FINAL REPORT

Team Number	34
Team Email	may1734@iastate.edu
Team Website	may1734.sd.ece.iastate.edu
Bill Adamowski	Client & Advisor
Lucas Collins	Communication Lead
Jonny Krysh	Webmaster
Jake Long	Technical Lead
Garret Meier	Team Lead
Derek Yu	Key Concept Holder

Last Revised: April 23, 2017

Table of Contents

1 Introduction

1.1 Project statement

Our aim with Expresso is to provide services to local coffee roasters for expanding and optimizing their business by creating an efficient, automated platform for coffee roasters to promote and sell their unique coffee brands to a broad range of customers.

1.2 Problem Statement

Small, local coffee shops struggle to make decent profits on roasting and distributing their beans. The low margins in the coffee industry are due to many factors like price competition with international chains (Starbucks, Dunkin, etc.), high cost of beans or roasting, and inventory optimization. Conversely, local coffee businesses often impact coffee growers most, paying them more reasonably than large chains, as well as influencing communities by providing a unique atmosphere for work and leisure. Creating a system similar to Expresso is highly cost prohibitive for individual roasters. Adding in the cost of building and maintaining an online customer base, only a few high-profile shops have entered the market (Counter Culture, Blue Bottle). By removing the necessity of shops to create their own e-commerce platform, we hope to provide the opportunity for local coffee shops to expand their businesses with minimal effort and cost. Coffee shops currently roast their beans in house to save money and create a unique tasting blend, but expanding that business takes significant technical and marketing effort. As a result, most local roasters don't sell their coffee outside of their brick and mortar shop.

1.3 Goals

From a consumer perspective, our service will help people discover their favorite coffee brands and pay to get them shipped to their door. We want to provide a convenient, reliable, and easy way for people to have just the right amount of coffee that fits their taste delivered to their door. We plan to do this by creating a fully automated order creation and delivery service for roasters, as well as a platform for customers to discover, purchase, and review a variety of coffee roasted by small shops around the country.

1.4 Existing Solutions

1.4.1 Bean Box

This coffee distributor, found at https://beanbox.co/, works with 21 of Seattle's roasters to distribute their coffee in consolidated boxes to Consumers monthly. This service uses the centralized shipping method for coffee, and offers subscriptions of one, 12oz bag for $20 monthly. They tout fresh (within 48 hours) roasts delivered with free shipping around the US. Bean Box has high marks in high quality roasts, but lacks in customizability and cost effectiveness. Consumers are tied to the roasting schedule and specific amounts monthly.

1.4.2 Blue Bottle Coffee

Formerly Tonx, Blue Bottle, at https://bluebottlecoffee.com, was one of the early entries into subscription-based coffee services. They run their entire vertical from sourcing beans through roasting and distribution. Blue Bottle gives among the highest quality beans and mid-level prices, but lacks in variety and customizability. All Consumers receive the same bags of coffee each month, and other options are few and far between.

1.4.3 Counter Culture Coffee

Counter Culture is one of the more fleshed out coffee subscription brands with a complete vertical from purchasing beans to shipping the roasted coffees. Found at https://counterculturecoffee.com/, they offer many, customizable roasts and subscription offerings. Their coffee isn't branded as high class as Blue Bottle or Ritual, but their technical features are quite sound. The primary drawback to the service is that it's central to Counter Culture roasts rather than offering a solution to local roasters.

1.4.4 Ritual Coffee Roasters

Ritual Coffee, a San Francisco based coffee roaster, found at https://www.ritualroasters.com, functions similarly to Blue Bottle, but uses an external service called Shopify (http://ritual.myshopify.com/) to handle their eCommerce functionality. Ritual has a few rotating coffees, priced higher than almost all other options. They offer reasonable customizability, but lack the personalization of Counter Culture of Blue Bottle, who host their own services. Their approach of using Shopify to host their eCommerce functionality, is one that undoubtedly has a high barrier of entry for Providers, but is worth looking into as we continue our implementation of Expresso.

1.5 Definitions

Term	Definition
Consumer or Customer	The external client who is viewing, purchasing, and receiving shipments; the coffee consumer.
Provider or Roaster	An external entity which holds a type and amount of coffee; the coffee roaster.
User	Either a Consumer or Provider with no necessary distinction in Expresso.

Figure 1

2 Deliverables

2.2 Overall Project Deliverables

- Consumer can order and receive periodic coffee shipments
- Providers can add to, view, and edit an inventory of available goods
- Providers can view orders that need to be fulfilled
- Providers can receive consumer shipment information to fulfill orders
- Consumers can be alerted to incoming and delivered shipments
- Consumers are billed for purchased goods
- Providers are paid for fulfilled shipments

2.3 Possible Solutions

2.3.1 Consumer and Provider Interface Solutions

- Mobile Application
- Desktop Application
- Responsive Website

2.3.2 Back End System Solutions

- Microservice architecture
- Monolithic server architecture

2.3.3 Shipment Method Solutions

- Centralized distribution center containing all goods
- Decentralized shipments sent to Provider

3 Design

3.1 System specifications

Detail any specifications given and/or assumed about the project.

3.1.1 Non-functional

- All parts of the system should be secure and protected against common attacks like:
    - XSS
    - SQL Injection
- Code should be easy to maintain and understand
- User interface should be easy to use
- Secure login and authentication

3.1.2 Functional

Providers shall:

- Post their different coffee bean types and prices
- View orders placed for their coffee beans
- Receive payments for orders
- Browse available coffee beans from various shop owners, and be able to place orders on these beans

Customers shall:

- Subscribe to periodical deliveries of recommended bean
- Send payments for orders they place
- Set preferences for their preferred coffee types
- View orders they have placed and track their orders

Both shall:

- Log in using an email and password—without both of these the user should not be allowed access to the service

3.2 Design Overview

Our solution for building Expresso involves a microservice architecture with servers interfacing through REST API's. Each microservice separates its concerns from the others and can scale independently. We chose this architecture since it is common

industry practice for web applications, and it offers benefits to our development and implementation cycle.

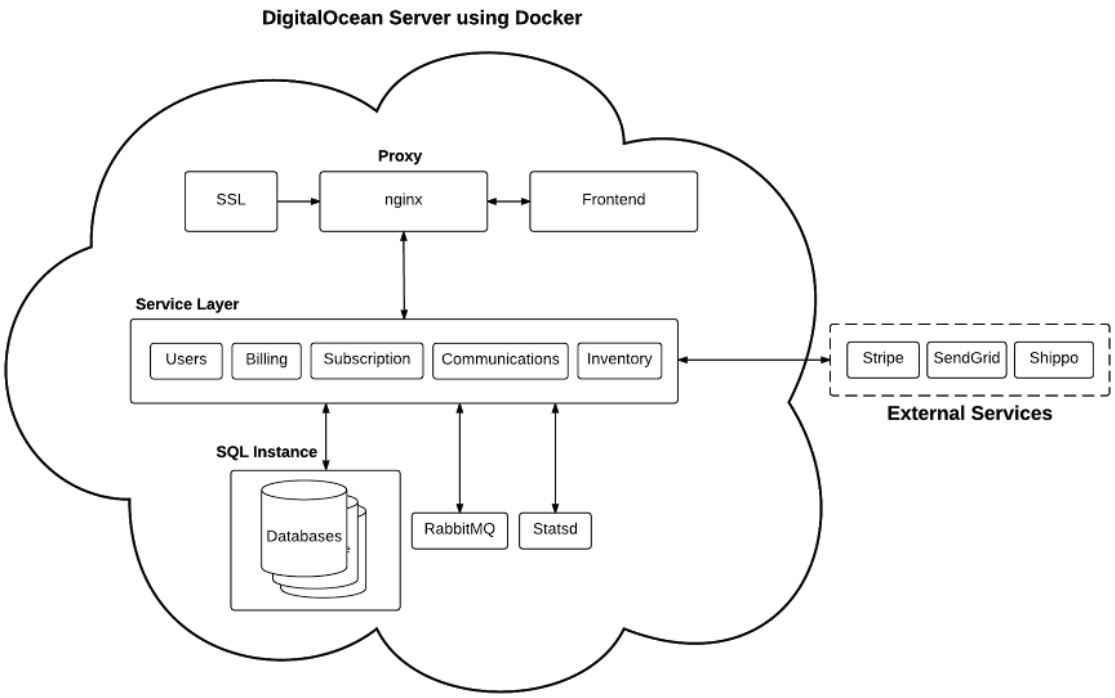

FIGURE 2

In the following sections we discuss the high level overview of each service, their design considerations, and the major data objects each service will be dealing with.

3.2.1 User Service

The user service contains all methods related to user management. This includes registering, updating, listing, getting users, and resetting user passwords.

Design Considerations

This service requires speed so that users can quickly create their account and get started using our service. Because of this we will rely on raw API calls rather than putting all of the requests in a queue to be processed one by one.

Consumer

This class stores information about a customer. The other services contain foreign keys referencing consumers in this service.

Provider
This class stores information about a provider. The other services contain foreign keys referencing providers in this service.

3.2.2 Billing Service

The billing service handles all payments and billing, including recurring payments. This service integrates with other Expresso services to gather relevant consumer and roaster information for payments as well as connecting items, subscriptions and invoices. Much of this service's usefulness comes in directly integrating with Stripe, a billing Service, for securely and consistently handling payment information. Rather than spreading Stripe integration across all our services, it connects our internal data models to their corresponding Stripe entities.

Design Considerations
All of the Stripe integration must be handled in the background to create a seamless user experience for both consumers and roaster, which constrains the billing service to only using programmatic methods for gathering user information. Because of this, the billing service creates an easy interface for other services to manage user information without coupling it to any Stripe credentials, tokens, or keys.

To adequately wrap the implementation details of the Stripe api in an accessible interface for Expresso's services, the billing service must maintain relations between many Stripe entities and their counterparts in our data model (Figure 3). In addition to storing relations, the billing service implements Stripe's beta Connect API to manage dynamically created accounts and transfer funds through invoices from users to roasters. These managed accounts hold plans and subscriptions for each roaster that create our recurring billing systems. However, in our design, we noted that one item in Expresso creates many plans for a managed account based on the offered subscription periods. So we must maintain the relationship in the Billing service.

Additionally, we chose to tie subscriptions to specific item types meaning that users can have only one subscription per item, but might have multiple across one or many roasters. This means that we have to uniquely identify payment information for customers through our Stripe account to the managed roaster accounts for each subscription.

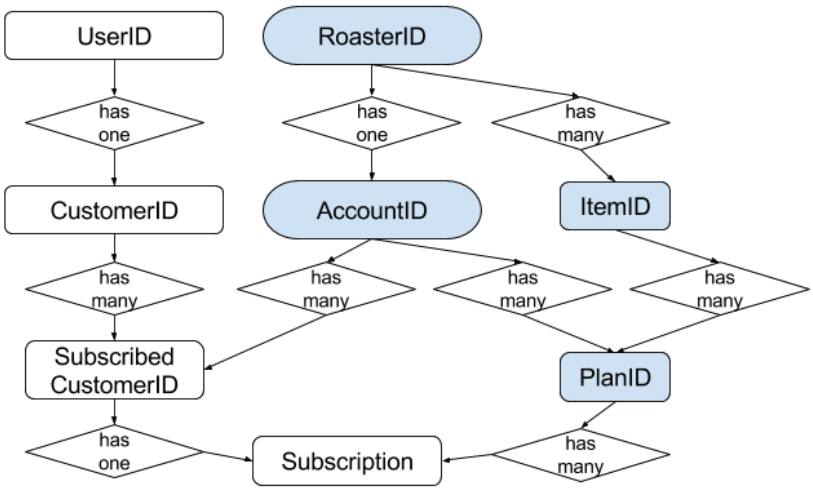

Figure 3

All of this complexity is hidden by the billing service which allows roasters, subscriptions and customers to be accessed and modified as single entities with their unique id's in Expresso's database schema rather than requiring knowledge of Stripe's inner workings.

Because our service is largely subscription-based, we automated usage of Stripe's API using Webhooks, that trigger actions on subscriptions only after a successful invoice on the account. A new invoice triggers activity throughout all the services and results in a new order being created along with email alerts. By responding to webhooks we can efficiently create new items without periodic polling or inaccurate cron jobs.

Customer Account
Stores a relation between User ID and Stripe Customer ID. Additionally, the Customer Id can relate to many subscribed Customer Ids for each customer subscription.

Roaster Account
Stores the relation between Roaster ID and Stripe Managed Account ID, as well as tokens for accessing and modifying Managed Account information through the roaster.

Plan
Stores the relation between a roaster's items and the multiple plans stored in Stripe under the Managed Account. Used to connect subscriptions to items to the desired plan.

3.2.3 Inventory Service

The inventory service will handle the inventories and automatically generated orders of each coffee provider. Each provider will be able to add to and update their inventory. The frontend view will reference the inventories of all providers in order to not sell product that is out of stock.

Design Considerations

We wanted the minimum viable product (MVP) to allow basic local inventory functions. More specifically, we ask that the coffee providers simply keep their individual inventory up-to-date with the latest inventory count. For the MVP, we combined the items, orders, and shipping details associated with orders into this one inventory service. This was done such that when orders are fulfilled with a particular item, a decremented number of that particular item can be updated easily to the database. In a future version, perhaps orders and shipping could be better abstracted out to be their own services. Figure 4 shows the basic diagram of the Inventory service. A Roaster has one inventory which has zero or more impending orders and zero or more items.

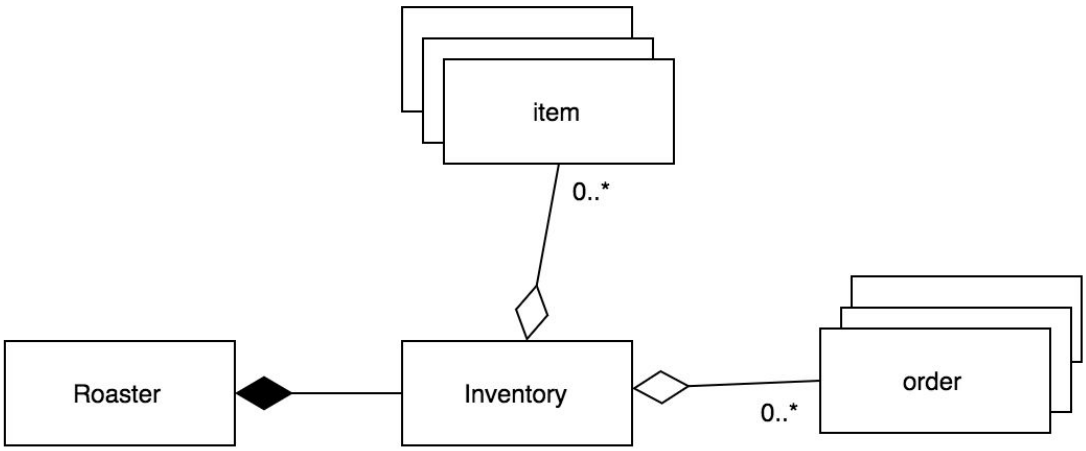

Figure 4

Order

Stores an item identifier, quantity information, customer id, shipping lablel, and a ship date to be used by the provider to ship the specified items.

Item

Represents a real bag of coffee beans to be sold by the provider. Stores the name, coffee type, description, picture, tags, in stock quantity, and other information about the item.

3.2.4 Subscription Service

The subscription service allows users to manage their coffee subscription(s). This service stores subscription data associated with the user and interfaces this data with the inventory and billing service.

Design Considerations:

A key issue was how the subscription service will handle multiple subscriptions per consumer. Currently, the subscription service follows a one to one relationship. Each user is able to create a single subscription per bean item. Theoretically, a user can create a subscription for each bean item listed. Because of how intertwined subscriptions are with the Stripe Billing service and orders within the Inventory service, specific considerations are need. First, a customer account must exist within the Billing service before subscription creation. Secondly, customers cannot make multiple subscriptions of the same bean item. Lastly, the trigger for orders must flow through Billing, Subscription, and Inventory services in order to gather the correct information for the periodic shipments.

Subscription

Stores subscription information for the specific user. Includes coffee information: what bean item, which roaster, and quantity of beans. Also includes the current status of the subscription and date for the next shipment, in order to trigger new orders within the Inventory service.

3.2.5 Communication Service

For the communication service, we centralize all methods of communicating externally to users in one service. Since Expresso relies on asynchronous distribution and shipping, we need to push communications to users in ways which will get their attention rather than requiring constant monitoring of our application. The best way to accomplish this is through email events which are triggered based on application state. The communication service, Bloodlines, is designed with that end goal in mind.

Design Considerations

In this service, we value accuracy and reliability over speed. Taking that in mind, we can rely on queuing and events rather than raw API calls. A critical feature of this service is being able to respond to user interactions easily. We want to send an email each time a person purchases an item or has a shipment inbound. By responding to events on webhook triggers (Figure 5) rather than always listening for send request, we can decouple queuing message with actually sending them. Since communication might be sprinkled through other services, Bloodlines has to quickly respond when enqueuing

messages which would drain server resources if it waited for the message to be sent. We want to avoid a situation where a change to Bloodlines' code requires changes in the way we dispatch events from other services which could trigger communications.

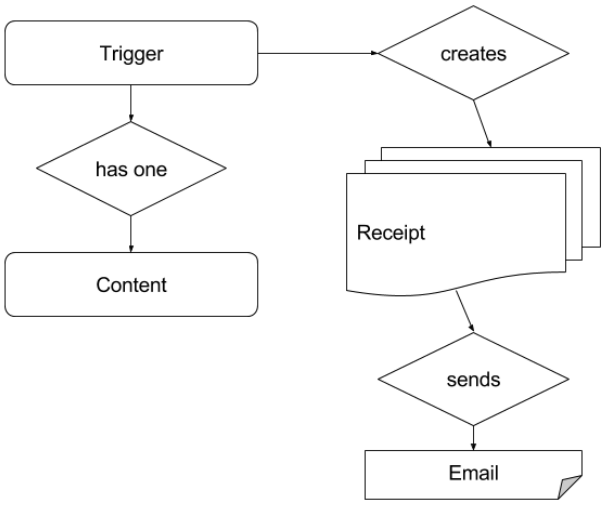

Figure 5

Content

This object gives reference to various static content messages with the metadata necessary to correctly send a message.

Receipt

The receipt object contains a record of some content being sent to a user. The state of the receipt indicates whether it's waiting to be sent, in the send queue, or finished.

Job

The job object represents a list of content to be sent at a given time. Jobs let content be scheduled for entry into the queue. Ready is a job that's waiting to be sent to the send queue, sending is a jobs that's currently queuing, and finished means that all the receipts have been sent to the send queue.

Preference

This object tracks a user's communication preferences. We don't want to send a user emails or sms if they've opted out of that type of communication.

Trigger

The trigger object sets a default pairing of content and parameters that can be updated and used from external services. This will help us update content that's sent quite often without making changes to the code itself.

3.2.6 Frontend

The frontend was originally set up utilizing Create React App (https://github.com/facebookincubator/create-react-app) provided by Facebook. This is an opinionated framework starter pack that offers a barebones NodeJS web server for running the frontend and a working React and Redux set-up (compiled using Webpack). In addition, it included React Router, which we use to handle all of our application routing.

We chose to go with this toolkit due to the ease of set-up for all team members for frontend development. In addition to what's already mentioned, the starter kit offered niceties like hot reload (for auto-refresh on save) and ESLint (for parsing Javascript for linting). Our design for development was strongly determined by React and Redux. The design pattern details will be fully covered in sections 3.3.4 and 3.3.5 below.

We decided on a single-page application which routes without refreshing. The single-page application functions as a dashboard where users (whether roasters or customers) can choose pages from the sidebar which then rerenders the content in the main view. Since Redux handles state and React Router works with React to change the route and page without actually refreshing, choosing this design was beneficial.

Modifying the Redux state by firing actions with AJAX requests to the server proves to be incredibly performant. From frontend to server, we can load many items (bean items, subscriptions, etc.) very quickly. We decided to display items using lazy loading and an infinite scroll technique rather than paging. This allows for users to continuously scroll content without having to interact with page numbers. A downside to this design is that users may not as easily be able to get back to previous items.

3.2.7 Proxy Layer

Since our services are hosted in the cloud and can be dynamically scaled as demand increases, we need a proxy layer to distribute requests to the correct IP addresses behind the proxy. To accomplish this, we use nginx as a web proxy between external requests and all Expresso's services (Figure 6). The nginx service receives incoming requests to "expresso.store" and routes them based on subdomain to the different IP and ports associated with the domain. As a built-in feature, the proxy will load balance

between multiple hosts under the same subdomain, ensuring that Expresso could scale if demand spikes simply by spinning up new hosts.

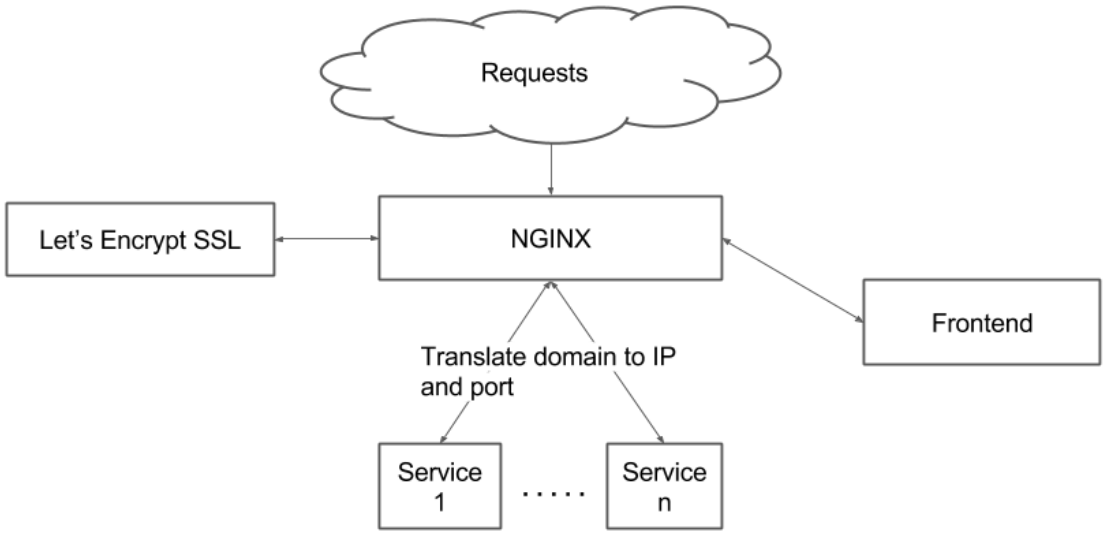

Figure 6

To take full advantage of the proxy layer, all expresso services also route their requests through the proxy rather than directly calling the IP of a given service. So, services can change IPs or ports as long as the proxy can find them and requests will be routed accordingly.

A final benefit of adding a proxy layer is securing our requests using SSL. Each request through any Expresso service is signed with the given subdomain's CA certificate to provide security assurance to any users connected to our servers.

3.3 Design Patterns

3.3.1 Service Design

With many microservices being built in Expresso, it was imperative to build each service similarly and with flexibility in mind as features need to be added during development. We also wanted to minimize code reuse while maximizing composability between services. This means that services shouldn't have to rewrite functionality that multiple services need, but they should rather be able to build on each other and share previously implemented functions.

To accomplish this goal, we built a standard framework for services comprised of 5 types: Routers, Handlers, Helpers, Gateways, and Configs.

Routers
The top level object, routers connect handler functionality to urls that can be accessed over HTTP requests. The router serves to construct all the context, gateways, and other objects that will be used throughout the rest of the program. The context a router creates, gets passed to all underlying handlers and contains any connection they have to other services whether within Expresso or not.

In addition, the router handles tracking metrics and api authentication, code that is shared between services.

Handlers
Called with incoming HTTP requests, handlers take the generic requests and validate their parameters, unmarshal their payloads and take action based on the contents. Handlers don't contain much logic other than sending error responses or passing out the result of helpers. By separating the request information from actual data manipulation, we can keep the code simple and readable in handlers to provide a clear picture of the path of a request through a service.

This was particularly helpful in debugging errors as well as providing a consistent API interface across all of our services. By providing a consistent API, we significantly increased our frontend development speed, and reduced the friction when adding new endpoints or updating functionality.

Helpers
The connection between request and database, helpers generally perform actions on the underlying data in the services. Helpers directly use gateways and expose interfaces that retrieve, update, or delete data based on calls from handlers. By keeping data interaction in handlers, we made composing functionality within a service as simple as adding a helper and calling its functions.

Helpers also helped consolidate database query building to a single set of functions rather than spreading them throughout the application. Generally functions in helpers clearly define their purpose, making code that uses them understandable and simple.

Gateways
These entities helped reduce code duplication throughout our application. A gateway provides a consistent interface in each service for creating and using connections to external services. Whether retrieving information from a database, or uploading photos, the gateways are used exactly the same in every service, and prevent connection management code from polluting handlers and adding complexity that bogs down development.

In addition to external services, each services implemented its own handler that serves as an interface between other go servers and itself. These gateways essentially wrap HTTP requests and response, but serve an important purpose of clearly defining the interactions between services. Rather than having to update all our code across all the services if an API changes, we can simply rebuild using the new gateway and guarantee consistent behavior across all the services. This helped add some peace of mind when making rapid changes in backend development.

Configs

Finally, the configs helped give each service a standard data type for defining and accessing application configuration. Given that Expresso's services always need connection information, having inconsistent ways for reading configuration variables would significantly hamper the portability of our code.

Using the configs objects, it's simple and consistent to share or update configuration in different microservices.

3.3.2 Event Queues

Expresso not only has to respond to predefined requests from its services, but it also has to respond appropriately to 3rd party webhooks. Responding to these webhooks required more than just a simple API endpoint because the volume from external providers could tax server resources that would be better spent serving data for our users. However, important events in webhooks should still be handled in a fault tolerant and timely manner. To accomplish this, we created an event queue interface of producers and consumers that allows webhook requests to be handled quickly, events placed into a queue, and then handled by a worker lazily.

With this webhook setup we can scale the number of workers to match demand and server resources while not losing API responsiveness, a critical factor in our design.

Worker implementation is integrated into the handlers and gateways of Section 3.3.1 and requires only implementing a single function interface for handling new messages. Any message produced in the service will be pushed to a queue denoted in the configuration file and then workers can read those messages whenever they arrive.

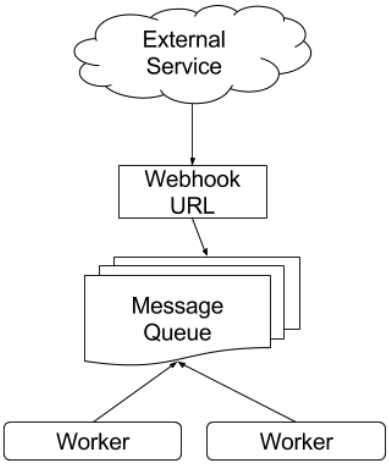

FIGURE 7

3.3.3 Database Schema

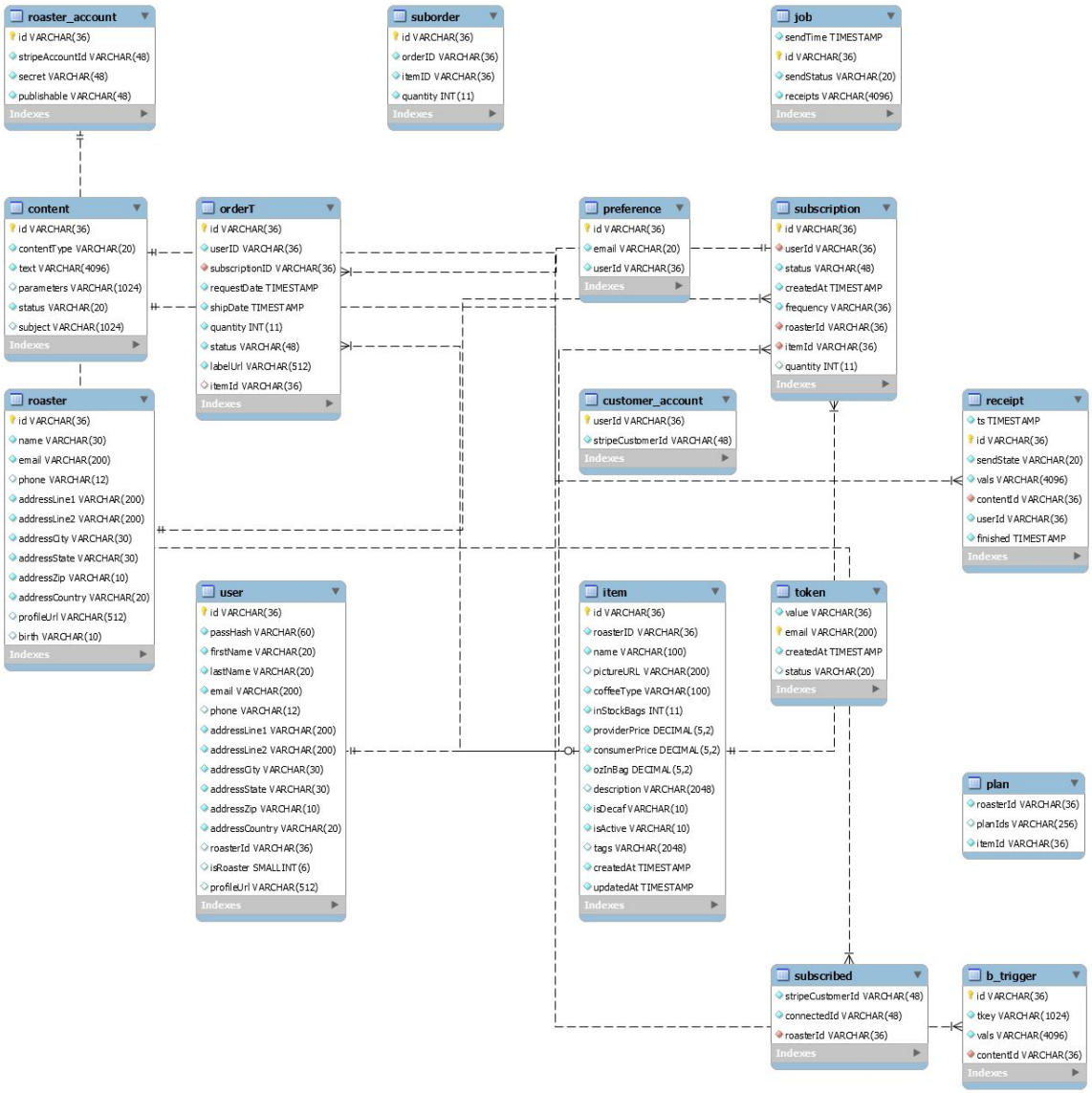

FIGURE 8

3.3.4 Frontend Store and Statefulness

On the frontend, we are using Redux to handle both a data store and statefulness of Reat components. In essence, the store is immutable, meaning there is no way to modify the data unless an action has been fired from a component. When an action is fired, there is typically a modification to the application's state. All of the actions and store updates are done seamlessly without requiring refreshing the application, giving the application a feeling of live updates. Common actions might be *REQUEST_ITEMS* or *LOGOUT*, described below.

The *LOGOUT* action is used to clear the user's JWT and log the user out. In additon, this action resets the store to it's initial state to continue a new session.

The *REQUEST_ITEMS* action is used to fetch data from the server with AJAX. Once the AJAX response is received, it is sent through a reducer which will map the data to a specific section of the store. Afterward, if there is new data in the store, it will be updated in any component using that data. We apply the data to components by mapping the data from the store to props on the component. This is typically done in a container component, which is explained more thoroughly in the following section.

3.3.5 Frontend Container Components

With React on the frontend, we followed a container design pattern for components. The idea behind using container components is to separate heavily logic-driven components into both presentational and container components. Presentational components are supposed to contain little-to-no logic and provide the necessary JSX (in-line Javascript html). Container components handle all events, callbacks, and fetching with AJAX. In addition, they fire all Redux actions and handle mapping of data from the store. In React, it is beneficial to pass data from top-level components downward. This design pattern provides a way for us to enforce the best prop handling.

4 Implementation

4.1 Development Process

4.1.1 Task Allocation

Tasks are prioritized and divided at our weekly meetings and on the fly- those that are vital to a minimum viable product are given precedence. If someone holds a clear preference towards a task, then that task is assigned to him. Unforeseen issues sometimes arise as development progresses; whoever feels they have ample time will address the issue.

4.1.2 Git Workflow

We follow a standard Git workflow. We have a base branch *master* that stays deployable - the branch is safe to compile and runs with no errors. To add new features, we create feature branches off of *master* and add new source code there. Once the feature is ready, we submit a pull request for the specified feature branch where other team members will review the changes. This allows the team to stay informed on what features on being added, catch mistakes, and create improvements. Once reviews are complete, we merge the feature branch into *master* and start the process over again with a different feature branch. We also utilize git's issues feature to create assignable tickets for enhancements and bug fixes. This helps prevent overlapping work and also facilitates discussion for each issue.

4.1.3 Continuous Integration

Our backend services utilize TravisCI for continuous integration. Each service has unit tests that TravisCI automatically runs, each time new code is pushed. This maintains code quality and catches erroneous behaviours before we containerize and deploy the updated services.

4.1.4 Phases

We have split development into flexible, continuous phases: backend, backend plus frontend, and frontend plus bug fixes. We first created the backend microservices, implementing RESTful API's for each service. Once basic CRUD operations were completed, we implemented the frontend to interact with the API endpoints, then finished up with bug fixes. The beauty of software development is that we are not permanently stuck in one phase. We have the ability to create changes on the fly, such as altering the backend to satisfy evolved requirements, while also developing the fronted at the same time. This flexibility creates an efficient development process,

allowing us to dynamically improve any aspect of the application, based on evolving requirements.

4.2 Technical Stack

4.2.1 Go

Go is a free and open source programming language. Go makes it easy to map routes to functions in the application. Because of this we will be using Go to implement our backend services.

4.2.2 MySQL

MySQL is a relational database management system. MySQL is scalable and high performance. We will be using a MySQL database to hold information about customers, shops, inventory, and orders.

4.2.3 Node

NodeJS is a JavaScript runtime environment which uses an event-driven architecture. NodeJS is capable of asynchronous IO to optimize throughput and scalability for web applications. We used Node for an entry point to our frontend and use the node package manager (npm) to download and install client javascript libraries.

4.2.4 React

React is a JavaScript library that provides views for data displayed using HTML. React allows for efficient updating of the web page when data changes. React can also be used to create a responsive website. Because of these features we will be using React to create our front-end website.

4.2.5 Redux

Redux is “a predictable state container for JavaScript applications” that links with React to create a seamless flow of data throughout the frontend of the application. In short, Redux has a store which holds the application’s data. Actions are fired from the from React components to populate the store. The store then updates the data, re-rendering components with the new data.

4.2.6 RabbitMQ

RabbitMQ is a message queueing service that allows different components to interact with each other.

4.2.7 Stripe

Stripe allows individuals to accept and make payments over the Internet. We will be using Stripe to handle charging customers, subscription payments, and paying the shops for the orders they ship.

4.2.8 Shippo

Shippo is an API that allows for the creation of shipping labels, shipping packages, and tracking the packages from various providers. We will be using Shippo to send the coffee beans to the customers and give shipment updates to users.

4.2.9 Sendgrid

Sendgrid is an email delivery service. We will be using Sendgrid in our communication service to keep customers updated about their subscriptions, shipped orders, and any messages they have received.

4.2.10 Digital Ocean

Digital Ocean host all of our servers, database, proxy, and stats information. By using cloud hosting we've ensured constant uptime during development, and Expresso is not locked to any one computer but can be deployed and expanded in minutes.

4.2.11 Docker

Docker creates a lightweight method for containerizing our services and deploying them either locally or in a cloud environment with no setup. The structure of docker-compose files also allows us to expand our hosting on the fly without having to worry about specific configuration on devices.

4.2.12 Amazon Web Services S3

To host static content (images), we use Amazon S3. As an industry standard with simple interfaces, we can store our files without wasting our limited server space.

4.3 Testing

4.3.1 Methodology

With many disparate services, integration testing from the start is a high priority for development. In order to guarantee reliable integration tests, reasonable unit testing should also be conducted on implemented interfaces.

At the very least, testing for each service should exercise mocks from other services which can be created through golang's built-in mocking library. Using go's built-in functionality for mocking in unit testing, unit tests can run without relying on correct functionality from other services, so development can continue concurrently without one implementation bottlenecking another.

In addition to depending on mocks for mocking integrations during unit testing, full integration testing is planned as soon as possible. To accomplish painless integration testing for each service, we plan on using containerization to allow developers to run

any dependent services locally while running integration suites or conducting manual integration tests.

4.3.2 Testing Criterion

We've decided on two primary testing criterion with which we'll evaluate tests for validity and intensity: branch coverage and requirements acceptance. With these two primary requirements guiding our test writing, we minimize time consuming errors, while avoiding spending too much time writing test cases.

Optimizing for high branch coverage will help ensure that most lines of code are run before pushed to development for integration testing and that we've handled error cases which we expect. Since error handling in go consists of testing for error values returned from function calls, branch coverage will help prioritize catching erroneous responses in testing before integration.

Requirements Acceptance is a focus of integration testing where we must test whether the system works together based on our functional and nonfunctional requirements. By meeting the requirements acceptance criteria, we can validate that the Expresso functionality meets our initial requirements for completion.

4.3.3 Testing Status

Throughout development, especially at early stages, we focused on building unit test suites for critical components of our infrastructure. These test suites focused on the core logic and prioritized for business logic rather than input validation. To balance new development and regression testing we decided not to set a team-wide standard of code coverage, but have managed to keep above 50% of the total on each of our microservices.

In addition to pure unit testing, integration testing was also a priority during development. Determining where and when services were malfunctioning can be difficult in microservice system, so we placed a high priority on gathering service metrics from the outset of development. To do this, we embedded stats gathering into every service request and response, including timing and response type. Gathering these metrics helped preliminarily determine where faults were occurring and quickly stop them without digging into the code too much.

4.4 Challenges

4.4.1 Learning New Technologies

For many of us, this was our first time working with Golang, React, and Redux. As such, we had to spend time learning these technologies instead of jumping right in and working. Even after we had gone through tutorials and started working on our actual

project, we still ran into problems that we didn't know the answer to. When this occurred we would talk to each other about the problems we were running into, and if no one knew what the solution was we would do further research into the topic.

4.4.2 Golang Dependencies

We use an external library, godeps, to manage backend dependencies. Some of godep's functionality clashes with Golang's built in library installation command *go get.* Originally we consistently used the godeps command *restore* to update dependencies. This detached the libraries, breaking *go get.* To fix this, we needed to cumbersomely navigate to each library and re-point the git repository to HEAD. To alleviate this problem, we now remove the generated godeps dependency file and execute the godeps command *save* to regenerate the list of dependencies. *Save* does not detach the library repositories, so we can continue development smoothly.

4.4.3 Mockery

We use Golang's built in mock library, mockery to create mocks for unit testing. A team member had consistent problems generating mocks, creating a small bottleneck for back end changes. His inability to mock was caused by specific parsing error native to the Windows environment, which is still currently being investigated by mockery maintainers.

4.4.4 Stripe Integration

Typically, Stripe transactions are kept between many users and one account owner. This account owner is normally the business offering the service, Expresso in our case. However, our needs required that we add additional accounts for the roasters on our platform, so they can receive funds from their subscribers and transfer funds to their own business bank accounts. This management required a great deal more involvement than was initially planned and required a deep understanding of the Stripe ecosystem.

In the end though, the Stripe integration made for smooth invoices and helped us avoid storing payment and bank information.

4.4.5 Scheduling

Because of the coupling between the microservices and the frontend there were often times when one of us was waiting on another to finish their work. This became a challenge because we all have different schedules and weren't always able to get features implemented as quickly as possible. At times this led to some features being set aside until the necessary features were implemented.

4.4.6 Managing Build Length

At one point in the project building all 5 services, database, proxy, and stats required approximately 10 minutes and over 5 GB of storage. With multiple builds each day, this was rapidly cutting into the development cycle, eating up a chunk of storage on Digital Ocean, so we investigated decreasing the build size as a way to both decrease the build time and open up storage on Digital Ocean. Previously, the Docker images for each of the 5 services was based on a regular version of Ubuntu taking 650 MB of storage for each service, and contributing to most of the delay in restart time. So, we investigated alternative Docker images with smaller versions of linux on them, eventually settling on using the minimal image taking only 5 MB.

After switching all services to use the minimal image size, we cut each service down to only 20 MB, and kept build time under 5 minutes. This reduced the total build size by two orders of magnitude, and helped keep our iteration fast.

4.5 Security

4.5.1 Passwords

The first step in securing a user's account is to let the user set a password. When a user registers their account they enter their password, which is then submitted to the user microservice. Before the user's information is stored in the database, it is encrypted using the bcrypt library. The resulting hash is then stored in the database and used to compare to the password users enter while logging in. On the frontend we require that passwords are at least 10 characters long.

4.5.2 Tokens

To ensure that only authenticated users are able to access our microservices we decided to use JSON Web Tokens (JWT). A JWT is a self-contained, compact method for transmitting data. The information in the token is signed upon creation, so that it can later be verified as an authentic token. When a user creates their account or logs in, a new JWT is created and signed using a secret stored in an environment variable on Digital Ocean. That token is then passed to the frontend application and sent with each HTTP request the application makes. Each route that contains protected data will verify the signature of the token and only supply the requested information if the token is valid.

The token is stored in local storage on the client. This means that when a user logs in, navigates away from our application, and then later returns to the application they will not have to log back in, as the stored token is sent to the server for verification.

4.5.3 Secure Socket Layer

We set up our website with a Secure Socket Layer (SSL) certificate, which enables encryption between the website and the server, so that any information sent cannot be read by anyone that intercepts it. When a connection is established between a client and the server, both go through a handshake process to verify the certificate and establish a session key, which is then used for encrypted communication between the client and server.

5 Conclusions

5.1 Goals

From a consumer perspective, our service will help people discover their favorite coffee brands and pay to get them shipped to their door. We want to provide a convenient, reliable, and easy way for people to have just the right amount of coffee that fits their taste delivered to their door. We plan to do this by creating a fully automated delivery service for roasters, as well as a platform for customers to discover, purchase, and review a variety of coffee roasted by small shops around the country. This will begin in Ames, eventually scaling to Iowa and (a lofty foresight) nationally.

5.2 Completed Work

At the outset, our goal was to create a scalable web-service with a modern architecture and programming practices. Thus far, we've succeeded in creating a minimum viable product that could be opened up for consumer use with only a few minor tweaks (like switching to a non-testing Stripe instance). Expresso accomplishes all of our stated requirements, allowing users to browse, subscribe and pay for shipments of coffee beans from roasters on the service. From a roaster perspective, we allow roasters to easily display their items, receive new orders, and act on those orders by shipping them out.

In addition to minimal functionality, we also have a more robust inventory management system, email alerts, and automation on the backend that gives user's more than just an amateurish experience.

From a technical perspective, we've succeeded in creating a modern microservices architecture that's easily scalable and deployable on cloud services. With the containerization between services, some of our implementation could even be used in other applications just by configuring it correctly. Our implementation, while not completely tested, has a significant test base that could be expanded with only a moderate amount of effort.

So, with that said, the completed work this semester meets our stated goals and could be easily deployed as a production scale system for ordering and shipping coffee from local roasters.

5.3 Future Work

There are no major plans for the team to continue work on this project after the completion of our course. Nonetheless, the following is a list of next steps that could be taken:

1. Consult with local coffee shops in Ames, demoing Expresso. Feedback could be taken into consideration and updates made to the site. If a coffee shop wishes to try out Expresso, we could help them get started.
2. Implement a centralized inventory version with automated shipping that would function in the event that a centralized inventory and shipping system is desired.
3. Implement better review functionality, allowing customers to give feedback. Similarly, the backend could possess machine learning functionality that would understand which items to recommend to customers based on reviews and habits.
4. Implement item bundles, allowing for single subscriptions to consist of multiple types of coffee products.
5. Generalize the system to allow for other types of item subscriptions other than coffee.

Appendices

Appendix 1: Operation Manual

We have our projected hosted on Digital Ocean. You can visit our final implementation of the web app at expresso.store. You can also download each of the parts and run them on a local machine.

Setting Up The Microservices

Prerequisites:

- Install Golang
- Install Make (also part of the build-essential package on Linux)
- Install git
- Install Docker
- Run the command “go get github.com/tools/godep”
- Run the “ghmeier/expresso-mysql” image
- Run the “ghmeier/rabbitmq-delayed” image

Steps:

1. To set up bloodlines, run the following commands in a terminal
    a. go get github.com/ghmeier/bloodlines
    b. cd $GOPATH/src/github.com/ghmeier/bloodlines
    c. godep restore
    d. make deps
2. To set up towncenter, run the following commands in a terminal
    a. go get github.com/jakelong95/towncenter
    b. cd $GOPATH/src/github.com/jakelong95/TownCenter
    c. godep restore
    d. make deps
3. To set up coinage, run the following commands in a terminal
    a. go get github.com/ghmeier/coinage
    b. cd $GOPATH/src/github.com/ghmeier/coinage
    c. godep restore
    d. make deps
4. To set up covenant, run the following commands in a terminal
    a. go get github.com/yuderekyu/covenant
    b. cd $GOPATH/src/github.com/yuderekyu/covenant
    c. godep restore
    d. make deps

5. To set up warehouse, run the following commands in a terminal
    a. go get github.com/lcollin/warehouse
    b. cd $GOPATH/src/github.com/lcollin/warehouse
    c. godep restore
    d. make deps
6. Each of the services needs a configuration file named config.json. An example config file can be found in Appendix 4. Enter the values according to how you have everything set up and place the config in the root of each microservice directory
7. In each microservice run the command make run from a terminal

Appendix 2: Other Considerations

During the design process we decided on a solution with responsive web applications for both Consumers and Producers and a microservice backend system powering the interactions with a distributed shipping method.

By using a web application as the primary user interface for Expresso, we can maximize usability across devices while condensing our development workflow into a single platform. While mobile applications provide easy access on phones and tablets, developing applications can be time intensive with limited abstraction between applications. Developing mobile solutions which function at a high level on both Android and iOS platforms would require significant development effort which could be instead applied to building the core product. An additional consideration in our decision to build the UI as a web application was the primary use cases of Expresso. From a user perspective, the ideal experience is ordering a subscription and continuously receiving high quality product within expected timeframes without visiting the website again. The value of Expresso isn't in an interface users spend a great deal of time using, but in reliability of our back end systems. Once again, this highlights that the best use of our development time is in building reliable systems for handling the logistics of shipping coffee subscriptions.

When considering whether to use a monolithic or microservices architecture, we weighed the technical as well as development process impacts of both options and decided on the microservice approach for a few reasons. One, microservices offer increased flexibility and autonomy for developers. Rather than performing rapid development on a code base with dependencies spread throughout, we can work on smaller services with mocked responses from areas which haven't been implemented. Scaling is an additional point in favor of using microservices which allow us to individually increase capacity for services which are facing a heady load rather than scaling the entire application. This would let us react rapidly to high demand situations.

Finally, we chose to work with the distributed model for shipping the Producer's product to our Consumers. This model allows Expresso to ensure fresh coffee being shipped along the shortest path from Producer to Consumer. The distributed model increases our logistical workload, with having to manage packaging and shipping information, but ensuring freshly delivered beans took priority. Additionally, this method removes a barrier to expanding Expresso to additional Producers by letting the Producers manage their own inventory and not sending their beans to a central distribution center.

Appendix 3: Links to the Codebase

The code for our front-end web application and each of our microservices can be found below:

- Expresso — https://github.com/jonnykry/expresso
- User Service — https://github.com/jakelong95/TownCenter
- Billing Service — https://github.com/ghmeier/coinage
- Subscription Service — https://github.com/yuderekyu/covenant
- Inventory Service — https://github.com/lcollin/warehouse
- Communication Service — https://github.com/ghmeier/bloodlines

Appendix 4: Example Configuration File

```
{
        "mysql": {
                "host": string,
                "port": string,
                "user": string,
                "password": string,
                "database": string
        },
        "sendgrid": {
                "api_key": string,
                "from_email": string,
                "from_name": string,
                "host": string
        },
        "towncenter": {
                "host": string,
                "port": string
        },
        "bloodlines": {
                "host": string,
                "port": string
        },
        "coinage": {
                "host": string,
                "port": string
        },
        "covenant": {
                "host": string,
                "port": string
        },
        "warehouse": {
                "host": string,
                "port": string
        },
        "rabbit": {
                "host": string,
                "port": string,
                "pubq": string
        },
        "statsd": {
                "host": string,
                "port": string,
                "prefix": string
        },
        "port": string,
        "tls": {
                "enabled": boolean
        },
        "s3": {
                "bucket": string,
        },
        "stripe": {
                "secret": string,
                "public": string
        },
         "shippo": {
       "token": string
    },
}
```