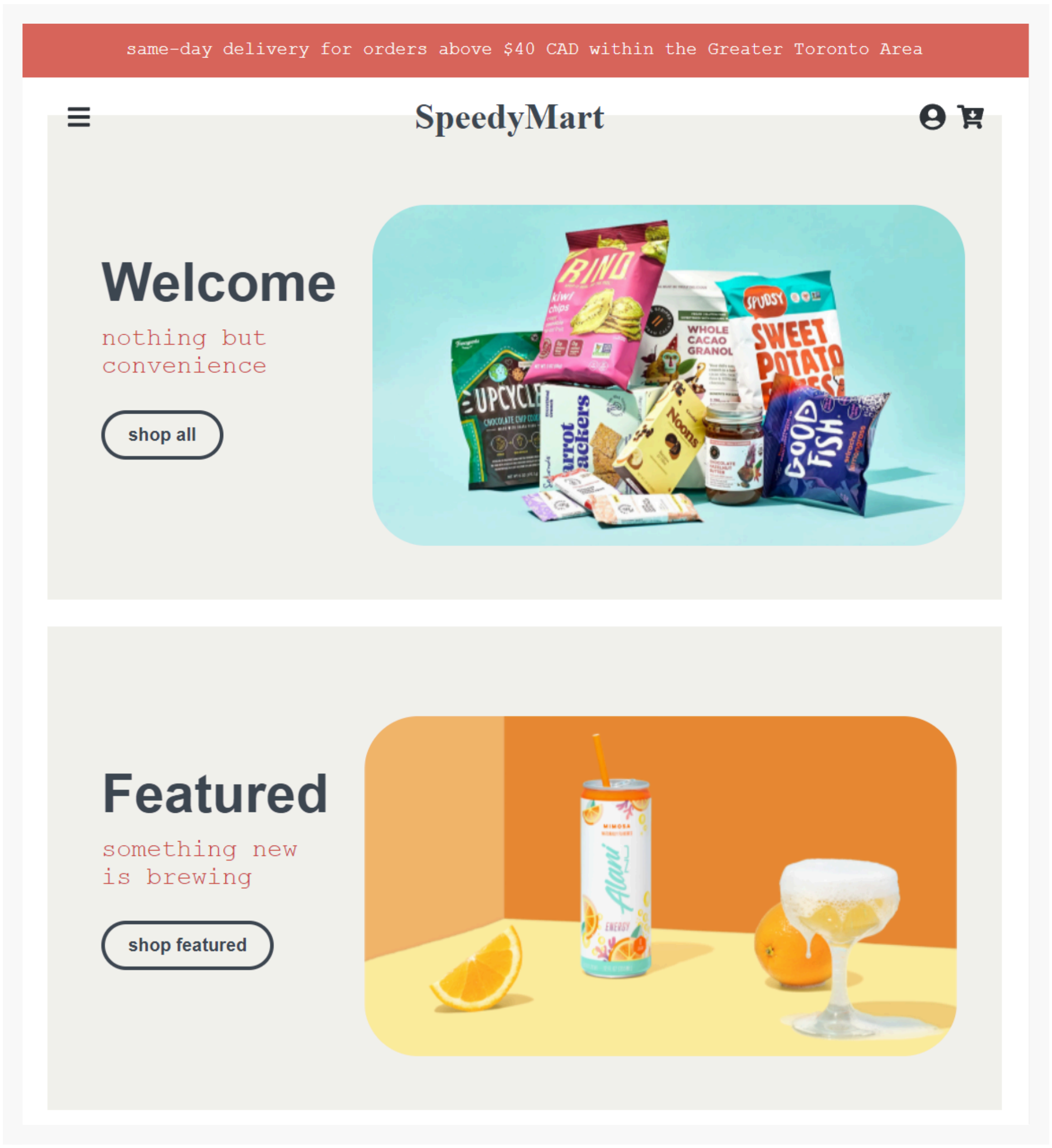
EECS4413 Final Project

SpeedyMart



same-day delivery for orders above $40 CAD within the Greater Toronto Area

SpeedyMart

Welcome

nothing but convenience

Featured

something new is brewing

Philip Nguyen | 213505722 | philipn7@my.yorku.ca

Matthew Tran | 215513666 | tran28@my.yorku.ca

Tarik Mohammad | tarikmoh@my.yorku.ca

Architecture

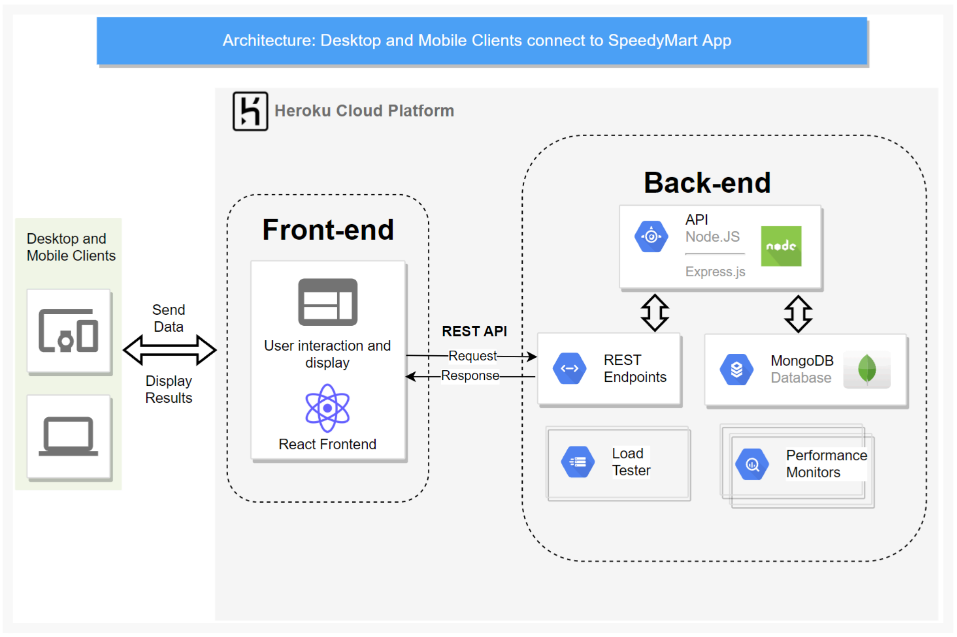


Figure 1: Architecture Overview

The SpeedyMart web app has **clear separation** between the front-end and back-end. A REST API is used to meditate communications between the front-end and the back-end services. The architecture leverages the availability and fault-tolerance of the **Heroku cloud platform**. This is achieved through Heroku's dynos which are scalable nodes in a cloud-based computer cluster. The following sections will detail the system components.

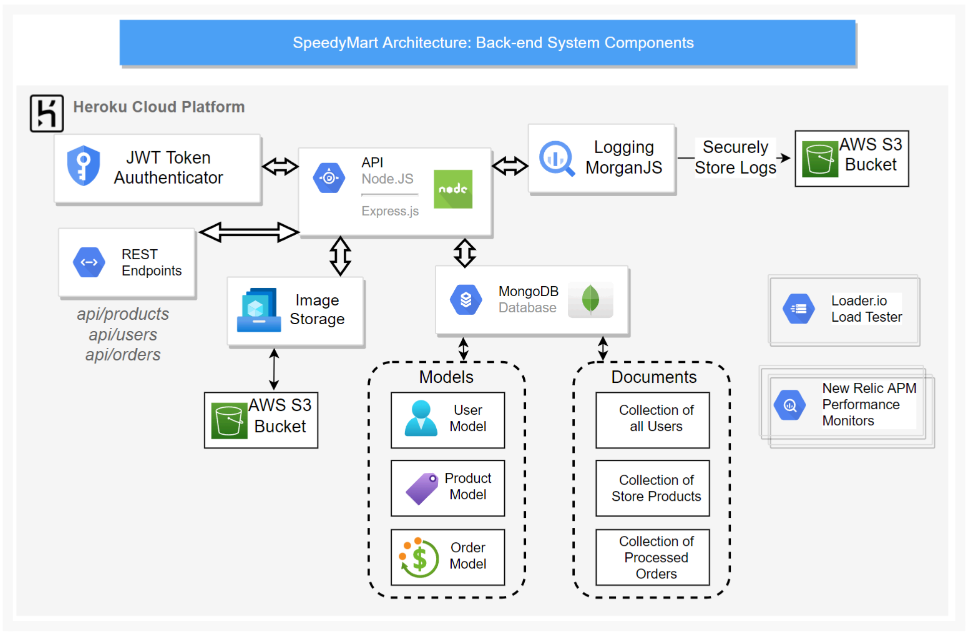


Figure 2: Backend Components

On the **backend**, **NodeJS** and **ExpressJS** are used as the web application framework.

The database we decided to use is **MongoDB**. A **non-relational database** gives us flexibility to choose model schemas to match the needs of the online store. The database models the users, products and orders of SpeedyMart. A SQL database would have worked as well, but we were able to iterate a working schema quickly with MongoDB.

The **REST API** endpoints are the only way the frontend communicates with the backend. Following the REST principles we had endpoints tailored for specific use cases. We tested the endpoints using **Postman** and curl commands to ensure availability.

Security is an important feature. Certain endpoints are protected using **JWT Token** authentication and verification of admin privileges. We applied the principle of least privilege where users are only given access to what they need and nothing more.

Logging and Image Storage are discussed in the implementation section.

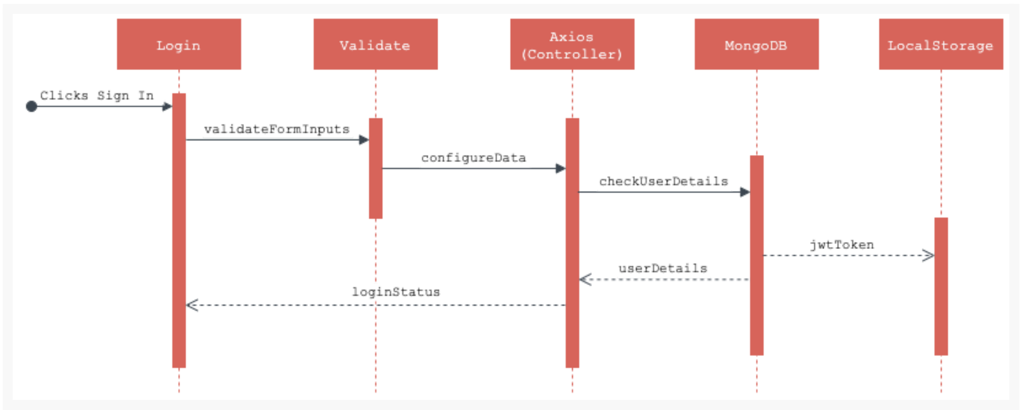


Figure 3: Sequence Diagram (Login)

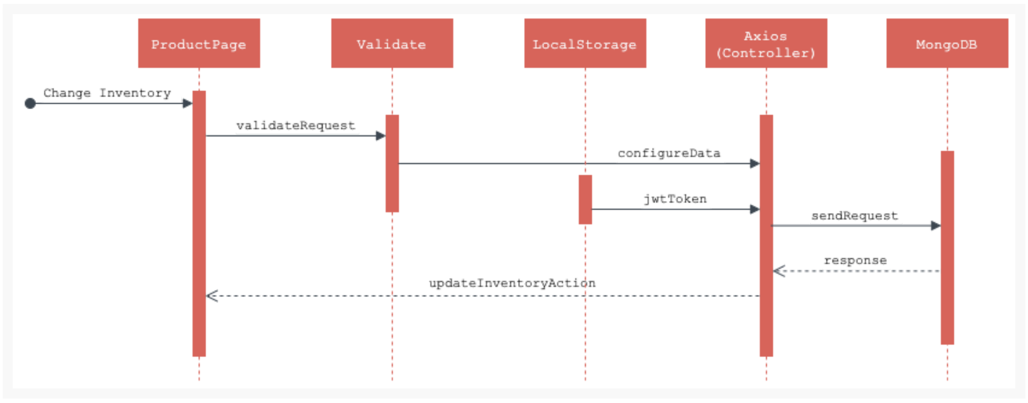
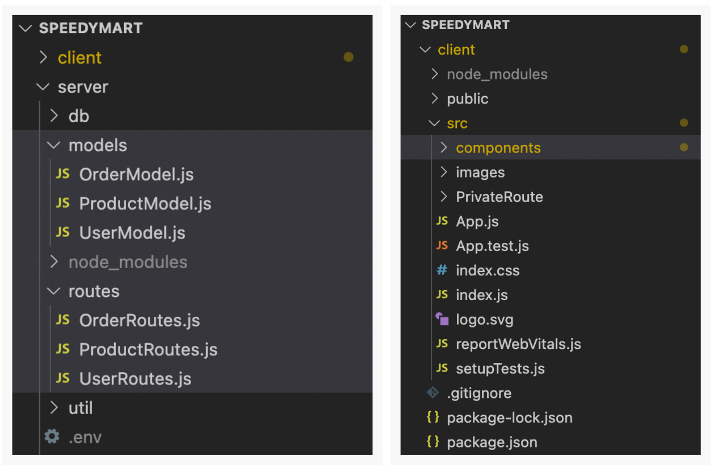


Figure 4: Sequence Diagram (Distinguished: Update Inventory)

Design

Model View Controller

We utilised the model, view, controller principle to organise the communication between our database, backend, and frontend. With **clear separation** between them, the backend **server** folder contains two subfolders within: **models** and **routes**. Models (the model) provides the schema structure for the non-relational database. Routes (the controller) provides the API structure to be called in the frontend. The frontend **client** folder contains a subfolder called **components** which renders all the views.



```
SPEEDYMART
  > client
  v server
    > db
    v models
        OrderModel.js
        ProductModel.js
        UserModel.js
    > node_modules
    v routes
        OrderRoutes.js
        ProductRoutes.js
        UserRoutes.js
    > util
      .env
```

```
SPEEDYMART
  v client
    > node_modules
    > public
    v src
      > components
      > images
      > PrivateRoute
        App.js
        App.test.js
        index.css
        index.js
        logo.svg
        reportWebVitals.js
        setupTests.js
      .gitignore
      package-lock.json
      package.json
```

Figure 5: MVC Structure (see Github for code organization)

Patterns

For the frontend, a primary pattern that was used is the **factory design pattern**. For an ecommerce website, branding is mandatory so it is important to build **complex**, **reusable UI** components. Evident through React, various hooks can be used to assign value to different components while maintaining the "look". One drawback is that props are needed to keep track of states. When a child component is changed, the parent component will not re-render. Other tools such as **Redux** can aid with this issue but for the scope of this project, we manually passed the states to each component, leading to slower load time.

For **example**, the cart view reuses a lot of components from the individual product pages. Navigating through the client folder (frontend), we see that **cartItem.js** is in subfolder *checkout*, the cart view is attached to the **NavBar.js** in subfolder *global*, and the function **addToCart()** is in subfolder *item*. They are all reusable components, yet their states exist **independent** of each other.

Inside **App.js** (the root component), there are **global states** that must be passed to the children to accommodate for this tradeoff (discussed more in the next design patterns).

As already mentioned with states and keeping track of changes, the other two design patterns are **observer** pattern and **state** pattern. Again, we chose **React** because it is a modern framework from **Facebook** and it has the tools readily available to be flexible when it comes to designing the **functionality** of the website. With its advantages of keeping component entities separate from one another, the same drawbacks occur. The observer and state changes must be passed through all components (the parent and all related ancestors). **Development-wise**, the developer must trace all props being passed (inefficient). **Performance-wise**, pages are loaded slower and there may be split-second delays. However, ReactJS is light on memory so these tradeoffs are **negligible**.

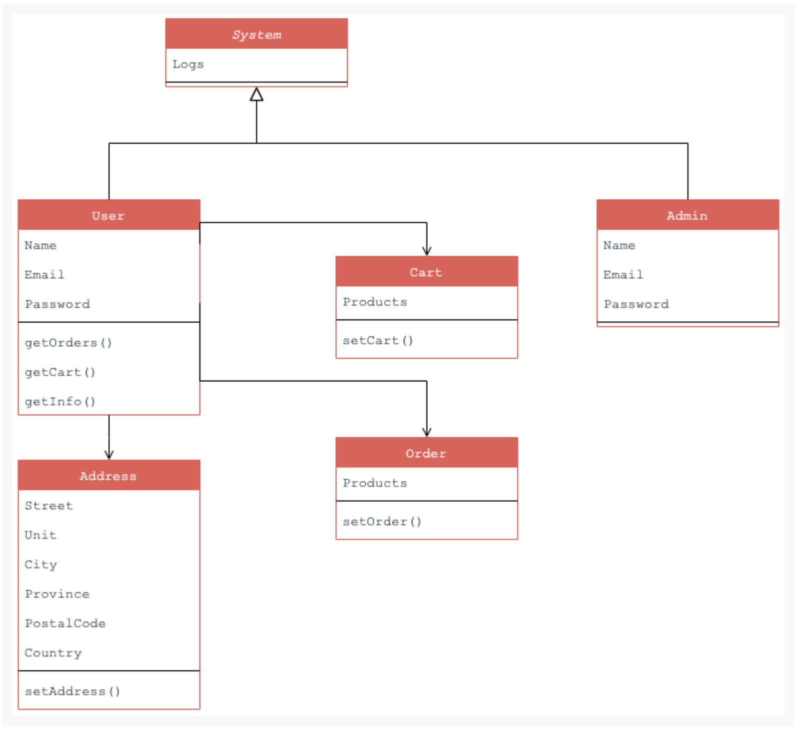


Figure 6: Class Diagram (User Account Component)

Implementation

In this section, we discuss some of the implementation decisions we made, how they deviated from our original plan and why the changes were made.

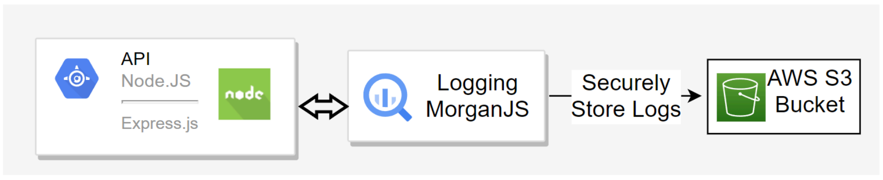


Figure 7: Cloud Usage and Logging

Logging using MorganJS and AWS

Logging of server API calls and events were done through **ExpressJS Morgan.** All log files are securely stored on a **AWS S3 bucket.** This storage is only **accessed by admins** and is **persistently stored.** Being a **cloud** based application, log files stored directly on the instance would be lost on restart which is why we store them on AWS S3. The decision to store on S3 was out of necessity because we did not take into account the ephemeral storage of the Heroku cloud platform. The tradeoff of this implementation is the added complexity of sending the logs out of the instance. **Multiple NodeJS packages** were also required to make this work. Logging events on the server increased our ability to test the web application. The logs show the **response time** of requests, the **number of requests** that were made and any **error codes** thrown.

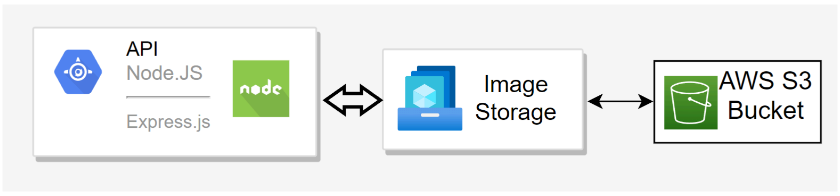


Figure 8: Cloud Usage and Storing Images

Media Storage using AWS

Persistent **image storage** is accomplished using a **AWS S3 bucket.** Heroku cloud platform dynos have limited storage volumes. Using S3 allows for **unlimited storage space** and allows SpeedyMart to be **highly scalable.** Hundreds of products can easily be added without affecting performance. Leveraging another cloud technology means our website is globally available without extensive maintenance overhead. The tradeoff is that more attention must be paid to create endpoints for each image. This adds complexity and latency as the images are stored on a different cloud service. Availability of the SpeedyMart website is **dependent on multiple platforms** being online which increases failure points. This approach limits our ability to test the image loading speeds since it is on a different cloud platform from our web application instance.

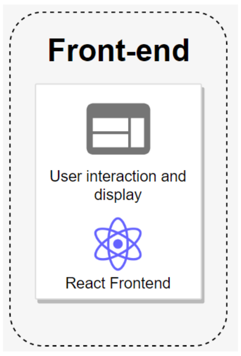


Figure 9: Fully built frontend with React framework

Why React?

The implementation of the frontend was highly based on an organised structure of **React components** and leveraging **REST API endpoints** to communicate with the backend. The front-end received the most amount of iteration while developing the web application. We intended to use React from the start, however we decided to limit usage of other packages such as Bootstrap to simplify our implementation. This limited us to **creating components from scratch** but allowed us full control of the look and performance of our website. React testing library allows for testing of the frontend. However, it is limited by how accurately it models every browser behaviour and the test coverage for the entire site would be time consuming to achieve.

Security

SpeedyMart runs under **SSL encryption.** The ‘https’ connection ensures that data is encrypted in-flight. SpeedyMart leverages best practices for securing environment variables by not exposing them on github and only storing them securely locally. We applied the principle of ‘least privilege’ where users are only given privileges needed to complete their tasks.

We tested for **SQL injection attacks** by inputting sql remove commands into text inputs on our website. For example, `SELECT * FROM Users WHERE Username='$username' AND Password='$password'` This was an attempt to trigger the database into performing something that was not meant to do. However, our backend is MongoDB(noSQL) so these commands would not work.

Cross-site scripting is another common security attack. Attackers attempt to execute malicious JavaScript within a victim’s browser. We attempted this attack by putting JavaScript code into text input elements on our webpage. For example, `<script>console.log('This should not print.')</script>`. This test resulted in no detected vulnerabilities from this kind of attack.

Admittedly, software security has a large background of specialisations. The attacks performed as **our test are not sufficient to test** a true attack by a professional malicious actor.

Performance

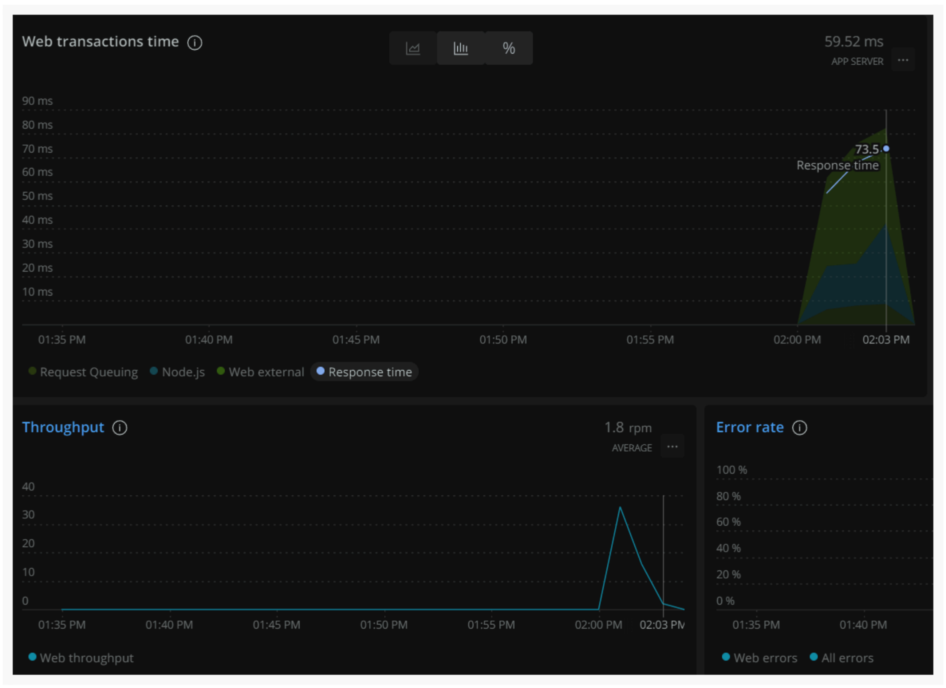
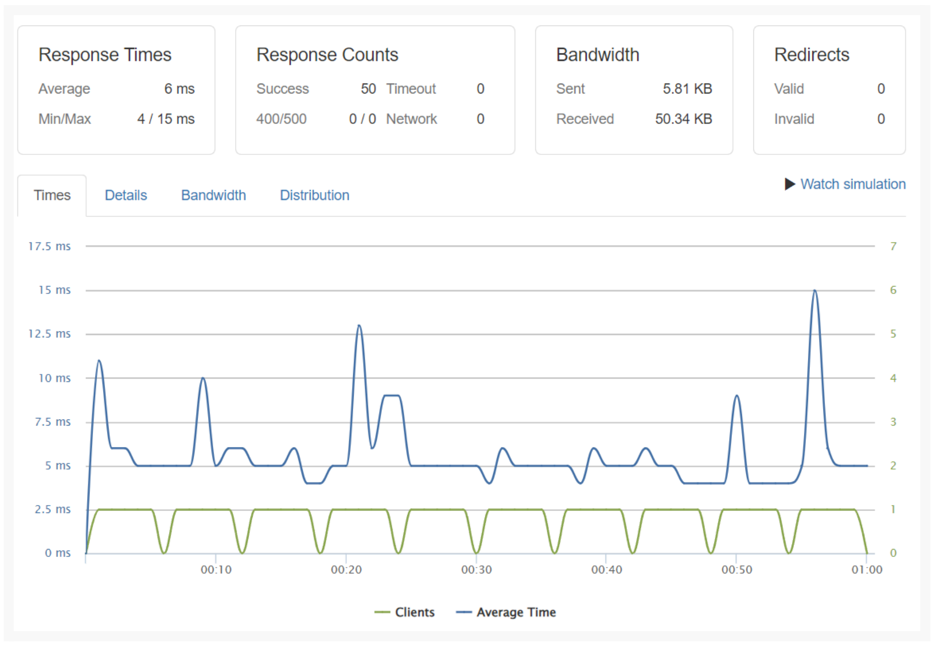


Figure 10: Throughput Graph and Transaction Time

Throughput

SpeedMart is generally **optimised** for web. A few factors are at play: hosting, images, component rendering, and HTML and CSS variations. Hosting is done on **Heroku** which **natively** allows developers to build, run, and deploy completely on the **cloud**. This eliminates communication with a local server and/or dependency on local files. Images are stored in **AWS S3** which is extremely efficient when it comes to **loading speed**. Component rendering is optimised through **React**, which is lighter and renders faster due to its **concurrent** rendering ability. Last but not least, during development, we knew there would be many calls to API endpoints. We used limited fonts and styling to create a **minimal website** to offset these call delays. **Scalability-wise**, using GraphQL (works nicely with noSQL databases) instead of REST API may lead to performance increases.

As mentioned, **response time** is crucial when it comes to performance. Related factors include **response count, bandwidth** availability, and number of **redirects**.



Response Times

Average	6 ms
Min/Max	4 / 15 ms

Response Counts

Success	50	Timeout	0
400/500	0 / 0	Network	0

Bandwidth

Sent	5.81 KB
Received	50.34 KB

Redirects

Valid	0
Invalid	0

Figure 11: Performance Tracking

Team Member Contributions

Work Division

Work was divided based on frontend and backend development.

Example workflow:

- Backend developer creates the database schema (MongoDB is a noSQL database so the structure must be created specific to the needs)
- Frontend developer creates the component to be rendered with static data (for testing purposes)
- Backend developer provides the endpoints and gets on a call to connect the backend with frontend components

Personal Statements

Phuong Tran: Worked on frontend components using React to render HTML components with CSS styling. Wrote functions for listener events using JavaScript (e.g. onClick(), onSubmit(), onChange(), ...etc.) Used states and React hooks to design and implement integration with API endpoints with help of “axios”. Created UML diagrams (Figures 3,4,5,6)

Philip Nguyen: Worked on the backend using NodeJS/ExpressJS to create REST API routes to handle requests/queries for the web application’s users, products and orders. Created MongoDB schemas and populated the MongoDB database. Added logging of the backend using Morgan and uploaded to AWS S3 bucket for security. Configured the project so that it automatically builds when pushed to Heroku cloud platform. Populated and hosted persistent image storage on AWS S3. Collaborated with frontend to create and test REST API endpoints. Wrote the Architecture, Security and part of the Implementation sections. Created diagrams of the application architecture (Figures 1,2,7,8,9).

Tarik Mohamad: Worked on checkout page (front-end)

Agreement Clause

Signature of agreement: By typing your name below, you agree that the above statements are true to the best of your knowledge.

Phuong Tran

Philip Nguyen

Tarik Mohammad

Relevant Project Information

Website

https://speedymart.herokuapp.com

Github

https://github.com/philipn7/SpeedyMart

Curl Commands

See **README.md**