Technical Manual 1.0

Project Team 3

Team Name: Clutch City Tech Solutions

UH | College of Technology

CIS 4375 Project Management & Practice

Fall 2020

PREPARED BY: Clutch City Tech Solutions

PREPARED FOR: CIS 4375 Project Management and Practice

DATE SUBMITTED: November 24, 2020

PROJECT SPONSOR: Lumico

CLIENT ACCEPTOR: Sil & Jackie Nguyen

PROJECT MANAGER: Daniel Howard

LAST EDIT: November 22, 2020

Contents

Application Development Lessons Learned

What worked?

React

- Using React for our application worked well in this case because it proved to be a fast-smooth experience for development as well as having tremendous online support. Many big E-commerce applications were written in React, so it made it a viable option for us. Since react is a JavaScript framework it was also very similar to Vue which is something all of us were familiar with the last term for our Enterprise Application Development course

GroupMe

- Using GroupMe proved to be useful for collaboration and communication. Everyone in the group was familiar with the user experience of GroupMe. This made it very simple to communicate our needs and group meetings

Microsoft SharePoint

- Microsoft SharePoint was a requirement for the project, but it turned out to be an excellent tool for the group to share documents and organize our documentation on the technical side. Bugs and issues were reported on a live excel document that could be viewed and edited by all members of the development team

Zoom

- To Demo our application for the rest of the group, the development team Used zoom for screen-sharing. This proved to work excellently by allowing other team members to take turns specifying what should be changed in the application.

Git/GitHub

- Git integration using GitHub was crucial in the collaboration of code between the members of the development team as well as members who were not actively working on the application. GitHub allows for different branches which allowed different members of the development team to work on different aspects of the application.

Mocha

- Mocha testing was beneficial in that it worked with our Mongoose schemas and we were able to create automated tests for our create, read, update and delete (CRUD) functionality in our application

Heroku

- By using Heroku and the Heroku CLI we were able to get our application ready and deployed to a live website in a matter of minutes. Heroku has paid and free options which make it flexible in that we can use the free tier until the application is accepted by the project sponsors to go live

PayPal/PayPal API

- In terms of payments, the PayPal API and documentation worked in our case because it provided us with a demo sandbox account that could easily be switched to live production mode to accept different forms of payments. PayPal is also a trusted and secure payment system which made it a simple pick for our payment method

Technical Documentation

- Reading all documentation of any JavaScript/React package proved to work in that the majority of packages had helpful documentation with examples to help us create different functionality in our application

Setting Early Deadlines

- Our application's first build was finished four weeks before the project's due date which gave us a lot of time to debug and look for different ways to improve our final build. I believe that this was crucial in us having a well-optimized application which very few bugs and functionality errors

What did not work?

Separate GroupMe Chats

- At first, we began a separate GroupMe chat for our application team to keep the development separate from the rest of the group project however we quickly found that it created communication problems between the development team and the rest of the group. We found that it was best to express all of our progress and problems in the main group to allow the entirety of the group to have input for the development

Custom CSS (Cascading Style Sheets)

- Editing each component in our application using vanilla CSS proved to be a menial task that was taking up more time than needed so we figured it would be better to use different CSS libraries such as React-Bootstrap, AOS, Material-UI, Tachyons and Semantic-UI react to create components in a quick manner that were functional and visually appealing.

React Hooks

- We began our application working with simple react hooks to manage our state, however as our application began to get more complex, we switched over to Redux for state management and found it better suited for our needs since we had many moving parts in our application state.

Stripe/Stripe API

- Stripe was an enticing option for us to use for payment, but we quickly ran to issues implementing it with our React application. One of the stripe packages we tried to use was outdated which would have created potential security problems for our users. Therefore, we implemented PayPal which was up to date and simple to implement while still being secure.

What would be done differently?

UML Diagrams

- Using a UML Diagram instead of an ERD would have been a better way to represent our data since we were using a MongoDB data model however the course project requirements required us to create an entity relational model.

AWS Deployment

- While Heroku was fine for us to use for our free tier, using a more established cloud service such as Amazon Web Services would have yielded us better performance for our application and faster loading times for processes. While deploying to AWS is more complicated than Heroku, it is more scalable and future proof.

Real Front-End Webserver (Apache/Nginx)

- One of the main things that could have been done differently was to use a real front end webserver such as Apache or Nginx so that we could do proper reverse proxying in our back-end server to ensure better performance and security

Different File Collaboration Tool

- Using SharePoint was able to yield us results but other collaboration tools such as google drive and Dropbox have more advanced features and more of our teammates were comfortable using these tools rather than Microsoft SharePoint.

New Technologies Utilized

Frontend technologies

React

Having used Vue previously in our Enterprise Application Development class we as a group felt that it would be beneficial as computer Information Systems students to be at the forefront of new technologies and work with the most popular web development framework used by major companies such as Netflix, Facebook, Yahoo!, Airbnb and others. React was at the forefront of our development and the ability to create reusable components helped speed up our development. React was also a relatively easy web development framework to use compared to vanilla JavaScript and Angular. React comes with numerous features and packages and has a large online support community that benefited us in development and debugging. For help with React, see its documentation at (https://reactjs.org/docs/getting-started.html).

Redux

One of the most challenging aspects of our development was how we were going to manage state within our application. We began by using react hooks to manage our different states and then moved on to Redux. This helped us manage how users, cart items, orders, and products were managed in our state. Redux maintains the state of our application using a state tree that cannot be changed directly. By using different actions and reducers in our application. For additional help with redux, see its documentation at (http://redux.js.org/introduction).

PayPal API

Something new to us that we had not learned in previous classes was the ability to rely on third-party services to assist with the more complex parts of our application. For us, at the beginning of the semester we realized that having a secure, reliable payment system was going to be crucial to the success of our application. We decided to use PayPal and integrate the PayPal API to manage all of the payments for our orders. The PayPal developer documentation made it very simple and straightforward and allowed us to set up a sandbox account to simulate store purchases. For more help working with PayPal, please see their documentation at (https://developer.paypal.com/docs/business/).

Email.js

To generate one of our reports we as a team decided that it would be beneficial to our customers that they receive a welcome email that gave them their Usernames and welcomed them to the website. This was possible by integrating a package called Email.js which took care of the email routing and automatically sent personally tailored emails to our new users after they had successfully registered. Email.js was simple and had great documentation as well as React compatibility which made it an easy choice to pick for our email client. You can find additional information about working with Email.js at (https://www.emailjs.com/docs).

Styling (Tachyons, AOS, React-bootstrap)

Coming into this application separate group members all had different knowledge that was self-taught when it came to styling our application. We decided to use CSS libraries because they had excellent documentation as well as being simple and easy to integrate into our application. One of the most important factors when building an application is to make sure that the user interface and user experience is simple while still being intuitive. Tachyons and React-bootstrap had many components that we were able to use such as input fields, different column formatting as well as shadows, and animations. The last styling library we implemented was AOS which is a library for transition animations. This library made our application flow well and create a pleasing user experience for customers as they are navigating through the different pages in our application. You can find Tachyons documentation at (https://tachyons.io/docs/). For more information about AOS, see (https://michalsnik.github.io/aos/). You can find additional information about working with React-bootstrap at (https://react-bootstrap.github.io/getting-started/introduction).

Backend Technologies

CSV-Stringify

This package along with Node.js (A technology that we have previously utilized in past courses) helped us convert our orders table in our application to an easily accessible CSV file that could be saved by the administration. This was the second report form that we used. This package also implemented the Node.js stream. Transform API and provided an easier synchronous and callback-based API. More information about working with CSV-Stringify can be found at (https://csv.js.org/stringify/).

Dotenv

More of a methodology than technology, the Dotenv package is a zero-dependency module that helps to load environmental variables from a .env file. This method and package create a more secure system because all of our important passwords and connection strings were saved as environmental variables in .env files and our deployment application. This made it so even if someone were to get access to our code, they could not connect to our database or access our PayPal and JWT keys. For more information about working with Dotenv, please refer to (https://www.npmjs.com/package/dotenv).

Multer

Multer is a middleware that works with node.js in the backend to handle multipart form data for uploading files. In our case, this was the best solution to upload images for the administration users to add new products to the store. This is because we are hosting our site on one server. If our site were hosted on many servers, we would need to have the images available in a centralized location. Multer does this by adding a body object and a file object to the request body JSON document that is parsed in the request handler. The body object contains the values of the text fields of the form. More information about Multer, see (https://www.npmjs.com/package/multer).

Browser Extensions

Redux Devtools

Redux Devtools is a browser extension the development team of the application used to make sure that the specific state actions and reducers were either failing or succeeding. This extension was compatible with Google Chrome and Mozilla Firefox. The redux Devtools also helped us see what information was being saved to the state such as the products listed, cart items, and current user logged in.

Tampermonkey

Tampermonkey is a browser extension that allows users to run custom JavaScript scripts on webpages. During testing, this extension allowed us to automate certain actions such as signing in, adding items to cart, or entering information at checkout. Effectively, this extension allowed our testing team to create debug tools in their browser that were completely independent of the project source code.

Communication Tools

GroupMe

Our team communicated on a common basis using GroupMe which is a popular team collaboration and communication social media application. GroupMe is compatible with android and iOS devices as well as being available on the web which made it accessible to all group members. In GroupMe, the team was able to communicate any hardships, conduct polls, and overall keep the flow of communication smooth throughout the project.

GitHub

GitHub is a website that provides hosting for software development and was used by the development team to collaborate and share code. The main repository was called LumicoMain and it contained separate branches for different members of the development team to work on. After every team member was completed with their code on their separate branch all of the data was merged to the main branch and made ready for deployment

Testing Technologies

Mocha

Mocha was the preferred library for testing our application. Unit testing began with the creation of assertion based Mocha scripts on our express server. Our mocha scripts used the server's internal methods to create, read, update, and delete data. After our development team developed all of these scripts, they were used to test the different functions within our application to make sure everything was running smoothly.

Business Requirements

User Interface Requirements

Most users today have the expectation that user interfaces will work at the home or office and on the go. The employees at Lumico were no exception. To accommodate this desire, our development team used React, a JavaScript framework, and Tachyons, a style sheet framework, to ensure that pages would be mobile friendly, fast, and responsive.

Security Requirements

At Clutch City Tech Solutions, we take security seriously. To maintain the reputation of Lumico and keep Lumico's clients safe, we outsourced as much sensitive information as possible to trusted third party platforms like Heroku, PayPal, and MongoDB Atlas. Payment processing will be handled by PayPal, a reputable payment processing platform. The entire site is hosted on Heroku. Sites hosted on Heroku have Transport Layer Security (TLS) enabled; therefore, all connections made to the site are secure. This level of security is guaranteed by Heroku's certificate that was issued by DigiCert Inc. The database is hosted on MongoDB Atlas. The main advantage of MongoDB Atlas is that it supports secure and scalable database connections. We could not, however, outsource all security.

Employees and users are responsible for using secure passwords. When a user or employee creates an account, their password must contain at least one number, one uppercase, and one lowercase letter. Additionally, passwords must be at least eight digits long. We believe that this policy will encourage more secure passwords; however, employees and users are responsible for making passwords that are difficult to guess.

Performance and Capacity

Part of Clutch City Tech Solution's goal was to provide Lumico with a database that could scale effectively. Consequently, we developed a transactional database that uses a document-based design. While transactional databases tend to use a relational design, we believe that the document-based design offered by MongoDB will scale well and be more maintainable. Using MongoDB also allowed our team to host the database on MongoDB Atlas. Consequently, as long as the database design is effective, scaling the database to handle more capacity should be as easy as paying for more capacity.

Data Input and Output Requirements

To detail the required data input and output, we created a data dictionary and a CRUD matrix. As seen below, our CRUD matrix explains what create, read, update, and delete operations admins, employees, and customers can perform. The CRUD matrix also helps clarify the flow of data. For instance, it is easy to see that admins need to be able to send and receive all types of data.

User Role → ↓ Document Name	Employee	User	Notes
User	CRUD	CRU	Users cannot update the isAdmin property
Product	CRUD	R	
orderItems	CRUD	CRUD	
shippingAddress	CRUD	CRUD	
paymentResult	CR	CR	Users do not create payment results directly: they are created by the payment provider
Order	CRU	CR	

Table 1:CRUD matrix describing each role's input and output requirements

We also modeled the Lumico Webstore's data flow with data flow diagrams. The context data flow diagram in Figure 2 shows the highest-level view of the data within the Lumico Webstore. In summary, users can create an account, browse the store, and create an order. Employees can manage products, manage users, browser orders, and update orders. Finally, we included PayPal as an external entity because users send their payment to PayPal for processing. Once PayPal processes the payment, a payment result is returned to the Lumico Webstore.

The level zero data flow diagram in Figure 3 shows the system in more detail. It shows the process of users browsing products, placing orders, and so on. The process of signing in users, placing orders, managing products, and managing users are shown in more detail in their respective level one data flow diagrams in Figure 4, Figure 5, Figure 6, and Figure 7.

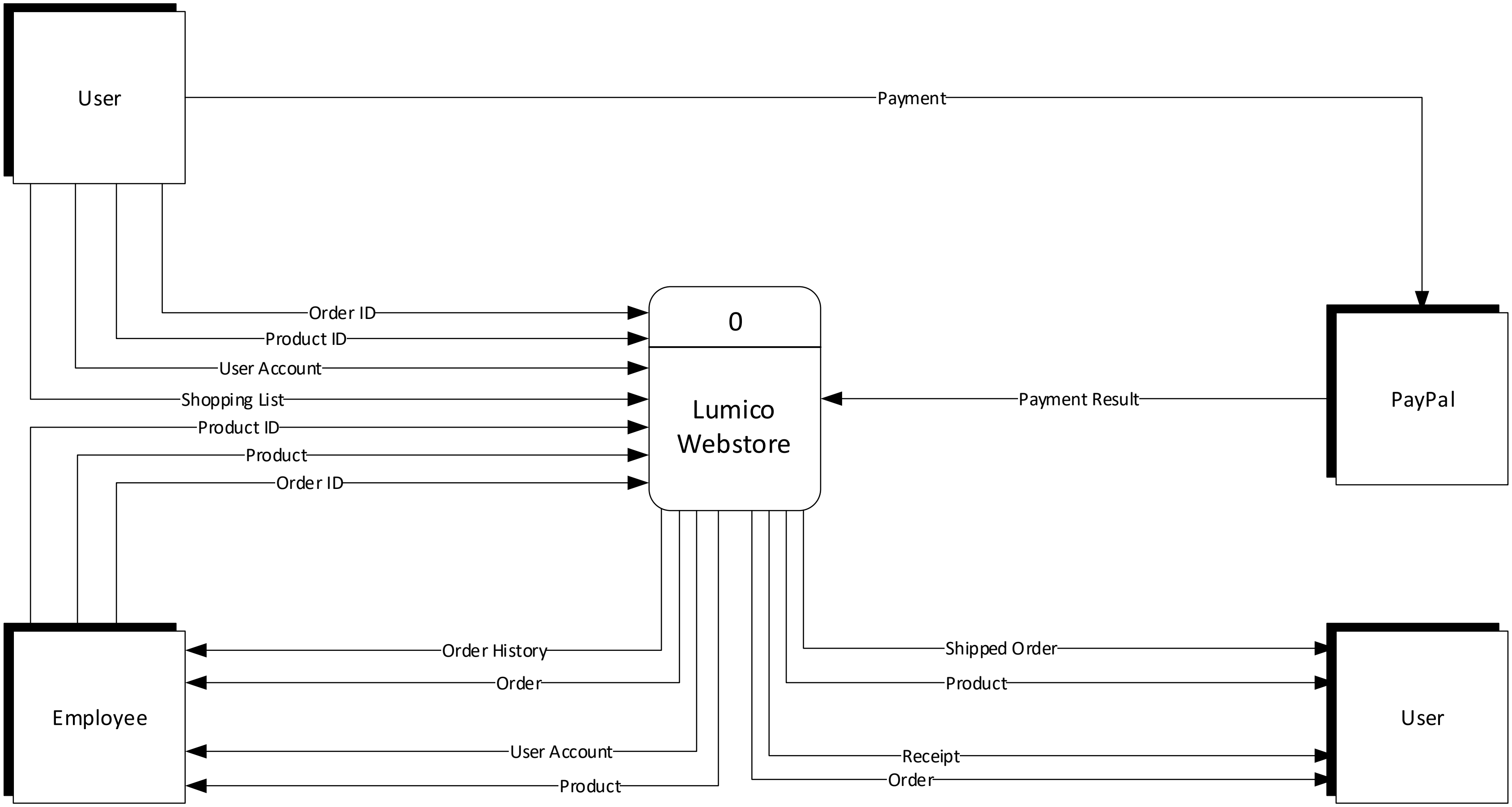
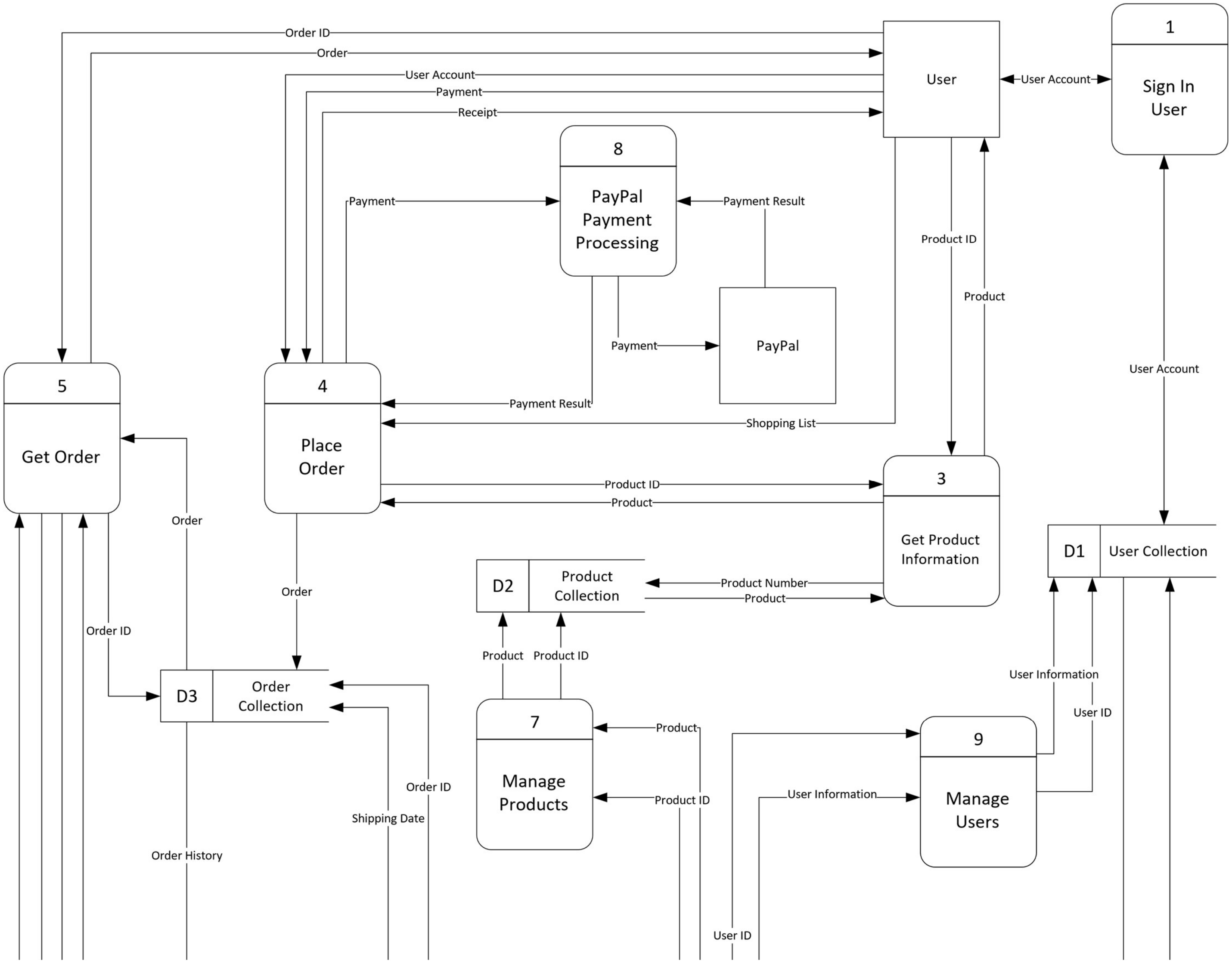


Figure 2:Lumico Webstore Context Data Flow Diagram



Order ID
Order
User Account
Payment
Receipt
User
User Account
1
Sign In User
8
PayPal Payment Processing
Payment
Payment Result
Product ID
Product
Payment
PayPal
User Account
5
Get Order
4
Place Order
Payment Result
Shopping List
Product ID
Product
3
Get Product Information
Order
Order
D1
User Collection
D2
Product Collection
Product Number
Product
Order ID
Product
Product ID
D3
Order Collection
User Information
User ID
7
Manage Products
Product
9
Manage Users
Order ID
Product ID
User Information
Shipping Date
Order History
User ID

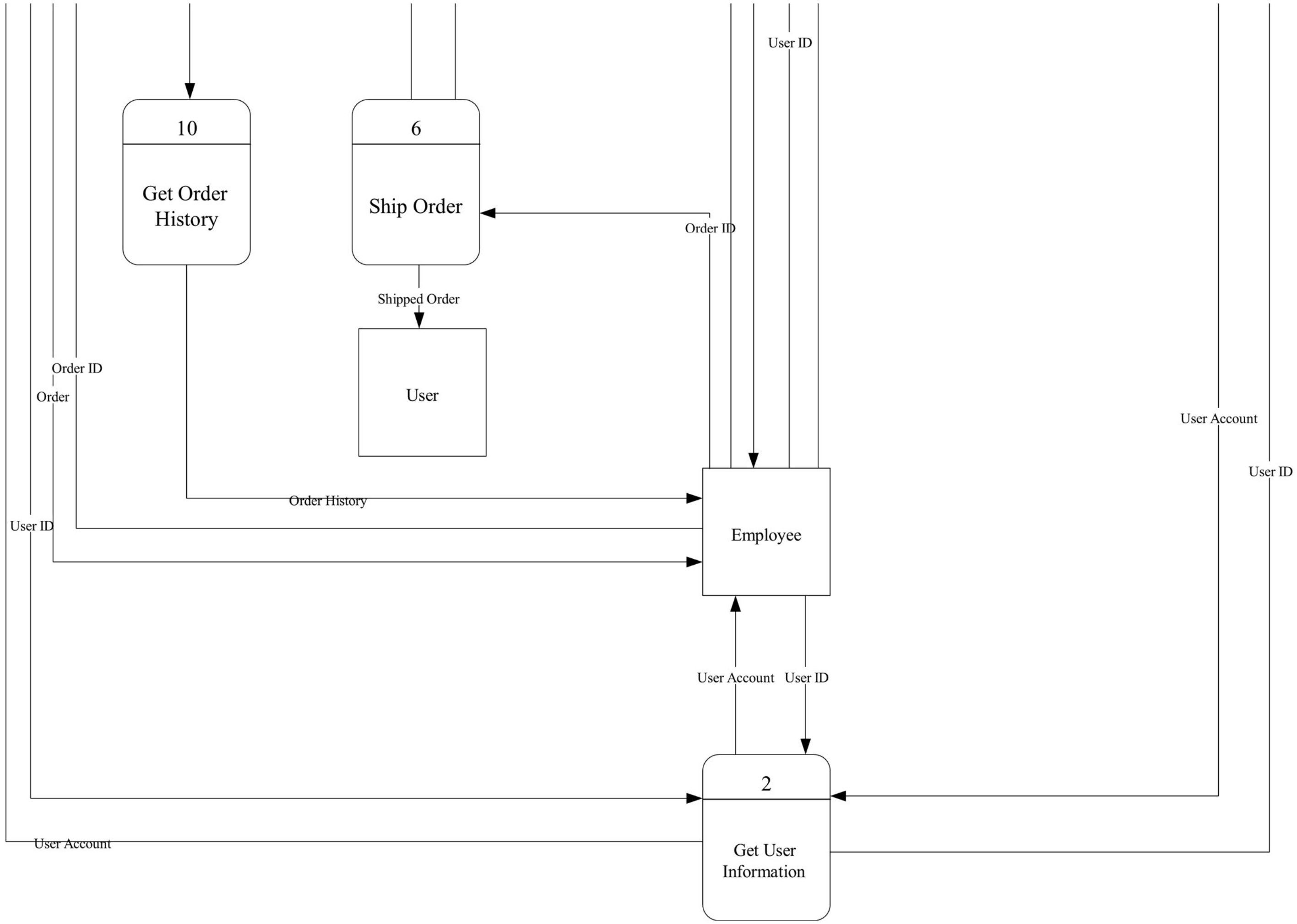


Figure 3:Lumico Webstore Level 0 Data Flow Diagram

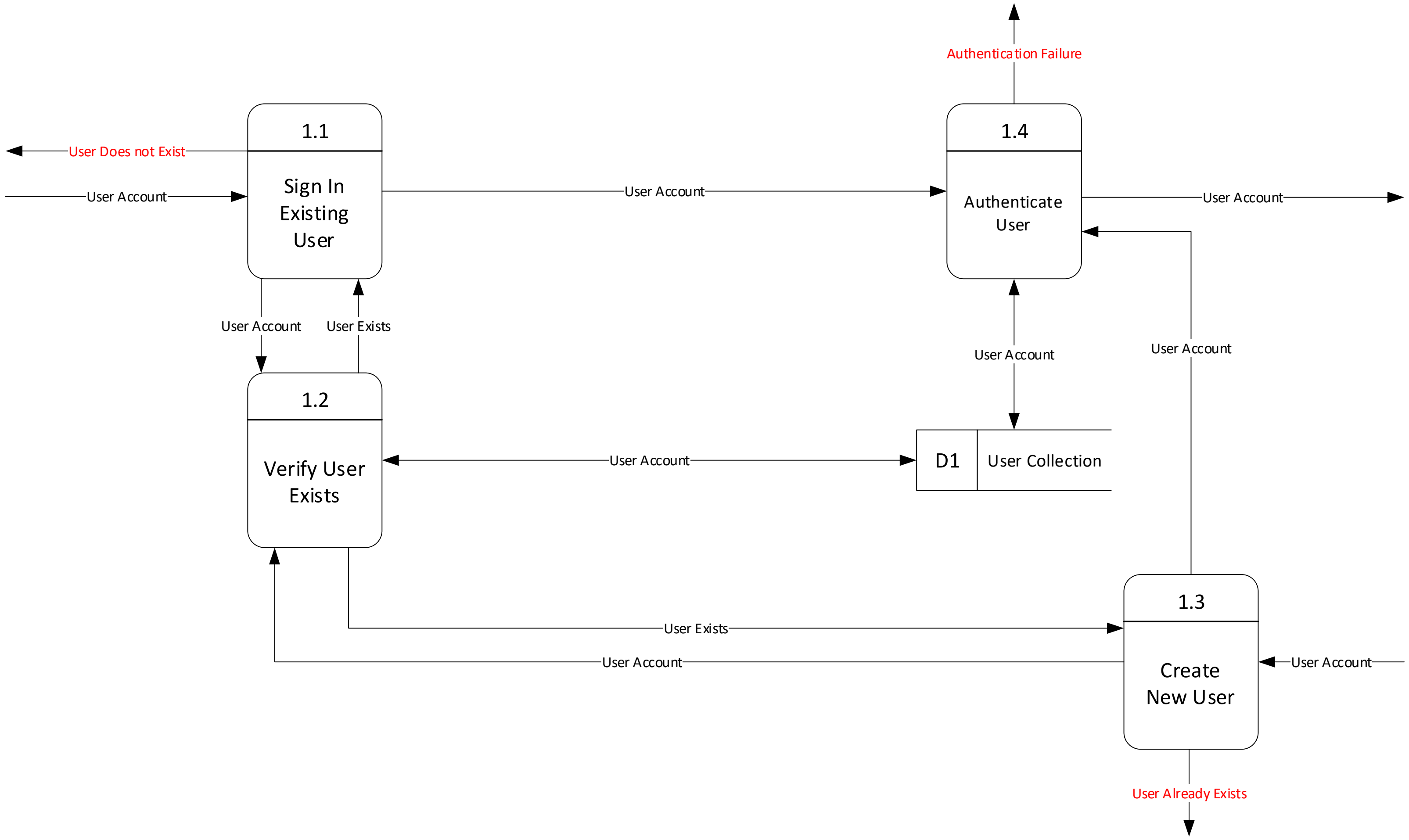


Figure 4:Lumico Webstore Level 1 Data Flow Diagram: Sign In User

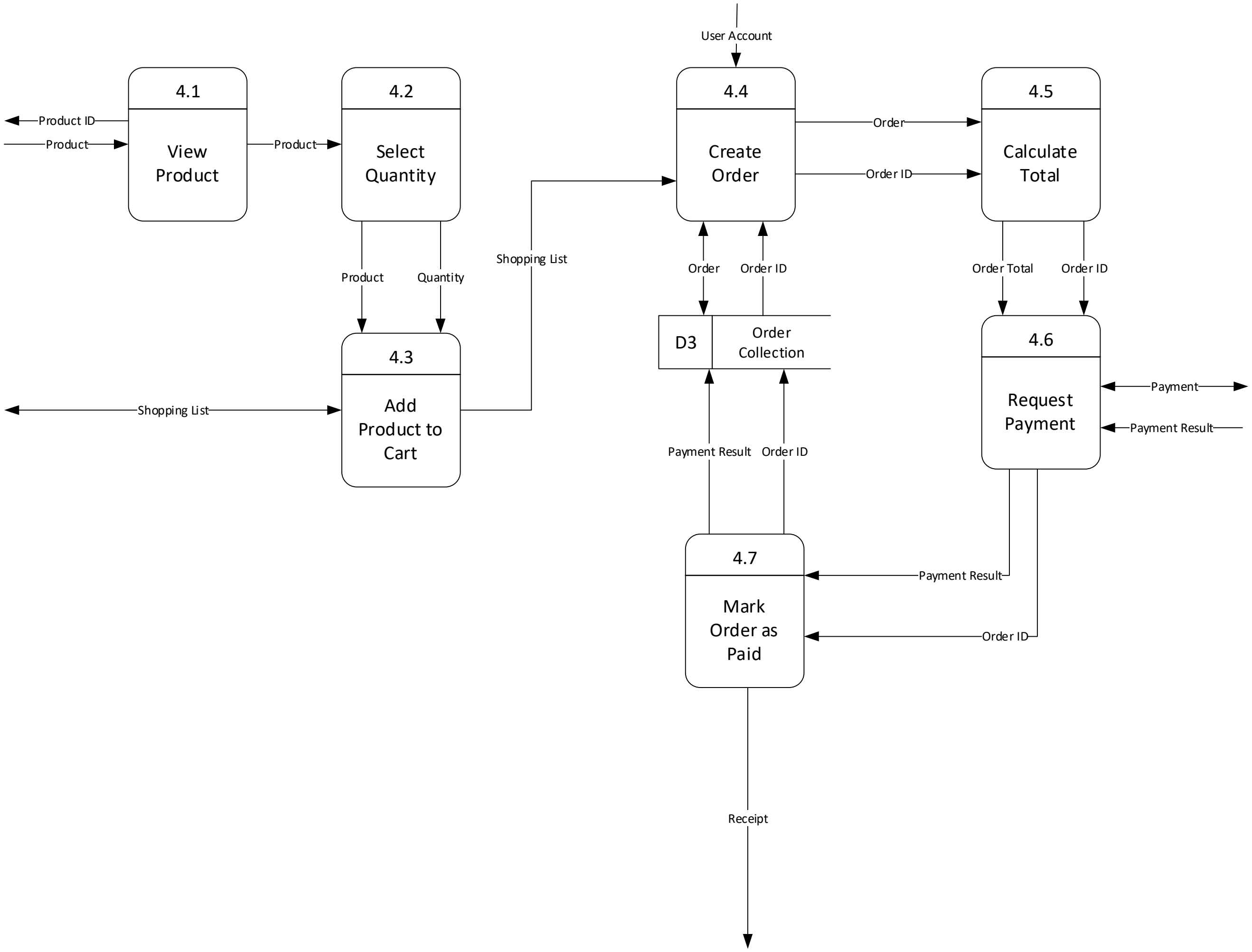


Figure 5:Lumico Webstore Level 1 Data Flow Diagram: Place Order

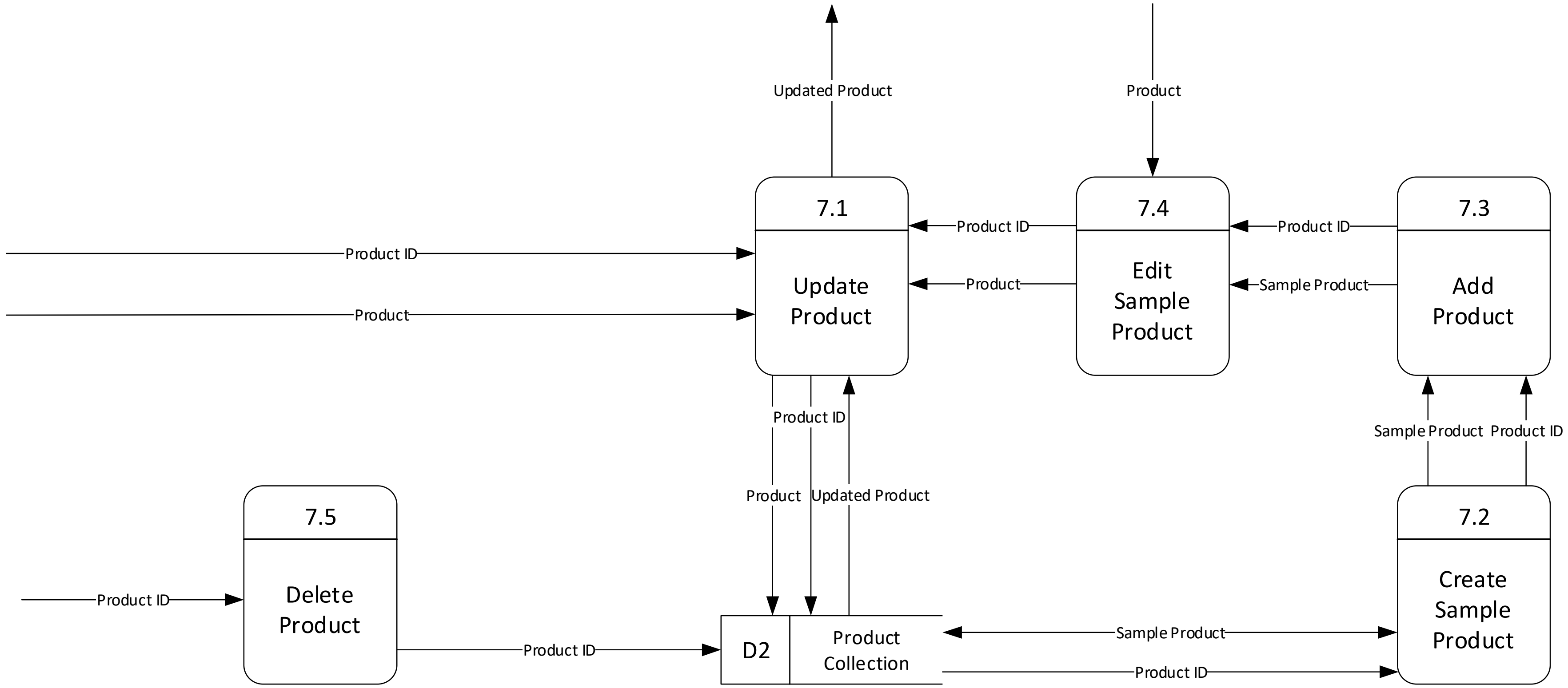


Figure 6:Lumico Webstore Level 1 Data Flow Diagram: Manage Products

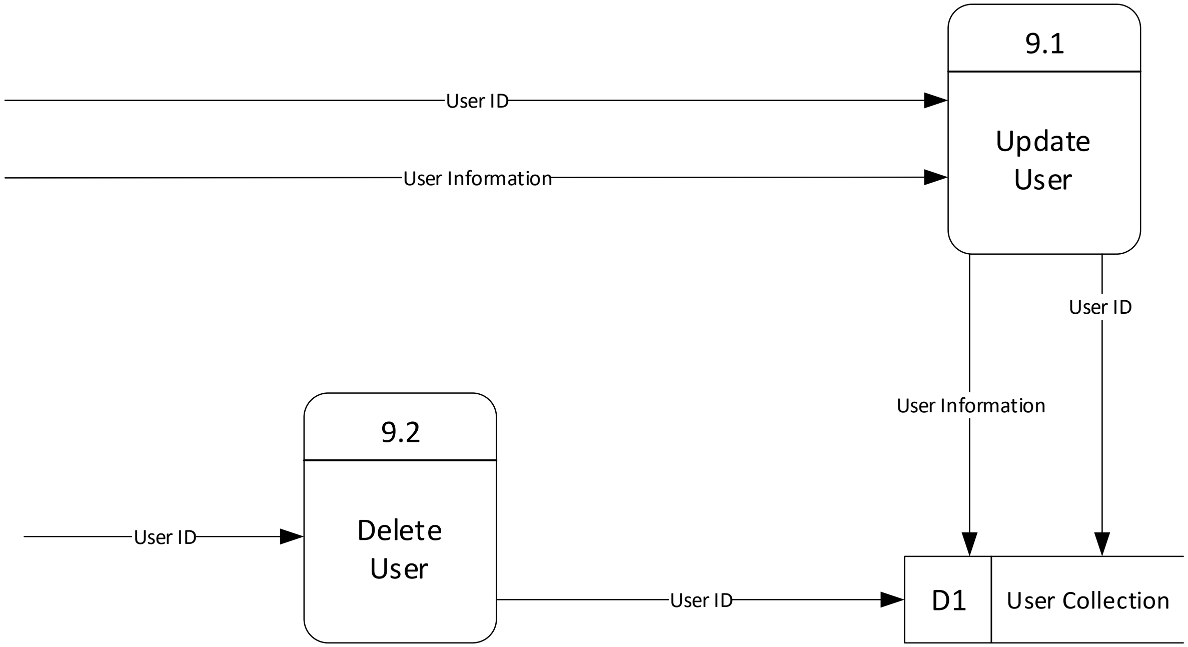


Figure 7:Lumico Webstore Level 1 Data Flow Diagram: Manage Users

Migration Requirements

As the client currently holds all data on Microsoft Excel and paper, our team aims to transition that to a database. We plan to coax the data in their Excel documents into the correct format for the database, export the data to csv format, and then import it directly into the database. The data they store on paper will be more challenging to import.

One idea we have is to use optical character recognition (OCR) to convert the paper data to a digital format. OCR is not perfect, however, so we will need to manually ensure that the data was recognized correctly. From there, we should be able to use the same process we used to import the Excel data.

Data Cleanup, Backup, Recovery and Archival

Data Clean Up

Since our database was built with the idea that everything was going to be unique, each product, user and order is assigned a unique ID number. This ensures that no duplicate IDs exist in the data. We also implemented data sterilization techniques on our web interface.

We used HTML forms to facilitate user input. Each HTML form has validation that ensures that data is in the correct format before it makes its way to the database. Additionally, we have validation on the server itself to check the validity of the data a second time.

Data Recovery & Backup

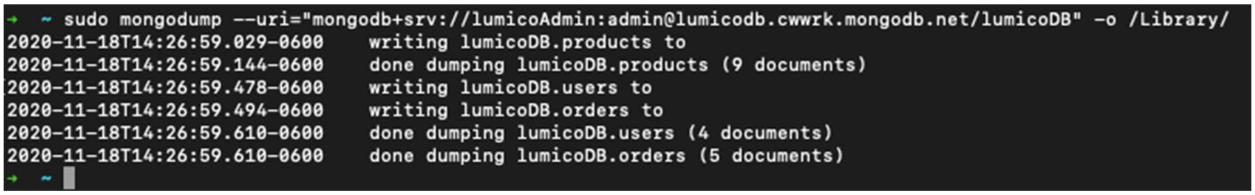
At the moment we are using a M0 Sandbox database tier in MongoDB stored on AWS. This free tier does not come with automated backup, so we decided to store our MongoDB backups as local files using the ***mongodump*** utility that creates a binary export of all contents that our in the database. After we used ***mongodump*** we then used ***mongorestore*** in order to load data from both our binary database dump or the standard input into a ***mongod*** or mongos instance.

Backup

To create a backup file with ***mongodump***, run the following command:

```
mongodump --uri="mongodb+srv://lumicoAdmin:admin@lumicodb.cwwrk.
mongodb.net/lumicoDB"
```

This will create a backup file that can be stored locally.



```
→  ~ sudo mongodump --uri="mongodb+srv://lumicoAdmin:admin@lumicodb.cwwrk.mongodb.net/lumicoDB" -o /Library/
2020-11-18T14:26:59.029-0600    writing lumicoDB.products to
2020-11-18T14:26:59.144-0600    done dumping lumicoDB.products (9 documents)
2020-11-18T14:26:59.478-0600    writing lumicoDB.users to
2020-11-18T14:26:59.494-0600    writing lumicoDB.orders to
2020-11-18T14:26:59.610-0600    done dumping lumicoDB.users (4 documents)
2020-11-18T14:26:59.610-0600    done dumping lumicoDB.orders (5 documents)
→  ~ 
```

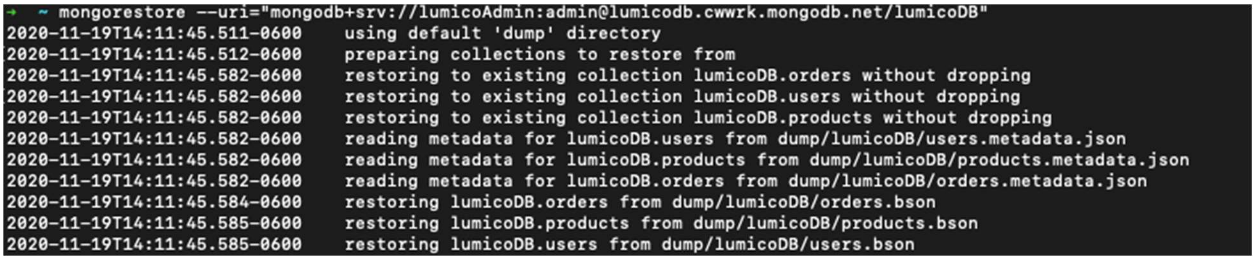
Figure 8:Example of running a backup with mongodump

Recovery

To restore the database from a backup file with ***mongorestore***, use the following command***:***

```
mongorestore --uri="mongodb+srv://lumicoAdmin:admin@lumicodb.
cwwrk.mongodb.net/lumicoDB"
```

This command will restore data from the mongodump file previously created.



```
➜  ~ mongorestore --uri="mongodb+srv://lumicoAdmin:admin@lumicodb.cwwrk.mongodb.net/lumicoDB"
2020-11-19T14:11:45.511-0600    using default 'dump' directory
2020-11-19T14:11:45.512-0600    preparing collections to restore from
2020-11-19T14:11:45.582-0600    restoring to existing collection lumicoDB.orders without dropping
2020-11-19T14:11:45.582-0600    restoring to existing collection lumicoDB.users without dropping
2020-11-19T14:11:45.582-0600    restoring to existing collection lumicoDB.products without dropping
2020-11-19T14:11:45.582-0600    reading metadata for lumicoDB.users from dump/lumicoDB/users.metadata.json
2020-11-19T14:11:45.582-0600    reading metadata for lumicoDB.products from dump/lumicoDB/products.metadata.json
2020-11-19T14:11:45.582-0600    reading metadata for lumicoDB.orders from dump/lumicoDB/orders.metadata.json
2020-11-19T14:11:45.584-0600    restoring lumicoDB.orders from dump/lumicoDB/orders.bson
2020-11-19T14:11:45.585-0600    restoring lumicoDB.products from dump/lumicoDB/products.bson
2020-11-19T14:11:45.585-0600    restoring lumicoDB.users from dump/lumicoDB/users.bson
```

Figure 9:Example of restoring a backup with mongorestore

Data Archival

Mongodump is also used for our data archival process. First to destroy all of the data using the terminal, navigate to the root directory in the project folder. After that, run the **npm run data:destroy.** This will run a seeder script that will load dummy data from our products.js file in order to restore the application data to its initial phase. After this, follow the steps for revering a backup file using mongostore to retrieve any backup data.

Documentation and Training Requirements

One of our team members studied instructional design in the past and developed an approach to instruction for a website very similar to this one. The approach they used to develop training included steps resembling the analysis, design, development, implementation, and evaluation (ADDIE) model. The first step is to identify if there is a problem that can be solved with training. This is something that CCT Solutions will do as application testing commences. The second step, if training would prove useful, is to identify the root causes of the problems with Pareto analysis. Given the root causes, we will perform context and learner analysis to help us identify the general characteristics of the learners such as their age, gender, and education level so that the training can be tailored to their needs. Following this, we will perform procedural analysis, a technique used to identify the steps involved in completing a task. Then we will sequence our instruction, develop videos and documentation about every problem identified, and implement guided design to walk employees through the training.

Data Model

The entity relationship diagram (ERD) presented in Figure 10 illustrates all entities, their attributes, and their relationships with other entities. As mentioned in the project improvements section, we quickly learned that ERDs are not well suited for MongoDB data modeling. MongoDB discourages relationships among entities because it has no built-in relationship integrity guarantees. Instead of using an ERD, it would have been better to use Unified Modeling Language (UML) to model documents and nested documents. Each document in MongoDB is akin to an object in an object-oriented programming language. To summarize, Figure 10 shows that users create orders. Each order is associated with a list of order items. An order item represents a certainty quantity of a product. An order is also associated with a shipping address. When a user pays for their order, a PayPal payment result is added to the order. Users can also create products if they have administrative access.

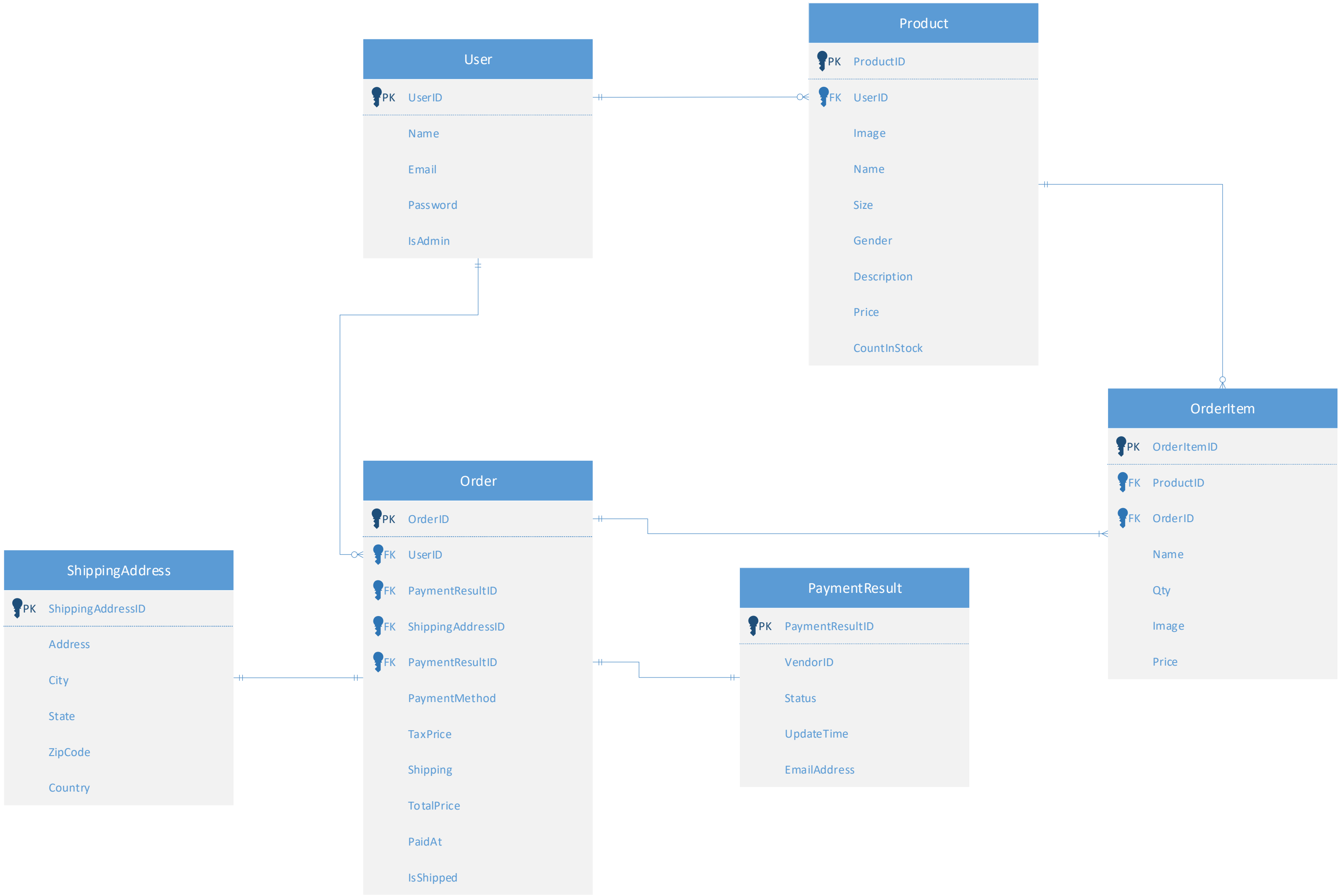


Figure 10:Lumico Webstore Entity Relationship Diagram

Table 2 shows our data dictionary. The data dictionary explains the format, storage location, and relationships of all data stored in the database. It is to the entity relationship diagram what the work breakdown structure dictionary is to the work breakdown structure.

Database Name	Document	Attribute Name	Data Type	Length	Format
LumicoDB	User	_id	ObjectId	N/A	N/A
LumicoDB	User	name	String	200	N/A
LumicoDB	User	email	String	200	^[\w\.=-]+@[\w\.-]+\.[\w]{2,3}$
LumicoDB	User	password	String	200	^(?=.*\d)(?=.*[a-z])(?=.*[A-Z]).{8,}$
LumicoDB	User	isAdmin	Boolean	1	N/A
LumicoDB	Product	_id	ObjectId	N/A	N/A
LumicoDB	Product	user	ObjectId	N/A	N/A
LumicoDB	Product	name	String	200	N/A
LumicoDB	Product	image	String	200	N/A
LumicoDB	Product	size	String	200	N/A
LumicoDB	Product	_id	ObjectId	N/A	N/A
LumicoDB	Product	gender	String	200	N/A
LumicoDB	Product	category	String	200	N/A
LumicoDB	Product	description	String	2000	N/A
LumicoDB	Product	price	Number	N/A	^\$[0-9]+?\.[0-9]{2}$
LumicoDB	Product	countInStock	Number	N/A	^[^-.][0-9]+$
LumicoDB	orderItems	name	String	200	N/A
LumicoDB	orderItems	qty	Number	N/A	N/A
LumicoDB	orderItems	image	String	200	N/A
LumicoDB	orderItems	price	Number	N/A	N/A
LumicoDB	orderItems	product	ObjectId	N/A	N/A
LumicoDB	shippingAddress	address	String	1000	N/A
LumicoDB	shippingAddress	city	String	1000	N/A
LumicoDB	shippingAddress	state	String	2	^[A-Z]{2}$
LumicoDB	shippingAddress	zipCode	String	10	^([0-9]{5})(-[0-9]{4}){0,1}$
LumicoDB	paymentResult	id	String	200	N/A
LumicoDB	paymentResult	status	String	200	N/A
LumicoDB	paymentResult	update_time	String	200	N/A
LumicoDB	paymentResult	email_address	String	200	^[\w\.=-]+@[\w\.-]+\.[\w]{2,3}$
LumicoDB	Order	_id	ObjectId	N/A	N/A
LumicoDB	Order	user	ObjectId	N/A	N/A

LumicoDB	Order	orderItems	orderItems	N/A	N/A
LumicoDB	Order	shippingAddress	shippingAddress	N/A	N/A
LumicoDB	Order	paymentMethod	String	200	N/A
LumicoDB	Order	paymentResult	paymentResult	N/A	N/A
LumicoDB	Order	taxPrice	Number	N/A	N/A
LumicoDB	Order	shipping	Number	N/A	N/A
LumicoDB	Order	totalPrice	Number	N/A	N/A
LumicoDB	Order	paidAt	Boolean	1	N/A
LumicoDB	Order	isShipped	Boolean	1	N/A

Table 2:Lumico Webstore Data Dictionary

Since we used MongoDB, the data dictionary looks a little bit different than it would if we had used a relational database. For example, in Figure 2, it says that the Order document contains orderItems. At the same time, Figure 2 shows orderItems as a discrete document in the data dictionary. This pattern describes the presence of a nested document. OrderItems appears as a discrete document so that its properties can be detailed; however, in practice, orderItems is contained inside of the Order document. The same is true for paymentResult, shippingAddress, and anywhere where the Data Type column is the name of another document.

Production and Deployment

Prerequisites

A few prerequisites are required prior to starting the production and deployment phases

1. **Git Setup**
    a. Make sure to have Git working on your machine by navigating to: https://git-scm.com/downloads
    b. Make sure that you have cloned the most recent repository **LumicoMain** into a safe local directory. In order to perform this task, navigate to your desired directory and run:

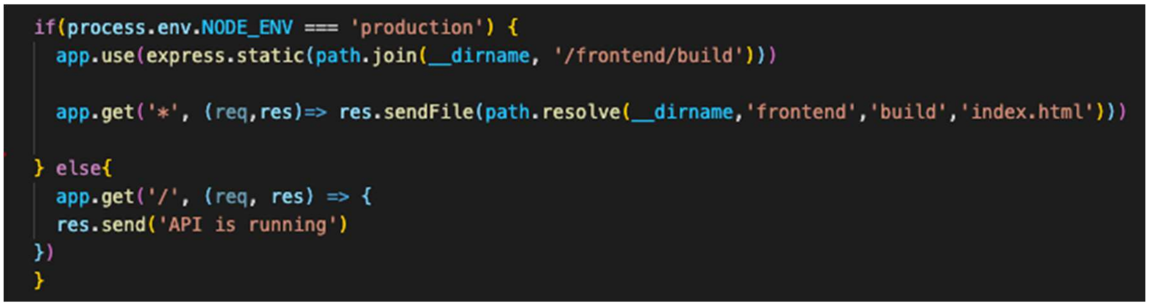
       git clone https://github.com/musab118/LumicoMain.git

    c. After the installation has finished run: **npm install** to install the backend dependencies
    d. After the first install is complete run **cd frontend** to navigate to the frontend folder of the project and once again run **npm install** to install all front-end dependencies
2. **Heroku CLI**
    a. **Steps for Windows**
        i. Navigate the heroku dev center and download the windows installer
    b. **Steps for MacOS**
        i. Install Homebrew if not already. Steps are located on the homebrew website: https://brew.sh
        ii. Run: **brew tap heroku/brew && brew install heroku**
    c. For more information refer to the Heroku developer documentation at **https://devcenter.heroku.com/articles/heroku-cli**

Production

a. To prepare for deployment our application must first be ready for production

b. Add the following block of code in the server.js if not already available as shown in Figure 11:



```
if(process.env.NODE_ENV === 'production') {
  app.use(express.static(path.join(__dirname, '/frontend/build')))

  app.get('*', (req,res)=> res.sendFile(path.resolve(__dirname,'frontend','build','index.html')))

} else{
  app.get('/', (req, res) => {
  res.send('API is running')
})
}
```

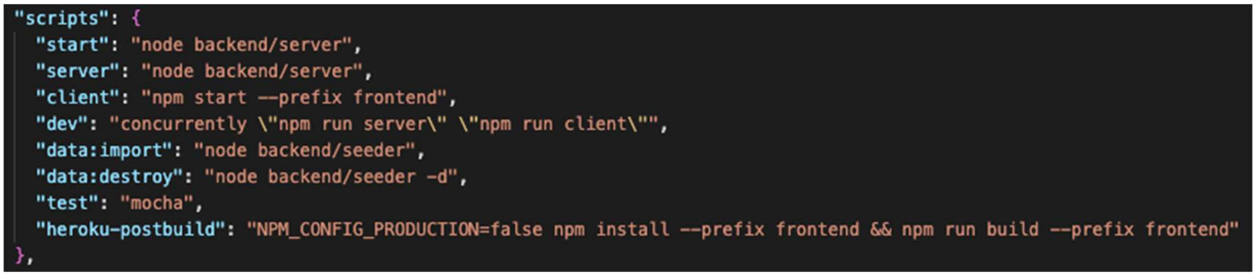
Figure 11:Configuring server.js for a production environment

c. This code block checks whether the application is in production mode according the environment variables that will be set up in the deployment phase

Deployment

a. After the production steps have been completed the application is ready for the deployment step

b. In the package.json file located in the root directory of the application add the heroku-postbuild script under the “scripts” section as shown below in Figure 12:



```
"scripts": {
  "start": "node backend/server",
  "server": "node backend/server",
  "client": "npm start --prefix frontend",
  "dev": "concurrently \"npm run server\" \"npm run client\"",
  "data:import": "node backend/seeder",
  "data:destroy": "node backend/seeder -d",
  "test": "mocha",
  "heroku-postbuild": "NPM_CONFIG_PRODUCTION=false npm install --prefix frontend && npm run build --prefix frontend"
},
```

Figure 12:What the npm scripts should look like in package.json

c. In the terminal since we already have the heroku CLI installed run **heroku login.** This will launch a browser windows which will allow you to login to your personal heroku account

d. In order to create an application run: **heroku create APP_NAME** with the APP_NAME being a unique application name that hasn’t been used by other heroku members in this case it was lumicoapp.

e. Create a file in your root directory simply named Procfile with not extension containing the following data: **web: node backend/server.js**

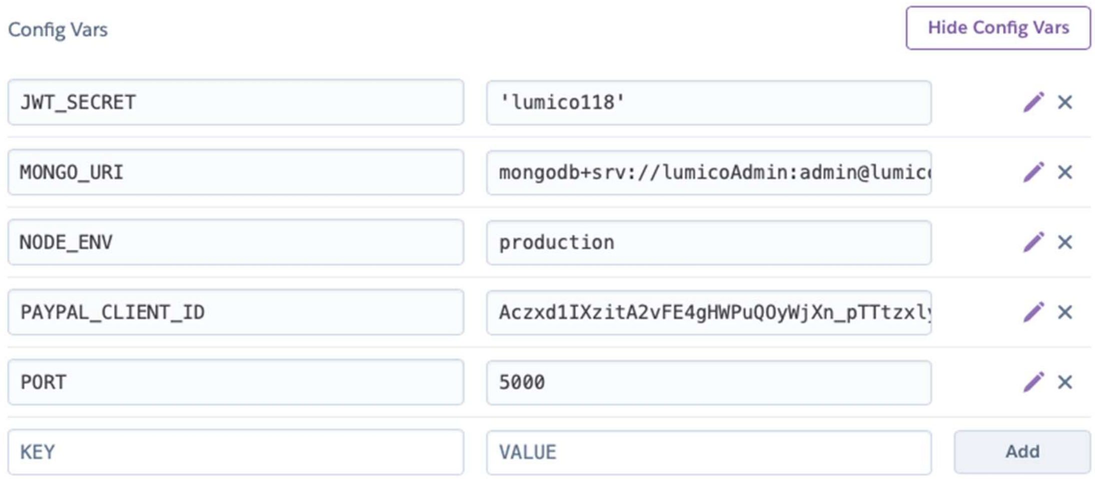
f. Run **git add .** and **git commit -m “YOUR_COMMIT_MESSAGE”** to add all changed files and commit them.

g. Navigate to the heroku.com and login with your account. You should see your created application. After selecting your application, navigate to the deploy tab and you should see the steps to add our heroku project as a remote repository. Run the command: **heroku git: remote -a lumicoapp.** This will add a heroku remote repository that we can make changes on and deploy at any time.

h. After all the previous steps have been completed run: **git push heroku main.** This will take some time but after the process is complete the application has been deployed with errors

i. In order to resolve these errors, we must once again navigate to heroku.com and in our application dashboard under the settings tab select the **Reveal Config Vars** button. This will allow us to add our unique environment variables

j. Add the environment variables as shown in Figure 13 (you can find these in Appendix VIII):



JWT_SECRET	'lumico118'
MONGO_URI	mongodb+srv://lumicoAdmin:admin@lumic
NODE_ENV	production
PAYPAL_CLIENT_ID	Aczxd1IXzitA2vFE4gHWPuQ0yWjXn_pTTtzxl
PORT	5000
KEY	VALUE

Figure 13:Example of configuring environment variables on heroku.com

k. After that the application will be live on heroku at our link: **https://lumicoapp.herokuapp.com**

Lumico Disaster Recovery Plan

The following Disaster Recovery Plan is to ensure that operations at Lumico can be restored and any damages repaired in a timely manner. The purpose of this plan is as follows:

- To minimize disruptions in normal day to day operations
- To limit the extent of damage and/or disruptions
- To minimize the economic impact of the interruption
- To establish alternate means to operate or conduct business in advance
- To provide training to personnel with emergency procedures
- To provide smooth and rapid restoration of services

In the Event of a Disaster

In the event of a disaster, the appropriate emergency personnel must be notified immediately and prior to any other actions taking place. The personnel that must be notified are the CEO, CFO, and CTO of Lumico and local emergency services if needed. These members must then immediately contact the disaster recovery team. By doing so, reducing the damage caused by the disaster could potentially be minimized and can aid in faster recovery and resumption of normal operating procedures.

Security

Proper security measures must be followed at all times by all members of the Lumico staff. These security measures include but are not limited to:

- Password sharing is prohibited
- Only approved applications may be downloaded on company equipment
- Access to databases will be limited to only the appropriate parties
- Access to the web application code is restricted to appropriate parties

Please refer to the Lumico Security Policy for further guidelines and policies.

By adhering to these security measures, Lumico can limit the possibilities of a security breach disaster and if there is a breach, identify the breach faster and resume normal operations.

Natural Disasters

In the event of a natural disaster, please follow and refer to the recommendations of local emergency services. Your safety is of more importance than continuing Lumico operations. Once an all clear has been announced, the disaster recovery team must be contacted in order to identify damages and proceed with a recovery plan.

Protection of Data

In order to protect the data of Lumico customers and employees, all data must be stored remotely. No localized storage of this data must occur unless approved and authorized by the CEO/CFO and CTO. Passwords to this remote storage must not be shared with anyone in order to protect the privacy of this personal data. In the event that this data must be stored locally, it must be stored on a removable drive and must be on your person at all times. Leaving this data in a vehicle or unsecured in any manner is unacceptable and could lead to a breach in security.

Conclusion

The purpose of the Lumico Disaster Recovery Plan is an attempt to mitigate damages and losses in the event of a natural disaster and/or breach of security. Implementing and following this plan can potentially reduce damages caused and help in a rapid recovery and resumption of normal business operations.

Customer Relationship Policy

Clutch City Tech Solutions' strives to produce the best Information Technology solutions and support suited for all types and sizes of businesses.

To accomplish this mission, we agree upon these values:

- Illustrate all the possible solutions that meets the client's goal.
- Plan the solution after thoroughly reviewing all the possible scenarios.
- Design the solution that is easy to use and understand.
- Implement the system in compliance with the client.
- Provide the best support service.

General Guiding Principals

- We only consider ourselves successful when our clients benefit from our solutions.
- Customer success will always be our priority.
- Our customers may not always be right, but they always come first in our considerations. Our policy is to provide a quality service and IT solution that is superior to that provided in our local industry. Our solution targets to provide effective service that is responsive to our clients' needs.
- We seek to provide a service that is based on trust and mutual respect. We promise to help our clients in the best way. Provide them with all the possible solutions that will help them to achieve their goal.
- One of the most important value of our company is to provide the best support for our clients. We intend on achieving this by maintaining effective and active communications with our client.

- Upon implementation of our IT solution, we will provide you with technical support that will help the continuity of your business's growth and efficiency. You can always reach out us on supportCCTS@gmail.com
- The support team will do their best to solve the client's problem and update them accordingly.

We ensure that all of our staff, engineers, and members fully understand our customer relations policy and protocols and implement all requirements of these.

Testing Plan

Introduction

This test plan is meant to establish standard testing procedures that team members should use for all application testing. It includes a view of the project's scope and risk and how they relate to testing. This document will identify procedures, deliverables, and tools as they relate to testing.

Technologies

We will use Mocha, a node library, for unit testing and data integrity testing. Mocha scripts must be placed in the test folder in the root of the project and named module_name.test.js where module_name is the name of the module being tested. To test the web application programming interface (API), we will create a node script that uses the node library Axios to perform API requests. We will also try to perform invalid actions to test validation and security. All test data should be written to a unique database named LumicoTestDB. These tests should not run on the production database because they will delete substantial amounts of data. We recommend downloading MongoDB at https://www.mongodb.com/try and running these tests on your local computer. You will also need to add MONGO_TEST_URI to your .env file. This variable should equal the URI of the MongoDB instance you are using. For example, MONGO_TEST_URI=mongodb://localhost:27017/LumicoDBTest.

Objectives

The scope of our project with Lumico includes a web-based apparel store front and a web-based administrative interface. Both of these interfaces will be developed with the same tools: Node, React, Tachyons, Express, and MongoDB. The testing team is responsible for testing both interfaces and ensuring that scope objectives are met, and risks related to development are monitored adequately. The test team includes various members from Clutch City Tech Solutions and users from Lumico.

Scope

For more details about the overall scope of our project, refer to the project execution plan. Aspects of the project's scope that will fall under the scrutiny of testing include user interface and user experience, CRUD functionality, data integrity, and security.

Risks

The risk report and risk register contain a complete listing of risks for our project. The two greatest risks our team identified were related to implementation planning and data specifications. To mitigate these risks, the testing team should ensure that implementation plans and data specifications are being followed. More details about how this will occur are included in the Testing Methodology section.

Testing Methodology

The testing methodology we tailored to this project includes five phases. Testing should be performed on a weekly basis as program updates occur. Each phase of testing should be repeated every time testing is performed. The first phase is discovery.

During the discovery phase, testers should perform a code audit to identify new API endpoints, methods, and classes. After identifying new items, the testing team should work with the developers to expand upon the documentation attached to each item. This step should help ensure that everyone knows how each item functions and what each item is intended to do. After expanding documentation, testers should review the project scope and project requirements documentation. If any items are outside of scope, this should be documented and communicated to the developers and project manager before any further testing occurs. Users at Lumico should then be allowed to give feedback about how well each item that was within scope meets scope requirements. Their feedback should be documented in detail. After the discovery phase, unit testing can begin to test each individual component of the program.

Unit testing should begin with the creation of assertion-based Mocha scripts on the express server. These scripts should use the server's internal methods to create, read, update, and delete data. Once developed, these tests are fully automated and can be launched with the npm test command. After unit testing, we should preform integration testing to ensure that each component works together properly.

To test integration, we must be able to test how the web API interacts with external inputs. We will use Axios to insert, update, and delete test data of each type through our project's web API. The test data should include both valid and invalid entries for each entity and each operation. After the test data is added or updated, Mocha scripts can be leveraged to test data integrity using our project's data schema. If any invalid data was present, this should be documented. Mocha scripts should also be used after deleting the data to ensure that it was deleted successfully. After integration testing, we will perform system testing to test the user experience and security.

System testing should take place in the web browser to better reflect how users interact with the system. Each type of user interaction should be tested to ensure that users can only alter data they are intended to. To aid in this step, testers might want to consider using the TamperMonkey browser addon to automate common tasks. User acceptance testing should follow system testing.

User acceptance testing is supposed to give users the ability to provide feedback on the best results we have available to them at the time. Consequently, there should be no obvious bugs identified in this phase. Any bugs that are uncovered in this phase should be examined further to identify flaws in the testing process. Ultimately, the goal of this phase is to obtain user feedback regarding fitness for use.

Deliverables by Phase

Test Phase	Deliverable
Discovery	
	Code Audit Report
	Technical Manual Updates
	User Feedback
Unit Testing	
	Mocha Test Results
Integration Testing	
	Mocha Test Results
System Testing	
	Error Report
User Acceptance	
	User Feedback
	Error Report

Table 3:Testing deliverables classified by testing phase

Documentation

Code Audit Report

A code audit report should be an Excel spreadsheet that lists each item that will be tested in each module. The module's path from the root of the project directory should also be identified. Items that are eligible for automated testing with Mocha and the status of each test's implementation—Forthcoming, Working, or Work in Progress—should also be identified in this report. As the documentation is updated, the technical manual should also be updated. Please refer to Appendix IV and Appendix V if you would like an example of the code audit report.

User Feedback

User feedback should be documented in an informal format. At a minimum, however, user feedback needs to be documented with the date, the person who gave the feedback, and the context of the feedback.

Mocha Test Results

The results of automated testing can be obtained by redirecting the output of the npm test command to a text file. The test data used when conducting the tests should also be included in the Mocha test results. Please refer to Appendix III if you would like an example of what the Mocha test results document should look like.

Error Report

The error report is intended to identify issues that did not occur during automated testing. Each entry should include all the information necessary to recreate the error. This includes the webpage the error occurred on, the status of the error, the priority of the error (based on how many areas of the website it impacts and how it will be to fix), preconditions that must be met to recreate the error, any other comments that might be useful, and a description of the error. Please refer to Appendix VI if you would like an example of what the Error Report should look like.

Project Improvements

Improvement	Details	Importance
Implement better state management	Implement state management so that quantity is automatically manipulated after purchase has been made	High
Customer Service	Implement customer service email form that allows customers to communicate with management for order or product issues	High
Order Cancel	Implement easy method for customers to cancel items before shipped	High
Add more payment methods	Implement different payment API's to work with newer methods such as Apple Pay, Stripe and bitcoin	Medium
Advertising Emails	Send out periodic marketing emails announcing new products and discounts	Medium
Recommended Items	Implement prediction system that recommends items to users based on perviously purchased items	Medium

Styling	Update the website with different festive themes depending on the time of year	Medium
Section for Sale Items	Implement separate section for items that are on sale	Medium
Implement reward system	System that creates customer loylaty by awarding 1 point per dollar spent that can be redeemed for future purchases	Low
3D product images	Implement three-dimensonal images in the products page to allow users to have more angles while choosing clothes	Low
Try before you buy	Allow users to buy items and try them out before deciding whether they want to pay for them	Low
Implement different login methods	Allow users to login and register with 3rd party accounts such as Google and Apple accounts to allow for easier registration	Low

Figure 13: Project Improvements

Service Level Agreement

This Service Level Agreement is effective as of **September 7th, 2020**

BETWEEN: Clutch City Tech Solutions

AND: Jackie Nguyen/Lumico

Introduction

This service level agreement (SLA) describes the levels and type of service that **Lumico** ("the client") will receive from **Clutch City Tech Solutions** ("the provider").

This SLA should be read alongside the service level agreements of hosting and data storage services Heroku, AWS, and MongoDB.

Purpose

The client (Lumico) depends on Software support services and IT support services from the provider (Clutch City Tech Solutions) to ensure dependable runtime, general maintenance, and live support. This service level agreement sets out what levels of availability and support the client is guaranteed to receive for the build web application and database.

This SLA forms an important part of the contract between the client and the provider to ensure that both parties work together effectively.

The SLA Agreement may be reviewed and changed at any time only through a mutual agreement between both parties.

Software and IT covered

This SLA covers the Software and IT services that are listed below. It is important to note that some of the services utilized contain separate and independent uptime and maintenance schedules that are to be kept independent from this SLA.

Services

- **MongoDB**
  The provider (Clutch City Tech Solutions) will be solely responsible for any issues regarding data being stored incorrectly as a result of the backend code and any corruption error that is deemed not caused by the client or a larger scale problem with a MongoDB or AWS outage. The provider is not responsible for any downtime caused by these two services.

- **Heroku**
  The provider (Clutch City Tech Solutions) is in charge of any deployment errors that are deemed not caused by the Heroku online platform and service outages. The provider will also be in charge of managing and monitoring application access, configuration, and overall maintenance of the deployed application.

- **General Codebase**
  The provider (Clutch City Tech Solutions) will be in charge of maintaining the working codebase and implementing any mutually agreed up features and functionality according to the clients' needs.

Exclusions

This SLA is written in a spirit of partnership. The provider will do everything in their power to ensure viable runtime and minimal downtime if they have control over the situation. That said, the SLA does not apply to:

- Any equipment/software not listed above
- Any services not listed above
- Unexpected downtime from third-party services

This SLA does not apply in circumstances that go beyond the supplier's control such as natural disasters, war, and other acts of God.

This SLA does not apply if the client is in breach of its contract with the provider for any valid reason.

Provider Responsibilities

The provider will provide and maintain the IT software web application that is used by the client. The IT Development contract between the provider and the client includes these responsibilities:

- Ensure relevant software is updated, maintained and that the application is running online
- Responds to support requests within applicable time frames as stated in the response times section of the SLA.
- Takes appropriate steps to escalate and resolve issues promptly
- Maintains good communication with the client

Client Responsibilities

The client will use the provider-based application as agreed upon and as designed. The IT development contract includes a full list of the IT software and its intended uses. Additionally, the client will:

- Notify the provider of any issues or problems promptly
- Allow the provider access to the codebase and any other resources such as login information, private tokens, and software keys to ensure proper incident resolution
- Maintain good communication with the provider

Uptime

Since the provider is currently relying on third-party platforms and infrastructure for hosting and database needs the uptimes to vary based on specific services detailed below:

Heroku

Heroku currently has not guaranteed uptime but has provided historical uptime at **99.98%** uptime for applications deployed.

MongoDB

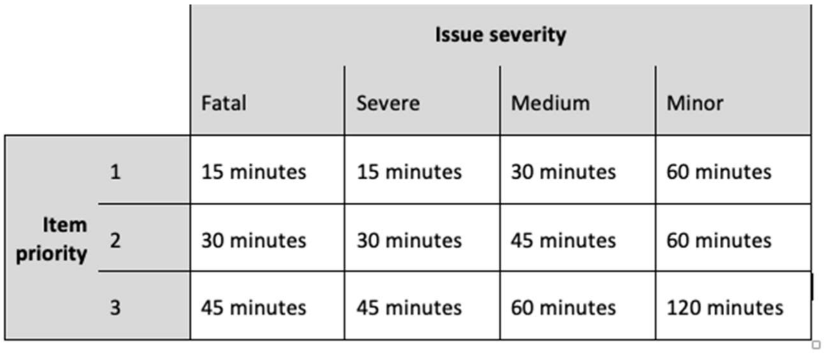
MongoDB guarantees that All MongoDB atlas clusters are highly available and backed by an industry-leading uptime SLA of **99.995%** across all cloud providers.

AWS

Amazon AWS provides a service commitment with Amazon DocumentDB clusters available with a monthly uptime percentage of at least **99.9%** availability.

Response Times

Response times are measured by how long it takes for the provider to respond to a support request that is raised by the client. The provider is deemed responsible after they have responded to the client's request through email. Response times are shown below in Figure 14.



		Issue severity			
		Fatal	Severe	Medium	Minor
Item priority	1	15 minutes	15 minutes	30 minutes	60 minutes
	2	30 minutes	30 minutes	45 minutes	60 minutes
	3	45 minutes	45 minutes	60 minutes	120 minutes

Figure 14:Provider response times

Termination

The provider recognizes that it provides services that are critical to the client's business. If the provider constantly fails to meet agreed-upon service and support standards the client has a right to terminate their entire contract with the provider with no penalty.

The right is available if the provider fails to meet client service levels and expectations more than five times in a single calendar month.

Signatures

This service level agreement is agreed upon as part of the IT software and support contract between Lumico and Clutch City Tech Solutions.

Signed on behalf of the client:

Name:
Position:
Date:

Signed on behalf of the provider:

Name:
Position:
Date:

References

Building Shopping Cart With Reactjs and Redux Part 2. (2020, June 23). Retrieved October 19, 2020, from https://webmobtuts.com/frontend-development/building-shopping-cart-with-reactjs-and-redux-part-2/

Kendall, K. E., & Kendall, J. E. (2019). Systems analysis and design (10th ed.). Hoboken, NJ: Pearson.

Kumar, M. (n.d.). CRUD Operations In ReactJS With Axios Using Web API And SQL Server. Retrieved October 19, 2020, from https://www.c-sharpcorner.com/article/crud-operations-in-reactjs-with-axios-using-web-api-and-sql-server/

Lewington, G. (2019, October 26). How to allow users to upload images with Node/Express, Mongoose, and Cloudinary. Retrieved October 19, 2020, from https://www.freecodecamp.org/news/how-to-allow-users-to-upload-images-with-node-express-mongoose-and-cloudinary-84cefbdff1d9/

Morrison, G. R., Ross, S. M., Morrison, J. R., & Kalman, H. K. (2012). Designing effective instruction. Hoboken, NJ: Wiley.

Openbase. (n.d.). Retrieved October 19, 2020, from https://openbase.io/

TACHYONS - Components. (n.d.). Retrieved October 19, 2020, from https://tachyons.io/components/

Appendix

Appendix I: Use Cases

Use Case Name: Create User Account	**Use Case ID:** 0001
Area: Lumico Webstore	
Actor(s): Users	
Description: Allow a User to create an account.	
Triggering Event: User clicks the Register Now! link on the Sign In page.	
Steps Performed	**Information for Steps**
1. User clicks Sign In button on landing page.	
2. User clicks the Register Now! link on the Sign-In page.	
3. User enters their information on the Registration page.	User name, user email, user password
4. User clicks the Sign Up button on the Registration page.	
5. System creates a new user account and logs in the new user.	User name, user email, user password
6. New user is logged in.	
Precondition: A user would like to create an account on Lumico Webstore.	
Postcondition: User was able to create a new account and is immediately logged in.	
Assumptions: User did not previously have an account.	
Success Guarantee: System creates a new user account and logs in the new user.	
Minimum Guarantee: System creates a new user account.	

Requirements Met: Allows users to create an account so that they can shop on Lumico Webstore.
Priority: High
Risk: Medium
Failure Mode: User enters improperly formatted information. **Response:** The registration form will reject the incorrect information.
Failure Mode: User enters correctly formatted information that is invalid. **Response:** Users are able to update their account details to fix any potential mistakes.

Figure APX-1: Use Case 1

Use Case Name: Create Administrator Account	**Use Case ID:** 0002
Area: Lumico Webstore	
Actor(s): Lumico Employee	
Description: Allow a Lumico Employee to create an account.	
Triggering Event: Lumico Employee clicks the Register Now! link on the Sign In page.	
Steps Performed	**Information for Steps**
1. Lumico Employee clicks the administration portal drop down menu on the top right of the interface.	User email, user password
2. Lumico Employee clicks the Users link.	User email, user password
2. Lumico Employee clicks the edit button next to one of the user accounts.	User email, user password
3. Lumico Employee checks the is admin checkbox.	User email, user password
4. Lumico Employee clicks the update button.	User email, user password, other user’s email, other user’s name, other user’s isAdmin property
5. User account is updated in the database.	
Precondition: A new employee has been approved to work on the webstore.	
Postcondition: The correct user account is promoted to an administrator account.	
Assumptions: Lumico management is able to effectively keep track of new employees.	
Success Guarantee: The correct user account is promoted to an administrator account.	
Minimum Guarantee: A user account is promoted to an administrator account.	
Requirements Met: Allows employees to login as an administrator so that they are able to manage the webstore.	

Priority: High
Risk: Medium
Failure Mode: Lumico Employee enters improperly formatted information. **Response:** The registration form will reject the incorrect information.
Failure Mode: Lumico Employee enters correctly formatted information that is invalid. **Response:** Users are able to update their account details to fix any potential mistakes.
Failure Mode: A user account not belonging to an employee is promoted. **Response:** Ask employees to format their user’s name in a specific, easily identifiable way.

Figure APX-2: Use Case 2

Use Case Name: Create Product	**Use Case ID:** 0003
Area: Lumico Webstore	
Actor(s): Lumico Employee	
Description: Allow a Lumico Employee to create a product.	
Triggering Event: Lumico Employee clicks the Create Product button on the administrator portal product page.	
Steps Performed	**Information for Steps**
1. Lumico Employee clicks Administration Portal to display the dropdown menu.	User email, user password
2. Lumico Employee clicks the Products link.	User email, user password
3. Lumico Employee clicks create product button.	User email, user password
4. Lumico Employee enters the product information on the product page.	User email, user password, product name, product price, product image, product category, product count in stock, product description
5. Lumico Employee clicks the Update Product button.	User email, user password
6. The new product is added to the database.	User email, user password, product name, product price, product image, product category, product count in stock, product description
Precondition: A Lumcio Employee is logged in with an administrative account.	

Postcondition: A new product is added to the webstore.
Assumptions: Lumico management is able to effectively describe new products.
Success Guarantee: A new product is added to the webstore with correct information.
Minimum Guarantee: A new product is added to the webstore.
Requirements Met: Allows employees to add products to the webstore.
Priority: High
Risk: Low
Failure Mode: The user is not logged in. **Response:** The user is redirected to the login page.
Failure Mode: Lumico Employee enters improperly formatted information. **Response:** The product form will reject the incorrect information.
Failure Mode: Lumico Employee enters correctly formatted information that is invalid. **Response:** Lumico Employees are able to update products to fix any potential mistakes.

Figure APX-3: Use Case 3

Use Case Name: Update Product	**Use Case ID:** 0004
Area: Lumico Webstore	
Actor(s): Lumico Employee	
Description: Allow a Lumico Employee to update a product.	
Triggering Event: Lumico Employee clicks the Edit button on the administrator portal product page next to the product they want to edit.	
Steps Performed	**Information for Steps**
1. Lumico Employee clicks Administration Portal to display the dropdown menu.	User email, user password
2. Lumico Employee clicks the Products link.	
3. Lumico Employee clicks edit button next to the product they wish to edit.	User email, user password
4. Lumico Employee enters the product information on the product page.	User email, user password, product name, product price, product image, product category, product count in stock, product description
5. Lumico Employee clicks the Update Product button.	User email, user password
6. The product is updated in the database.	User email, user password, product name, product price, product image, product category, product count in stock, product description

Precondition: A Lumcio Employee is logged in with an administrative account.
Postcondition: A product is updated in the webstore.
Assumptions: Lumico management is able to effectively describe new products.
Success Guarantee: A product's information is updated with correct information.
Minimum Guarantee: A product's information is updated.
Requirements Met: Allows employees to edit products in the webstore.
Priority: High
Risk: Low
Failure Mode: The user is not logged in. **Response:** The user is redirected to the login page.
Failure Mode: Lumico Employee enters improperly formatted information. **Response:** The product form will reject the incorrect information.
Failure Mode: Lumico Employee enters correctly formatted information that is invalid. **Response:** Lumico Employees are able to update products to fix any potential mistakes.

Figure APX-4: Use Case 4

Use Case Name: Administrator Edits User Account	**Use Case ID:** 0005
Area: Lumico Webstore	
Actor(s): Lumico Employee	
Description: Allow a Lumico Employee to edit a user account.	
Triggering Event: Lumico Employee clicks the Edit button next to the user they want to edit on the Users administration portal page.	
Steps Performed	**Information for Steps**
1. Lumico Employee clicks the administration portal dropdown menu on the top right.	User email, user password
2. Lumico Employee clicks the Users link.	User email, user password
3. Lumico Employee clicks the edit button next to one of the user accounts.	User email, user password
4. Lumico Employee enters the new user information.	User email, user password, other user's name, other user's email, other user's isAdmin property
5. Lumico Employee clicks the update button.	User email, user password
6. The user's account is updated in the database.	User email, user password, other user's name, other user's email, other user's isAdmin property
Precondition: A user needs help updating their account information.	
Postcondition: The user's account information is updated by Lumico Employee.	
Assumptions: Lumico Employee is able to effectively identify the correct user to edit.	
Success Guarantee: The correct user account is updated.	

Minimum Guarantee: A user account is updated.
Requirements Met: Allows employees to edit user accounts.
Priority: Medium
Risk: Low
Failure Mode: Lumico Employee enters improperly formatted information. **Response:** The registration form will reject the incorrect information.
Failure Mode: The user is not logged in. **Response:** The user is redirected to the login page.
Failure Mode: Lumico Employee enters correctly formatted information that is invalid. **Response:** Lumico Employees are able to update account details to fix any potential mistakes.

Figure APX-5: Use Case 5

Use Case Name: Administrator Deletes User Account	**Use Case ID:** 0006
Area: Lumico Webstore	
Actor(s): Lumico Employee	
Description: Allow a Lumico Employee to delete a user account.	
Triggering Event: Lumico Employee clicks the Delete button next to the user they want to edit on the Users administration portal page.	
Steps Performed	**Information for Steps**
1. Lumico Employee clicks the administration portal drop down menu on the top right of the interface.	User email, user password
2. Lumico Employee clicks the Users link.	User email, user password
3. Lumico Employee clicks the delete button next to one of the user accounts.	User email, user password
4. The user is deleted from the database.	User email, user password, other user’s name, other user’s email
Precondition: A user needs help deleting their account.	
Postcondition: The user’s account is deleted by Lumico Employee.	
Assumptions: Lumico Employee is able to effectively identify the correct user to delete.	
Success Guarantee: The correct user account is deleted.	
Minimum Guarantee: A user account is deleted.	
Requirements Met: Allows employees to delete user accounts.	
Priority: Medium	
Risk: Medium	
Failure Mode: The user is not logged in.	

Response: The user is redirected to the login page.	
Failure Mode: Lumico Employee deletes the wrong user. **Response:** Have a prompt that asks employees to double check their decision before continuing.	
Use Case Name: User Updates their Account	**Use Case ID:** 0007
Area: Lumico Webstore	
Actor(s): User	
Description: Allow a user to edit their account.	
Triggering Event: User clicks the Update Info button on the User Profile page.	
Steps Performed	**Information for Steps**
1. User clicks their name on the top right to display the dropdown menu.	User email, user password
2. User clicks the Profile link in the user dropdown menu.	User email, user password
3. The user's Profile is selected from the	User email, user password
4. The user's orders are selected from the database.	User email, user password
5. The user's orders and profile are displayed on the user's profile page.	User name, user email, user password, orders
4. User fills in their new information on the profile page.	User email, user password, new user name, new user email, new user password
5. User clicks the Update Info button on the User Profile page.	
6. The user's account is updated on the database	User email, user password, new user name, new user email, new user password

Precondition: A user needs to update their account information.
Postcondition: The user’s account information is updated.
Assumptions: The user will only update their account when necessary.
Success Guarantee: The correct user account is updated.
Minimum Guarantee: A user account is updated.
Requirements Met: Allows users to edit their accounts.
Priority: Medium
Risk: Low
Failure Mode: User enters improperly formatted information. **Response:** The registration form will reject the incorrect information.
Failure Mode: User enters correctly formatted information that is invalid. **Response:** Users are able to update account details to fix any potential mistakes.
Failure Mode: The user is not logged in. **Response:** The user is redirected to the login page.

Figure APX-6: Use Case 6

Use Case Name: User Adds Item to Cart	**Use Case ID:** 0008
Area: Lumico Webstore	
Actor(s): User	
Description: Allow a user to add items to their cart.	
Triggering Event: User clicks the Add To Cart button on the Product page.	
Steps Performed	**Information for Steps**
1. User clicks a product they wish to buy.	product name, product image, product description, product price, product quantity in stock
2. User selects the quantity they want.	
3. User clicks the Add To Cart button on the Product page.	
4. The product is added to the user's cart on the client side.	product name, product image, product price, qty
Precondition: A user must know what product they want.	
Postcondition: The product is added to the user's cart.	
Assumptions: The user will have a user account on Lumico webstore before checkout.	
Success Guarantee: The correct product and correct quantity is added to the correct user's cart.	
Minimum Guarantee: A quantity of a product is added to a user's cart.	
Requirements Met: Allows users to add items to their cart for purchase.	
Priority: Medium	
Risk: Low	
Failure Mode: User adds an item that is out of stock to their cart.	

Response: If an item is out of stock, users cannot add the item to their cart.

Figure APX-7: Use Case 7

Use Case Name: User Completes Checkout	**Use Case ID:** 0009
Area: Lumico Webstore	
Actor(s): User	
Description: Allow a user to order items in their cart.	
Triggering Event: User clicks the Place Order button on the Order Summary page.	
Steps Performed	**Information for Steps**
1. User clicks the cart icon on the top right to go to their cart.	
2. User clicks the checkout button.	
3. User fills in their shipping address.	User email, user password, address, city, state, zip code, country
4. User selects their payment method	Payment methods
5. User reviews their order	User email, user password, order items
6. User clicks the Place Order button on the Order Summary page.	
7. PayPal creates a new payment.	PayPal ClientID, payment total
8. A new order is created on the database.	User email, user password, address, city, state, zip code, country, order items, shipping, tax, total price, payment result, payment provider

Precondition: A user must be logged in.
Postcondition: A new order is created.
Assumptions: The user will have a user account on Lumico webstore before checkout.
Success Guarantee: The correct quantities of the correct products are ordered by the user.
Minimum Guarantee: Quantities of a product are ordered by a user.
Requirements Met: Allow a user to order items.
Priority: High
Risk: High
Failure Mode: PayPal payment fails. **Response:** The ordering process should check for PayPal success before adding orders to the database.
Failure Mode: The user is not logged in. **Response:** The user is redirected to the login page.

Figure APX-8: Use Case 8

Use Case Name: User Deletes an Item from their Cart	**Use Case ID:** 0010
Area: Lumico Webstore	
Actor(s): User	
Description: Allow a user to delete an item from their cart.	
Triggering Event: User clicks the trashcan button next to the item they want to delete on the Cart page.	
Steps Performed	**Information for Steps**
1. User clicks the cart icon on the top right to go to their cart.	
2. User clicks the trashcan button next to the item they want to delete.	
3. The item is deleted from their cart client side.	Product name, product price, product description, product image
Precondition: A user must have an item in their cart they wish to delete.	
Postcondition: An item is removed from the user's cart.	
Assumptions: The user has an item in their cart.	
Success Guarantee: The correct item is deleted from the user's cart.	
Minimum Guarantee: An item is deleted from the user's cart.	
Requirements Met: Allow a user to delete an item from their cart.	
Priority: High	
Risk: Low	
Failure Mode: The user deletes the wrong item from their cart. **Response:** The user can add the item to their cart again.	

Figure APX-9: Use Case 9

Use Case Name: User Updates the Quantity of an Item in their Cart	**Use Case ID:** 0011
Area: Lumico Webstore	
Actor(s): User	
Description: Allow a user to update the quantity of an item in their cart.	
Triggering Event: User selects a new quantity from the dropdown next to the item they want to update on the Cart page.	
Steps Performed	**Information for Steps**
1. User clicks the cart icon on the top right to go to their cart.	
2. User selects a new quantity from the dropdown next to the item they want to update on the Cart page.	
3. The quantity of that product in the user's cart is updated client side.	product name, product price, product description, product image, qty
Precondition: A user must have an item in their cart they wish to update.	
Postcondition: An item's quantity is updated in the user's cart.	
Assumptions: The user has an item in their cart.	
Success Guarantee: The correct item is updated to the correct quantity in the user's cart.	
Minimum Guarantee: An item's quantity is updated in the user's cart.	
Requirements Met: Allow a user to update the quantity of an item in their cart.	
Priority: High	
Risk: High	
Failure Mode: The user updates the quantity of the wrong item in their cart. **Response:** The user can change the quantity of any item in their cart again.	

Figure APX-10: Use Case 10

Use Case Name: User Views their Orders	**Use Case ID:** 0012
Area: Lumico Webstore	
Actor(s): User	
Description: Allow a user to view their order history.	
Triggering Event: User clicks the Profile link in the user dropdown menu.	
Steps Performed	**Information for Steps**
1. User clicks their name in the top right to display the user dropdown menu.	User email, user password
2. User clicks the Profile link in the user dropdown menu.	User email, user password
3. The user's profile is selected from the database.	User email, user password
4. The user's orders are selected from the database.	User email, user password
5. The user's orders and profile are displayed on the user's profile page.	User name, user email, user password, orders
Precondition: A user must be logged in.	
Postcondition: All of the user's orders and profile are displayed on the user profile page.	
Assumptions: The user has an account with Lumico.	
Success Guarantee: The correct user's orders are displayed.	
Minimum Guarantee: A user's orders are displayed.	
Requirements Met: Allow a user to view their order history.	
Priority: High	
Risk: Low	
Failure Mode: The user is not logged in. **Response:** The user is redirected to the login page.	

Figure APX-11: Use Case 11

Use Case Name: Lumico Employee Views All Orders	**Use Case ID:** 0013
Area: Lumico Webstore	
Actor(s): Lumico Employee	
Description: Allow a Lumico Employee to view all orders.	
Triggering Event: Lumico Employee clicks the Orders link in the Administration Portal dropdown menu.	
Steps Performed	
1. Lumico Employee clicks Administration Portal on the top right.	**Information for Steps**
2. Lumico Employee clicks the Orders link in the Administration Portal dropdown menu.	User email, user password
3. All orders are selected from the database.	User email, user password
4. All orders are displayed on the Orders page.	User email, user password
Precondition: A Lumico Employee must be logged in.	User email, user password, orders
Postcondition: All orders are displayed on the Orders page.	
Assumptions: The Lumico Employee has administrative access.	
Success Guarantee: All orders are displayed correctly.	
Minimum Guarantee: All orders are displayed.	
Requirements Met: Allow Lumico Employees to manage orders.	
Priority: High	
Risk: Low	
Failure Mode: The user is not logged in. **Response:** The user is redirected to the login page.	

Figure APX-12: Use Case 12

Use Case Name: Lumico Employee Exports Orders to CSV	**Use Case ID:** 0014
Area: Lumico Webstore	
Actor(s): Lumico Employee	
Description: Allow a Lumico Employee export all orders to comma separated values (CSV) format.	
Triggering Event: Lumico Employee clicks the Export CSV button on the Orders page.	
Steps Performed	**Information for Steps**
1. Lumico Employee clicks Administration Portal on the top right.	User email, user password
2. Lumico Employee clicks the Orders link in the Administration Portal dropdown menu.	User email, user password
3. All orders are selected from the database.	User email, user password
4. All orders are displayed on the Orders page.	User email, user password, orders
5. Lumico Employee clicks the Export CSV button on the Orders page.	User email, user password
6. All orders are displayed on the Orders page.	User email, user password, orders
7. Order data is converted to CSV format.	User email, user password, orders
8. Lumico Employee is prompted to download the CSV file named Orders.	User email, user password, orders csv
Precondition: A Lumico Employee must be logged in.	
Postcondition: Lumico Employee is prompted to download the CSV file named Orders.	
Assumptions: The Lumico Employee has administrative access.	

Success Guarantee: All orders are correctly exported to CSV and user is prompted to download the newly created file.
Minimum Guarantee: All orders are exported to CSV.
Requirements Met: Allow Lumico Employees to export order data.
Priority: Medium
Risk: Low
Failure Mode: The user is not logged in. **Response:** The user is redirected to the login page.

Figure APX-13: Use Case 13

Use Case Name: Lumico Employee Marks an Order as Shipped	**Use Case ID:** 0015
Area: Lumico Webstore	
Actor(s): Lumico Employee	
Description: Allow a Lumico Employee to view all orders.	
Triggering Event: Lumico Employee clicks the Orders link in the Administration Portal dropdown menu.	
Steps Performed	**Information for Steps**
1. Lumico Employee clicks Administration Portal on the top right.	User email, user password
2. Lumico Employee clicks the Orders link in the Administration Portal dropdown menu.	User email, user password
3. All orders are selected from the database.	User email, user password
4. All orders are displayed on the Orders page.	User email, user password, orders
5. Click the Details button next to the order to mark as shipped.	User email, user password, order id
6. Click the Mark as Shipped button.	
7. The order is updated to the shipped status in the database.	User email, user password, order id, shipping date
Precondition: A Lumico Employee must be logged in.	
Postcondition: All orders are displayed on the Orders page.	
Assumptions: The Lumico Employee has administrative access and is able to correctly identify the order to be marked as shipped.	
Success Guarantee: The correct order is marked as shipped.	

Minimum Guarantee: An order is marked as shipped.
Requirements Met: Allow Lumico Employees to manage orders.
Priority: High
Risk: Low
Failure Mode: The user is not logged in. **Response:** The user is redirected to the login page.

Figure APX-14: Use Case 14

Appendix II: Activity Diagrams

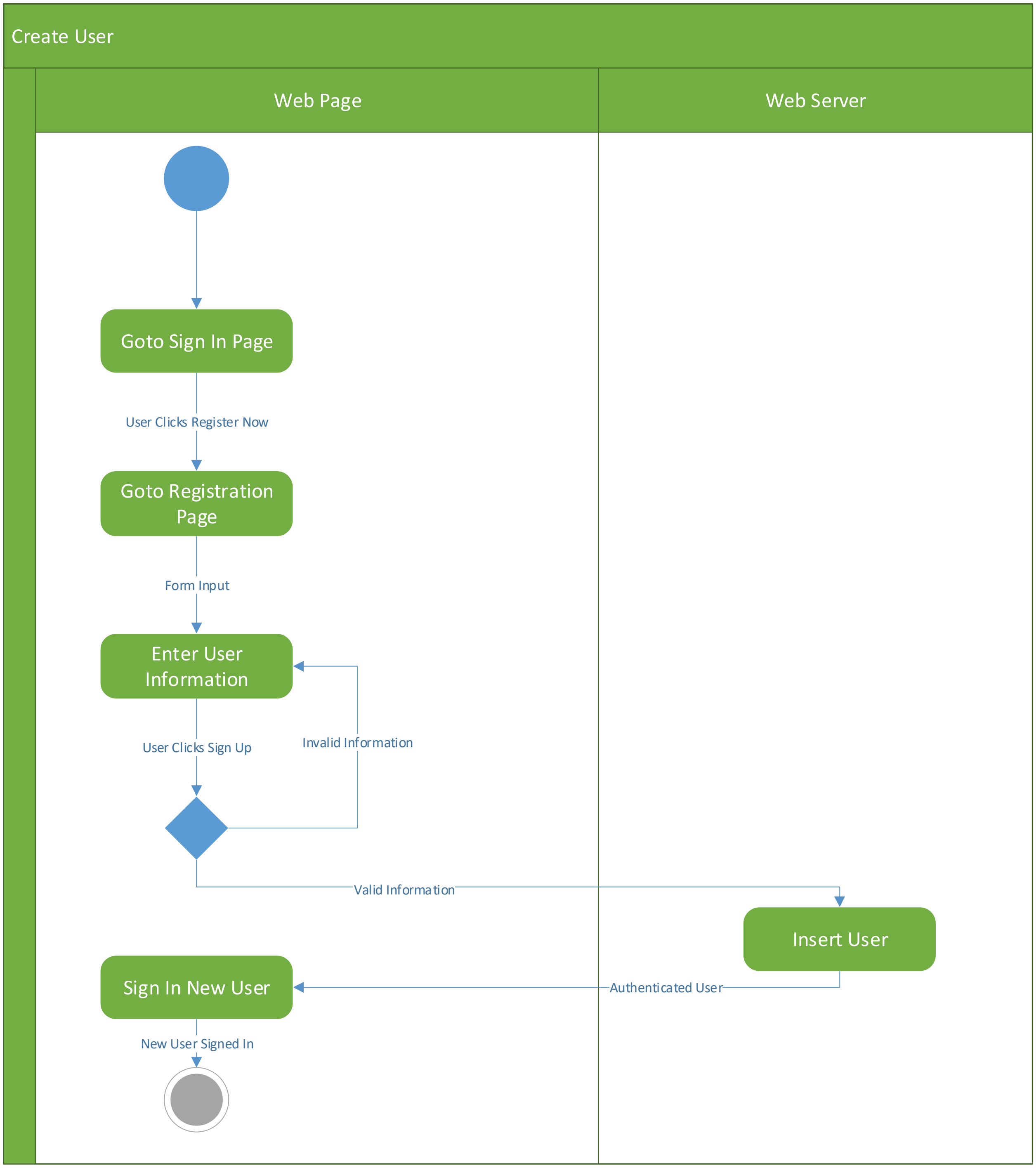
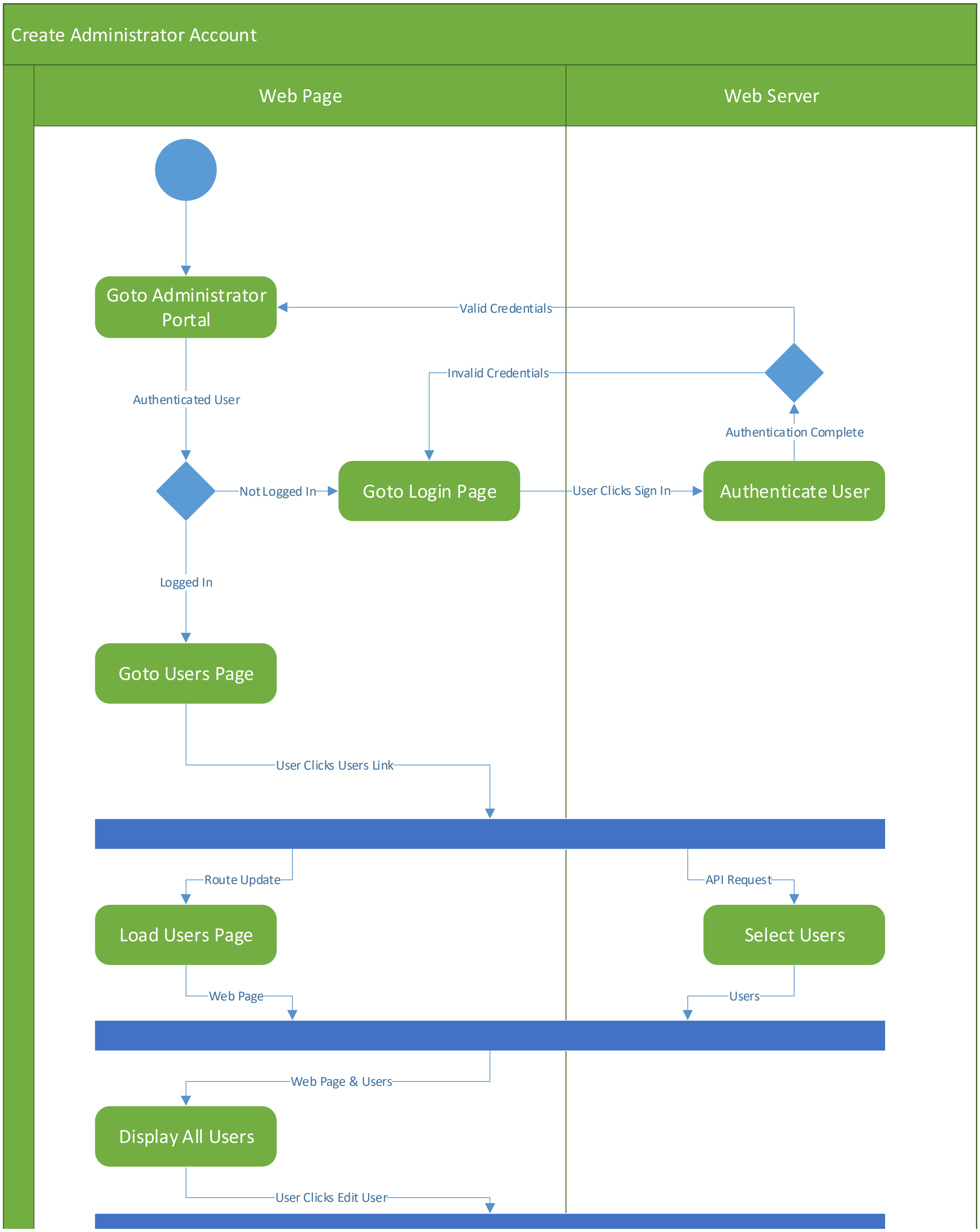


Figure APX2-1: Activity Diagram for Creating a User



Create Administrator Account
Web Page
Web Server
Goto Administrator Portal
Valid Credentials
Invalid Credentials
Authenticated User
Authentication Complete
Not Logged In
Goto Login Page
User Clicks Sign In
Authenticate User
Logged In
Goto Users Page
User Clicks Users Link
Route Update
API Request
Load Users Page
Select Users
Web Page
Users
Web Page & Users
Display All Users
User Clicks Edit User

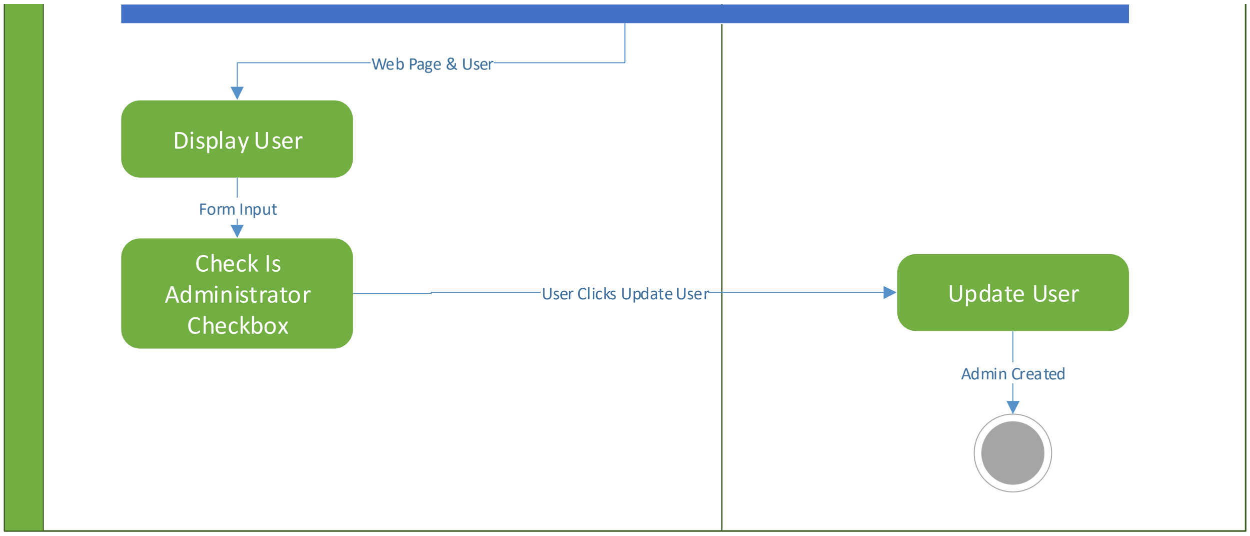
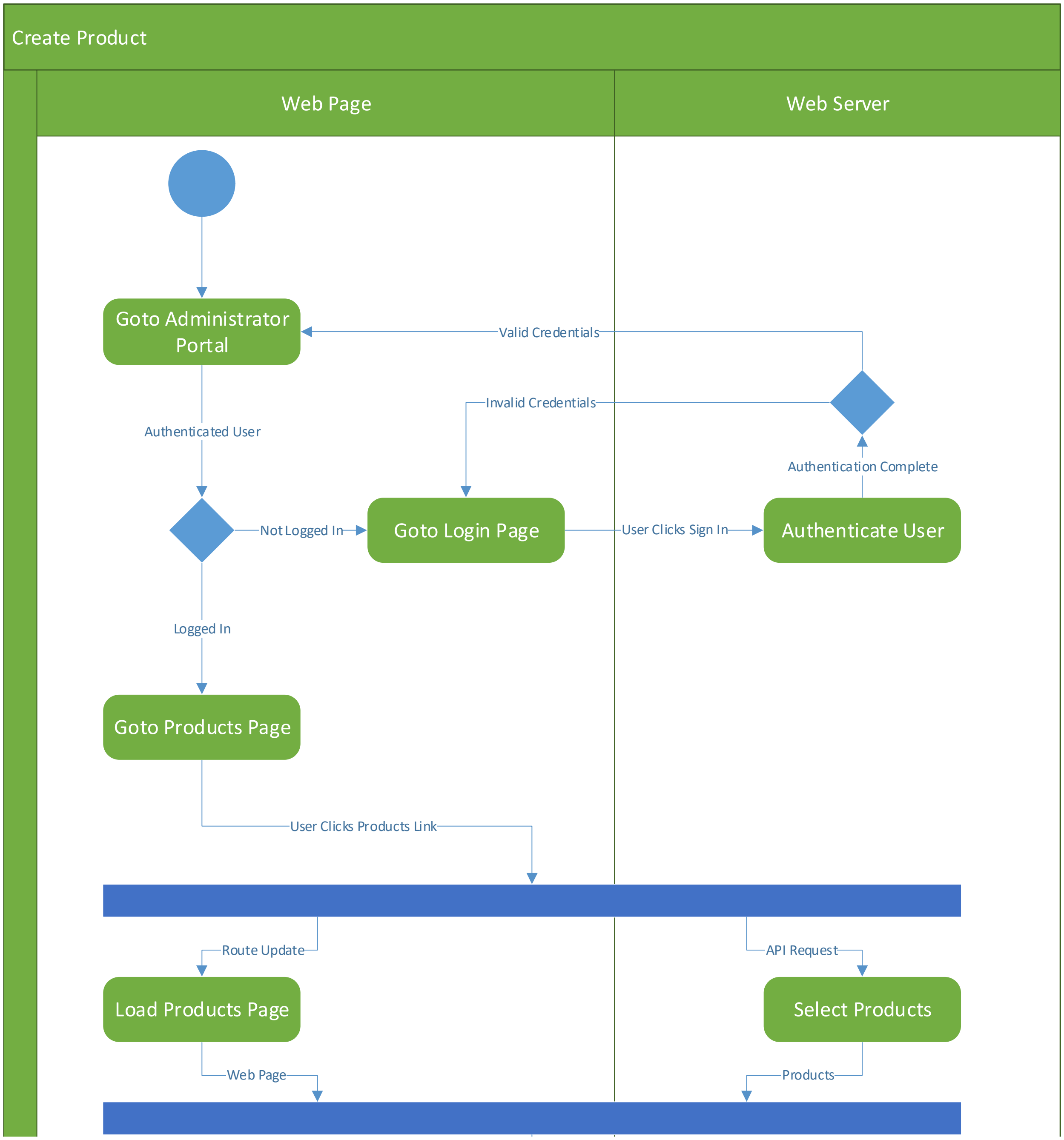


Figure APX2-2: Activity Diagram for Creating an Administrator Account



Create Product
Web Page
Web Server
Goto Administrator Portal
Valid Credentials
Invalid Credentials
Authenticated User
Authentication Complete
Not Logged In
Goto Login Page
User Clicks Sign In
Authenticate User
Logged In
Goto Products Page
User Clicks Products Link
Route Update
API Request
Load Products Page
Select Products
Web Page
Products

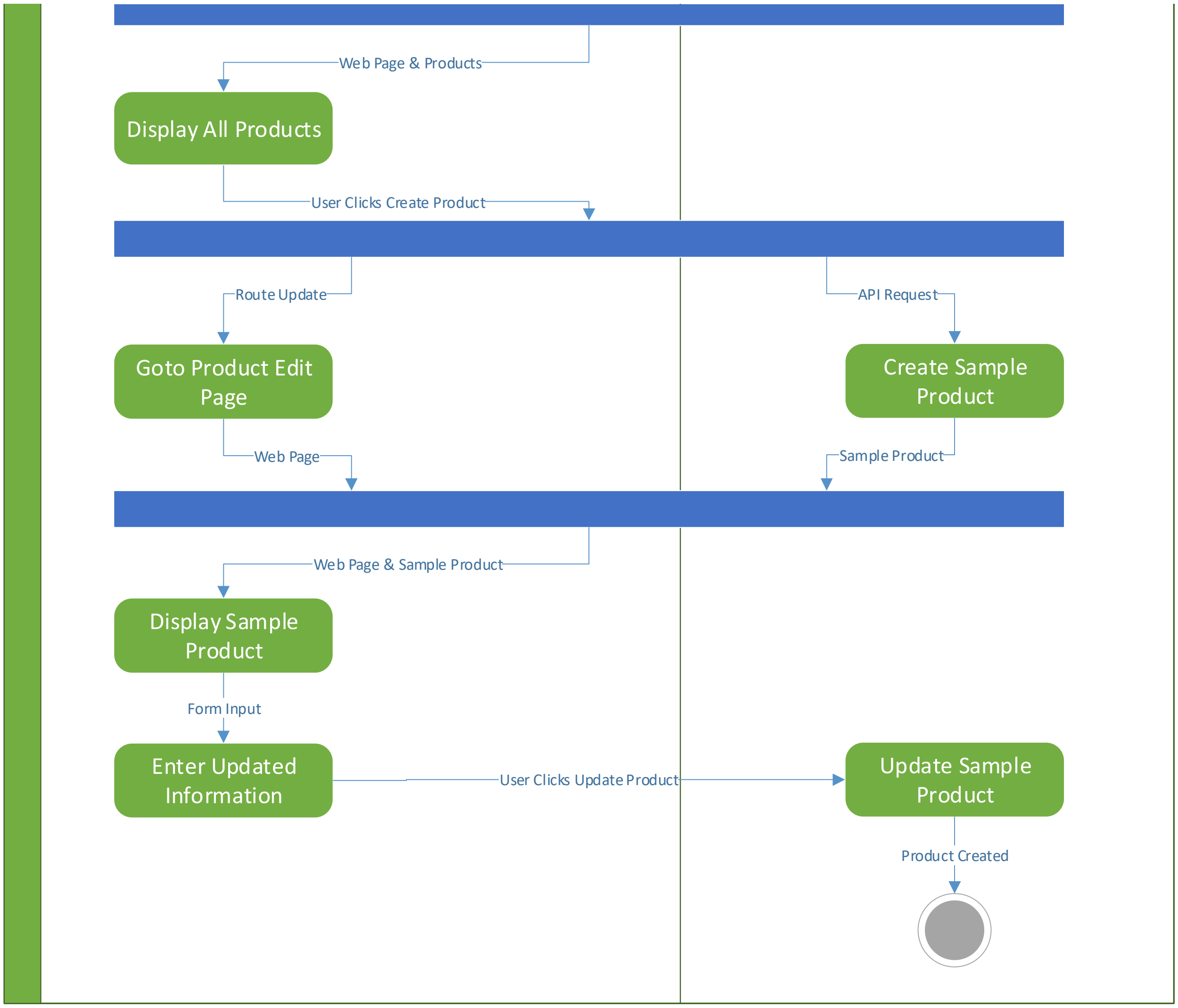
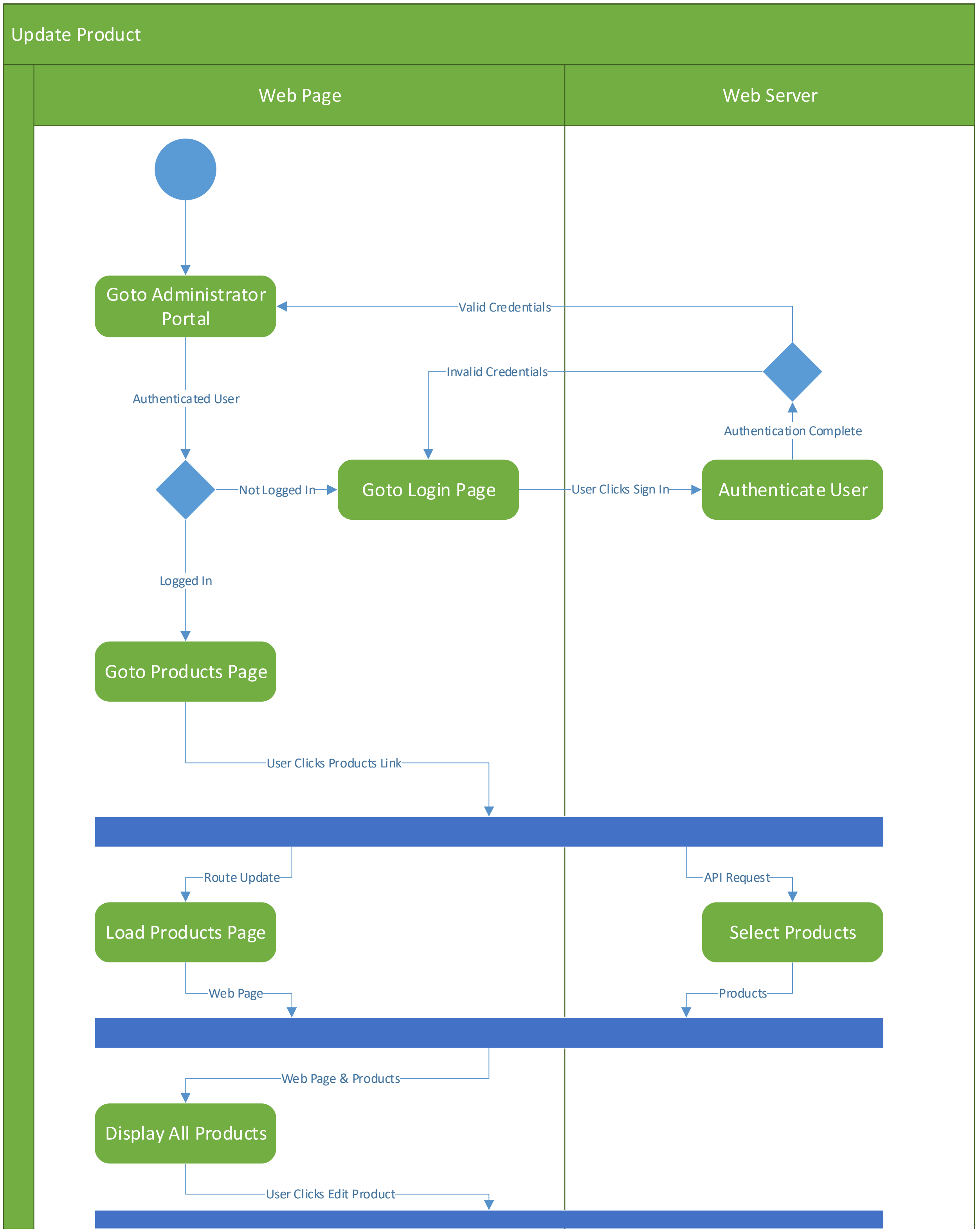


Figure APX2-3: Activity Diagram for Creating a Product



Update Product
Web Page
Web Server
Goto Administrator Portal
Valid Credentials
Invalid Credentials
Authenticated User
Authentication Complete
Not Logged In
Goto Login Page
User Clicks Sign In
Authenticate User
Logged In
Goto Products Page
User Clicks Products Link
Route Update
API Request
Load Products Page
Select Products
Web Page
Products
Web Page & Products
Display All Products
User Clicks Edit Product

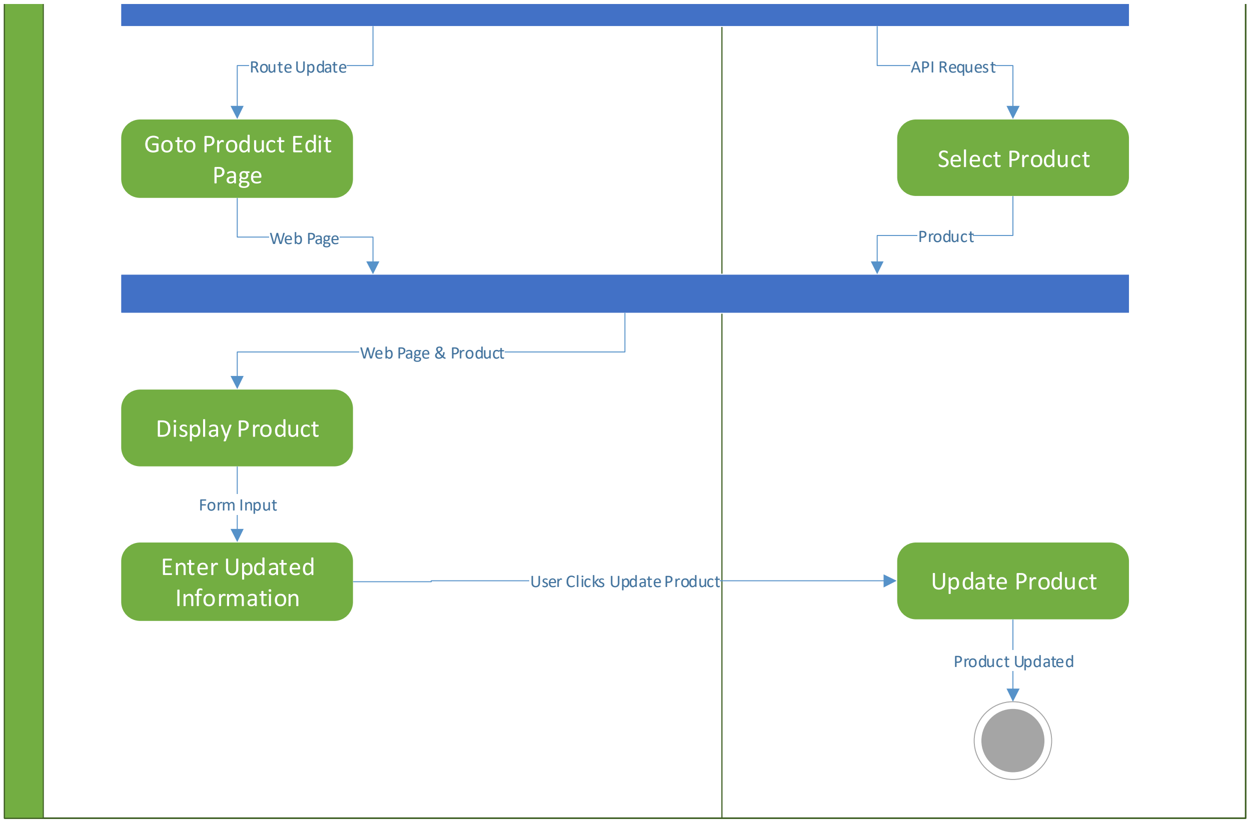
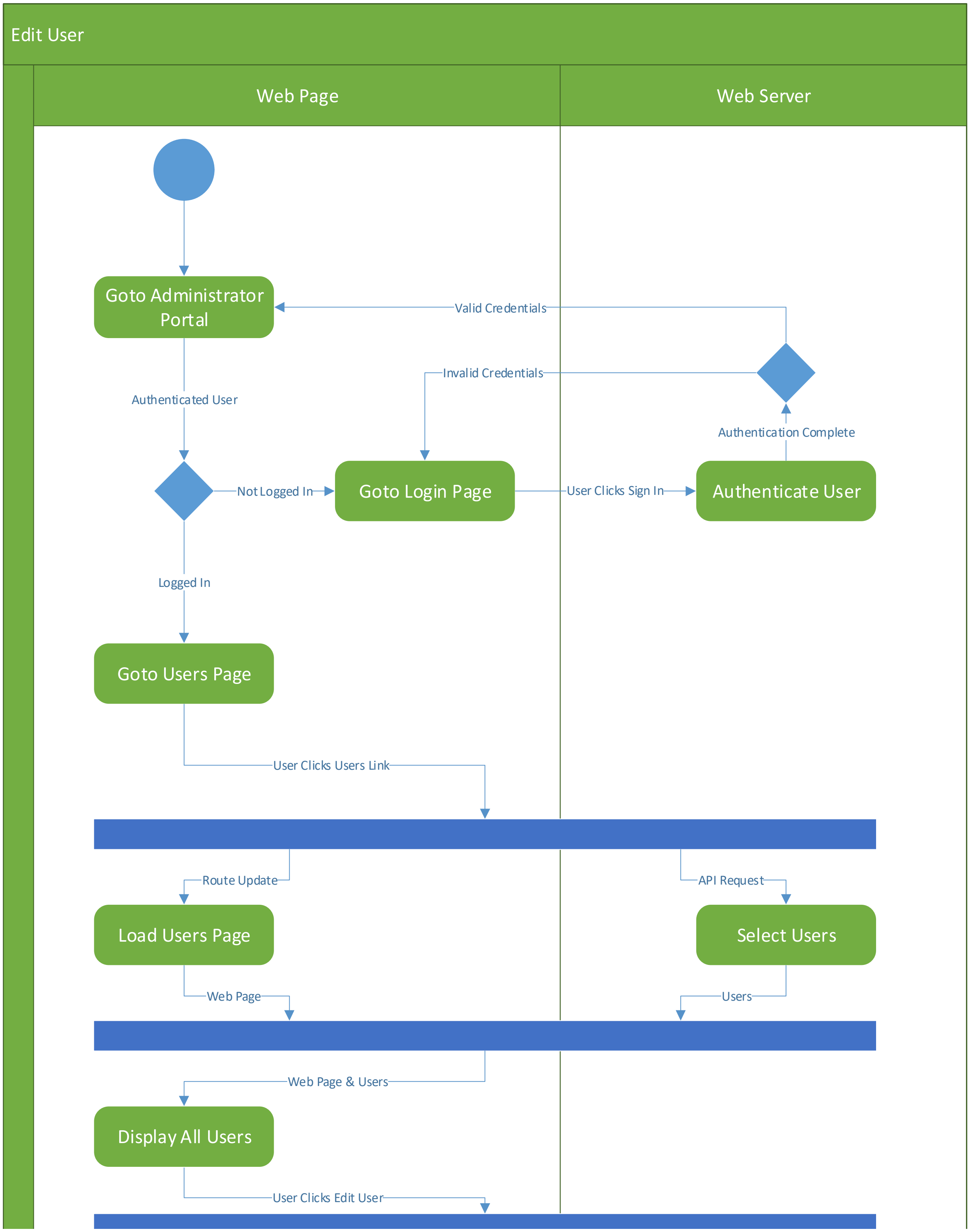


Figure APX2-4: Activity Diagram for Updating a Product



Edit User
Web Page
Web Server
Goto Administrator Portal
Valid Credentials
Invalid Credentials
Authenticated User
Authentication Complete
Not Logged In
Goto Login Page
User Clicks Sign In
Authenticate User
Logged In
Goto Users Page
User Clicks Users Link
Route Update
API Request
Load Users Page
Select Users
Web Page
Users
Web Page & Users
Display All Users
User Clicks Edit User

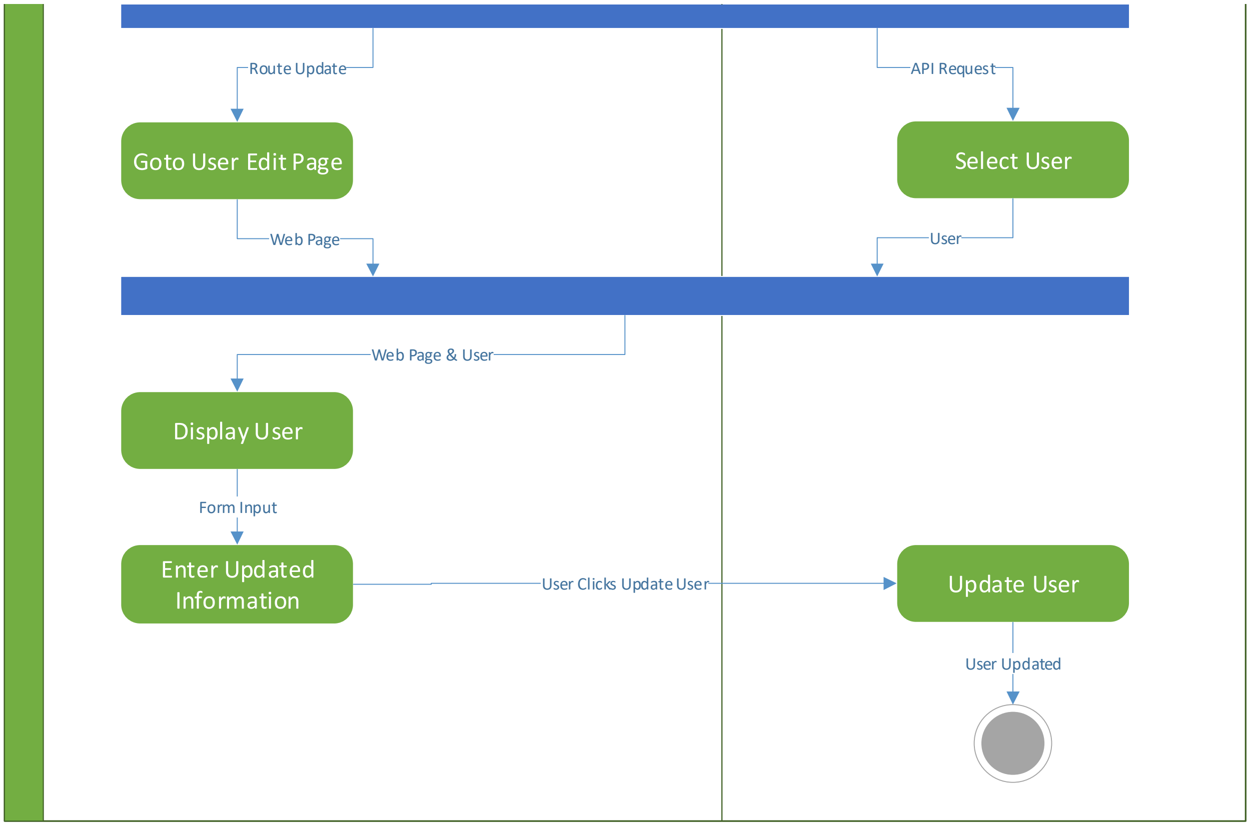


Figure APX2-5: Activity Diagram for Editing a User

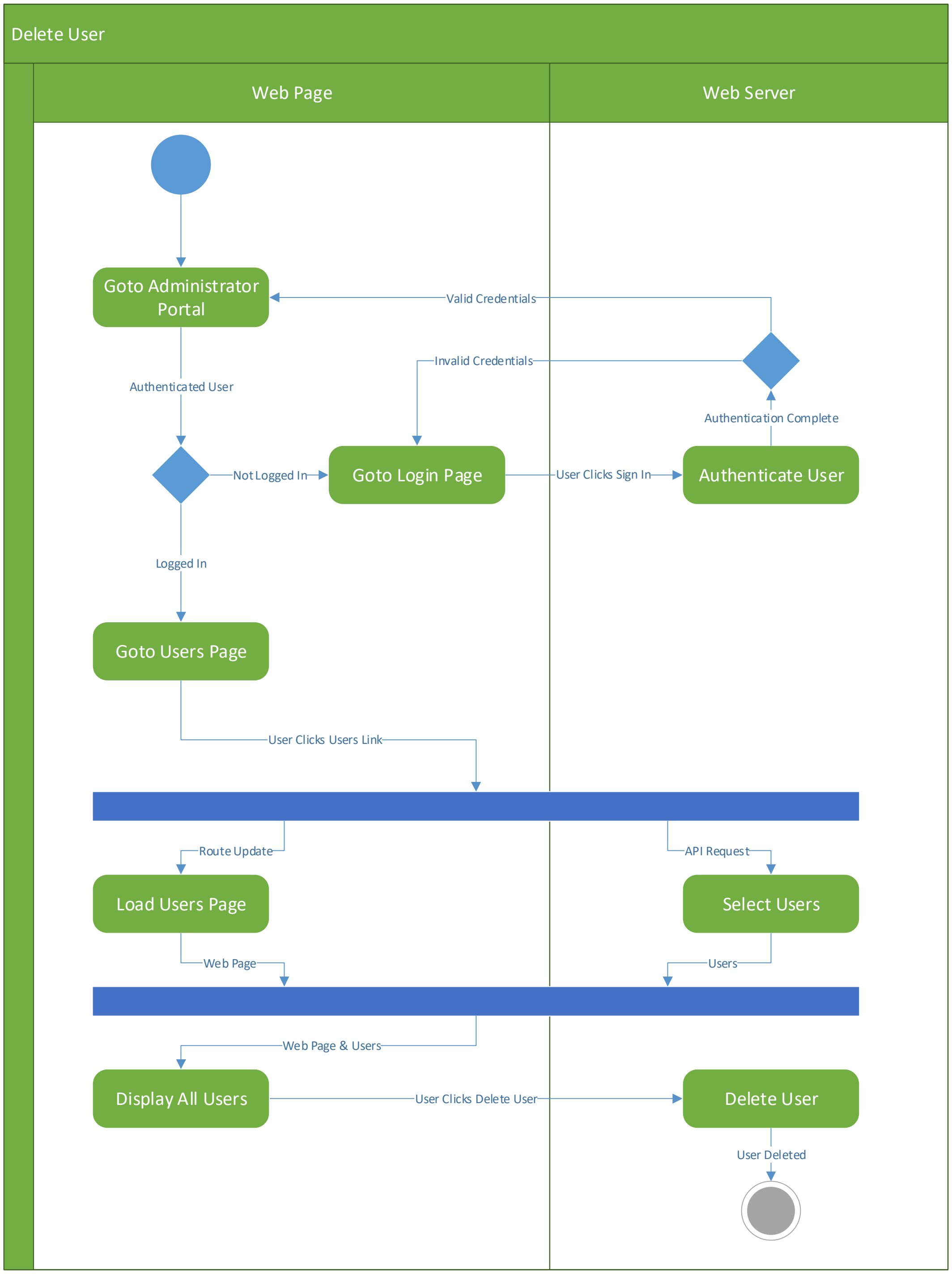
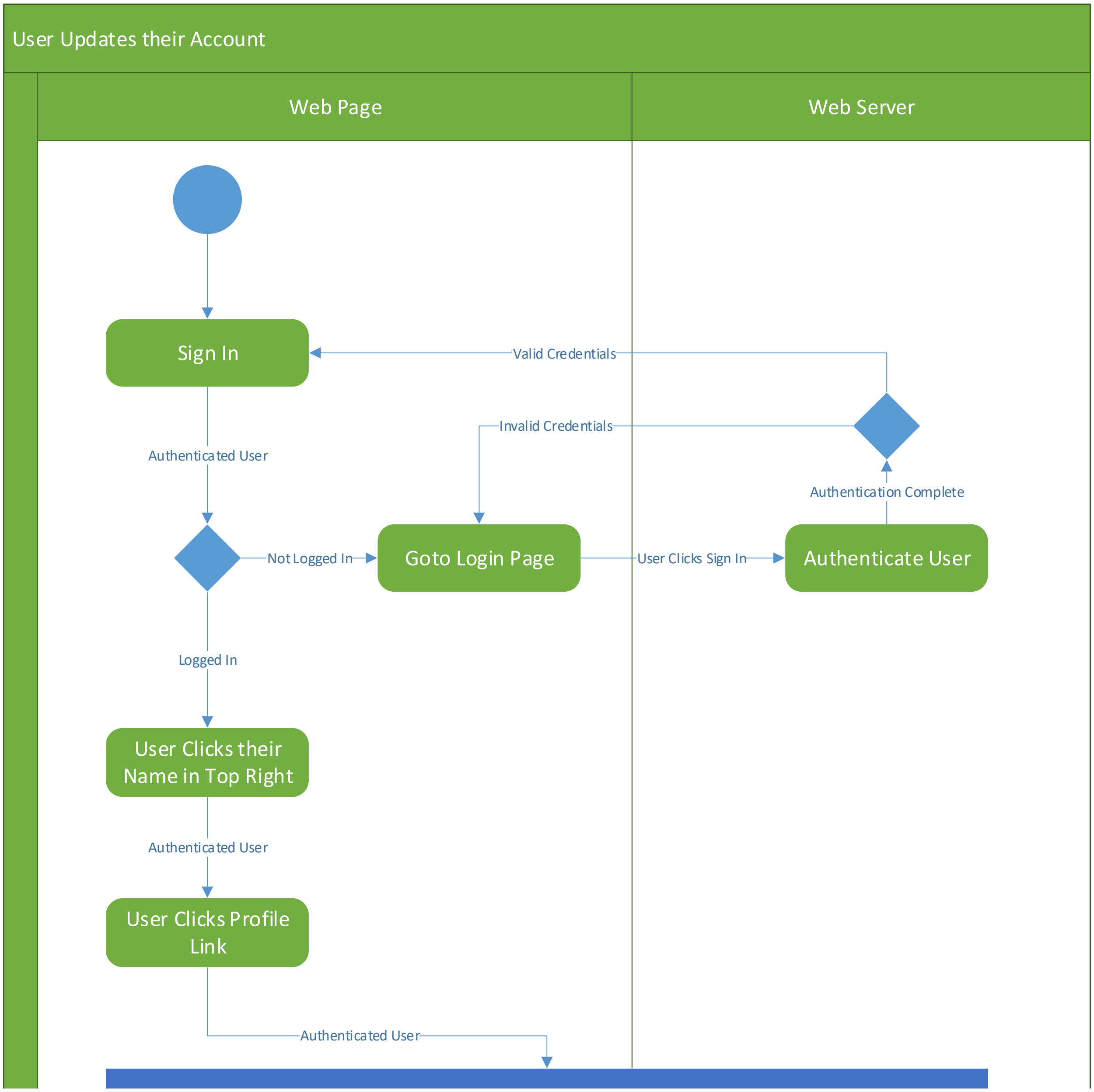


Figure APX2-6: Activity Diagram for Deleting a User


User Updates their Account
Web Page
Web Server
Sign In
Valid Credentials
Invalid Credentials
Authenticated User
Authentication Complete
Not Logged In
Goto Login Page
User Clicks Sign In
Authenticate User
Logged In
User Clicks their Name in Top Right
Authenticated User
User Clicks Profile Link
Authenticated User

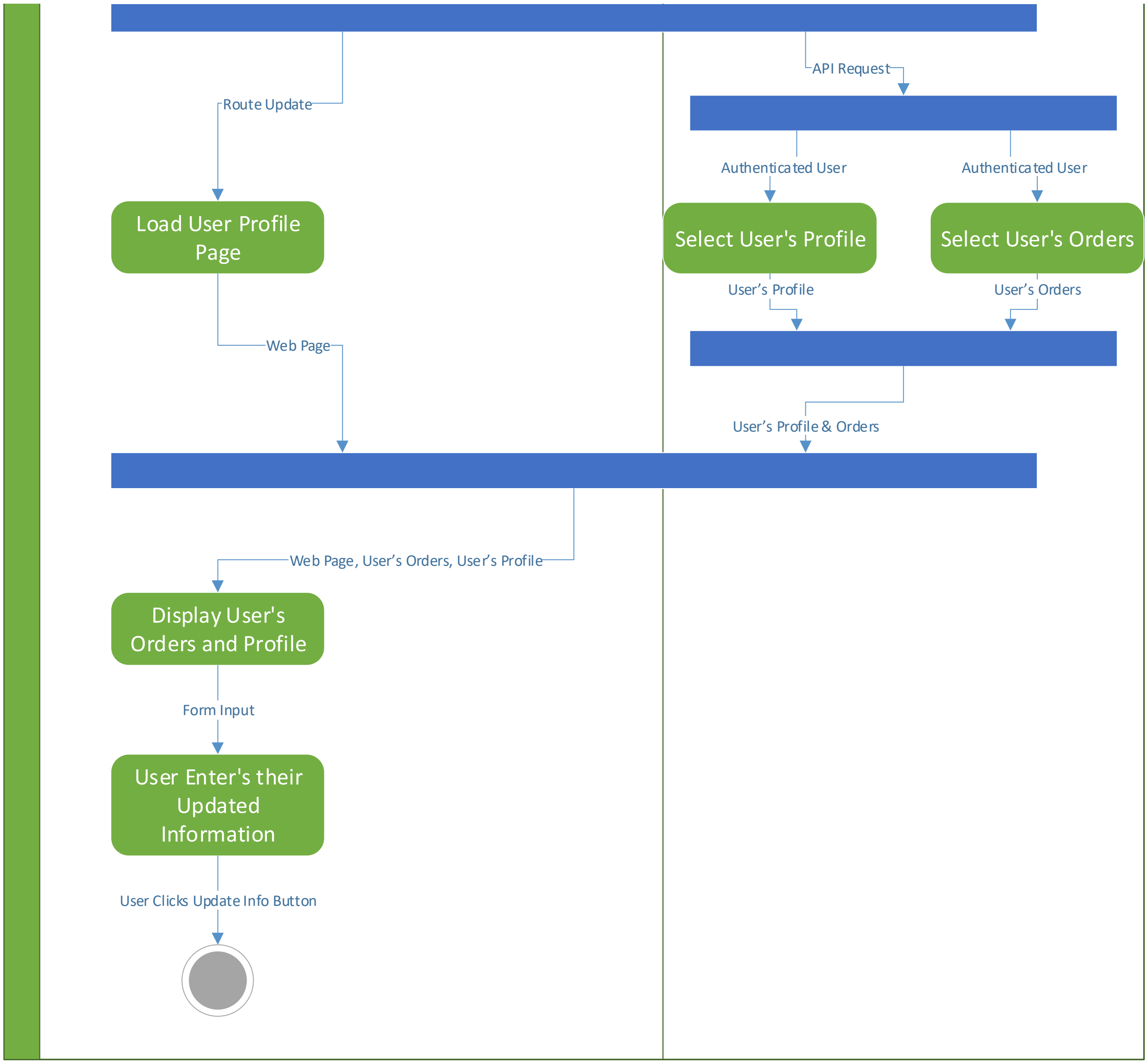


Figure APX2-7: Activity Diagram for User and Updating their Account

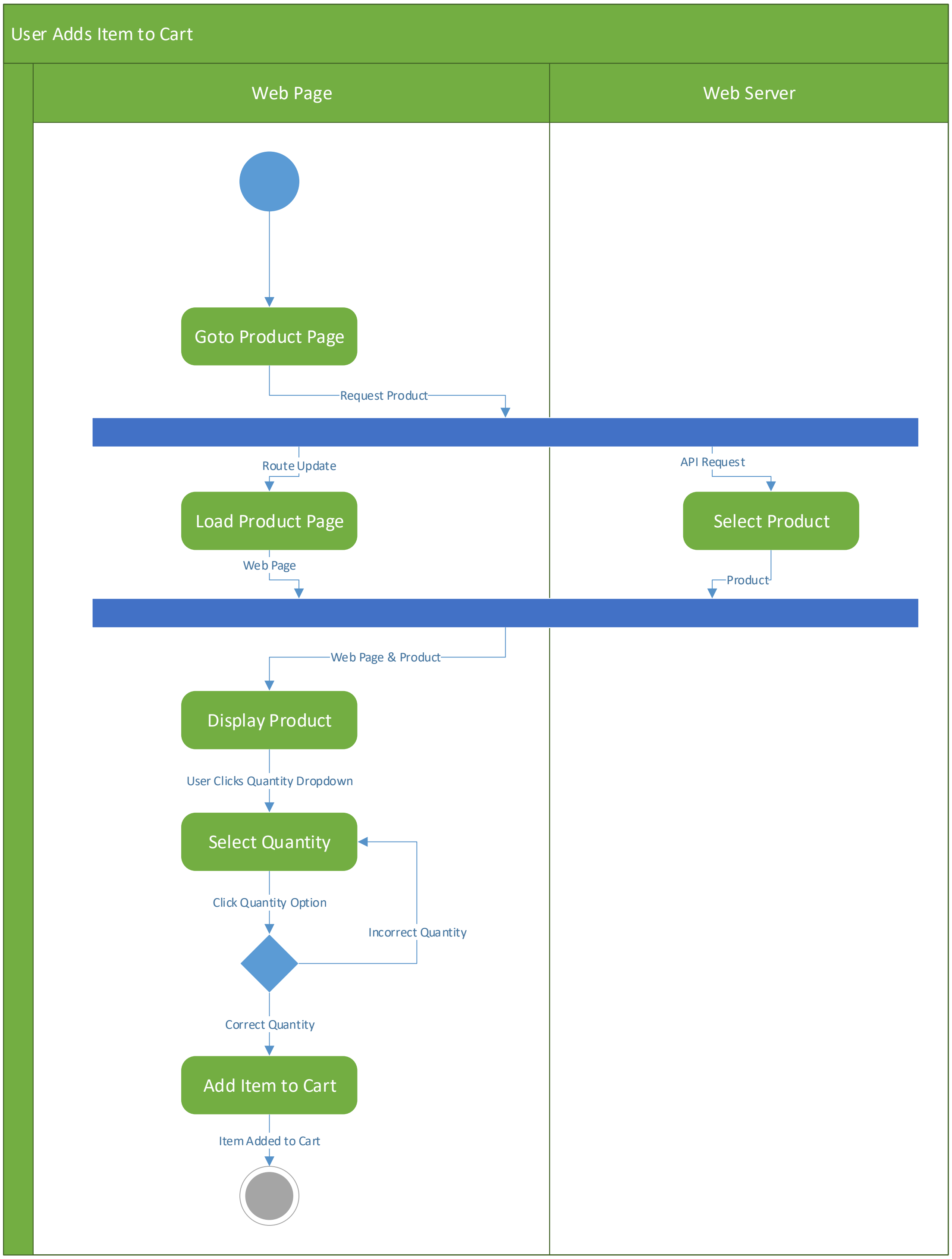
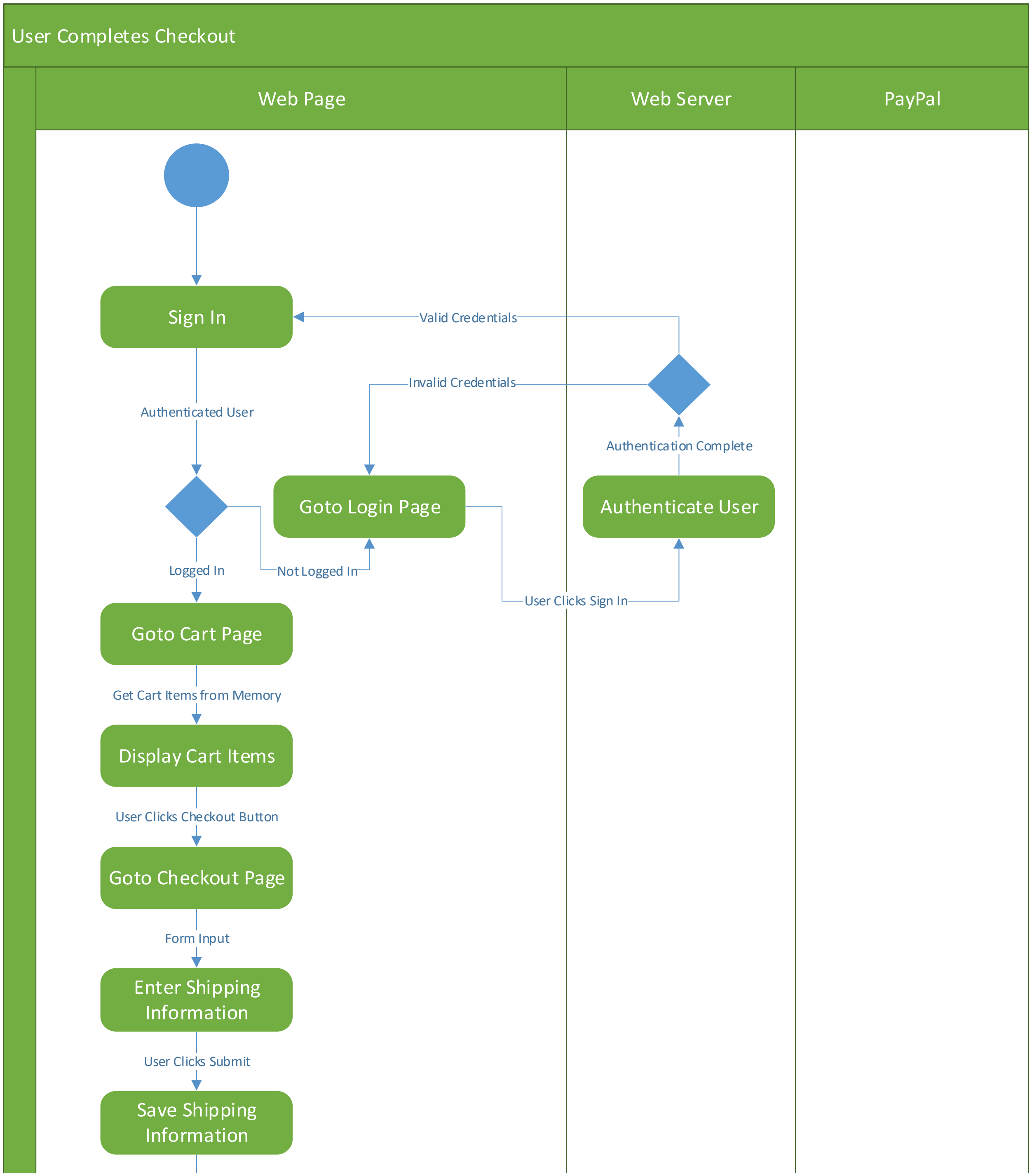
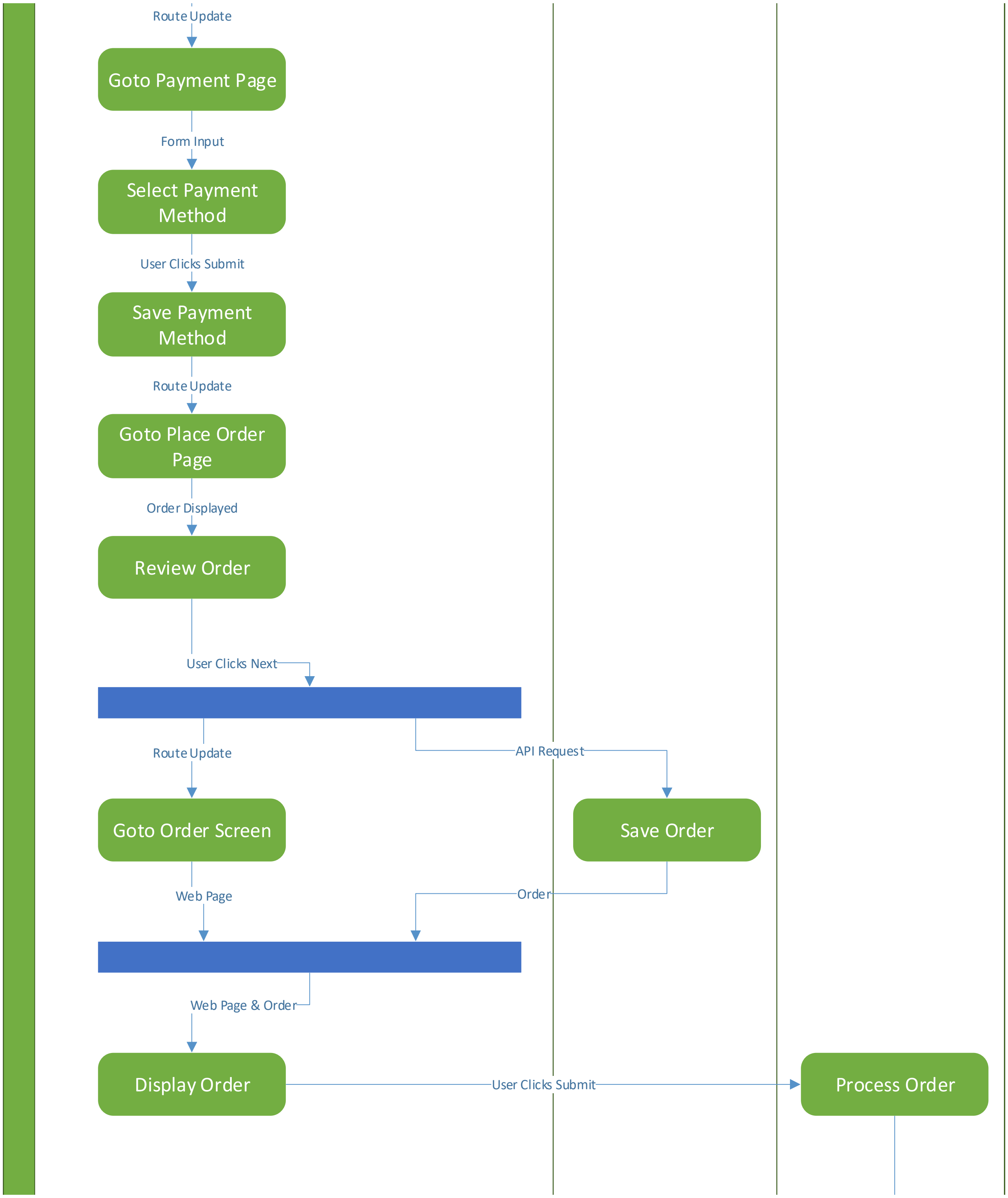


Figure APX2-1: Activity Diagram for Adding an Item to Cart



User Completes Checkout
Web Page
Web Server
PayPal
Sign In
Valid Credentials
Invalid Credentials
Authenticated User
Authentication Complete
Goto Login Page
Authenticate User
Logged In
Not Logged In
User Clicks Sign In
Goto Cart Page
Get Cart Items from Memory
Display Cart Items
User Clicks Checkout Button
Goto Checkout Page
Form Input
Enter Shipping Information
User Clicks Submit
Save Shipping Information



Route Update
Goto Payment Page
Form Input
Select Payment Method
User Clicks Submit
Save Payment Method
Route Update
Goto Place Order Page
Order Displayed
Review Order
User Clicks Next
Route Update
API Request
Goto Order Screen
Save Order
Web Page
Order
Web Page & Order
Display Order
User Clicks Submit
Process Order

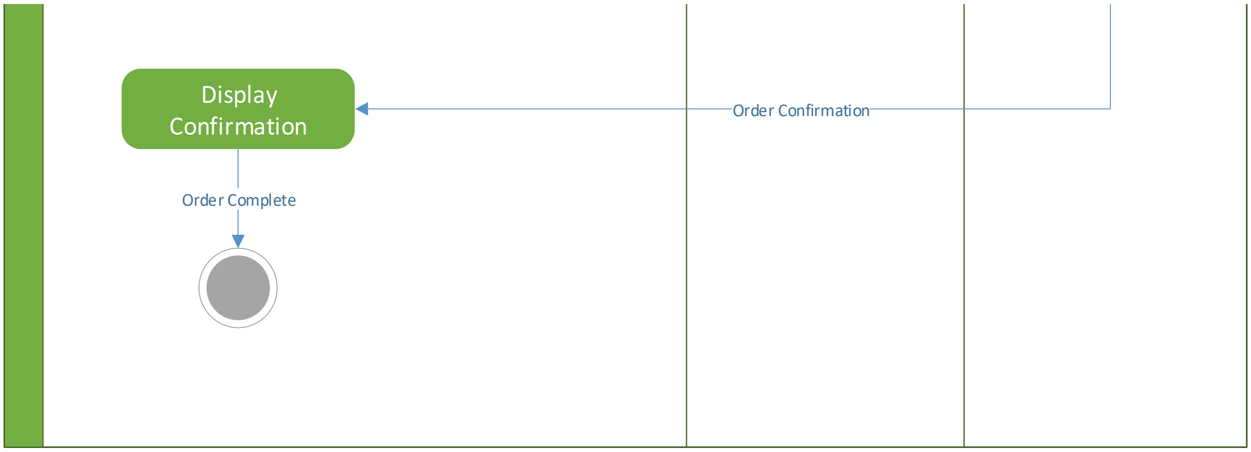


Figure APX2-9: Activity Diagram for Completing Checkout

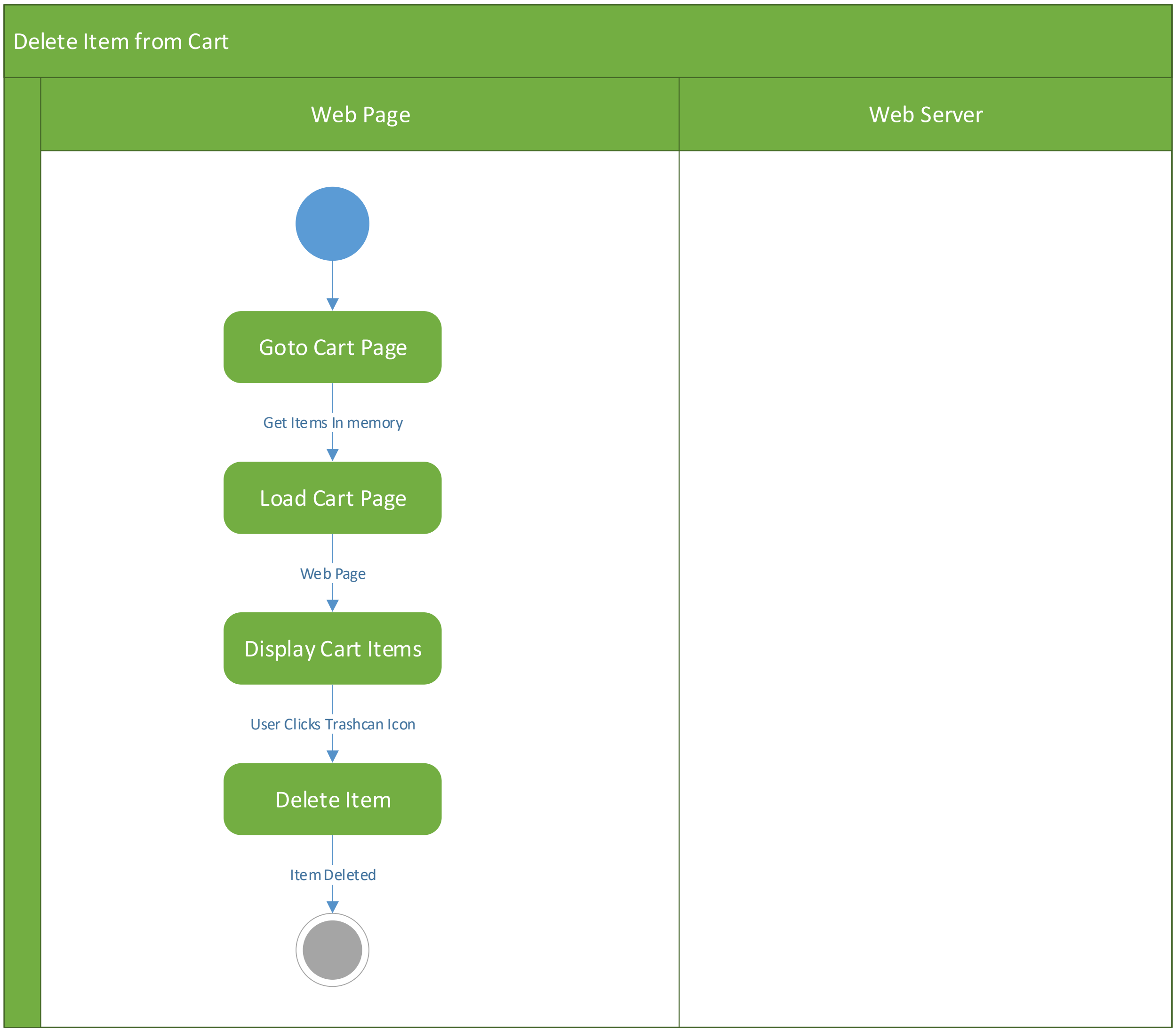


Figure APX2-10: Activity Diagram for Deleting an Item from the Cart

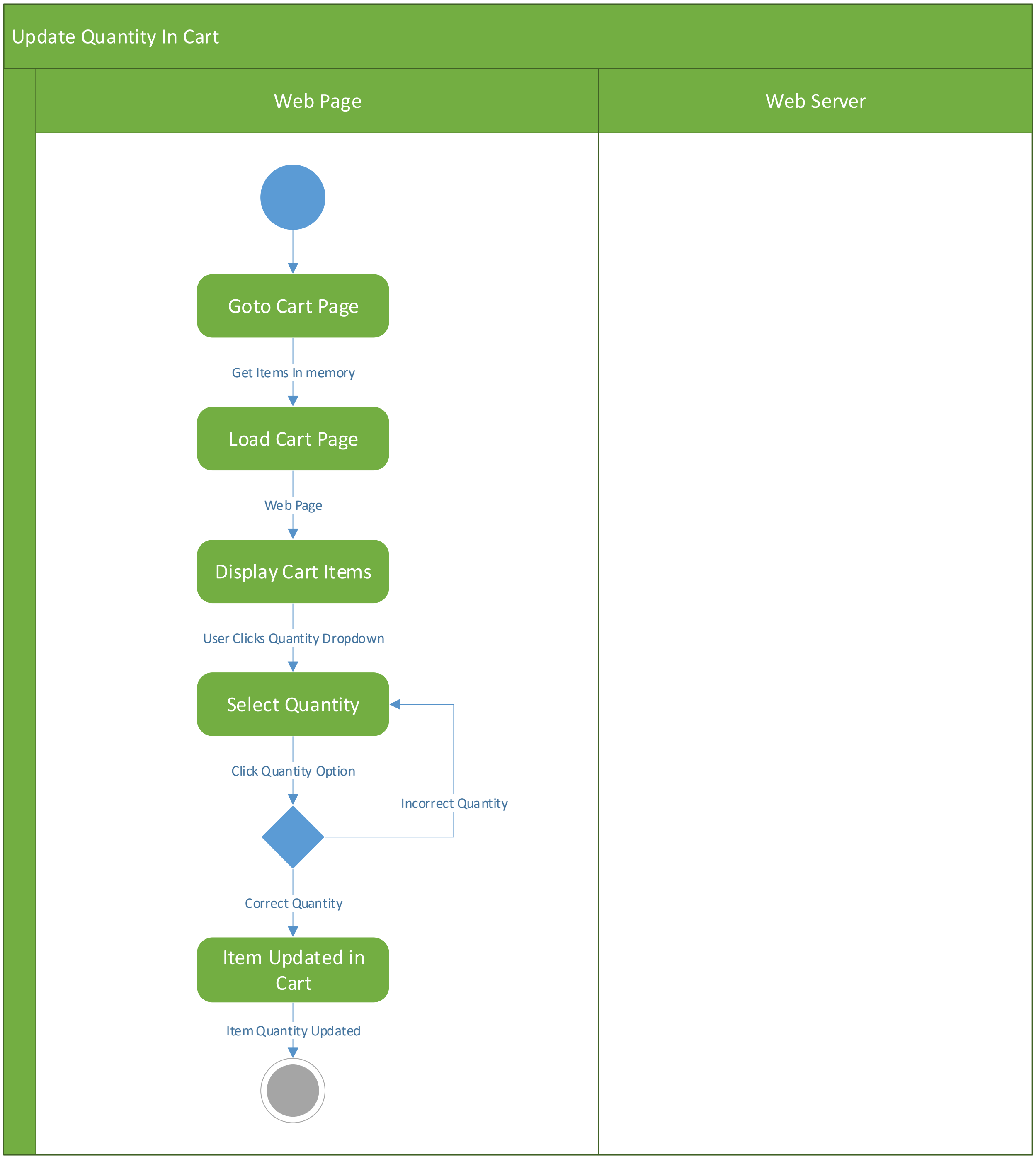
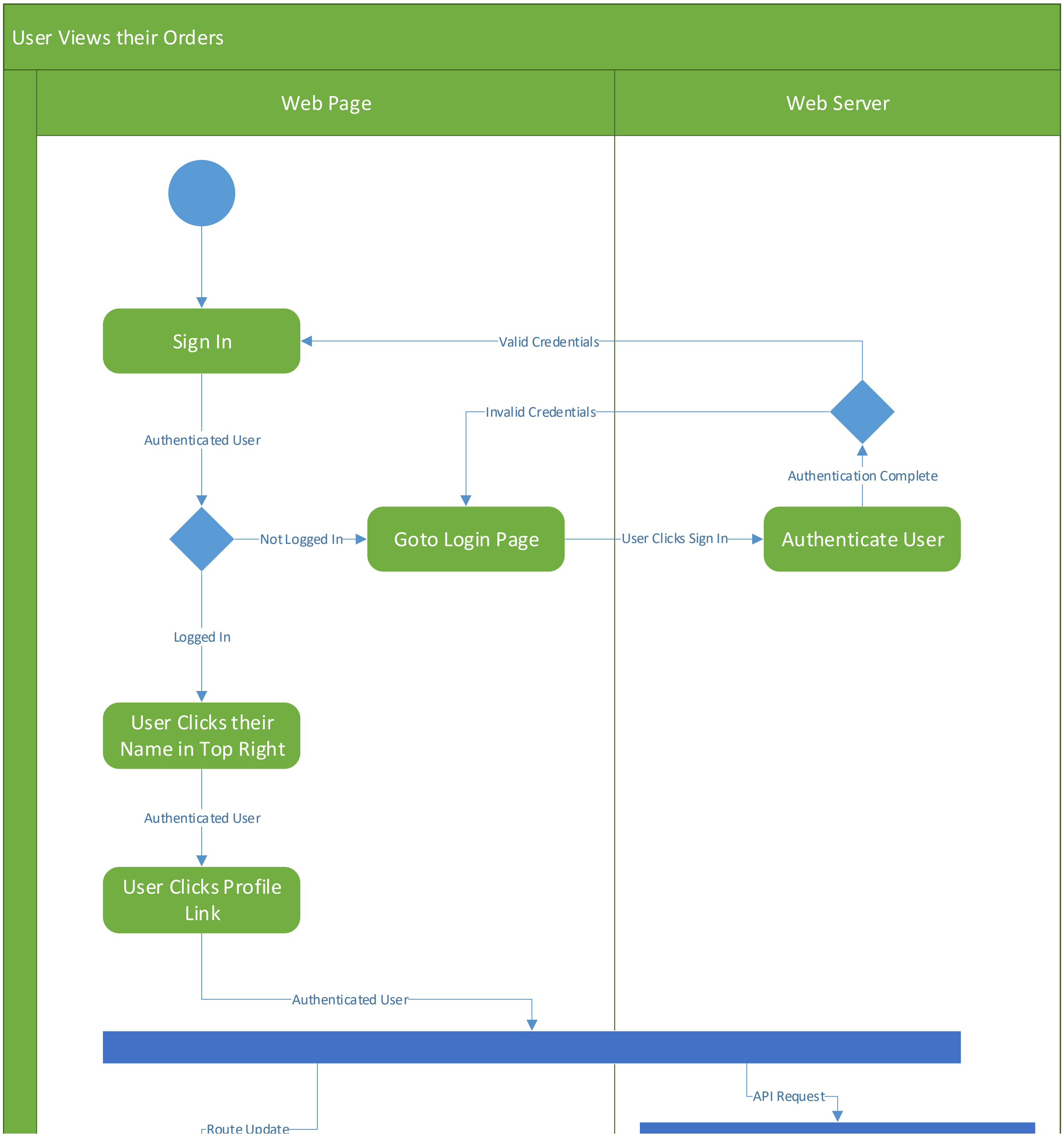
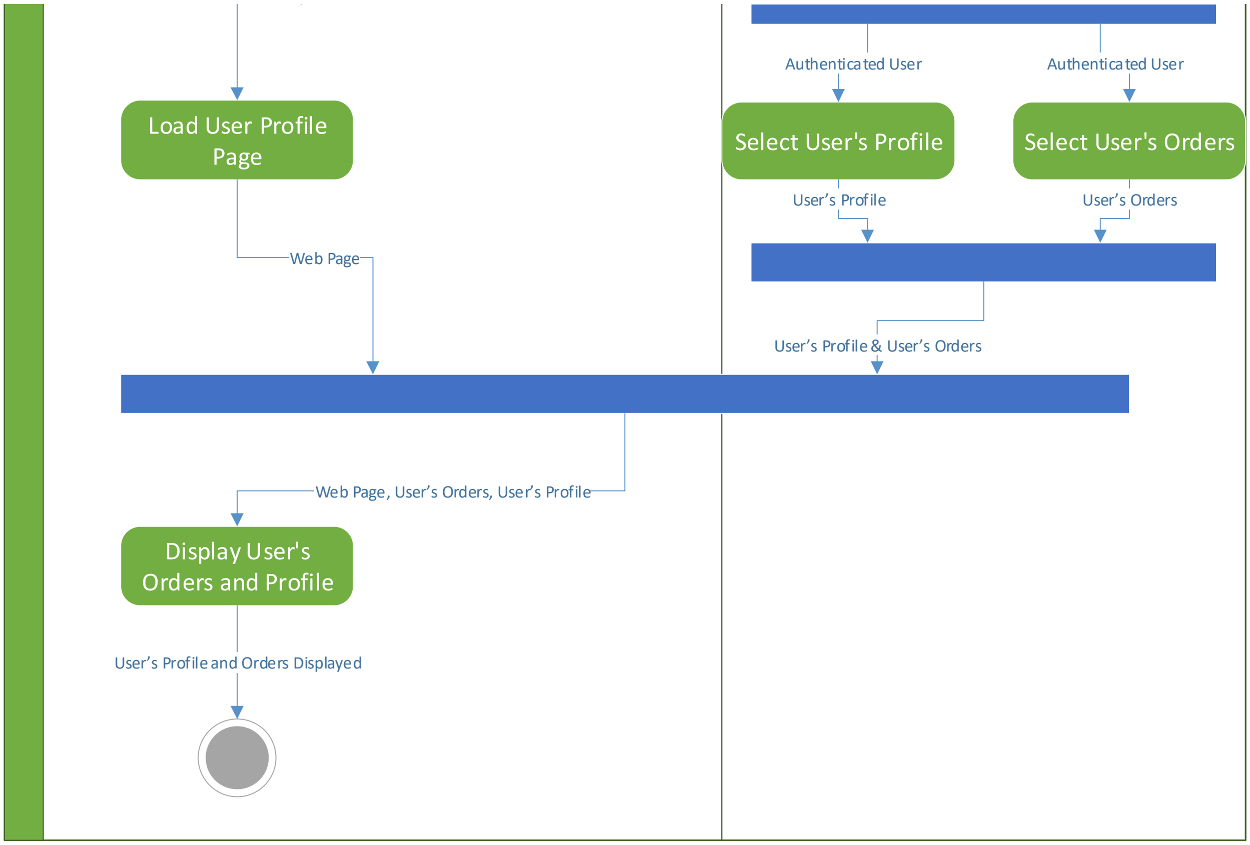


Figure APX2-11: Activity Diagram for Updating Quantity of items in Cart


User Views their Orders
Web Page
Web Server
Sign In
Valid Credentials
Invalid Credentials
Authenticated User
Authentication Complete
Not Logged In
Goto Login Page
User Clicks Sign In
Authenticate User
Logged In
User Clicks their Name in Top Right
Authenticated User
User Clicks Profile Link
Authenticated User
API Request
Route Update



Authenticated User
Authenticated User
Load User Profile Page
Select User's Profile
Select User's Orders
User's Profile
User's Orders
Web Page
User's Profile & User's Orders
Web Page, User's Orders, User's Profile
Display User's Orders and Profile
User's Profile and Orders Displayed

Figure APX2-12: Activity Diagram for a User Viewing their Orders

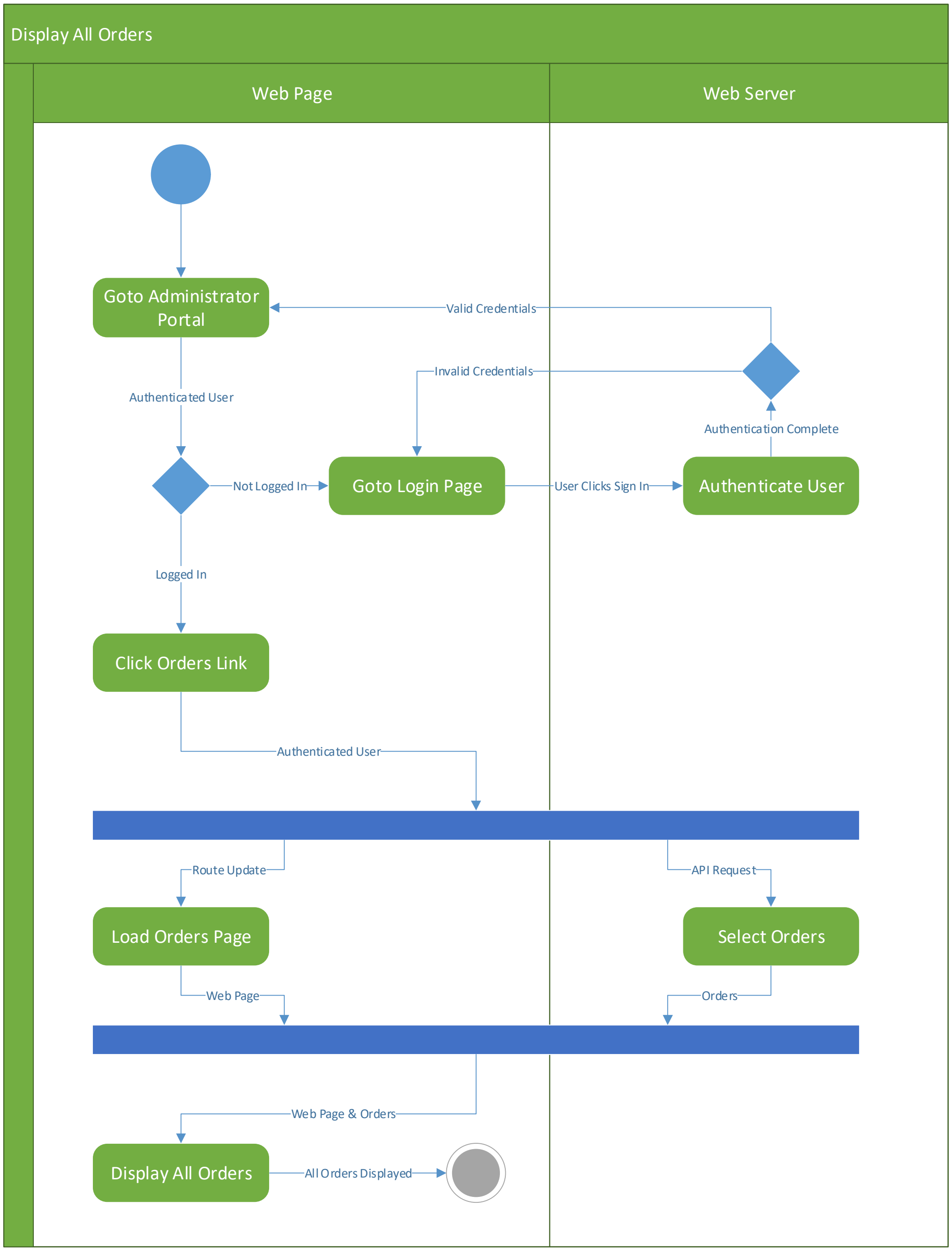
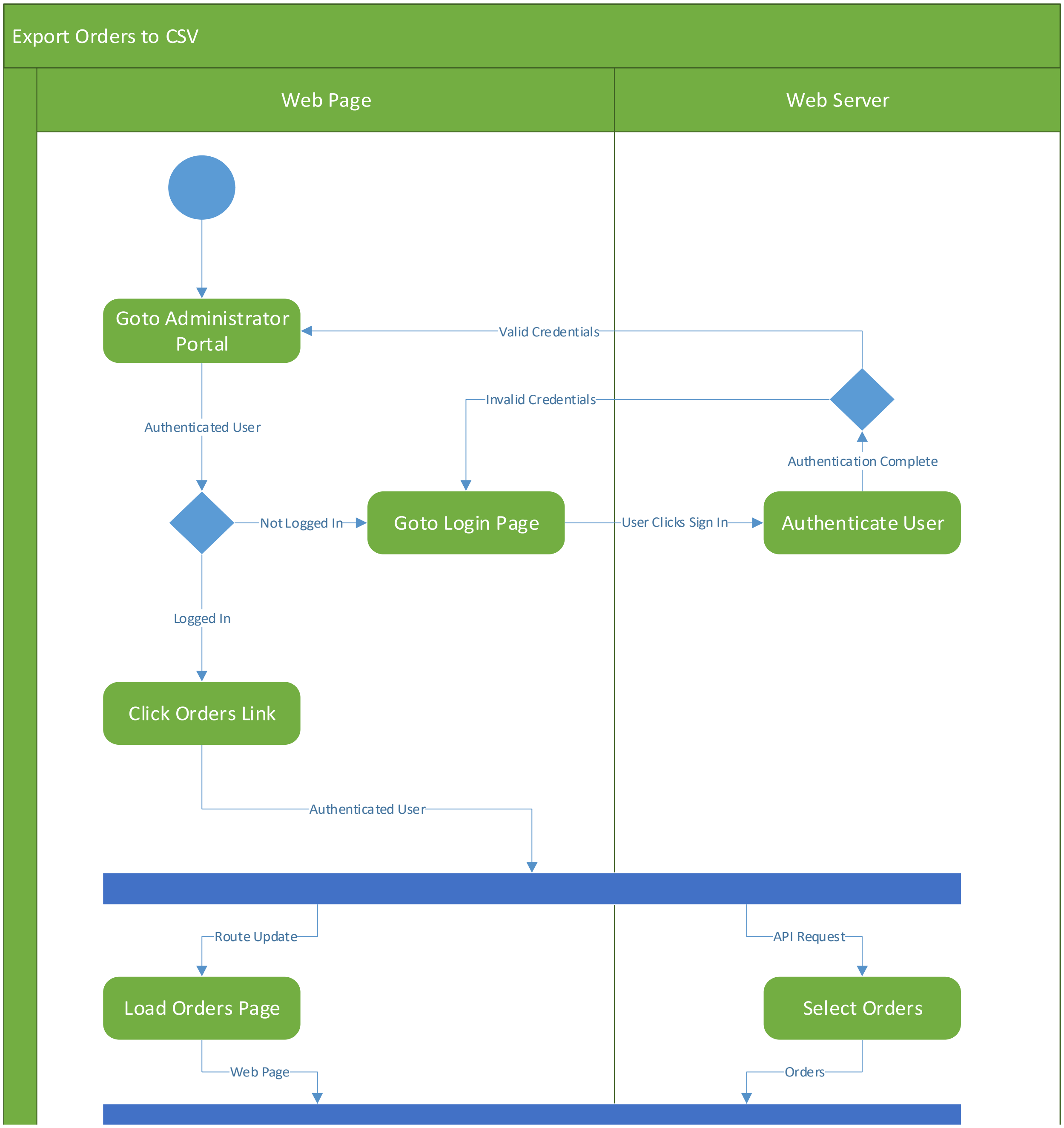


Figure APX2-13: Activity Diagram for Viewing all Orders


Export Orders to CSV
Web Page
Web Server
Goto Administrator Portal
Valid Credentials
Invalid Credentials
Authenticated User
Authentication Complete
Not Logged In
Goto Login Page
User Clicks Sign In
Authenticate User
Logged In
Click Orders Link
Authenticated User
Route Update
API Request
Load Orders Page
Select Orders
Web Page
Orders

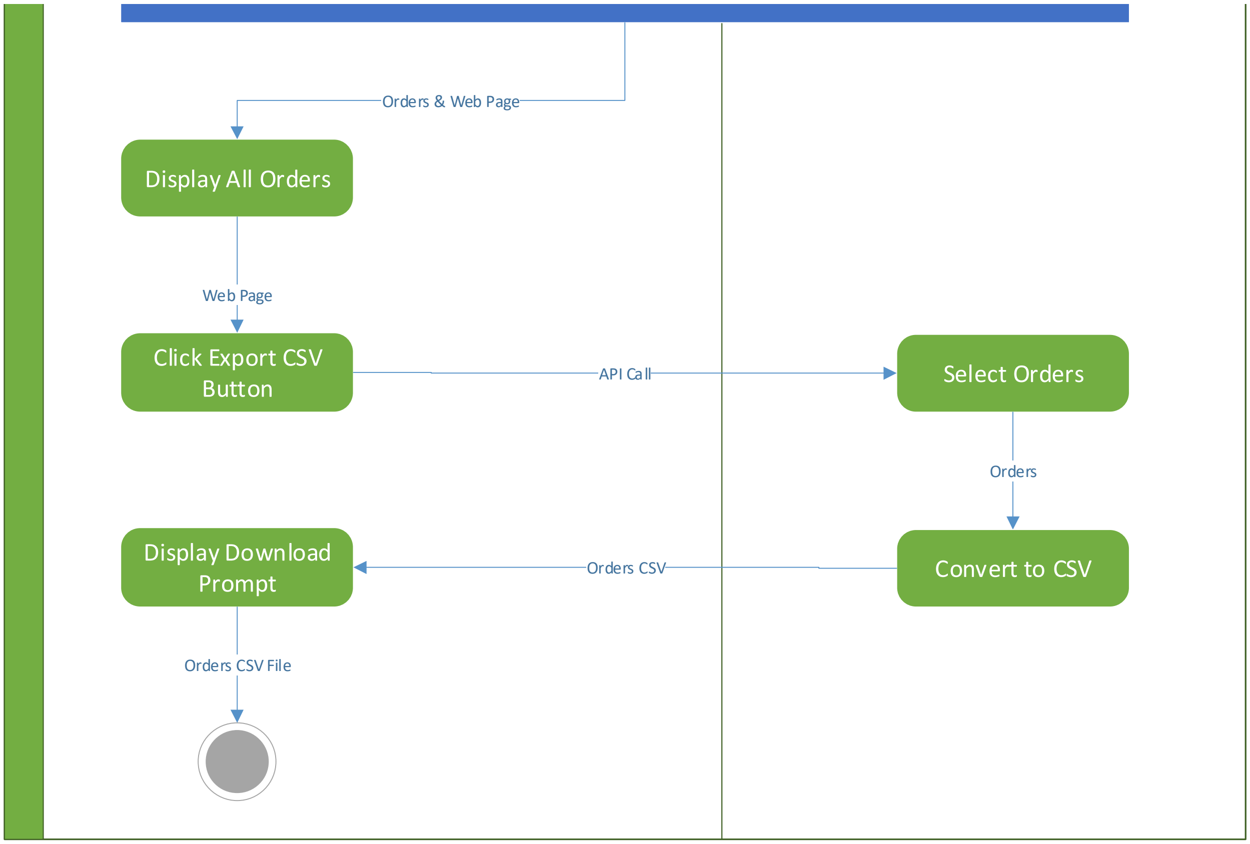
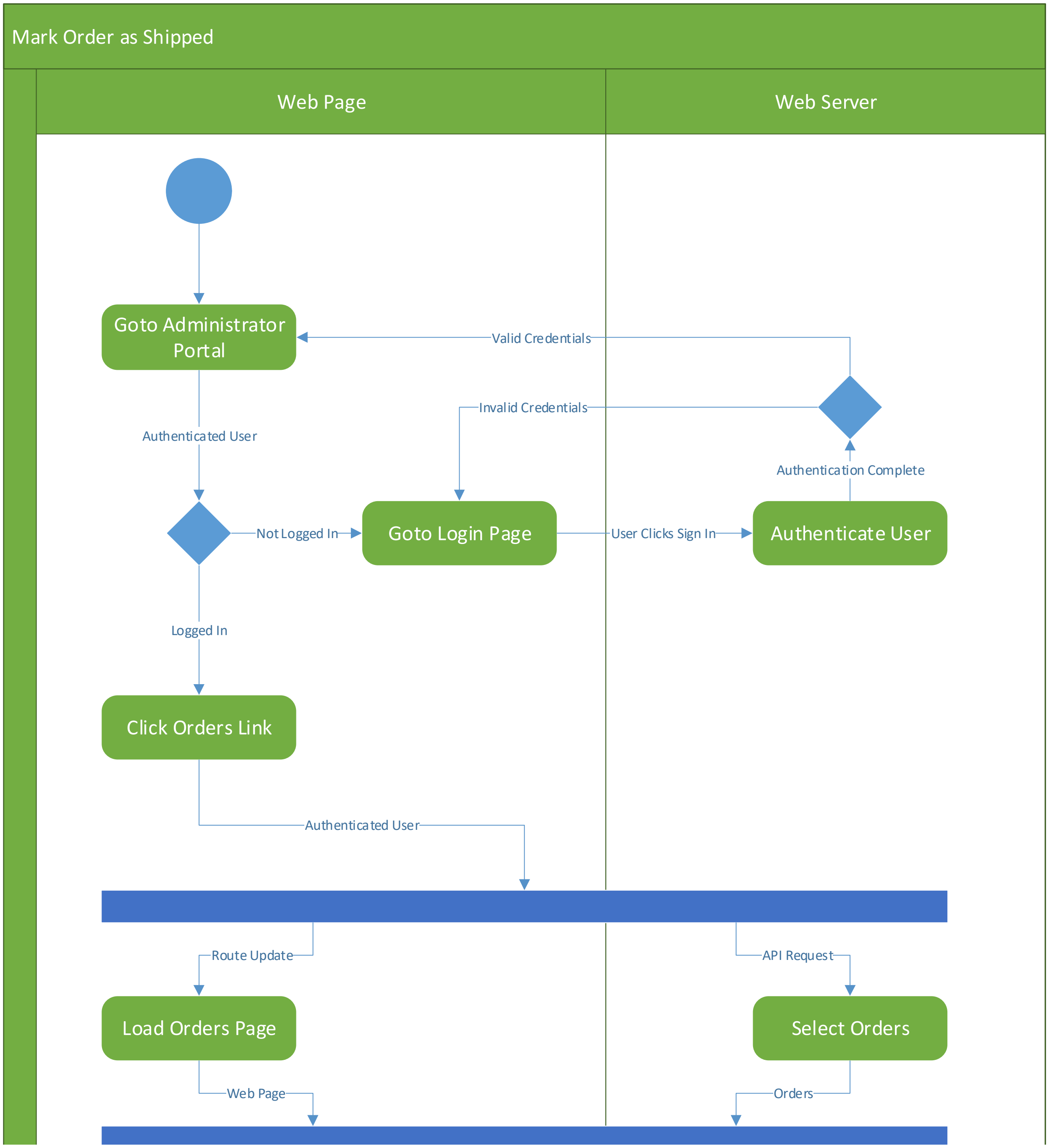


Figure APX2-14: Activity Diagram for Exporting Orders to CVS


Mark Order as Shipped
Web Page
Web Server
Goto Administrator Portal
Valid Credentials
Invalid Credentials
Authenticated User
Authentication Complete
Not Logged In
Goto Login Page
User Clicks Sign In
Authenticate User
Logged In
Click Orders Link
Authenticated User
Route Update
API Request
Load Orders Page
Select Orders
Web Page
Orders

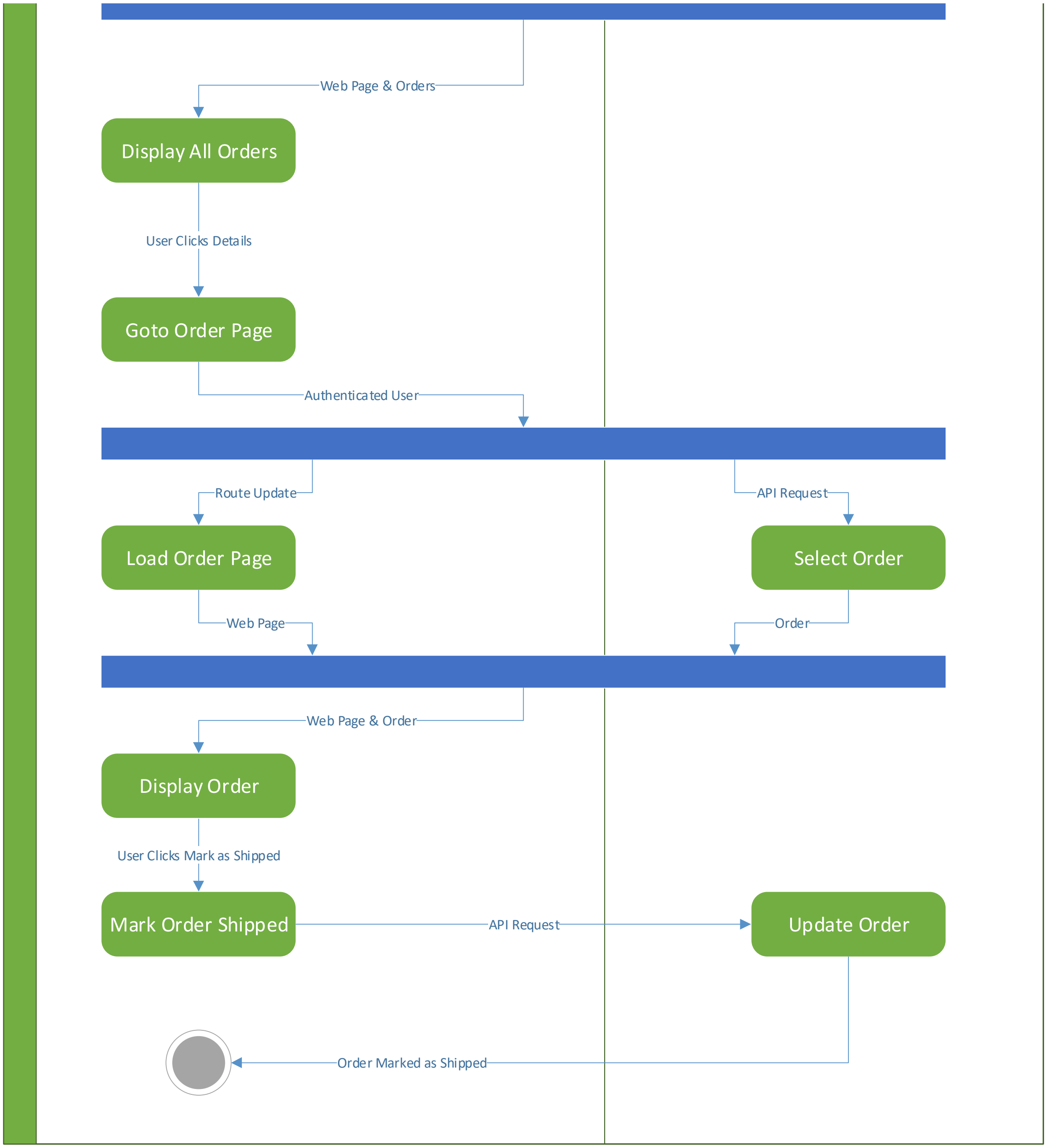


Figure APX2-15: Activity Diagram for Marking an Order as Shipped

Appendix III: Mocha Test Results

Symbol	Meaning
√	Success
n)	Failed Test

Table APX3-1: A Guide to Test Result Symbols

10/31/2020

Test Results

```
MongoDB
        User CRUD
                delete users
                        √ delete all users
                insert users
                        √ should not add user #1
                        1) should not add user #2
                        √ should add user #3 (99ms)
                find users
                        √ test #1 should        find user(s)
        Product CRUD
                delete products
                        √ delete all products
                insert products
                        √ should not add product #1
                        √ should add product #2
                        2) should not add product #3
                find products
                        √ test #1 should find product(s)
```

```
Lumico API
        Products
                √ get all products
        Users
                register user
                        √ should not add user #1
                        √ should add user #2 (105ms)
                login user
                        √ should not login user #1
                        √ should login user #2 (100ms)
                get user profile
                        √ should not get user profile #1
                        √ should get user profile #2 (104ms)

15 passing (673ms)
2 failing
```

Failed Tests

```
1) MongoDB
        User CRUD
                insert users
                        should not add user #2:
                Error: User should not have been added to collection.
                        Reason: Invalid email address

2) MongoDB
        Product CRUD
                insert products
                        should not add product #3:
                Error: Product should not have been added to collection.
                        Reason: Invalid countInStock
```

Test Data

```
const users = [
  {
    doc: {
      name: "",
      email: "",
      password: ""
    },
    fail: true,
    reason: "Empty"
  },
  {
    doc: {
      name: "Someone",
      email: "This",
      password: "something123"
    },
    fail: true,
    reason: "Invalid email address"
  },
  {
    doc: {
      name: "Greg",
      email: "GreggyG@mycooldomain.com",
      password: "something"
    },
    fail: false
  }
];
const products = [
  {
    doc: {
```

```
    user: null,
    name: "",
    image: "",
    size: "",
    gender: "",
    description: "",
    price: null,
    countInStock: null
  },
  fail: true,
  reason: "Empty"
},
{
  doc: {
    user: null,
    name: "Abcd",
    image: "/srv/images/default.png",
    size: "udsjihd",
    gender: "jshdf",
    description: "gfhjsdlfj",
    price: 1.20,
    countInStock: 10
  },
  fail: false
},
{
  doc: {
    user: null,
    name: "dasjkh",
    image: "a/b/c/d/e/f/g/h.png",
    size: "jsdkaknl",
    gender: "klnsdfjkl",
    description: "jkdsfjhksdf",
      price: 1.20,
      countInStock: 5.5
    },
    fail: true,
    reason: "Invalid countInStock"
  }
];
const userFilters = [
  {
    filter: {
      name: "Greg",
      email: "GreggyG@mycooldomain.com"
    },
    length: 1
  }
];
```

```
const productFilters = [
  {
    filter: {
      name: "Abcd",
    },
    length: 1
  }
];

const userTests = [
  {
    doc: {
      name: "",
      email: "",
      password: ""
    },
    fail: true,
    reason: "Empty"
  },
  {
    doc: {
      name: "Test",
      email: "test@email.com",
      password: "3(GoodluckHaveBatman)"
    },
    fail: false
  }
];
```

11/03/2020

Test Results

```
MongoDB
    User CRUD
        delete users
            √ delete all users
        insert users
            √ should not add user #1
            1) should not add user #2
            √ should add user #3 (111ms)
        find users
            √ test #1 should find user(s)
    Product CRUD
        delete products
            √ delete all products
        insert products
            √ should not add product #1
            √ should add product #2
            2) should not add product #3
        find products
            √ test #1 should find product(s)
    Order CRUD
        delete orders
            √ delete all orders
        insert orders
            √ should not add order #1
            √ should add order #2
        find orders
            √ test #1 should find order(s)
            √ test #2 should find order(s)
```

```
Lumico API
    Products
        √ get all products
    Users
        register user
            √ should not add user #1
            √ should add user #2
            √ should add user #3
        login user
            √ should not login user #1
            √ should login user #2 (106ms)
            √ should login user #3 (105ms)
        get user profile
            √ should not get user profile #1
            √ should get user profile #2 (99ms)
            √ should get user profile #3 (100ms)
    Orders
        Creates a new order
            √ should not place order #1 as user #1
            √ should not place order #2 as user #1
            √ should not place order #1 as user #2 (102ms)
            √ should place order #2 as user #2 (105ms)
            √ should not place order #1 as user #3 (103ms)
            √ should place order #2 as user #3 (102ms)
        Get a user's orders
            √ should not get orders for user #1
            √ should get orders for user #2 (104ms)
            √ should get orders for user #3 (104ms)
        Updates an order to set its status to paid
            √ should not pay orders for user #1
            √ should pay orders for user #2 (173ms)
            √ should pay orders for user #3 (178ms)
        Admin order handling
            √ get all orders (113ms)
            √ deliver all orders (118ms)

37 passing (2s)
2 failing
```

Failed Tests

```
1) MongoDB
       User CRUD
              insert users
                     should not add user #2:
              Error: User should not have been added to collection.
                     Reason: Invalid email address

2) MongoDB
       Product CRUD
              insert products
                     should not add product #3:
              Error: Product should not have been added to collection.
                     Reason: Invalid countInStock
```

Test Data

```
const users = [
  {
    doc: {
      name: "",
      email: "",
      password: ""
    },
    fail: true,
    reason: "Empty"
  },
  {
    doc: {
      name: "Someone",
      email: "This",
      password: "something123"
    },
    fail: true,
    reason: "Invalid email address"
  },
  {
    doc: {
      name: "Greg",
      email: "GreggyG@mycooldomain.com",
      password: "something"
    },
    fail: false
  }
];
```

```
const products = [
  {
    doc: {
      user: null,
      name: "",
      image: "",
      size: "",
      gender: "",
      description: "",
      price: null,
      countInStock: null
    },
    fail: true,
    reason: "Empty"
  },
  {
    doc: {
      user: null,
      name: "Abcd",
      image: "/srv/images/default.png",
      size: "udsjihd",
      gender: "jshdf",
      description: "gfhjsdlfj",
      price: 1.20,
      countInStock: 10
    },
    fail: false
  },
  {
    doc: {
      user: null,
      name: "dasjkh",
      image: "a/b/c/d/e/f/g/h.png",
      size: "jsdkaknl",
      gender: "klnsdfjkl",
      description: "jkdsfjhksdf",
      price: 1.20,
      countInStock: 5.5
    },
    fail: true,
    reason: "Invalid countInStock"
  }
];
const userFilters = [
  {
    filter: {
      name: "Greg",
      email: "GreggyG@mycooldomain.com"
    },
    length: 1
  }
];
const productFilters = [
  {
    filter: {
      name: "Abcd",
    },
    length: 1
  }
];

const userTests = [
  {
    doc: {
      name: "",
      email: "",
      password: ""
    },
    fail: true,
    reason: "Empty"
  },
  {
    doc: {
      name: "MochaTestUser",
      email: "test@email.com",
      password: "3(GoodluckHaveBatman)"
    },
    fail: false
  },
  {
    doc: {
      name: "MochaAdmin",
      email: "jfkdhsf",
      password: "dfhujisdf"
    },
    fail: false
  }
];

const validAddress = {
  address: "1234 Origin Ln",
  city: "Alfred",
  zipCode: "77777",
  country: "Why is this here? Where is state?"
};

const orderTests = [
  {
    doc: {
      orderItems: [],
      shippingAddress: {},
      paymentMethod: "",
      paymentResult: {},
      taxPrice: 0,
      shipping: 0,
      totalPrice: 0,
      isPaid: false,
      paidAt: null,
      isDelivered: false
    },
    fail: true,
    reason: "Order is empty"
  },
  {
    doc: {
      orderItems: [],
      shippingAddress: validAddress,
      paymentMethod: "Something",
      paymentResult: null,
      taxPrice: 123.12,
      shipping: 15.00,
      totalPrice: 123.12 + 15,
      isPaid: false,
      paidAt: null,
      isDelivered: false
    },
    fail: false,
    reason: ""
  }
];
```

11/10/2020

Test Results

Same results as on 11/03/2020.

Failed Tests

Same results as on 11/03/2020.

Test Data

Same data used on 11/03/2020.

11/17/2020

Test Results

Same results as on 11/03/2020

.

Failed Tests

Same results as on 11/03/2020.

Test Data

Same data used on 11/03/2020.

Appendix IV: Code Audit Endpoints

Module	Module Path	Endpoint	Request Type	Args	Authentication	Description	Success	Testing Status	First Observed	Outstanding Issues
productController	backend\controllers\productController.js	/api/products/	GET	null	None	Get all products	product document array	Complete	10/28/2020	
productController	backend\controllers\productController.js	/api/products/:id	GET	id url argument	None	Get one product	product document	Forthcoming	10/28/2020	When running with NODE_ENV = "production" using an invalid id gives back an internal error message. {"message":"Cast to ObjectId failed for value \"1\" at path \"_id\" for model \"Product\"","stack":null }
productController	backend\controllers\productController.js	/api/products/	POST	null	Admin	Create a product	201 Created product document	Forthcoming	11/03/2020	The product is hard coded There is no way to add admin users
userController	backend\controllers\userController.js	/api/users/	POST	{ name, email, password }	None	Register a new user	201 Created { _id: user._id, name: user.name, email: user.email, isAdmin: user.isAdmin, token: generateToken(user._id) }	Complete	10/28/2020	

userController	backend\controllers\userController.js	/api/users/login	POST	{ email, password }	None	Creates a new user session	200 Success
							{
							_id: user._id,
							name: user.name,
							email: user.email,
							isAdmin: user.isAdmin
							}
								Complete	10/28/2020	When running with NODE_ENV = "production" sending a GET request results in a 404 that gives json error message.

userController	backend\controllers\userController.js	/api/users/profile	GET	{ user }	User	Get a user's profile	200 Success
							{
							_id: user._id,
							name: user.name,
							email: user.email,
							isAdmin: user.isAdmin
							}
								Complete	10/28/2020	

userController	backend\controllers\userController.js	/api/users/profile	PUT	{ name, email, password }	User	Update a user's profile	200 Success
							{
							_id: updatedUser._id,
							name: updatedUser.name,
							email: updatedUser.email,
							isAdmin: updatedUser.isAdmin,
							token: generateToken(updatedUser._id),
							}
								Void	10/28/2020	

orderController	backend\controllers\orderController.js	/api/orders/	POST	{ orderItems, shippingAddress, paymentMethod, itemsPrice, taxPrice, shipping, totalPrice }	User	Creates a new order	201 Created order document	Complete	11/03/2020	
orderController	backend\controllers\orderController.js	/api/orders/	GET	null	Admin	Get all orders	200 order document array	Complete	11/03/2020	Never used There is no way to add admin users
orderController	backend\controllers\orderController.js	/api/orders/myorders	GET	{ user }	User	Get a user's orders	200 order document array	Complete	11/03/2020	
orderController	backend\controllers\orderController.js	/api/orders/:id	GET	id url argument	User	Gets an order by id	200 order document	Void	11/03/2020	This should probably be admin only since it allows someone to look up any order.
orderController	backend\controllers\orderController.js	/api/orders/:id/pay	PUT	id url argument { id, status, update_time, payer: { email_address }}	User	Updates an order to set its status to paid	200 order document	Complete	11/03/2020	
orderController	backend\controllers\orderController.js	/api/orders/:id/deliver	PUT	id url argument	Admin	Updates an order to set its status to delivered	200 order document	Complete	11/03/2020	There is no way to add admin users

orderController	backend\controllers\orderController.js	/api/orders/export-csv	GET	token query argument (the user's token)	Admin	Exports all orders a CSV	200 CSV of all order documents	Void	11/09/2020	

Table APX4-1: Details about All Server Endpoints (Trimmed Down Version)

Testing Plan Code Audit.xlsx

Figure APX4-1: Attached full version of the testing plan code audit

Appendix V: Code Audit Models

Module	Module Path	Description	Potential Failure Points	Testing Status	Date of Observation	Outstanding Issues
usermodel	backend\models\usermodel.js	Defines a user. Includes logic for password matching.	Email pattern not specified.	Complete	10/28/2020	Email has no validation beyond being required.
productmodel	backend\models\productmodel.js	Defines a product.	countInStock could be invalid. No validation for image data (this will be placed in html unsanitized, making XSS possible)	Complete	10/28/2020	
ordermodel	backend\models\ordermodel.js	Defines an order.	No validation on any information added from web interface	Complete	11/03/2020	

Table APX5-1: Details about each Data Model

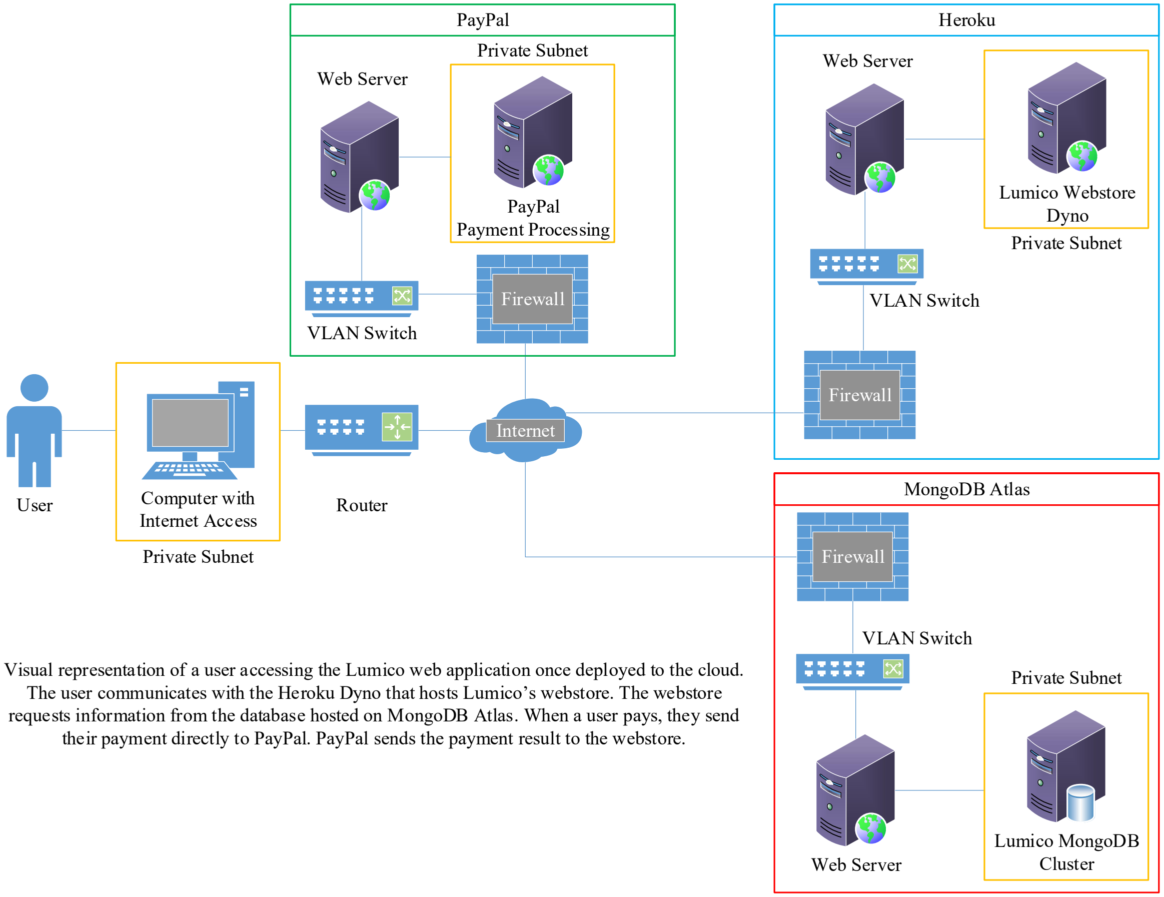
Appendix VI: Web Interface Error Report

Type	Status	Priority	Page	Preconditions	Other Notes	Description
Error	Not Fixed	Medium	Order Page		We can disable the breadcrumbs for admin users	Admins can click on the breadcrumbs at the top of the order screen when reviewing an order
Error	Not Fixed	Medium	Order Page (Checkout)	Order is created first to display the summary page	Ideally, if PayPal fails, the whole order should be cancelled.	If the PayPal payment errors out, the order is still created
Error	Not Fixed	Medium	User Profile	The PayPal payment failed, but the order was created	The Order Details button links to http://localhost:3000/order/$undefined	When an order is placed and PayPal fails, trying the view order details gives TypeError: order is undefined
Error	Not Fixed	Medium	Order List	The PayPal payment failed, but the order was created		When an admin views an order from the OrderList page, they are not notified if the paypal api call fails, it just loads forever.
Error	Not Fixed	Medium	Product	Have the backend server's console open before clicking a product on the home page	The product page still functions properly, there is just an error printed in the backend server's console	getProductByID includes two instances of res.json(product) which causes an ERR_HTTP_HEADERS_SENT error
Error	Fixed	High	Registration	Email confirmation sends even before registration is completed		
Error	Fixed	High	Order Page		The order.itemsPrice displayed immediately after the Order Summary header needs to have .toFixed(2) added.	Some prices appear very oddly due to precision problems. For example, in Order #:5fa1b52458946e061d7416d9, item price is $249.95000000000002
Error	Fixed	High	Shipping Page			Zip Code should be type text, not number State should be a dropdown box, not text Country should not be here at all, or should default to United States and be disabled
Error	Fixed	High	Edit Product	Upload folder does not exist in LumicoMain		When selecting an image for a new product, the upload fails with the following message "Error [ERR_HTTP_HEADERS_SENT]"
Error	Fixed	High	Order List			If an admin user reloads the orderlist page, they get an error message, "TypeError: userInfo is null"
Error	Fixed	Medium	Order Page (Checkout)		This is because I do not have the client id in my env file.	Checking out gives a 400 status code from PayPal
Error	Fixed	Medium	User Profile	The PayPal payment failed, but the order was created	Implemented simple refresh button	Orders do not appear in the user's profile until the user logs out and back in
Error	Fixed	Medium	Shipping Page	I was logged in as an admin at the time		There are steps at the top of the page that say "Sign In"

Error	Fixed	Low	User Profile	Admin clicks edit to edit a user on the Users admin page.	The user is not notified of any error taking place.	Going to the currently signed in admin user's profile page displays the wrong user's information
Suggestion	Not Implemented	Low	Order Page			Ideally, there would be a back button to take admins back to the order list
Suggestion	Not Implemented	Low	Registration			The format errors just pile up: once an error is resolved, the message does not go away, more are added on top.
Suggestion	Implemented	Low	Edit User		Completed. Added a simple window alert	When an admin updates a user, a message should be displayed telling them that changes have been saved

Table APX6-1: Web Interface Error Report

Appendix VII: Network Architecture



Visual representation of a user accessing the Lumico web application once deployed to the cloud. The user communicates with the Heroku Dyno that hosts Lumico's webstore. The webstore requests information from the database hosted on MongoDB Atlas. When a user pays, they send their payment directly to PayPal. PayPal sends the payment result to the webstore.

Figure APX7-1: Lumico Cloud Network Architecture Diagram

Appendix VIII: Application Environment Variables

REDACTED

Appendix IX: Application Source Code

REDACTED