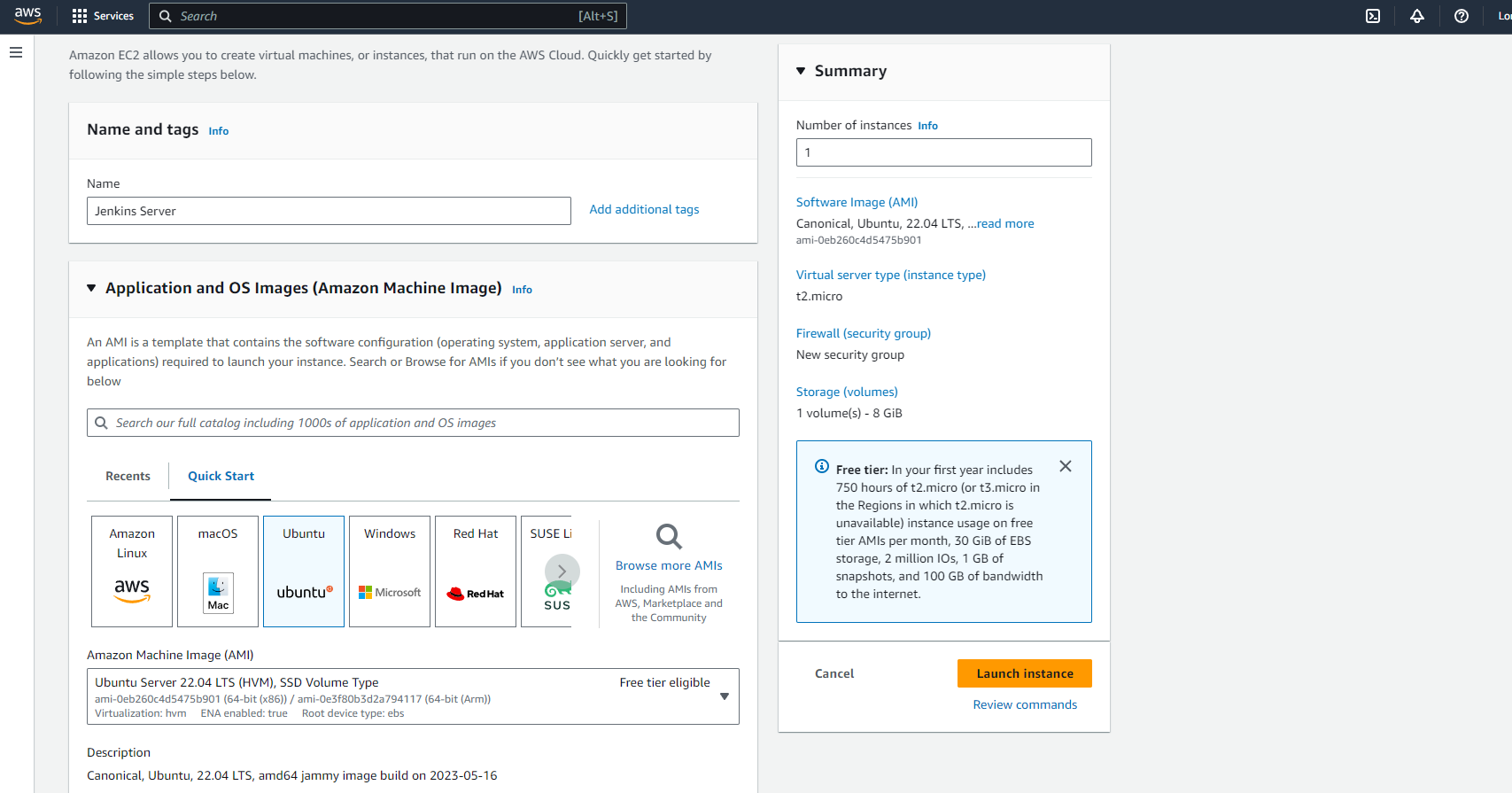


To enable integration with Jenkins, a webhook was set up in the repo that pointed at the Jenkins instance.

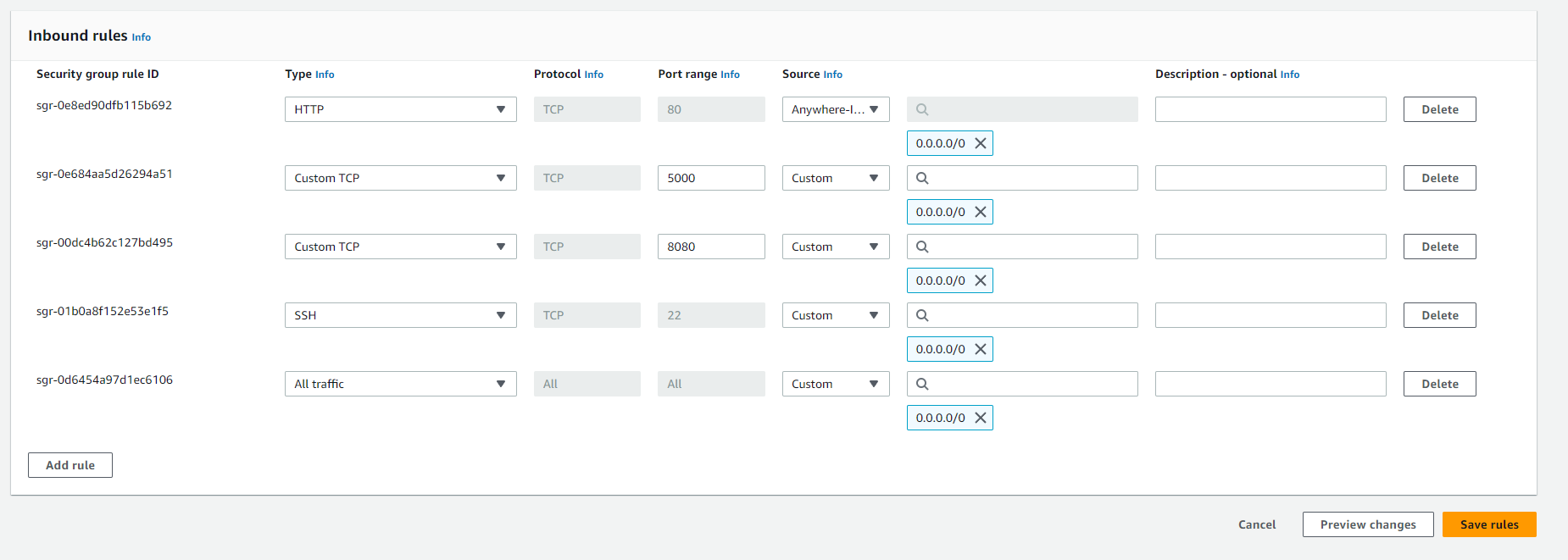


## Infrastructure

Virtual infrastructure was all created from the aws instance creation UI.



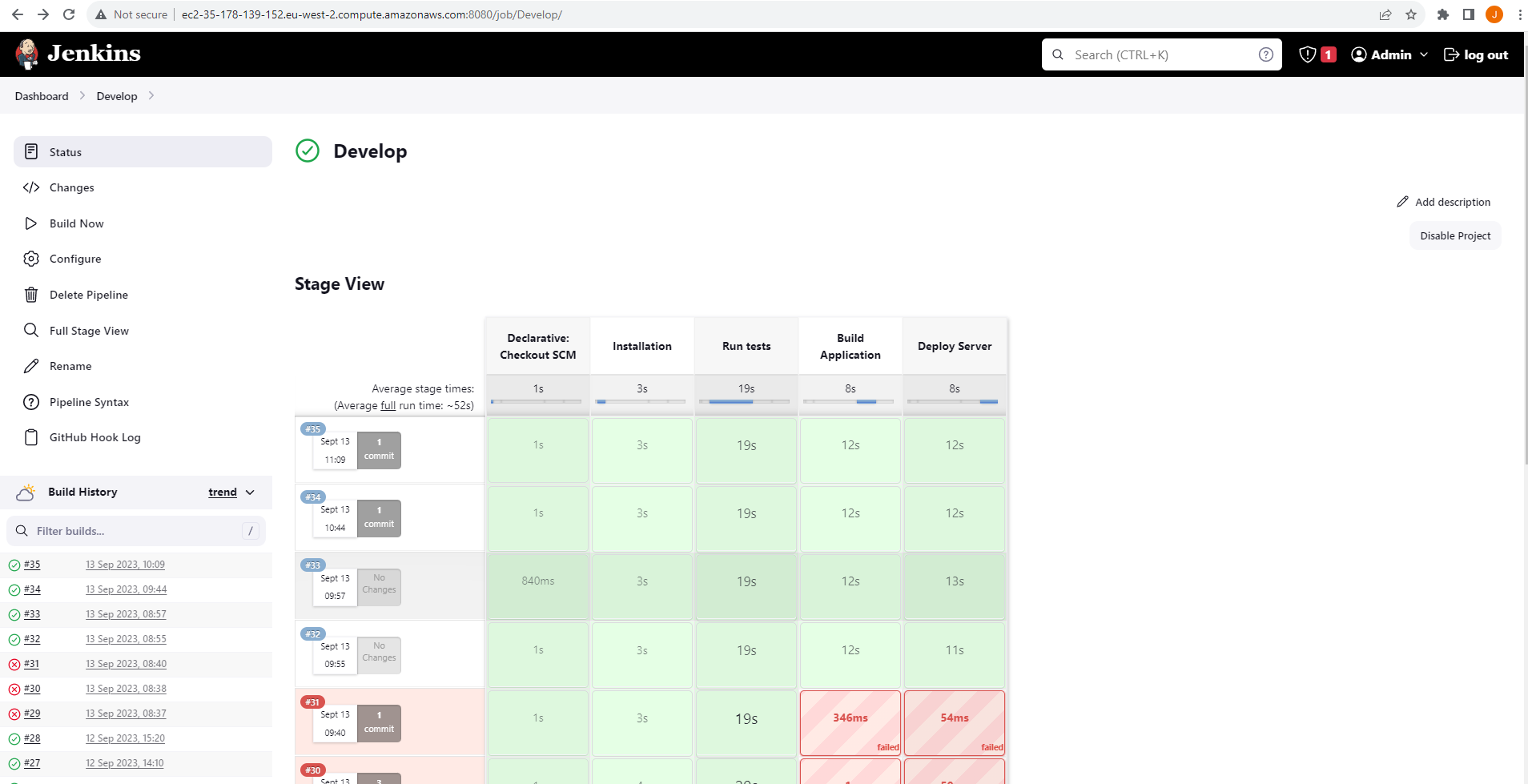
Rules to allow other machines to connect with the virtual instance were added:

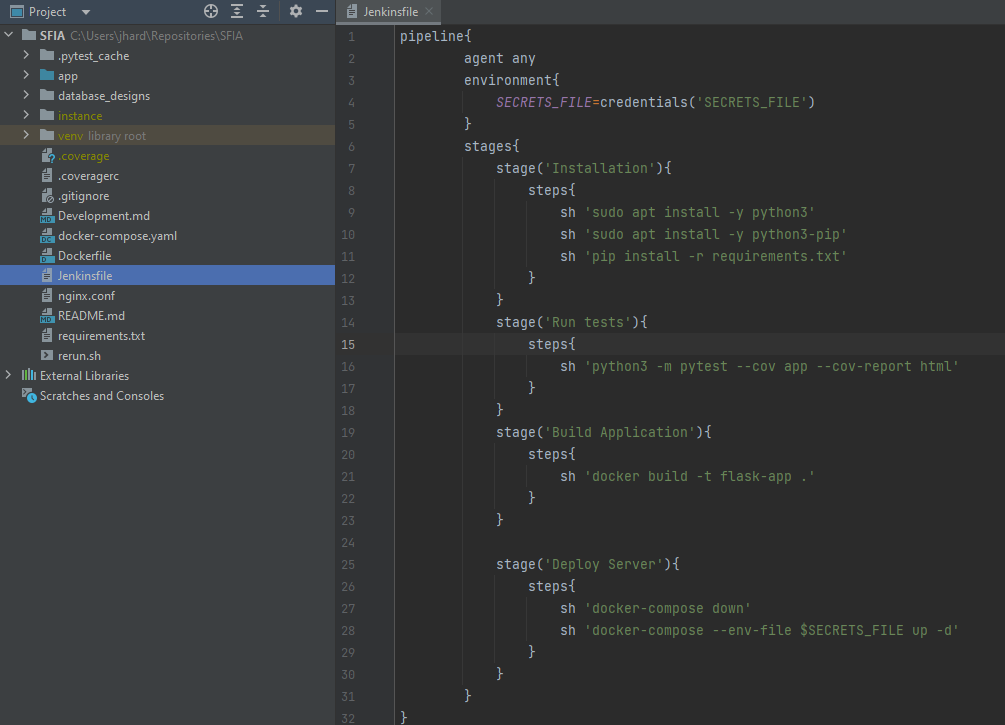


In its current state, CI and deployment are done on the same server, however with more time the builds could be published to docker hub or nexus and installed from the jenkins server onto other instances using ansible.

## Continuous Integration

Jenkins was used for the build process management with the pipeline managed from a file in the target directory.





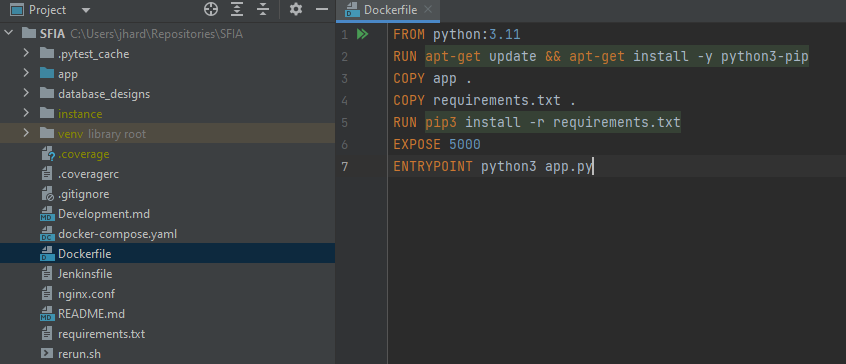
To provide the appropriate permissions the line:

jenkins ALL= NOPASSWD: ALL

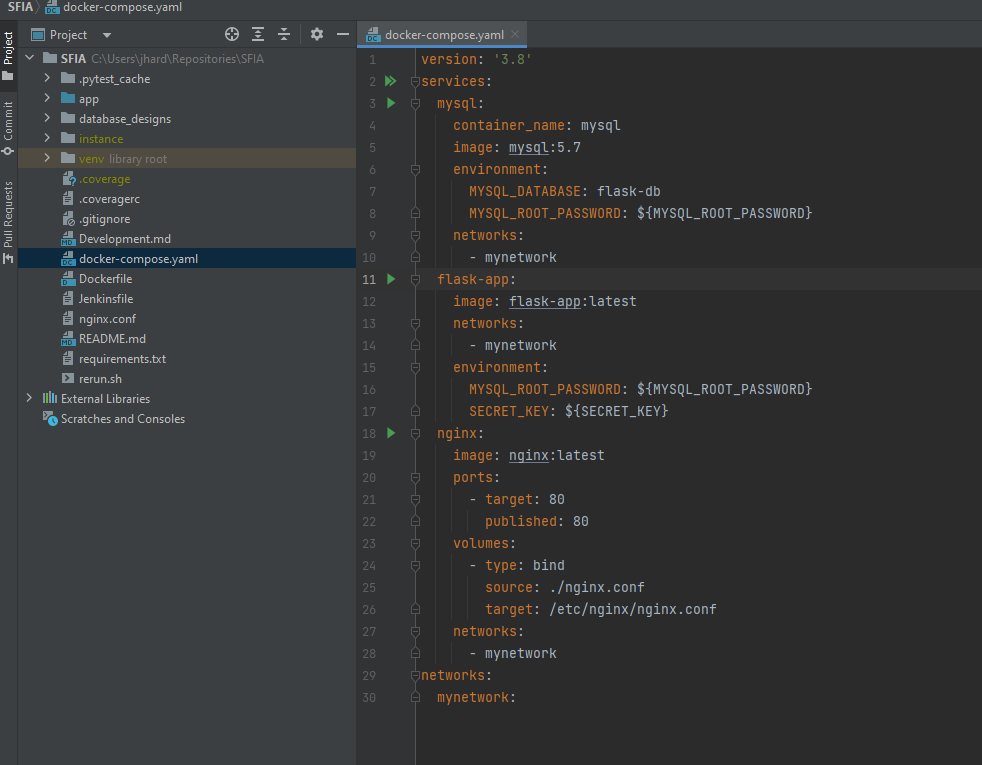
Was added to the sudoers file.

## Containerization

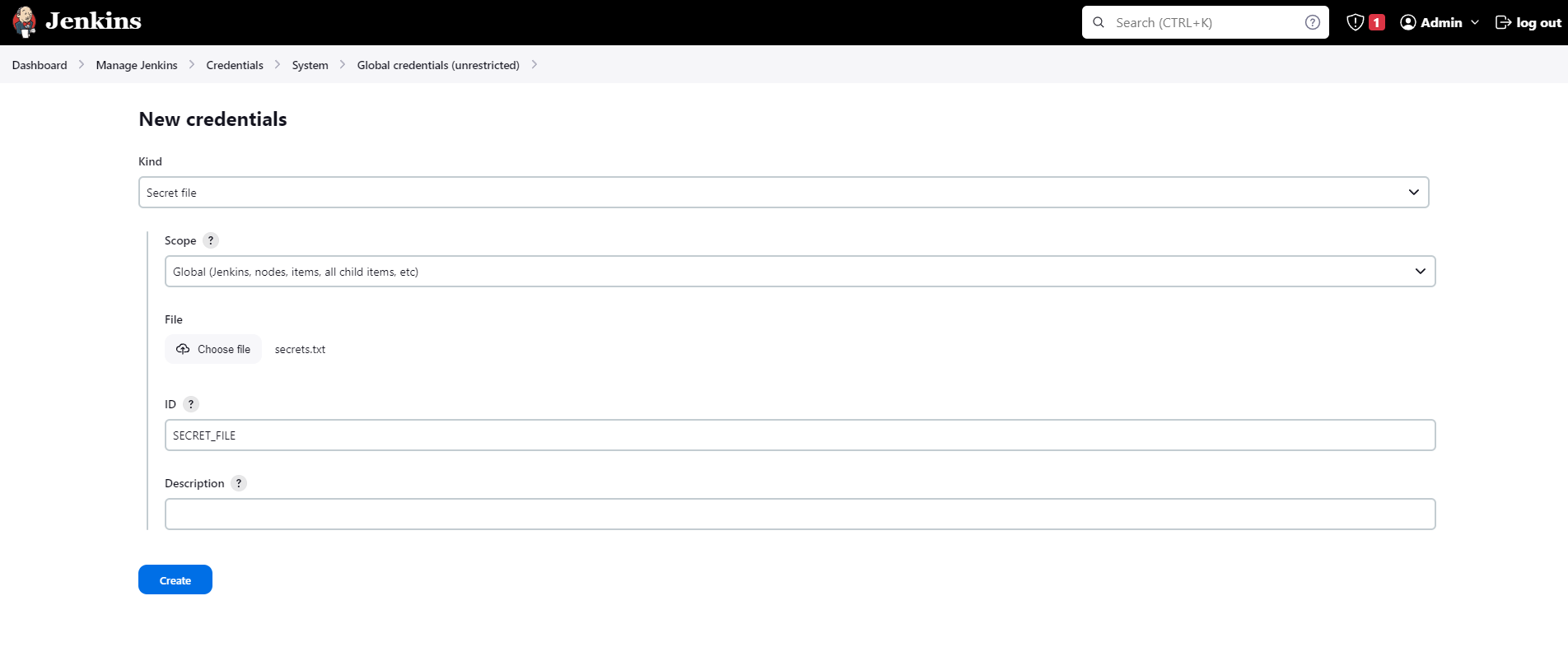
Docker was used for containerization, and docker-compose to streamline the tasks so they could be used in the jenkins build. The application was built using the Dockerfile:



With the full pipeline being built from the compose file:



To keep sensitive information secure, the variables were kept in a jenkins secret file that would be injected by jenkins at runtime.



To give jenkins the appropriate permissions to run docker commands, jenkins was added to the docker group on the instance, and then restarted:

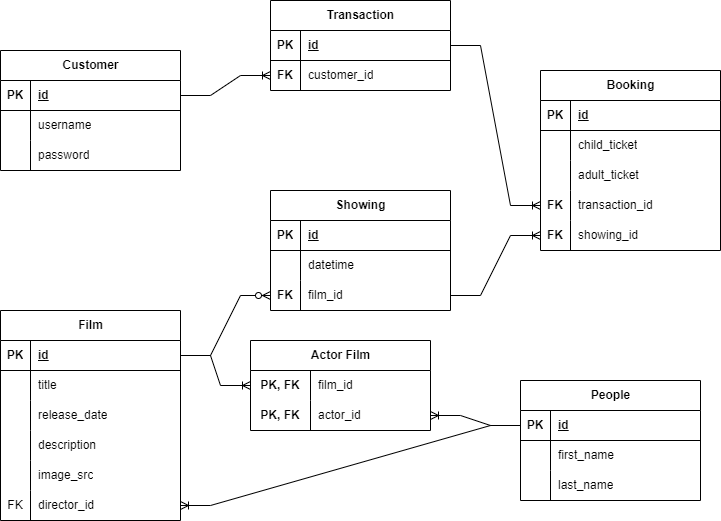
sudo usermod -a -G docker jenkins

systemctl restart jenkins

## Database Design



This was an initial design to give a broad base for the initial construction with the understanding models would be extended where needed.



Actor model renamed to People with an additional reference to account for director reference requirement.

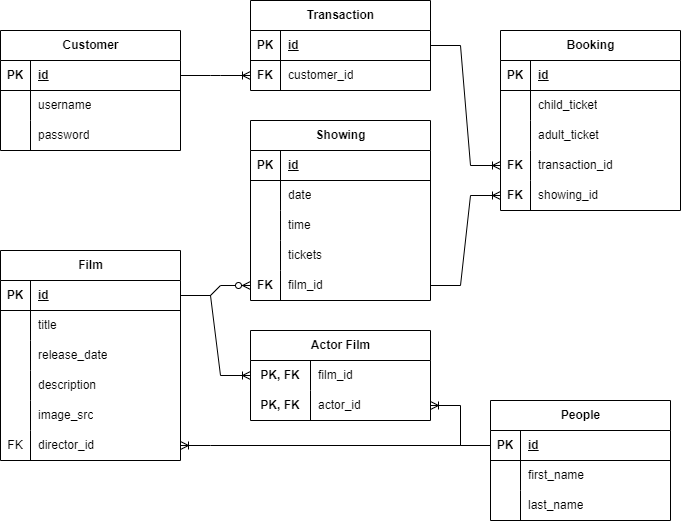
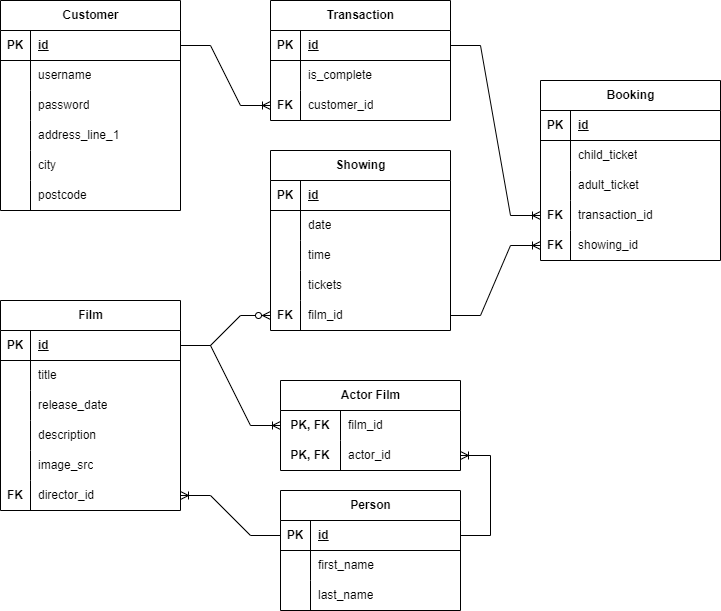


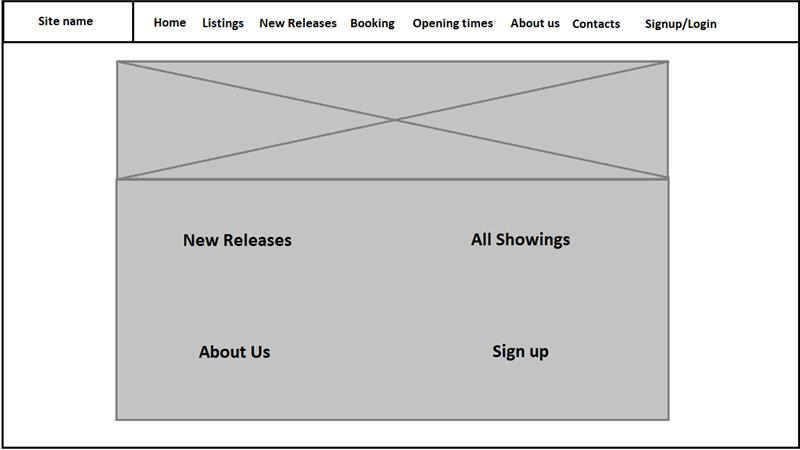
Diagram with showing model being expanded to facilitate sorting showings by date and hosting a limited number of tickets.



Customer model expanded to account for payment processing page.This is the most up to date iteration of the diagram.

# Web Page Design

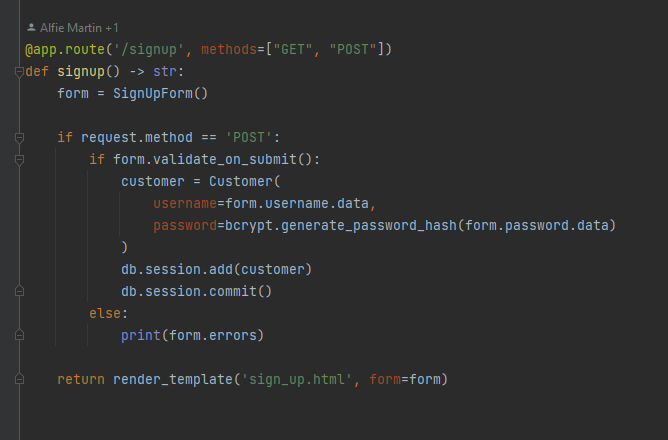
## Home Page



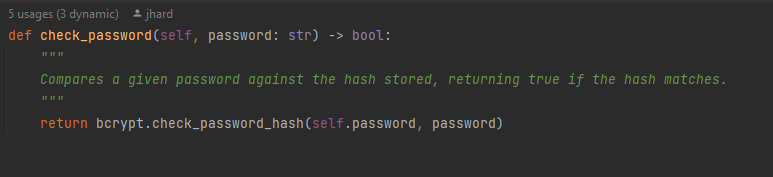
It was noted that many “competing” cinema pages have very minimalistic home pages, often just using them as slightly alternate variations on the listings page. As such, it was determined the home page didn’t need to be particularly complex as long as it conveyed in loose terms what the website was and what it did.

# Product Design

Passwords submitted to the database were hashed, with password confirmation done using hash comparison.



Signup route, hashing passwords as the customer model is created.

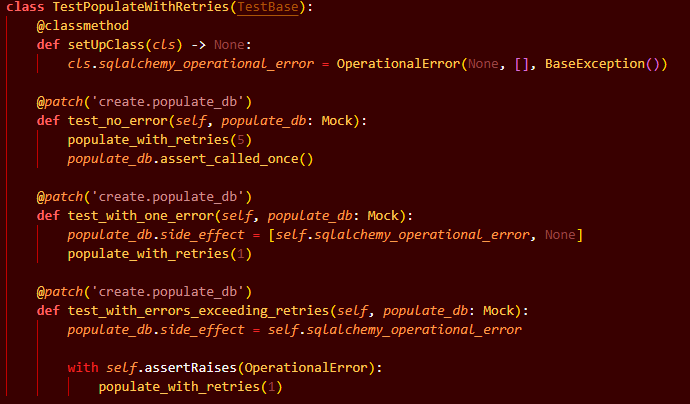


Password confirmation.

Test Design

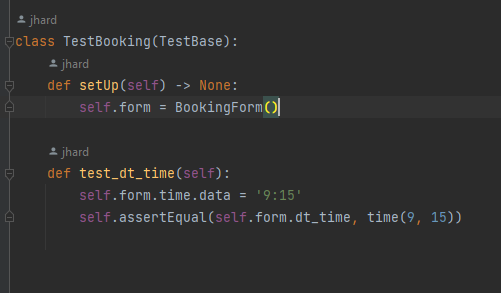
create.py Tests:

The create.py tests are used to specifically test to check that the file attempts multiple retries to populate the database, we have included checks for this in the tests.



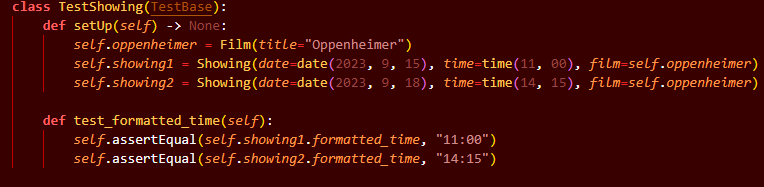
Forms.py Tests:

Any non-validation function added into a form was tested to ensure proper use.



models.py Tests:

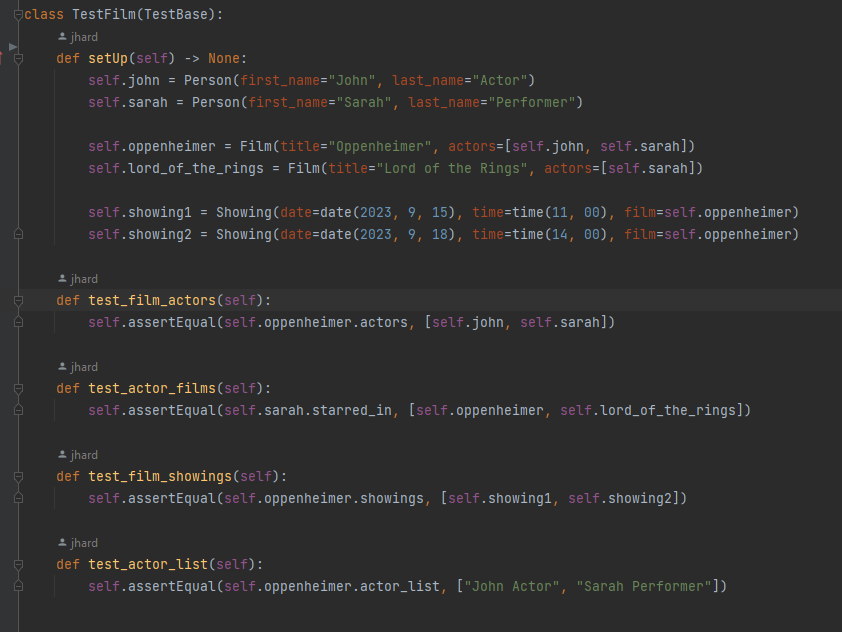
Any additional functions implemented in models had tests to verify they were working correctly.





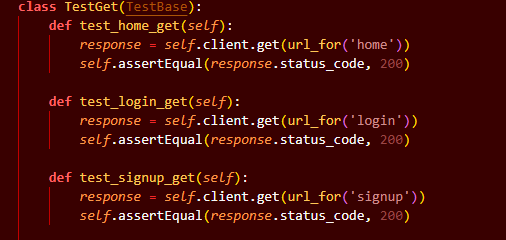


Because the film-actor relationship was modestly complicated using a many-to-many relationship, tests were done to ensure functionality.



views.py Tests:

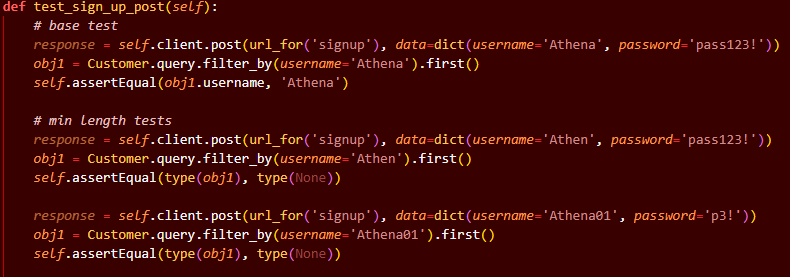
For the views.py tests, we have created various Get and Post requests to each of the pages in the website and ensure they interact with the backend to serve up the appropriate information.



Tests to check response status code.



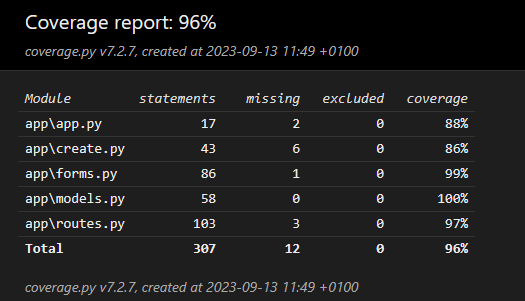
When creating the tests for the pages which required additional information based on information stored in the URL, we added a few test records to the necessary tables in which we used to pull required data from the URL links. We tested multiple records to ensure that the page links would be able to change based on the incoming records and display the page without issues.



Once all of the get requests had been created, we started working through adding the post requests.

When testing these requests, A base test was used to make sure when given data that should follow all the guidelines of the validators, the correct response would be taken. In this instance, a customer record was added to the table.

After covering that the base case works, we move on to testing the extreme points of the validators, this allows us to check the points in the program with the most risk of failure. Instead of testing to see if a change was made to the table. We instead look for a lack of change, due to the validators catching anything outside of the forms rules.



Once all the tests had been implemented, we found that the large majority of our code had been covered by the tests.

Issues Encountered

Merge Conflicts

As most commits were done on the develop branch, merge conflicts would have to be resolved by the person working on the changes by pulling down the development branch and resolving them locally. In the cases where branches were made that conflicted with develop on a pull request, the conflict would have to be resolved by merging locally into develop.

git checkout develop

git pull

git checkout [branch]

git merge develop

(resolve conflicts, either manually or using IDE tools)

git commit -m "message"

git push

Booking pages JS implementation

During the design of the booking page, we wanted to be able to instantly change the select items in the time field of the form, depending on which Movie and date the user had selected in order to attempt to remove the need to refresh the page to load these changes.

We had the idea to create a JS file to use event handlers to keep track of values in the movie and date field and run a search using SQLAlchemy for any relevant times to populate the time field. The issue was this would require an adjacent API which would’ve had to have been queried via AJAX, which was out of the scope of the requirements. After consideration, it was determined that post requests to the web page with alternate submit buttons was sufficient, as flask\_wtf forms contain information on which submit button was pressed.

## Deployment Issues

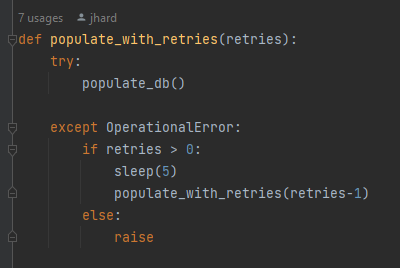
## MySQL Loading

There was a persistent issue encountered where the web app would crash upon loading. Isolating why this happened took a non-trivial amount of development time, but it was eventually attributed to the fact that when the flask-application started up it would attempt to connect to the mysql database to create the models.



The issue was that even if a docker container running MySQL claimed to be running, MySQL would take a few seconds to fully start up, and during this time docker had no way of verifying MySQL was running before starting the flask application.

The solution was to add some resiliency in the flask application by including a retry in the event an sqlalchemy error was encountered.



This allowed for the application to continue to attempt to connect until the MySQL database was running.