The Shopping Cart App

Final Report

Developed by Makan D and Daniel S.

Summary

With this project, we've set out to create an all-in-one shopping cart app for online businesses. Our core ideals for this project were to:

1. Create a web-based platform that customers can use to purchase various products.
2. Attach this system with a backend that can be dynamically updated from a database to introduce new products, update old prices, manage the various discount codes, and more.
3. Design a UI that is capable of functioning both on desktop and mobile devices.

After looking at all the available languages, libraries, and frameworks - we've settled on a **React.js** (JavaScript) frontend with a **FastAPI** (Python) backend. Information between these technologies is communicated via a REST API to make it as flexible as possible. We wanted to allow businesses to mix/match their own technologies simply by linking them to the API.

As for deployment, we've made use of **Github Actions** and **Heroku**. This CI/CD pipeline allows us to test both the backend & frontend, build the frontend, and deploy the final product to Heroku where it's publicly available. Each framework we looked at has its own set of testing libraries. With our solution, we used **PyTest** with **Selenium** for CI to test our backend and frontend.

As an example, we've launched a sample at a1-storefront-production.herokuapp.com that showcases its capabilities.

We've also included instructions on running and deploying the app within the README.md at the root of the repo.

Backend

A functional frontend would be nothing without a backend, so this is where we looked at first when deciding on our app's roadmap. In the end, we came down with these three options:

1. Node.js backend with Express[1]
2. Python backend with FastAPI
3. Java backend with Spring Boot
4. PHP backend with Laravel

Ease of Development

We both have a lot of experience in all of them except for PHP. So it'd be best if we'd choose one of the first three options if we don't want to spend most of our time learning the basics.

As for the frameworks, the only framework both of us have majority of our experience in is Express (Node.js). However, we can't just base our decision off which one we have the most experience in. We'd need to factor in the *ease of use* as well. Does this library provide us enough tools to get up and running in as short of a time as possible?

This is where FastAPI shines. There are many useful features bundled with FastAPI, such as endpoint request schemas, model wrappers, built-in form validation, and more, that will reduce the amount of time we spend writing code. Compared to Express, where OpenAPI schemas must be manually typed per endpoint, we can make use of FastAPI's interpolation of request & response models alongside Python's type annotations to create our app as efficiently as possible.

Maturity

All of these languages were created before we were born and have been used in production environments at many large corporations.[2] In terms of raw dates, however, Spring Boot is built on the Spring Framework, which in itself is almost two decades old.[3] This age advantage could provide us with more access to more documentation, resources, and libraries to use within our app.

[1] This includes Next.js that bundles an Express backend with it.
[2] https://en.wikipedia.org/wiki/Programming_languages_used_in_most_popular_websites
[3] https://en.wikipedia.org/wiki/Spring_Framework

All of these languages provide us access to helpful package managers for libraries to use for our app. PHP has Composer, Python has PIP, Node.js has NPM, and Java has Maven.

PHP and Node.js have many more modules built for web-based applications, with Node.js having them for both frontend and backend (Sometimes these modules can even be used on both). This will provide us with various wrappers for APIs, drivers for databases, and more.

Domains

Node.js and Express will give us access to a tech stack that we can use on both the frontend and backend (If we decide we want to use a Framework on the frontend). This would bridge the gap between them, since they'd both be relatively in the same language, using the same coding practices, resulting in less time spent learning.

Laravel and Spring Boot both have been built with server-side page rendering in mind. They both support complex user flows with support for controllers, models, etc to create websites with MVP design principles. This could result in much cleaner code and increased room for expansion. However, for the scope of this project, this complexity is not needed. Also, alongside that, we're looking to make an API, which makes the server-side rendering functionality a nice to have instead of a necessity.

FastAPI has our backend use-case in the name. A tool developed to create APIs, with the various features to make creating it as simple as possible. Down the line we might want to implement data analysis for recommendations to our app, so having access to libraries like Numpy and the plethora of machine learning libraries will make data analysis fit nicely within our app.

Popularity

Age alone does not define the adoption of a framework. Looking at stats alone[4], both Express and Laravel are being used by the majority of websites right now. This could result in more up-to-date guides, actively updated libraries, and a more active community to get us up and running with the basics. FastAPI is relatively new to the space being launched just 2~ years ago[5], so access to resources could be limited.

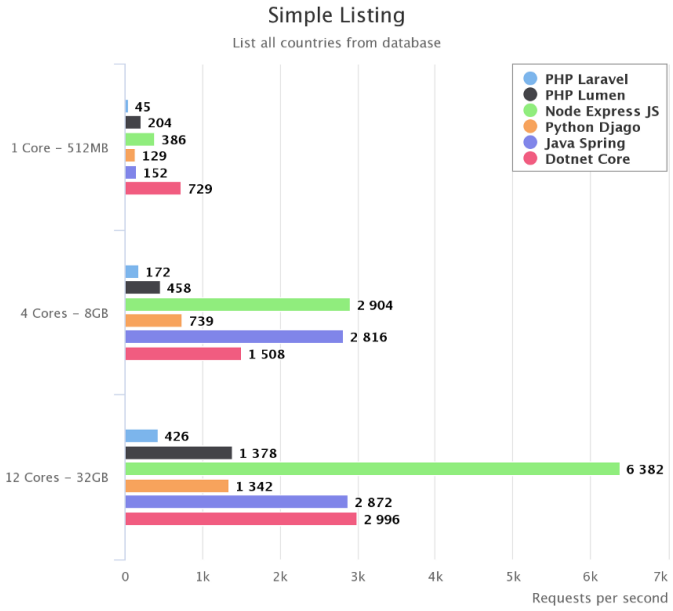
Performance and Scalability

[4] https://www.similartech.com/categories/framework
[5] https://en.wikipedia.org/wiki/FastAPI

In terms of raw performance, Node.js is king[6]. On top of that, FastAPI boasts that it's got performance that is on par with Node.js and Go, which makes it one of the fastest Python frameworks available[7].

Threading our apps is also possible on all of these except for PHP. Adding more points for speed to Node.js and FastAPI. (+ Spring!)

All of them provide the modularity to build SOLID design compliant and scalable apps.



Monetary

From a monetary standpoint, each backend has a hidden cost when it comes to how fast it can be deployed. This comes into play with the "lightness" of the framework and dependency installation.

Python itself and dependency installation is extremely quick without any actual building/compiling for FastAPI with a small bundle size and runs on a lightweight server that can scale. This allows our build times to be extremely quick. Node.js installations have to NPM binary installations and even with caching builds, NPM installs have more overhead when deploying especially as node modules build times increase.

[6] https://medium.com/@mihaigeorge.c/web-rest-api-benchmark-on-a-real-life-application-ebb743a5d7a3
[7] https://fastapi.tiangolo.com/

Our solution: FastAPI

Here are some examples of how FastAPI makes writing code so short and simple, yet so powerful and flexible at the same time:

```python
class NameOrUid(str, Enum):
    NAME = "product_name"
    UID = "id"

class ReductionType(str, Enum):
    CASHOFF = "flat"
    PERCENTOFF = "percent"


class DiscountModel(BaseModel):
    discount_code: str
    reduction_type: ReductionType
    amount_off: float

class ProductResponseModel(BaseModel):
    id: int
    name: str
    description: str
    price: float
    img_url: str

class ProductsResponseModel(BaseModel):
    products: List[ProductResponseModel]

class DiscountsResponseModel(BaseModel):
    discounts: List[DiscountModel]
```

```python
@product_router.get(
    "/discounts/",
    response_model=DiscountsResponseModel
)
def discount_info():
    return get_all_discounts()
```

That isn't it though, FastAPI has built-in many more features such as:

- Easy middleware integrations.
- Exception-based responses (less manual status codes, more reusable).
- Production-ready ASGI server integration.
- Worker queues.
- Sophisticated test infrastructure **PyTest**.

As well, includes a tremendous amount of easy-to-use functionality for common application usages like OAuth2 and MongoDB/Postgres-SQL drivers. It was a no-brainer to go with FastAPI given our experience in Python and the simplicity of learning the library.

Frontend

The top choices for our frontend included:

1. React.js[8]
2. Vue.js
3. Angular.js
4. jQuery with Bootstrap
5. Vanilla HTML, CSS, and JavaScript.

Shortly after finding these options, we realized Angular.js is not something we would want to use. It's currently on its last release and is getting its final updates until December 31, 2021, where it'll go EOL. So it's best not to create the app on a platform that is coming to the end of its lifetime.

The other four can be split up into two groups, frameworks (React.js & Vue) and not frameworks (jQuery & Vanilla).

Ease of Development

In terms of experience, we're both well-versed in the vanilla trio and React.js. For the others, it's kind of a mixed bag of experience between us. Although, knowing simple web languages makes the transition to frameworks pretty simple. They're just built off these web languages with their own way of "creating" the app in a sense. (Source: We both learnt React.js in a week over the summer) JSX was built with HTML and CSS in mind, which is why it's so adaptable.

So if we were to learn a framework, it shouldn't be much of an issue.

Maturity

Frameworks themselves have been a relatively new way of creating websites with the introduction of React in 2013[9]. By not using a framework, we're opening doors to a stack that has had more time to mature. However, this is not to say that frameworks have been developing fast. Being backed by large companies like Facebook, the development of these frameworks has been done at an accelerated rate. Alongside all the resources provided directly within the documentation, the age of something alone is not a factor.

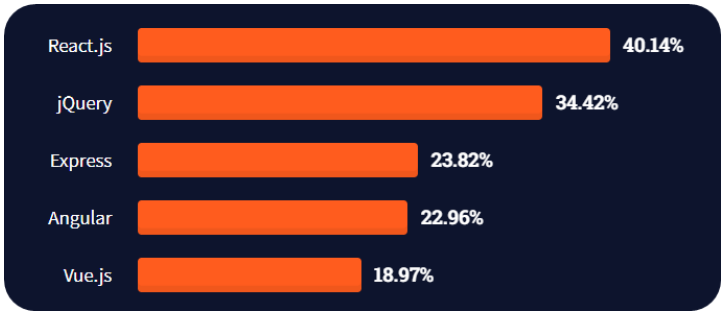
[8] We also considered Next.js, which is a React.js wrapper which includes a functional Express backend built in, significantly reducing development time.
[9] https://en.wikipedia.org/wiki/React_(JavaScript_library)

Domains

We decided not to use a Node.js backend, so frameworks don't get an advantage here in terms of using the same language. Frameworks make use of JSX to create components with their own internal logic, which would let us re-use various aspects of our website resulting in much less code. The same can be done without a framework, but we won't get the advantages of the virtual DOMs performance and the benefits of using JSX components. This would result in a lot of manual writing of DOM elements in strings, which could result in some more obfuscated and hard-to-read code. This could limit the extensions we can make to the app in the future.

Popularity

Based on the StackOverflow developer survey, React.js has been the most used web framework within the past year.[10] Vue.js is lagging a bit behind being a more recent framework compared to React.js.



jQuery is also the second most used, so the distribution of using or not using a framework is relatively close. It'd come down to what we want to use for our project.

Performance and Scalability

Performance-wise, for larger and more dynamic apps, frameworks win here. Making use of the virtual DOM and event handlers, updating states is done at much faster speeds compared to directly manipulating the DOM with jQuery (or vanilla JavaScript).

For size, however, if we want our website to load in the least amount of time, it'd be better to go with no framework at all. By simply using vanilla JavaScript we'd save on the resources required for jQuery, React, etc. This could drastically improve page load times and could even result in faster speeds compared to what we save with a virtual DOM.

Also, for this project, the speed improvements will be almost insignificant since we're not re-rendering large aspects of the screen at once. However, within the future, if we decide to scale the project with product pages and extra functionality, this could result in drastic speed improvements.

[10] https://insights.stackoverflow.com/survey/2021#most-popular-technologies-webframe

Monetary

From a monetary standpoint, we once again look at the build time used to build our production files and deploy them remotely. React, Vue, and Angular all utilize node libraries and thus have dependencies that must be resolved on build time. They also must convert their JavaScript into build processable HTML and JavaScript. This takes time and thus increases total build minutes. When compared to vanilla JavaScript, the only thing to deploy is the HTML, CSS, and JavaScript without dependency installation or conversion. (We might want to minify, but processes like that take a few seconds at most) Therefore, it will reduce the total build time. However, we compromise and take the overhead of using a framework for its more advanced features as it activates easier development versus its build time.

Our solution: React.js

To follow on from the point from performance and scalability, not only will the speed of the website be improved, but the code structure would be improved drastically as well. Being able to re-use components across the website would result in much cleaner-looking code and reduce development time.

We chose React.js over Vue.js because React.js currently is dominating the framework space, giving us access to more documentation and resources for learning the best practices.

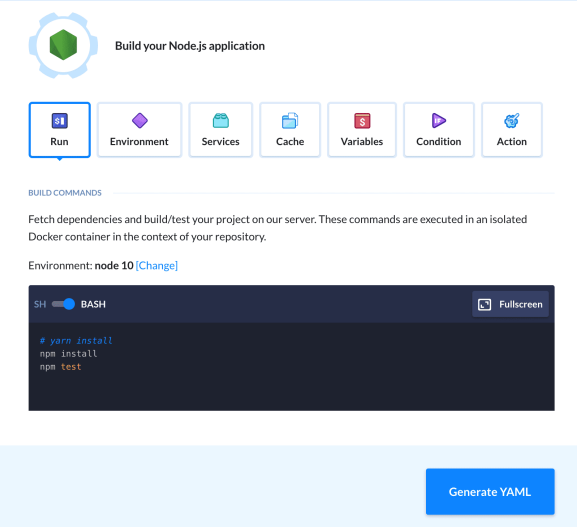
CI/CD

We want our CI/CD tool to test, build, and deploy our frontend & backend combo onto a Heroku server. Here are some of the choices we've considered:

1. GitHub Actions
2. Travis CI
3. Buddy
4. Jenkins

Ease of Use

All of the options except Jenkins are fully managed solutions. This means the setup for these is relatively simple apart from the YAML file creation. Jenkins would require us to get a server ourselves and deploy Jenkins on it, which in itself could be a complicated task. Alongside the installation, the process of setting up each plugin for the various tools we'd want to use is also a challenge in itself since everything has to be set up manually. This makes Jenkins the most difficult to use.



As for the builds themselves, the easiest one to use is Buddy. GitHub Actions, Travis CI, and Jenkins run their builds through YAML file configurations. Buddy features a GUI-based platform that allows us to set the build settings, alongside a YAML configurator as well for more fine-tuned control. It's the best of both worlds.

Maturity

Out of the four, GitHub Actions is the youngest of the bunch having its CI/CD element release just 2 years ago[11]. This might mean that GitHub Actions has the fewest number of actions[12], but that's not the case.

Due to GitHub Actions documentation and the ability to create "actions" straight on GitHub, the open-source community has already created 10,000+[13] actions from setting up environments to deploying apps to various hosting vendors. Alongside that, GitHub

[11] https://github.blog/2019-08-08-github-actions-now-supports-ci-cd/
[12] This is what they call the scripts that actually run your flows.
[13] https://github.com/marketplace?category=&query=&type=actions&verification=

has many guides and has made it easy to take YAML configurations from other CI/CDs such as Travis CI[14] and have it running on GitHub actions.

Despite being newer, in terms of flexibility, GitHub Actions is king here.

Domains

These tools except for Jenkins are cloud platforms for CI/CD whereas Jenkins must be run on your own server farm or locally.

They all relatively do the same thing. Their goal is to make automating various tasks as simple as possible. They can all link to repos, run commands on specific actions, test in containers, etc. The only key differences are in the way actions are configured, build pricing, and the fact that Buddy has a GUI which is entirely up to user preference.

Buddy and GitHub Actions offer **free** monthly build times which is a pro.

Popularity

Before GitHub Actions, the market was primarily dominated by Travis CI for open-source and private applications (open-source had the largest due to their free plan). With the simple migration of GitHub actions, many applications have simply ported over their deployments to GitHub Actions.

There isn't much information on Buddy's market share, but based on the plugin list and amount of documentation available online we can conclude that Buddy is probably one of the least popular of the four.

Jenkin's is kind of in its own category of being the only large CI/CD application that's open source and can be deployed by the developers themselves.

Performance and Scalability

Performance isn't key metric for deployment apps, so we're going to focus on scalability. At the base level, all of these apps are pretty much the same with minor tweaks within their setup scripts with access to different plugin libraries. So in terms of scaling your app up, it'll largely depend on the app. For ours, we looked through each of the platforms and they had all the tools we needed to deploy, plus more tools for future expansion. (Database plugins, deployment to AWS, etc)

[14] https://docs.github.com/en/actions/migrating-to-github-actions/migrating-from-travis-ci-to-github-actions

One thing we found with GitHub Actions is that you can deploy your own self-hosted runners[15]. This could potentially be helpful for some specific use cases (such as deploying some sort of machine learning system using in-house hardware) without having to pay for more machines. These self-hosted runners can be deployed alongside your GitHub Actions scripts and integrate nicely within GitHub, so that's a win in GitHub Action's books.

Monetary

Just last year Travis CI announced they would no longer be doing a forever free plan[16], while still costing more than their competitor, GitHub Actions, in terms of pricing[17].

Jenkins comes with the benefit of being able to self-host the solution. This could result in cheaper monthly bills and could provide more flexibility in what can be done with scripts since you have full access to the machine you're testing on. Jenkins can be more difficult for beginners and trade-offs need to be made by which platform allows for ease of access and effective deployment strategies.

Our solution: GitHub Actions

GitHub Actions easily wins the CI/CD competition here. Its giant library of actions and the fact that it's built right into GitHub allows us to remain in a single ecosystem, simplifying our development experience.

[15] https://docs.github.com/en/actions/hosting-your-own-runners/about-self-hosted-runners
[16] https://blog.travis-ci.com/2020-11-02-travis-ci-new-billing
[17] https://www.nymedia.no/en/blog/why-we-decided-migrate-from-travis-to-github-action

Databases

Although we didn't implement a real database within our project, these are our choices if we were to:

1. MySQL (or MariaDB)
2. MongoDB
3. CockroachDB
4. Amazon Aurora

Ease of Use

We have experience in only MySQL and MongoDB, so if we want to stick to something familiar we'd use one of those two. (More info about ease of use is in the *Domain* section)

Amazon Aurora[18] and CockroachDB[19] are both MySQL and Postgres compatible managed services[20] that will allow us to migrate our knowledge over with minor changes (if any). We could make use of any of these services with our existing knowledge of databases.

Maturity

In terms of maturity, MySQL is the oldest with its release date going back to the 1990s[21]. In terms of age, MySQL has been here longer than MongoDB so it could have more resources online.

Another benefit is that MySQL has been forked by many communities and improved. Solutions like MariaDB exist that try to improve on MySQL itself, providing that extra flexibility compared to MongoDB.

Domain

All these services cover very different domains. MySQL vs MongoDB is essentially the difference between a relational and nonrelational database. Depending on our use case for our app in the future, we'd need to decide whether we'd like to go relational or not. For now, in the current state, either can be used.

---

[18] https://aws.amazon.com/rds/aurora/mysql-features/
[19] https://www.cockroachlabs.com/docs/stable/migrate-from-mysql.html
[20] CockroachDB can be self-managed too, more about this later on.
[21] https://en.wikipedia.org/wiki/MySQL

The major advantage of relational databases comes from joining information from various other tables. This could be useful when doing things like order history, where we want to join the history of user orders and various product IDs. This could result in more space saved, faster query times, etc, if used properly. MongoDB is generally simpler to set up and get up and running since it's pretty much stored as JSON objects within the database, which is helpful for beginners or simple apps.

CockroachDB takes the elastic architecture approach where data is distributed across multiple servers globally. Databases will experience speed improvements (using servers closest to the user), redundancy (spreading the load), and uptime (one server goes down, multiple available). For the scope of our app, all of these features aren't needed but it's helpful to look for in the future.

Amazon Aurora takes database software and simply makes them faster with their server-side optimizations to the software and hardware. The only downside to this is that you'd have to deploy your database on AWS to get this.

Popularity

In terms of popularity, MySQL and MongoDB are head-to-head in terms of which is most used[22] (MySQL typically comes on top).

The popularity of each would not typically be something we'd use to compare these two since they use completely different ways of organizing and storing info. If we ever decide that what we want to do with a database would require a relational database we'd go with MySQL, otherwise with MongoDB.

In terms of managed services, AWS has the edge over CockroachDB[23]. This is simply because AWS is more well known and CockroachDB has its own "use case". Not every app requires the redundancy that CockroachDB offers, so it's typically just better to go with AWS.

Performance and Scalability

[22] https://db-engines.com/en/ranking
[23] https://db-engines.com/en/system/Amazon+Aurora%3BCockroachDB

In terms of raw performance, MongoDB is superior in terms of inserts and MySQL has the edge with selects[24]. Once joins are introduced, MySQL gets yet another edge over MongoDB.

Deciding based on performance would come down to the use case of the app and which queries are used more often. You can always use a relational database without relations, but you can't use MongoDB as a relational database (technically you could but it's not recommended). So if you realize you want to go relational midway through, switching databases could be difficult in the future. These are choices that need to be decided on when designing the database structure.

As for managed services, CockroachDB has the edge in terms of scalability. Being able to spread the load across various servers could introduce significant speed improvements.

Monetary

By going with a self-hosted option we'd always save more. We could spin up a Vultr instance and install our database engine of choice onto it for half the price of AWS. The only downside is the amount of installation and management required since the server would have to be maintained with updates.

AWS and CockroachDB offer managed services where this process is already completed for us so we can simply set up and start running our scripts out of the box. CockroachDB offers a self-managed service as well, but it's quite difficult to set up and maintain (based on what we read in the docs).

Our solution: Amazon Aurora (MySQL)

After weighing our options we came down to Amazon Aurora with a MySQL engine. This will allow us to get the speed improvements from AWS while being able to use something we're both familiar with.

Spending more to get this service managed for us will save us a lot of time[25] so we can put that towards building our app.

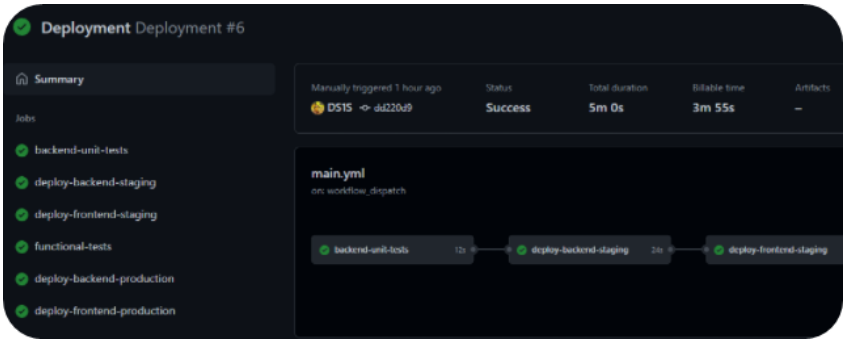
[24] https://github.com/webcaetano/mongo-mysql

[25] I (Makan) have some history with managing a MySQL server and it's no fun. Managing backups, updates, installations takes a lot of time out of development time.

Deploying using GitHub Actions

1. Deployments ran automatically for every push to master branch.
2. Deployments ran by manual dispatch of GitHub workflow (still end-to-end deployment).

Note, please check in .github/workflow/main.yml for the specifics of each job procedure to ensure that these holistic images are actually doing commands. The entire workflow is automatic on pushes, the images seen are from a manual push due to having to fork and no reason to push fake files to trigger it. **On:push:branches:[main]** is a valid hook that will do this. All jobs are dependent on the previous to succeed before being run.

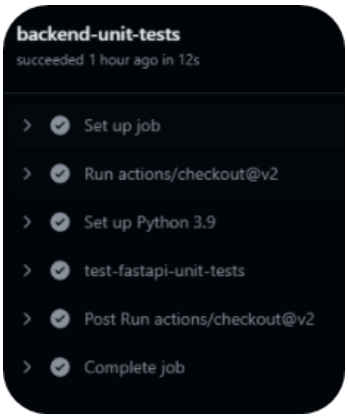


We've split our deployment into 6 steps:

1. backend-unit-tests
2. deploy-backend-staging
3. deploy-frontend-staging
4. functional-tests
5. deploy-backend-production
6. Deploy-frontend-production

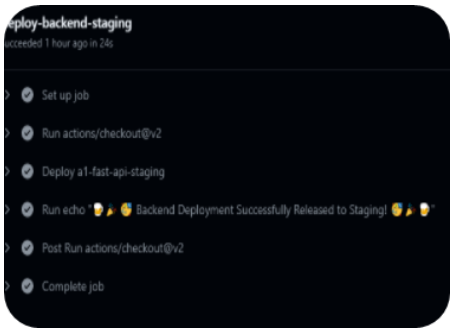
Backend Unit Testing

In this stage we run unit tests for FastAPI backend before trying to deploy to staging. It runs our test suite and will fail if any tests do. It will also fail if the coverage is below 100%. The pipeline/workflow will terminate if these are not satisfied.



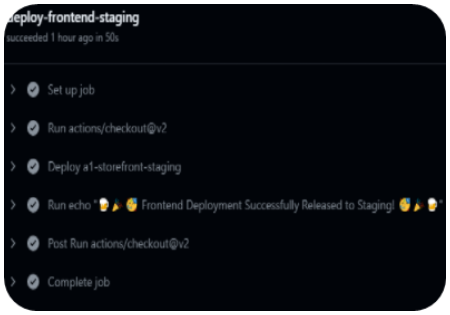
Deploy backend to staging

We use a GitHub Actions for Heroku deployment that makes calls to the Heroku CLI with our secret tokens. It will then request Heroku to look for our app a1-fastapi-staging and push our files over to be deployed and call the procfile to know how to run our entrypoint. If any step fails, the pipeline terminates. We perform a health check by querying the /health endpoint.



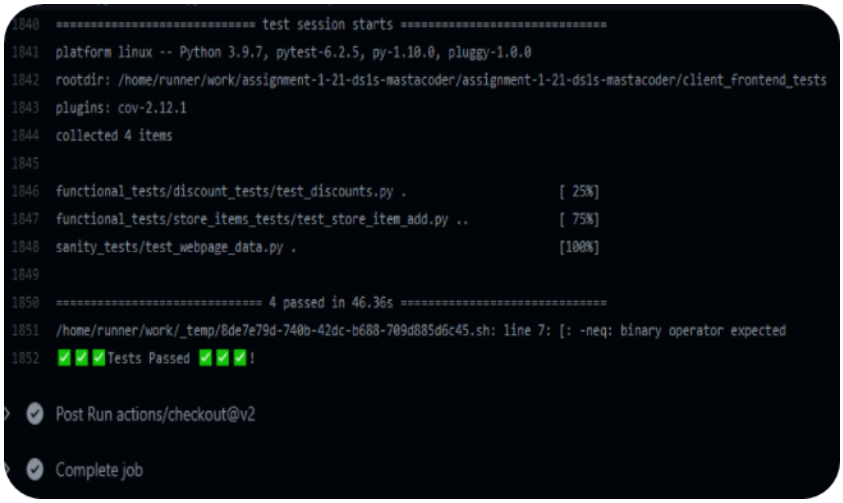
Deploy frontend to staging

Once the backend deploys then we deploy the front end directory to heroku into a separate app a1-storefront-staging calling its own procfile. We perform a health check to ensure we get some 200 responses from the server.



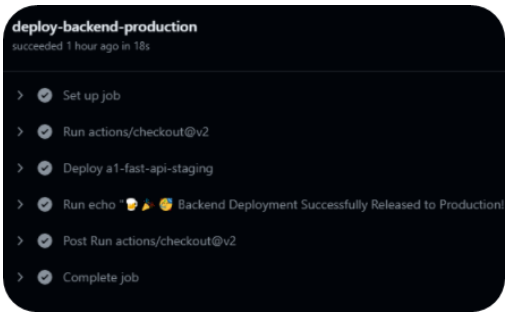
Functional Tests

Once both the backend and frontend are in staging, we can run functional tests on the staging application. To do so we use github actions to run pytest on our client_frontend_tests passing as an environment variable the base url of our frontend web app. The test files declare tests that use selenium and perform queries through a headless client to our frontend checking components and actions.



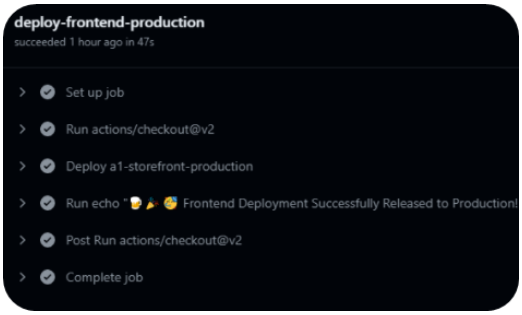
Deploy backend to production

Once our functional tests pass in the staging environment, we then tell heroku to update our production app a1-fastapi-production with the new files and environment variables for production. We perform a health check to /health endpoint on the server to move to the next stage.



Deploying frontend to production

Once our production backend is deployed we deploy our production frontend with the same style of commands: update files with new ones in the staging environment etc. We perform a health check on the server to ensure we get a 200 response that it's up.



Final Discussion

Our deployment to production is now complete! We can log into heroku to see that the builds passed and look at the detailed build logs.

Heroku dashboard: successful deployments of each environment.