Team Amazonian Prime

Technical Design

April. 4, 2023
Version 2.0

Document Information

Revision History

Date	Version	Status	Prepared by	Comments
Feb 8, 2023	1.0	Graded	Amazonian Prime	
April 3, 2023	2.0	Complete	Amazonian Prime	Revisions for final submission: ● Updated database schema to match our final design database exactly, (added BlockedUsers table, changed camel case to pascal case, added several new attributes to multiple tables) ● Updated ER diagram to include all the changes noted above ● Updated API table to reflect our final APIs used ● Added note to Production and Test Environment section that we were unable to maintain two environments due to budget constraints ● Updated high-level architecture overview (diagram and description) ● Updated the backend AWS service diagram flows in the appendix

Document Control

Role	Name	E-mail	Telephone
Professor	Jerry Jim		
TA	Marie Salomon		
TA	Shijun Shen		
Project Sponsor	Peter Smith		
	Mahmoud Al Khatib	mahmoudalkhatib.ubc@gmail.com	
	Michael He	michaelhe17@gmail.com	
	Joshua Luong	joshualuong@hotmail.com	
	Tristan Martinuson	tmartinuson@gmail.com	
	Elaine Shi	elaineshi328@gmail.com	
	William Suryawidjaja	suryawidjajaw@gmail.com	

Approval

Role	Name	Signature	Sign-off Date
Amazonian Prime Member	Mahmoud Al Khatib	Mahmoud Al Khatib	April 4, 2023
Amazonian Prime Member	Michael He	Michael He	April 4, 2023

Amazonian Prime Member	Joshua Luong	Joshua Luong	April 4, 2023
Amazonian Prime Member	Tristan Martinuson	Tristan Martinuson	April 4, 2023
Amazonian Prime Member	Elaine Shi	Elaine Shi	April 4, 2023
Amazonian Prime Member	William Suryawidjaja	William Suryawidjaja	April 4, 2023

Table of Contents

Introduction

Overview
The AWS Vancouver team is facing a challenge in fostering staff engagement and building stronger relationships among employees, especially with the new norm of remote working. To address this challenge, the team is proposing to develop an AWS-based SaaS solution that allows for virtual and physical connections among staff through an internal commerce marketplace.

Goals
The project aims to deliver a fully functional e-commerce application that is easy to use, secure, and accessible by all employees. The application will include a user profile creation system, a listing and purchasing system, a secure payment system, a shipment status system, and an admin panel for managing listings and users.

The successful completion of this project will result in a platform that facilitates employee interaction and collaboration, leading to improved employee morale and a more positive work environment.

Assumptions of The System

1. We will be using Material UI to develop our Frontend components.
2. We will also assume that the selected AWS services are compatible with one another
3. We will have a maximum user base of 500 and a minimum concurrency of 10 users
4. If the sponsor decides to use our solution, they will implement their own shipping and payment systems. Therefore our mock payment and shipping systems should be clearly isolated from the rest of our project

Programming Environment

Programming Tools and Languages

IDE: Visual Studio Code for IDE (version 1.75)
- Extensions: AWS Toolkit, ES7+ React/Redux/React-Native snippets

BackEnd
Language: Nodejs (version 18.14.0) for Lambda
AWS Services:
- AWS API Gateway
- AWS Step Functions
- AWS Aurora Database MySQL (MySQL engine version 8.0)
- AWS S3
- AWS CloudFormation
- AWS SES
- AWS SNS & SQS

Libraries:
- Google OAuth (google-auth-library version 8.7.0)

Tools/Platforms:
- Docker (version 4.16.3)
- Git (version 2.39.1)

FrontEnd
Language: Typescript (version 4.4.2)
Framework:
- React (version 18.2.0)
- React Dom (version 18.2.0)
- Redux (version 5.0.0)
- SASS (version 1.58.0)

Libraries:
- Material UI Design (version 5.11.8)
- Axios (version 1.3.2)

Databases

We will be using the Amazon Aurora MySQL database which currently supports MySQL Community Edition versions 5.7. The list of databases that we will be making to store our data is detailed in the Data Design Section.

Source Control System

We will be using Git (version 2.39.1) for source control. We will also be using a semantic versioning in our pushes to ensure that we can keep a track of the version of our changes. We will maintain two main branches that will correspond to two different environments; **main** and **dev**.

All of our branches will be merged into **dev** first through a Pull Request. The pull request will need the approval of two group members before being merged to the **dev** branch. Afterwards, it can be merged into **main** after testing on the AWS environments.

UML Tool

We will be using draw.io to generate our UML Diagrams

Production and Test Environment (CI/CD)

Due to budget constraints with AWS-services, we are unable to maintain two environments (production and prod). As a result we are only maintaining a production environment. Our original plan is detailed below:
Our team will focus on using GitHub actions for maintaining production and test environments hosted on AWS services where both environments can be deployed and tested in conjunction with our sponsor services in defined CI/CD pipelines.

Any changes made to our GitHub repository will funnel through our test deployment pipeline in order to identify any bugs and/or issues with features before building a snapshot to our production deployment which will funnel through our other production pipeline.

Our team plans to maintain 2 branches, one used for development in test and the other being the main branch used for production. The test environment will deploy on changes made within this branch and will utilize SQL scripts to populate dummy data within the database.

Well-tested changes will be pushed to a main branch from the dev branch after a successful pipeline build and an approved team review on GitHub. A two-environment setup should ensure a working deployed product at all times after initial deployment and will prevent failures in production.

We will take the following steps to set up and create our GitHub workflow pipelines:

1. Create a YAML workflow within /.github/workflows/{*file_name*}}.yml that will contain build and test jobs.

2. Within this YAML workflow:
    - Set the name of the GitHub Action
    - Define when to trigger the event on
    - Define the jobs to run within the workflow
    - Define our tests to be run on trigger
    - Define the environment to run the workflow on
    - Define the steps taken within the workflow
3. Commit and push this workflow to our project repository
4. Add AWS environment variables to be able to push to our AWS services
5. View GitHub Actions to verify the workflow is running on branch changes
6. Verify that changes are being pushed to our end-services and are deployed with the updated changes.

UI/UX Mockups

Figma Links

The below links represent possible workflows a user can take. These mockups are a work in progress and are likely to change in the future- linked are the first iterations. Note: The “click” and arrows should be followed for a better understanding of the mockups.

Login/ Registration Page:
https://www.figma.com/file/50VTfCdh18muBvy34y8yco/Amazonian-Prime?node-id=59%3A910&t=vnnFMnRIU98tU6VC-0

- This is the basic page for user registration/ login
- During the first time registering, we may ask more questions to properly “register” a user

Buying Page:
https://www.figma.com/file/50VTfCdh18muBvy34y8yco/Amazonian-Prime?node-id=0%3A1&t=vnnFMnRIU98tU6VC-0

- Starting from the landing page, this mockup will guide the user through major buying workflows (i.e. browsing, looking at product details, searching, adding to cart, viewing cart, looking at order history)
- This workflow also includes how to switch to ‘Admin privileges’ if the user is also an admin

Selling Page:
https://www.figma.com/file/50VTfCdh18muBvy34y8yco/Amazonian-Prime?node-id=59%3A911&t=vnnFMnRIU98tU6VC-0

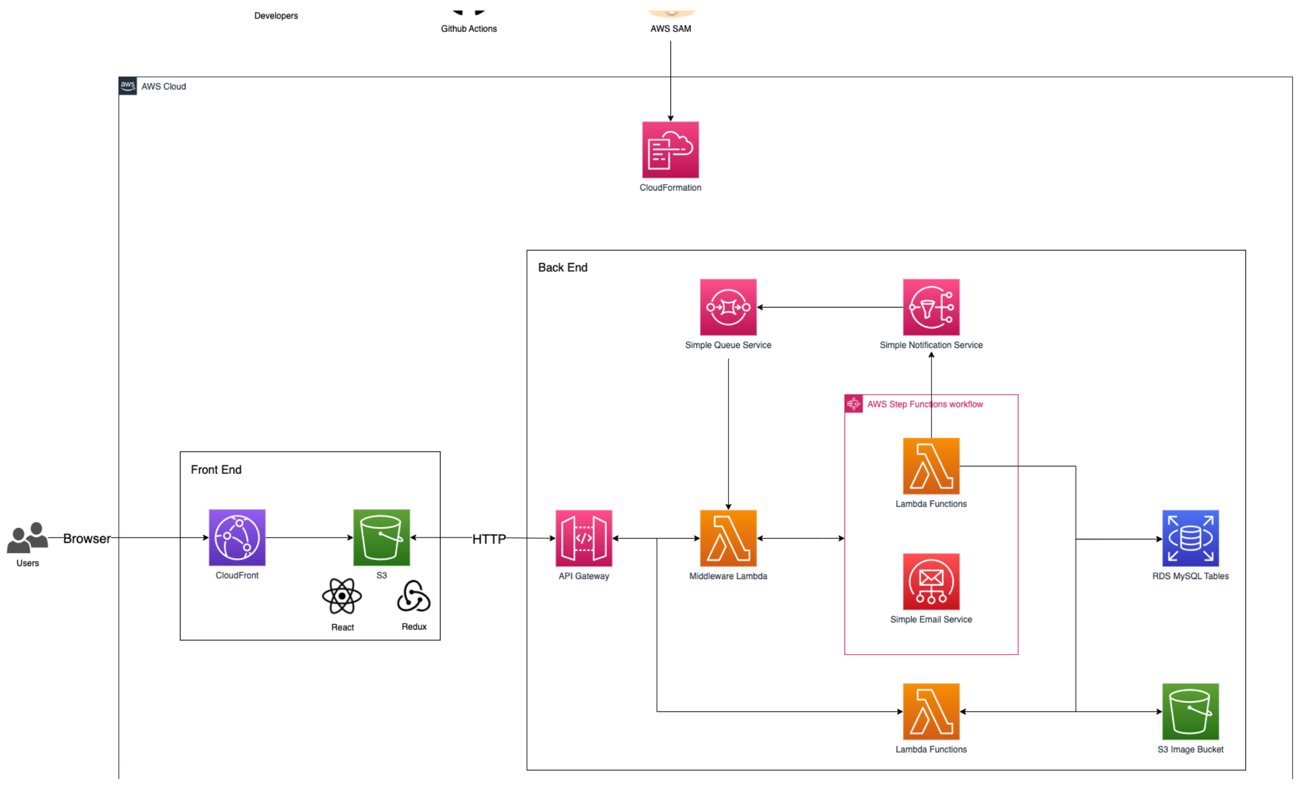
- This workflow begins once the user clicks “Sell”. Depending if the user has sold before, we may ask them for more information
- This includes major selling workflows (e.g. providing details for a listing, listing an item, viewing the listed item, and viewing past listings)

Admin Page:
https://www.figma.com/file/50VTfCdh18muBvy34y8yco/Amazonian-Prime?node-id=59%3A909&t=vnnFMnRIU98tU6VC-0

- This workflow is only available for administrators of the application. This involves major workflows: promoting and demoting users with admin privileges, canceling orders, deleting postings, and also the ability to switch back to normal user privileges (to go back to normal buy and sell page)

Software Architecture

High-Level Architecture Overview



This high-level architecture overview displays how our front end system will interact with our AWS back end.

The users will access our application through the Browser and a CloudFront URL, which will redirect the user to the S3 bucket where our Frontend lives. The front end system will be built on React + Redux, which will use Axios to create API requests to our AWS API Gateway Endpoints.

The API Gateway will service HTTP requests to GET requests will be able to access data from AWS Aurora through a defined Lambda access function.

All of the services will reside within the same VPC in order to allow access between the services. In our AWS Step Functions, we have 3 generic workflows illustrated that will focus on the Buyer, the Seller and the Admin workflow when interacting with the back end. Each Step Function may engage with other AWS services that we have defined such as AWS Aurora, AWS SES and Amazon S3 in various Lambda functions within a workflow.

Backend AWS Service Diagram Flows

The Backend Workflow diagrams are attached in the appendix.

Data Design

Normalized Specifications (to 3NF)

Users(<u>UserID</u>, FirstName, LastName, Email, Department, IsAdmin)

BlockedUsers(<u>BlockingID</u>, Email)

BankingDetails(<u>BankingID</u>, **UserID, AddressID**, InstitutionNum, AccountNum, TransitNum, NameOnCard)

PaymentDetails(<u>PaymentID</u>, **UserID**, **AddressID**, CreditCardNum, ExpiryDate, CVV, CardHolderName)

ShippingAddress (<u>**UserID**</u>, <u>**AddressID**</u>)

Address(<u>AddressID</u>, **CityName**, **Province**, **StreetAddress**)

Country(<u>CityName</u>, <u>Province</u>, <u>StreetAddress</u>, PostalCode, Country)

Listing(<u>ListingID</u>, **UserID**, ListingName, Description, Cost, Quantity, Category, Size, Brand, Colour, ItemCondition, isActiveListing, PostedTimestamp)

- isActiveListing will indicate whether the item has been sold or not. We don't want to delete listings from the database after they've been sold as we need them for order history

ListingImage(<u>PictureID</u>, **ListingID**, S3ImagePath)

- A listing can have multiple pictures associated with it (limit to 5)

Orders(<u>OrderID</u>, **UserID**, **AddressID**, **PaymentID**, ShippingStatus, OrderTimestamp, PurchaseAmount, GSTTax, PSTTax, TotalAmount)

OrderItem(<u>**OrderID**</u>, <u>**ListingID**</u>, OrderQuantity)

ShoppingCartItem (<u>ShoppingCartItemId</u>, **UserID**, **ListingID**, Quantity)

- If user adds the same listing to the shopping cart, then add up the amount to increase the quantity (only if the shoppingCartItem hasn't been purchased)

Legend
Underline - Primary Key
Bold - Foreign Key

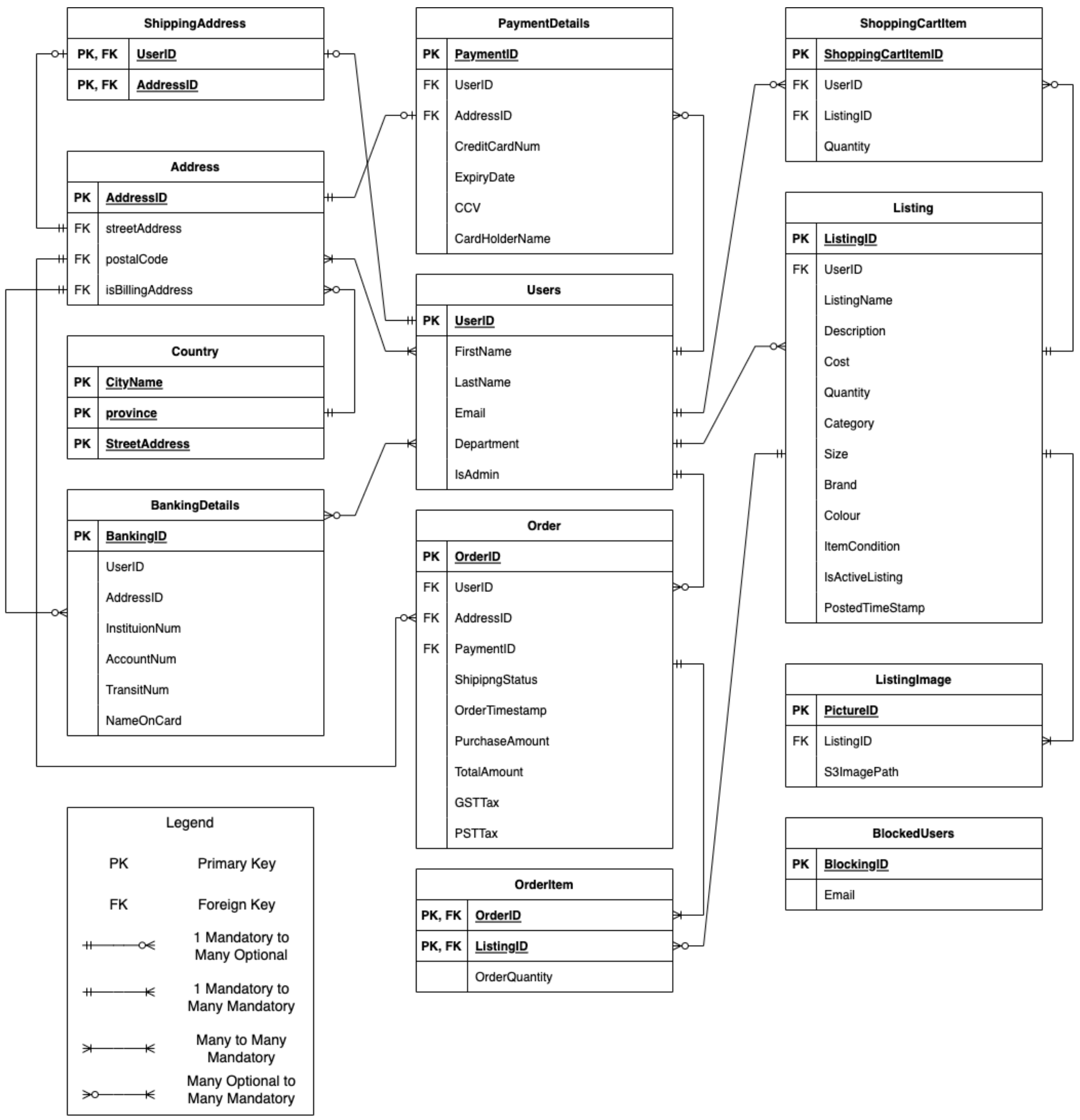
Each of the columns has a single value, which suggests that it satisfies the 1NF state.

All of the tables have a single column as the primary key aside from the BankAccount, PaymentMethod, and ShippingAddressTable, whose only columns are part of the composite key. Therefore, there are no partial dependencies and the tables satisfy the 2NF state.

The functional dependencies in the AddressTable were City, Province, StreetAddress -> Country, Postal Code. To normalize this, we have split the table into a subtable, CountryData.

Finally, there are no functional dependencies between non-key columns of the remaining tables, so the tables are satisfied by 3NF.

ER Diagram



- Users are the Amazon employees that use the application. They encompass buyers, sellers, and administrators. There is a field “isAdmin” that denotes if the user has admin privileges.
  - A user will have at least one shipping address associated with their account. This means that they must fill in this section at registration. They can have multiple shipping addresses.
  - A user can have banking (direct deposit) information associated with their account. They will fill in this information when accessing the selling workflows for the first time. They can have multiple bank accounts linked.
  - A user can have multiple payment methods (debit/credit cards) associated with their account. They will be prompted when registering for the buying workflows for the first time but can fill out this information at checkout.
  - A user can have multiple items in their shopping cart, or none.

    - A user can have multiple item listings or none.
    - A user can have multiple orders in their order history, or none.
- BlockedUsers are users who have had their accounts deactivated and deleted by an administrator. They can no longer access the website as their emails are checked when registering through google auth.
- Address contains all the necessary info for both shipping and billing addresses. There is a field “isBillingAddress” and “isShippingAddress” that indicates which one they are, or if they are both.
    - An address is associated with at least one user. An address might be associated with multiple users, for example, if many Amazon employees use the same office as their address.
    - An address can be associated with a PaymentDetails, if it is a billing address
    - An address can be associated with one or more orders, if it is a shipping address
    - An address has exactly one country. The country is determined by the province and city of the address.
- Country is the country that is associated with a city or province. This distinction is necessary for the third normal form (no transitive dependencies)
    - A country can be associated with multiple addresses, or none
- BankingDetails contains all the necessary details for a direct deposit when a user has sold an item on the marketplace
    - Banking details are associated with exactly one user
- PaymentDetails contains credit/debit card information such as number, CVV, and billing address
    - PaymentDetails must be associated with exactly one address
    - PaymentDetails are associated with exactly one user
- ShoppingCartItem is an item that has been placed in a user’s shopping cart. These entries are removed when a user completes are purchase or if the user removes the item from their shopping cart
    - A shopping cart item is associated with exactly one user (the user whose shopping cart it belongs to)
    - A shopping cart item is associated with exactly one listing (the item listing which the user has placed into their shopping cart)
- Listing is an item listing which a user (seller) has placed for sale.
    - A listing is associated with exactly one user (the seller who posted the listing)
    - A listing can be associated with multiple shopping cart items or none. For example, a listing might be in multiple users’ shopping carts at the same time.
    - A listing has at least one image of the item, but can have more
- ListingImage is an image of an item listing, saved as an image path
    - A listing image is associated with exactly one listing
- Order is the order details of a user’s past order. It includes all items that the user checked out in their shopping cart, the date they were purchased, as well as the shipping address associated with the order.
    - An order is associated with exactly one shipping address
    - An order is associated with exactly one user
    - An order is associated with one or more OrderItems
- OrderItem is an item that was purchased by a user
    - An order item is associated with exactly one order
    - An order item is associated with exactly one listing (the listing that was purchased)

API Design

Endpoints

API Endpoint	Method	Description	Response Codes
/admin/users?offset=x&limit=y&name?=name	GET	Get all users for with pagination limits	200 - Admin receives paginated results of users and their column values passed as a list 401 - Unauthorized
/admin/users/:userid	DELETE	Deactivates/ deletes a user by userID	200 - User was successfully deleted. Returns id of the deleted user 401 - Unauthorized 404 - User with provided userID not found.
/admin/orders?offset=x&limit=y	GET	Get the list of orders with pagination limits	200 - Successfully returns list of orders within pagination limit 401 - Unauthorized 404 - Orders within pagination limits not found
/admin/users/privilege/{UserID}	PUT	Modifies Admin Privileges for a specified User in the path parameters. AdminID and IsAdmin passed in the request body.	200 - User privileges were modified. 401 - Unauthorized 404 - User with provided userID not found.
/listings?name=value&category=value&offset=x&limit=y	GET	Gets the list of listings with pagination limits and optional filters (name and category)	200 - Returns a list of listings in the given range 404 - Resource with the given limits not found
/listing	POST	Creates a product listing using seller supplied details contained within the request body	200 - Returns the posting id of the product listing 400 - Bad request 401 - Unauthorized
/listings/:listingid	GET	Get the listing details given the listing ID	200 - Returns the listing with details 404 - Resource with the id not found 401 - Unauthorized

/listings/delete/:listingid	**DELETE**	**Delete the listing given the listing ID**	**200 - Listing was successfully deleted. Returns id of the deleted listing 401 - Unauthorized 404 - Listing with provided id not found.**
/users/:token	**GET**	**Get the specified user given their token.**	**200 - Returns the user details 401 - Unauthorized 404 - user with provided id not found.**
/user	**POST**	**Create a new user given user information in the request body**	**200 - Returns the user id 400 - Bad request 401 - Unauthorized**
/users?name=value&offset=x&limit=y	**GET**	**Gets the list of users given optional name with pagination limits**	**200 - Returns the users details within pagination limit 404 - Resource with the given limits not found 401 - Unauthorized**
user/shipping	**POST**	**Adds a new shipping address to the User**	**200 - Returns the shipping address added 400 - Bad request**
/orders/:userId?offset=x&limit=y	**GET**	**Get the list of orders from a given user within the pagination limit**	**200 - Returns the orders details 404 - Resource with the given user id not found 401 - Unauthorized**
/orders/:userId	**POST**	**Create an order for a user given information in the request body (e.g. listing details, price, etc.)**	**200 - Created an order and returns the order id 400 - Bad request 404 - Resource with the given user id not found 401 - Unauthorized**
/orders/:userId	**PUT**	**Modify an order for a user given information in the request body (e.g. shipping status)**	**200 - modified an order and returns the order id 400 - Bad request 404 - Resource with the given user id not found 401 - Unauthorized**

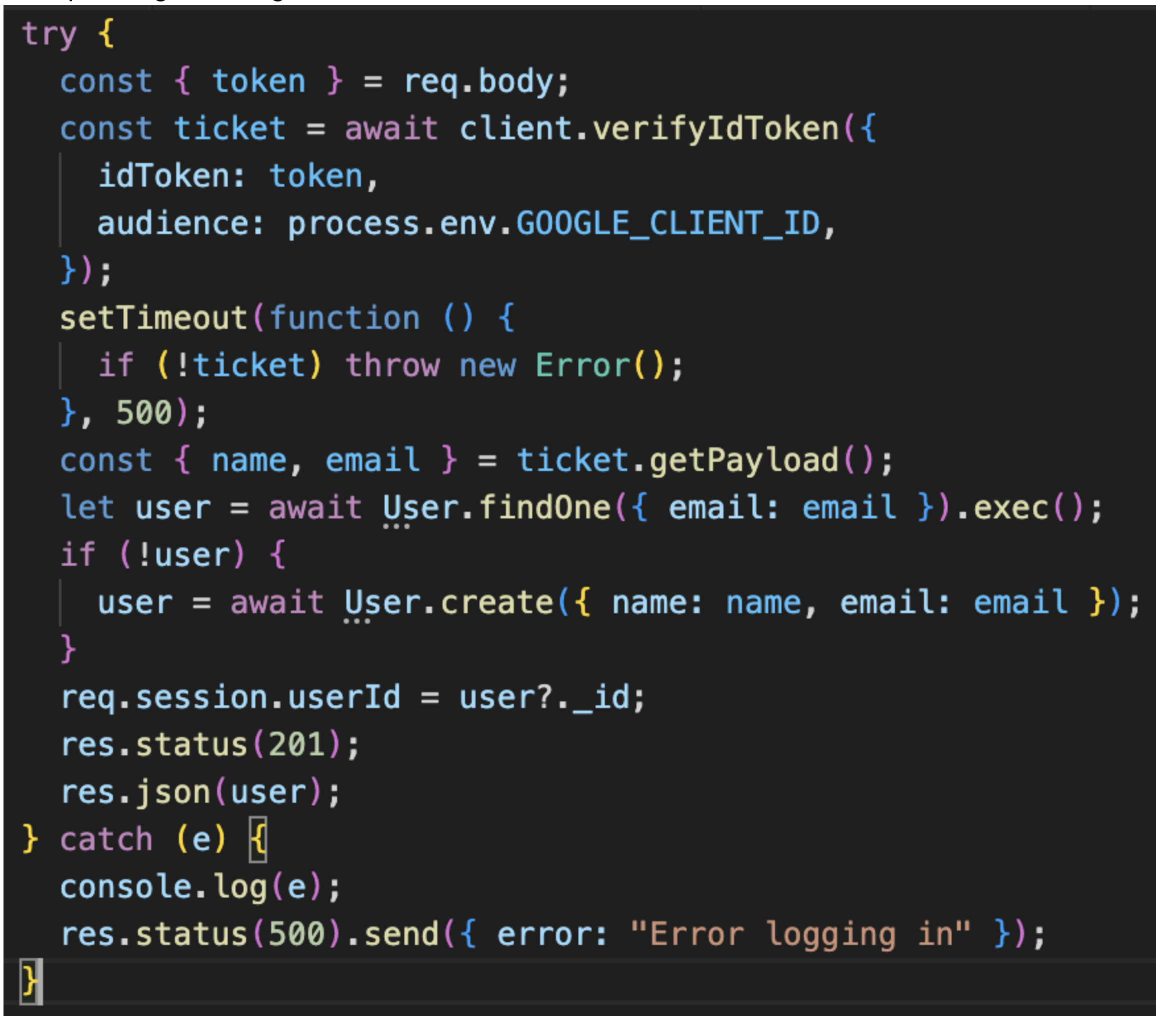
/user/shopping-cart/:userid	GET	Returns the shopping cart items for the user	200 - Returns the shopping cart items 400 - Bad request 404 - Resource with the given user id not found 401 - Unauthorized
/user/shopping-cart/:userid	PUT	Modify the shopping cart to add a listing to the cart or update an existing item to the cart.	200 - Returns the listing just added 400 - Bad request 404 - Resource with the given user id or listing id not found 401 - Unauthorized
/user/shopping-cart/:userid	DELETE	Modify the shopping cart to remove a listing from the cart	200 - Returns the listing just deleted 400 - Bad request 404 - Resource with the given user id or listing id not found 401 - Unauthorized
/checkout/:userid	POST	Create a payment request to checkout items within the shopping cart	204 - Success No Content 400 - Bad request 404 - Resource with the given user id not found 401 - Unauthorized
checkout/retry	POST	Retries processing the payment if the payment has failed before	200 - Success
/payment	GET	the payment details of a particular PaymentID. If the payment is valid we generate a random number [0,100]	200 - Success returns payment status
/shipping/id	GET	Gets the shipping information fro the provided id	200 - Success
/user/banking	PUT	updates the banking details of a user	200 - Success 400 - Bad request
/user/shipping/{id}	GET	retrieves the shipping address associated with a user ID	200 - Success
/user/banking/{id}	GET	retrieves banking details associated with a specific user ID	200 - Success
/blockedUser orders/create/{UserId}	GET	retrieves all Blocked Users	200 - Success
	POST	creates an order in the database	200 -Success 400 - Bad request

/listings/test	GET	retrieves shopping cart information from a database, calculates the total cost including taxes	200 - Success
/user/payment/{id}	GET	retrieves payment details for a given user ID	200 - Success
/user/payment	POST	Creates a payment done by a user	400 - Bad Request 200 - success payment
/user/address	POST	adds a new address	400 - Bad Request 200 - Success
/user/banking	POST	adds banking details	200 - Success 400 - Bad Request
listing/update	PUT	updates a product listing	200 - Success 400 - Bad Request

Algorithms

We will be using GoogleOAuth for the authentication (which will be handled by Google OAuth packages) and we will be using simple algorithms to manipulate our databases using Lambda functions. Hence, we will not use any complex algorithms.

Example usage of GoogleOAuth:



```
try {
  const { token } = req.body;
  const ticket = await client.verifyIdToken({
    idToken: token,
    audience: process.env.GOOGLE_CLIENT_ID,
  });
  setTimeout(function () {
    if (!ticket) throw new Error();
  }, 500);
  const { name, email } = ticket.getPayload();
  let user = await User.findOne({ email: email }).exec();
  if (!user) {
    user = await User.create({ name: name, email: email });
  }
  req.session.userId = user?._id;
  res.status(201);
  res.json(user);
} catch (e) {
  console.log(e);
  res.status(500).send({ error: "Error logging in" });
}
```

Notable Tradeoffs and Risks

Tradeoffs

Development on a production and test environment vs single deployment environment

- We chose to develop on two environments for the following reasons:
  1. Production environment can always maintain a working model of our product while the new features and bug fixes can be maintained in the development/test environment.
  2. We can test safely without crashing our workable model
  3. Proper code vetting and reviews can be maintained when establishing two workable branches
- Some of the tradeoffs for choosing to maintain two environments are:
  1. It is financially expensive to maintain two deployment environments given they can exhaust more resources than solely maintaining one.
  2. There is more overhead upfront in setting up a two-environment system on GitHub and AWS
  3. It is time-consuming to maintain releases between two environments with code reviews, working branches, pipeline building and maintenance.
- Despite the tradeoffs, we have decided to pursue the production and test environment approach for its safety and organization in deliverables.

Authentication: Google OAuth vs Firebase vs Amazon Amplify Auth

- We chose Google OAuth for the following reasons:
  1. Users are most likely going to already own a Google account, therefore they will be familiar with the login method and don't need to manage an additional password.
  2. Google API library ensures account safety, fast authentication and smooth integration with Node.js
  3. Easy to use, we only need to use the library functions, and not develop the underlying system and difficult algorithms.
- Some tradeoffs for choosing Google OAuth over options such as Amazon Amplify Auth or Firebase:
  1. Firebase and Amazon Amplify Auth may come with more features that could be used to improve the product
  2. If we choose otherwise, we have more freedom in designing the authentication system
  3. By using Google, users are forced to use their personal Google account
- Despite the tradeoffs, we chose Google OAuth for its simplicity and security.

Backend Programming Language: NodeJS vs Go

- We chose to develop with Node.js for several reasons:
  1. Node.js is built on Chrome's V8 JavaScript engine and it is suitable to build fast and scalable network applications
  2. Node.js excels in the instance of creating real-time systems and web applications
  3. Node.js has a large community with many available libraries for added support
  4. Our developers have more experience with Node.js and JavaScript
- Some tradeoffs for choosing NodeJS over Go:
  1. Go is good for building high-performance, concurrent systems
  2. It excels at building web servers and other network applications
  3. Go makes it easy to catch errors during development since it is statically typed
- Despite some of these tradeoffs, we believe that NodeJS would be ideal for our use, as our goal is to build a fast web application that might benefit from several existing libraries.

Database Selection: MySQL vs PostgreSQL

- We chose to develop with MySQL over PostgreSQL for several reasons:
  1. **Structure:** Our current Data Base design follows a traditional relational database design. Both MySQL and PostgreSQL are compatible with our data design. PostgreSQL supports a wider range of data types such as arrays, store (key/value store), and JSONB, but since we do not see a significant need for those data types according to the design, MySQL will accommodate our needs.
  2. **Performance:** MySQL's storage-engine framework supports demanding applications with high-speed partial indexes, full-text indexes and unique memory caches, which makes

MySQL's performance better. MySQL compatibility layer in Amazon Aurora can provide faster performance for simple, **read-intensive** workloads due to the lighter-weight architecture and focus on fast query execution. In our solution, most of the operations will be read operations, so it would be more beneficial to opt for MySQL.

3. **Scalability**: Our database will be managed by AWS Engines, and will automatically scale.
4. **Community Support**: Both MySQL and PostgreSQL have large and active communities, but MySQL has a larger and more dedicated community of developers and users.
5. **Prior Experience:** Our backend team is mostly experienced with MySQL vs PostgreSQL which decreases the overhead to get used to PostgreSQL.

Frontend Frameworks: React vs Vue

- We chose to develop with React for several reasons:
    1. React avoids the traditional DOM rendering and only updates certain objects within the DOM, if it is changed in the virtual DOM
    2. React has a large community support with numerous available packages/ libraries that may be beneficial to include
    3. React doesn't create a separate HTML page for each route
    4. Availability of libraries (e.g. MaterialUI)
- Some tradeoffs for choosing React over Vue:
    1. Vue has an elegant language syntax
    2. Vue is customizable and scalable between the library and framework
    3. One member of our team has experience with Vue
- Ultimately, we chose to work with React due to its large community and support (versus Vue's).

Frontend Language: TypeScript vs JavaScript

- We chose to develop with Typescript for several reasons:
    5. It is easily maintainable, as the typing system allows the developer to know exactly what certain objects are
    6. This offers code navigation
    7. Supports interfaces, classes, etc.
- Some tradeoffs for choosing Typescript over JavaScript:
    1. JavaScript is a cross-platform language (and what we use in the backend)
    4. It is easy to learn and to start with
    5. Ultimately, JavaScript has more freedom (with no typing), which might save developer time
- Despite the tradeoffs, we chose TypeScript for better maintainability and readability in the long run.

Risks

Risk	Impact to System	Solution
Exceeding free tier AWS service limits	May force a change in design and usage of AWS and cause delays	Careful planning of usage; constantly monitoring our usage
Members do not become proficient in AWS within due date	Increased chance of producing bugs and poor design	Have members peer review each other's code to reduce bad code; assign some members to dive deeper into AWS tutorials and fundamentals
Switching programming tools	May run into integration issues with the rest of the system	Research well before implementing the new tool; keep documentation constantly updated and clean
Unable to identify all the required functionalities on time	May cause the need to massively rework parts of the system	Take notes during TA and sponsor meetings; double check any ambiguity with the sponsor; prepare and demo as much of our design as possible for the sponsor to review

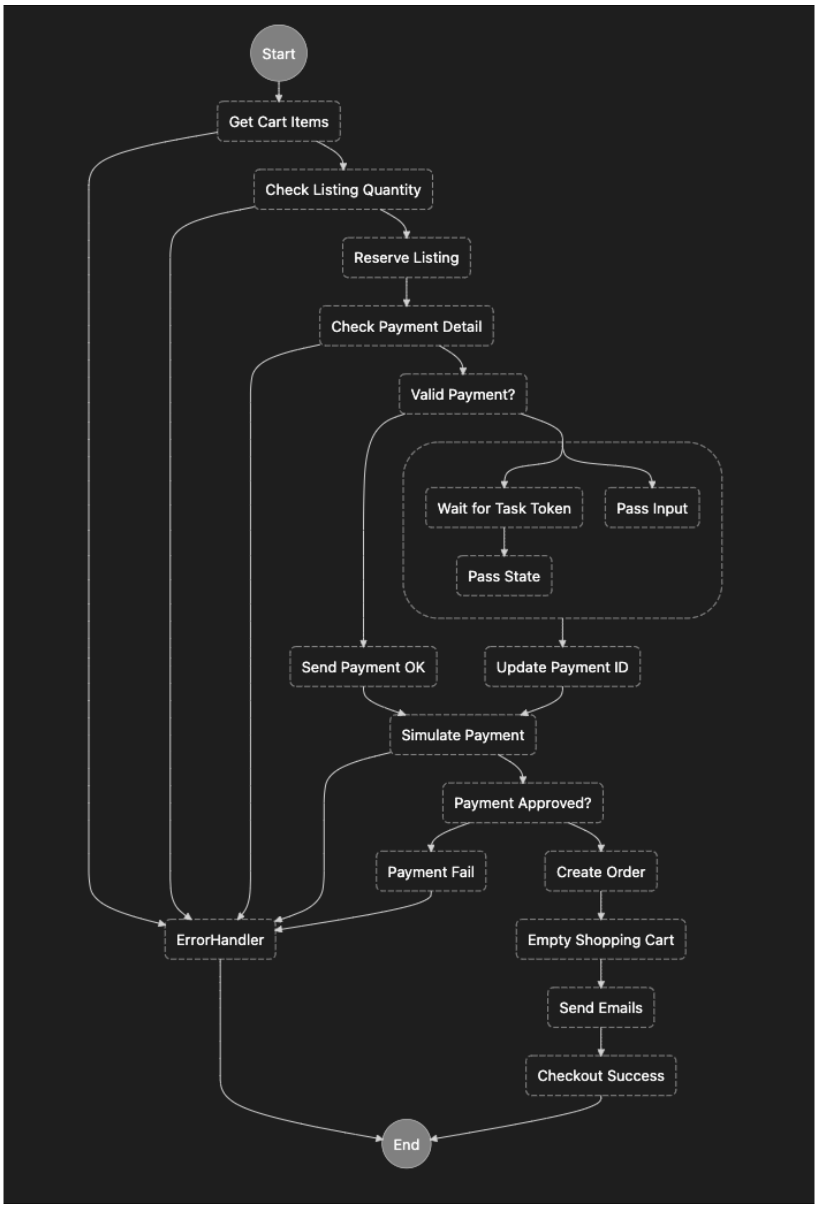
Appendix

Glossary of Terms / Abbreviations

Terms / Abbreviations	Description
SaaS	Software as a Service, a method of software delivery and licensing in which software is accessed online via a subscription
React/Redux	React is a front-end JavaScript library for building user interfaces managed and based on modular components. Redux is a JavaScript library for managing and centralizing application state used in conjunction with React.
Lambda	Serverless compute service that allows code to be hosted and run without provisioning or managing servers. It automatically scales and monitors with the application.
Step Function	Service that enables coordination and initiation of distributed workflows and applications through the use of visual workflow diagrams
Aurora RDS	Relational database service provided by Amazon Web Services (AWS) designed to provide high availability, performance and scalability for cloud-based applications
Google OAuth	Provides user authentication with a service without the need of maintaining user private information such as usernames, passwords and other sensitive details.
SCRAM	Salted Challenge Response Authentication Mechanism (SCRAM) is a family of modern, password-based challenge-response authentication mechanisms providing authentication of a user to a server
Amazon S3	Amazon S3 or Amazon Simple Storage Service is a service offered by Amazon Web Services that provides object storage through a web service interface

Backend AWS Service Diagram Flows

Checkout Workflow



The **checkout** workflow for this project will be handled using a step function workflow that involves various Lambdas and services such as SQS and SNS. Firstly for the input, we will get the user's ID, paymentID and shippingID. Using the UserID, we can retrieve the user's shopping cart.

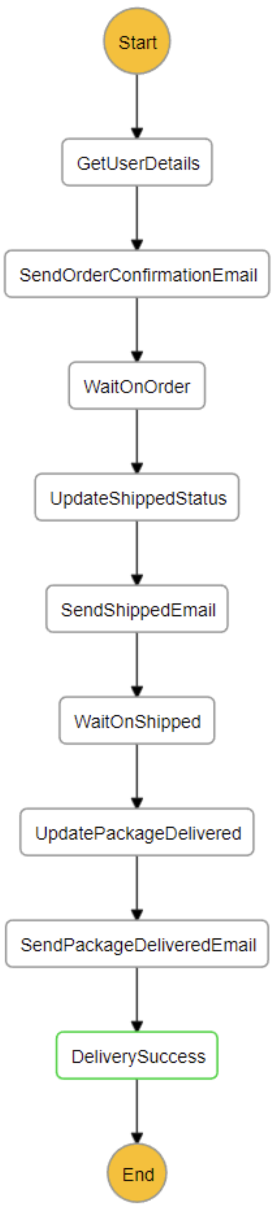
With the shopping cart, we can then check if the user has some items in the shopping cart and if they do, we check whether the quantity that they would like to purchase is lower or equal to than the amount available in the listing. If any of the conditions aren't satisfied, then the flow will terminate.

If both requirements are satisfied, then the checkout workflow will reserve the items by updating the amounts in the respective listings. Note that any errors thrown from here on will result in the items being unreserved when the error is handled.

We will check if the user's payment methods are valid, and if they are, they will proceed in the flow by creating an order entry in the database. However, if the payment method isn't valid, then the workflow stops for 5 minutes whilst we provide some time for the user to adjust their payment method. They are able to do so using the waitForTaskToken feature of Step Functions, which will let the step function be paused for as long as we want. However, we will specify that once the 5 minute grace period has passed, it will be handled as an error and the items will be unreserved.

If a new payment method is provided, then we will perform a second check to ensure that the payment method is valid and if so, proceed with creating an order entry. Once the order entry has been created, we will start another step function workflow that will update the shipping status and send emails to the buyer confirming that their purchase has been made.

Email Workflow



The email workflow for this project is a Step Function that handles a mock shipping status update with various timeouts added at steps labeled with “Wait”. The start of the workflow retrieves the user order details such as the items purchased and the user specific information such as their email using their UserID and OrderID. Upon retrieving this information in the database, the next step at SendOrderConfirmationEmail sends an email to the user notifying them their order has been confirmed with relevant details and is currently being processed by Amazonian Prime. The next step WaitOnOrder contains a timeout of thirty seconds to simulate the time delay of moving from processing to shipped. UpdateShippedStatus engages with the database to update the status of the order with “Shipped’ using the OrderID to signify that the package has now been shipped. SendShippedEmail then notifies the user of this change in order status by adding more relevant details such as mock shipping companies and tracking id. WaitOnShipped contains another timeout, this time longer in the span of five minutes to simulate the delay of transporting a package to a user. UpdatePackageDelivered now updates the status of the order with “Delivered” using the OrderID to signify the terminal status of a transaction. SendPackageDeliveredEmail then notifies the user of this status update regarding their package and finally the terminal state is DeliverySuccess in which the Step Function completes the email workflow. This function is dispatched asynchronously at the end of the Order Workflow which does not wait for a return on this separate Step Function.