Expresso

Design Document

Team Number	34
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: October 24th, 2016

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 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.

1.4 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 view and edit an inventory of available goods
- Providers can receive consumer shipment information to fulfil orders
- Consumers can be alerted to incoming and delivered shipments
- Consumers are billed for purchased goods
- Providers are paid for fulfilled shipments

These deliverables will take the form of a software application comprised of an interface for Consumers, an interface for Provider, a back end software system for managing data collected from Consumers and Producers, and a method for shipments.

2.3 Possible Solutions

2.3.1 Consumer and Provider Interface Solutions

We will have unique interfaces for each developed as:

- 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

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 Proposed Design

Our proposed 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.

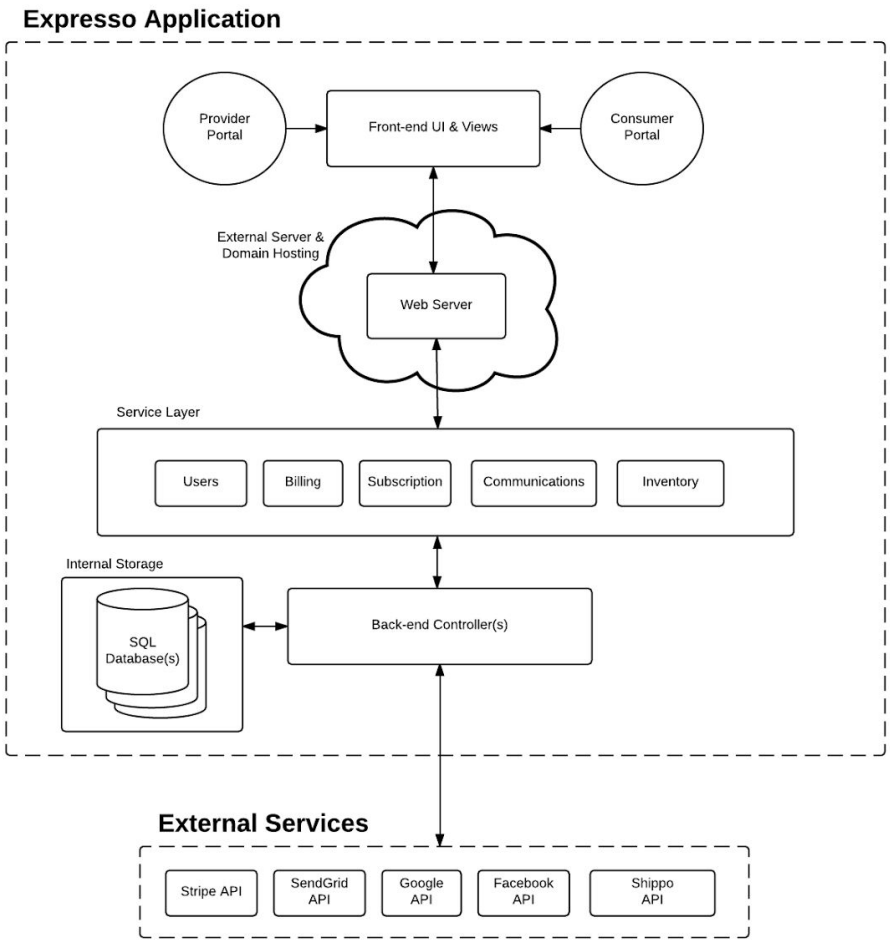

FIGURE 2

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 will contain all methods related to user management. This includes registering, updating, listing, and getting users.

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. This class stores foreign keys to the subscription service, billing service, order service, and communication service. This acts as a method to find this information about a specific consumer in the other services.

Provider
This class stores information about a provider. This class stores foreign keys to the billing service, order service, inventory service, and communication service. This acts as a method to find this information about a specific provider in the other services.

3.2.2 Billing Service

The billing service will handle all payments and billing, including subscription-based payments. This service will need to utilize other services, such as the messaging service to send emails to users regarding payment information. The majority of this service is dependent upon the Stripe API and acts mostly as a wrapper to their service. When a user signs up for a subscription, they will need to pay for the subscription using Stripe's API. Our billing service will store and maintain all tokens related to each individual user and their Stripe information.

Design Considerations
There are many considerations due to our dependence on the Stripe API to handle accounts for Expresso, our customers, and our roasters. We can achieve this through using Stripe's Connect, which allows use to programmatically create and maintain Managed Accounts for Roasters to receive payments. In addition, we will add Customers which will have a single, unique subscription to many roasters. The limitations of this make multi-product subscriptions difficult, so we may want to offer subscriptions only to a single source at first.

Because our service is largely subscription-based, we will be able to automate much of our usage of Stripe's API using Webhooks, allowing for updates based on changing statuses on accounts, subscriptions, and customers. This also promotes a lightweight billing service on our end with less network communication. More of these technical details are described in the billing development section.

Account (User)

This will store all Stripe keys and Billing-related account information for the customers.

Account (Roaster)
This will store all Stripe keys and Billing-related account information for the roasters.

Subscription
This will store all Stripe keys and Billing-related subscription information, as well as the keys to both user and roaster(s).

3.2.3 Inventory Service

The inventory service will handle all local (individual coffee providers) and global (union of all local inventories) inventories. Each coffee provider will be able to manage and update their inventory. Expresso will mostly use the global inventory to display and sell subscriptions based on what is available from the union of all local inventories.

Design Considerations
We want the MVP to allow advanced global and basic local inventory functions. More specifically, we will enforce strict control over available global inventory, asking the coffee providers to simply keep their individual inventory up-to-date with the latest inventory count. It is likely that the Inventory Service will expand to include another major piece of API functionality, the Orders Service. This would allow more direct control over how orders are requested, both on the local and global levels. But for this iteration, the Inventory Service will suffice. On top of that, If we choose to move beyond the MVP, more advanced local inventory control and functionality can be implemented to help coffee providers plan, predict, reorder supplies, and interact with any current inventory software they might currently use.

Inventory (Global)
This will store all inventory information regarding the union of all local coffee shop inventories.

Inventory (Local)
This will store all inventory information regarding local coffee providers.

3.2.4 Subscription Service

The subscription service will allow users to create, update, and cancel their coffee subscription. This service will store all relevant subscription data associated with the user.

Design Considerations:
A key issue was how the subscription service will handle multiple subscriptions per consumer. Currently, the subscription service follows a one to many relationship. Each user will be able to create a single subscription. Within this subscription will be a list of orders, which specify which coffee the user has selected to include within their subscription. This creates flexible subscriptions for users, resulting in a customizable experience.

Subscription
This object contains subscription data relevant to the user.

Orders
This object contains a list of coffee products the user has selected to be included within the subscription

3.2.5 Communication Service

For the communication service, we want to 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 want to 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. Aka, we want to send an email each time a person purchases an item or has a shipment inbound. Events logically could make our lives far easier. We want to avoid a situation in which a change to Bloodlines' code requires changes in the way we dispatch events from other services which could trigger communications.

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.

3.3 Design Analysis

3.3.1 Technical Feasibility

Currently, the MVP so far is technically feasible. All of the previously mentioned design choices thus far will be within our grasp for the following reasons:

1. We will be using various existing api's for payments and shipping
2. We will be leaving the storing of inventory up to the coffee providers instead of trying to store it in some other location.
3. We will be creating several manageable microservices.

The bulk of our potential issues lie in the hands of the coffee providers. Should one or more of them be happy to work with us, keep their inventory data up-to-date, and adequately respond and ship out product on time, then all will be well. Other than that, there are currently no major technical challenges that we expect to come across as all five team members are either well experienced in the languages and tools we will be using, or are more than capable of picking up and using new tools as we go.

4 Development

The primary driver for development of our services is the interface with which they communicate with other services and the front end. For this reason, the following sections detail the interface specifications for each service including return data models and API specifications.

4.1 User Service

Data Objects

Consumer		Provider	
name	string	business_name	string
email	string	email	string
phone	string	phone	string
id	int	id	int
subscription_id	int	billing	int
billing_id	int	inventory	int[]
address_line1	string	address_line1	string
address_line2	string	address_line2	string
address_city	string	address_city	string
address_zip	string	address_zip	string
address_country	string	address_country	string

Consumer API Interface

POST */consumer*

Parameters: firstname, lastname, email, phone, password, address_line1, address_line2, address_city, address_zip, address_country
Action: Creates a new consumer entry and returns the id that references that user, otherwise returns an error.

GET */consumer*

Action: Returns an array containing all of the registered consumers.

PATCH */consumer/:consumer_id*

Parameters: firstname, lastname, email, phone, password, address_line1, address_line2, address_city, address_zip, address_country, billing, subscription
Action: Updates the information for the specified consumer.

DELETE */consumer/:consumer_id*

Action: Deletes the specified consumer.

GET */consumer/:consumer_id*

Action: Returns information about the specified consumer.

Provider API Interface

POST */provider*

Parameters: name, email, phone, password, address_line1, address_line2, address_city, address_zip, address_country
Action: Creates a new shop entry and returns the id that references that provider, otherwise returns an error.

GET */provider*

Action: Returns an array containing all of the registered providers.

PATCH */provider/:provider_id*

Parameters: name, email, phone, password, address_line1, address_line2, address_city, address_zip, address_country, billing, inventory
Action: Updates the information for the specified provider.

DELETE */provider/:provider_id*

Action: Deletes the specified provider.

GET */provider/:provider_id*

Action: Returns information about the specified provider.

4.2 Billing Service

Data Objects

BillingCustomerAccount	
id	int
user_id	int
stripe_customer_id	string
stripe_card_id	string
email	string
created_at	date
BillingRoasterAccount	
id	int, PK
roaster_id	int
stripe_account_id	string
stripe_account_secret	string
email	string
created_at	date
BillingSubscription	
id	int
subscription_id	int
roaster_id	int
user_id	int
stripe_subscription_id	string
stripe_plan_id	string
amount	double
created_at	date
due_at	date

Billing Account Interface

POST */billing/customer/account/:user_id*

Parameters: user_id, email, (optional) parameters
Action: Creates a new billing account for the customer and returns the id of the account or errors

GET */billing/customer/account/:id*

Parameters: id, (optional) parameters
Action: Retrieves the billing account for the customer, returning an object of the account or error if it does not exist.

DELETE */billing/customer/account/:id*

Parameters: id, (optional) parameters
Action: Deletes the billing account found by id and returns success or errors otherwise

POST */billing/roaster/account/:roaster_id*

Parameters: roaster_id, email, (optional) parameters
Action: Creates a new billing account for the roaster and returns the id of the account or errors

GET */billing/roaster/account/:id*

Parameters: id, (optional) parameters
Action: Retrieves the billing account for the roaster, returning an object of the account or error if it does not exist.

DELETE */billing/roaster/account/:id*

Parameters: id, (optional) parameters
Action: Deletes the billing account found by id and returns success or errors otherwise

Billing Subscription Interface

POST */billing/subscribe*

Parameters: user_id, roaster_id, subscription_id, amount, (optional) parameters
Action: Creates a billing subscription for the passed subscription. Returns the subscription id of the created subscription or error if none was created.

GET */billing/subscription/:id*

Parameters: id, (optional) parameters
Action: Retrieves the billing subscription for the given billing subscription id returning an object of the subscription or error if it does not exist.

GET */billing/subscription/:customer_id*

Parameters: customer_id, (optional) parameters

Action: Retrieves the billing subscription by customer id, returning an object of the subscription or error if it does not exist.

GET */billing/subscriptions/:roaster_id*

Parameters: roaster_id, (optional) parameters
Action: Retrieves all billing subscriptions for the roaster, returning an array of all subscriptions or error if empty.

Stripe Customer Interface

We will use Customer to represent customers in our product, which are the individuals subscribing to roasters. Here is a detailed description of the Stripe Customer object and API: https://stripe.com/docs/api#customer_object

In our application, we will maintain and store the Customer ID from Stripe in BillingCustomerAccount. The Stripe Customer is created using Stripe's API when our internal customer billing API is called.

Stripe Account Interface

We will use Account to represent roasters (or shops) in our product, which are the companies offering beans for subscription. Here is a detailed description of the Stripe Account object and API: https://stripe.com/docs/api#account

In our application, we will maintain and store the Account ID from Stripe in BillingRoasterAccount. The Stripe Account is created using Stripe's API when our internal roaster billing API is called. We will be using a Managed Account (https://stripe.com/docs/connect/managed-accounts), which is created and maintained programmatically by our application. Because of this, we need to handle all identity verification. Stripe offers many variations of secure ways to collect customer information, including secure libraries of their own.

Stripe Connect Interface

Connect is Stripe's way of allowing a single account to maintain other accounts along with customers. For our use case, we need to have customers paying for subscriptions, and roasters receiving money for the subscriptions when they're paid. To do this, we can use Stripes Connect to programmatically create Stripe Accounts for our roasters, as well as maintain subscriptions between customers and roasters.

We will be creating Customers for our single Expresso Stripe Account, which link to (Managed) Accounts for roasters. Subscriptions will be maintained in our Expresso

Account, as described below. More details on Connect can be found here: https://stripe.com/docs/connect

Subscription Interface

We will use Subscription to handle billing and payment between our customer accounts and roaster accounts. Stripe's Subscription object can be viewed in detail here: https://stripe.com/docs/api#account

In our application, we will maintain and store the Account ID from Stripe in BillingRoasterAccount. The Stripe Account is created using Stripe's API when our internal roaster billing API is called.

Since subscriptions for Stripes Connect do not offer a destination, we "can create a customer and a subscription in a connected account using a token created with either the platform's or the connected account's publishable key." On our end, we need to maintain these customer and account keys. (https://stripe.com/docs/connect/payments-fees#creating-subscriptions)

Payment Interface

Payments will be made by recurring charges to the Customer's debit card. We will have Webhooks (https://stripe.com/docs/webhooks) set up to listen for these events. When the status of a subscription changes, we will take the money from the Customer and pay it to the Roasters on the subscription.

Each Customer will have an Invoice when their Subscription is created. Stripe automatically handles creation and payment of Invoices on a Subscription. If the Customer for some reason did not successfully pay their Invoice, our Webhooks will inform us and we will not allow the Customer to access their Subscription. In addition, we will inform the Roaster to not ship the product. The Subscription will be paused, or put on hold, and we will follow-up with the Customer to find out details of why the payment did not go through.

4.3 Inventory Service

Data Objects:

Packaged_Product	
name	string
picture	string (image url)
type	string
In_stock_bags	Int
price	double
oz_in_bag	double
id	int
shop_id	int
lead_time	int (*number of days it takes to create/receive more bags of this type of beans*)
reorder_level	double (*If in_stock_bags goes below this number, a new shipment should be ordered*).
pipeline_stock	double (*number of bags of beans currently*
Raw_Materials	
bean_name	string
type	string
in_stock_lb	double
id	int
lead_time	int (*number of days it takes to receive a new shipment of beans*)
reorder_level	double (*If in_stock_lb goes below this number, a new shipment should be ordered*).
pipeline_stock	double (*amount of beans, in pounds, currently being shipped to the coffee shop*)

Global Inventory Interface

GET */inventory*

Action: Returns an array of packaged product objects.

GET */inventory/:name*

Action: Returns the coffee product object with the given product name.

Local Inventory Interface

GET */inventory/:bean_name*
Action: Returns the coffee bean object with the given bean name.

POST */order*

Parameters: coffee bean type, manufacturer, payment method, etc
Action: Sends out a message with the given parameters, ordering more coffee beans. This can either be a message to the coffee shop owner(s) or directly to a bean supplier.

PUT */package*

Parameters: A set of one or more tuples, each specifying a coffee name, a number of ounces
Action: Subtracts the amounts withdrawn for each coffee bean type

DELETE */inventory/:bean_name*

Action: Removes the object with the specified bean_name

4.4 Subscription Service

Data Objects:

subscription	
subscription_id	*int* Unique id created for this subscription
subscription_type	*string* Type of subscription
user_id	*int* Unique id of user associated with this subscription
status	*string* Status of subscription:

	<table><tr><td>active</td><td>Subscription is live</td></tr><tr><td>pending</td><td>Subscription has yet to be paid</td></tr><tr><td>cancelled</td><td>Subscription is cancelled and user will no longer be billed</td></tr><tr><td>inactive</td><td>Subscription has been paused to a later date</td></tr></table>
created_at	*date* The date this subscription was created
start_at	*date* The date this subscription will be active
total_price	*double* Total price of all orders within the subscription
orders	
shop_id	*int* Unique id created for roaster
oz_in_bag	*double* Amount of coffee beans, in ounces
bean_name	*string* Type of coffee bean
type	*string* Type of roast
price	*double* Price of order

Subscription API Interface

POST */subscription*

Parameters: user_id
Action: Create a new subscription.
Response: 200 OK, Content-type: application/json


```
{
  "subscription": {
    "subscription_id": "1",
    "subscription_type": "tier2",
    "user_id": "10",
    "status": "live",
    "created_at": "03-10-15",
    "start_at": "03-15-15",
    “total_price”: “19.00”
    "orders": [
      {
        "shop_id": "1",
        "amount": "7",
        "bean_name": "Arabica",
        "type": "Medium Roast",
        “price”: “10.00”
      },
      {
        "shop_id": "2",
        "amount": "14",
        "bean_name": "Robusta",
        "type": "Dark Roast",
                                “price”: “9.00”
      }
    ]
  }
}
```


GET */subscription/:subscription_id*
Parameters: subscription_id
Action: View a subscription
Response: 200 OK, Content-type: application/json

POST */subscription/:subscription_id*
Parameters: subscription_id, status
Action: Update current subscription. Only active, pending and inactive subscriptions can be updated
Response: 200 OK, Content-type: application/json

POST */subscription/:subscription_id/deactivate*
Parameters: subscription_id
Action: Pause a subscription
Response: 200 OK, Content-type: application/json

POST */subscription/:subscription_id/cancel*
Parameters: subscription_id
Action: Cancel a subscription
Response: 200 OK, Content-type: application/json

4.5 Communication Service

State Enums:

content_type - EMAIL
send_status - READY, QUEUED, SUCCESS, FAILURE
send_state - SUBSCRIBED, UNSUBSCRIBED, MINIMAL

Data Objects:

Content		Receipt	
content_type string []string uuid bool	type (EMAIL) text parameters id active	uuid []string uuid uuid timestamp send_status	recipient values content_id id time_sent status
Job		Preference	
[]uuid timestamp uuid send_status	receipts time_sent id status	uuid send_state send_state	user_id email sms
Trigger			
uuid uuid string	id content_id key		

[]string	parameters	

Content Interface

POST */content*

Parameters: type, text, (optional) parameters
Action: Creates a new content entry and returns the id that references that content, otherwise returns an error.

GET */content?count=<max>&offset=<amount>*

Action: Returns an array of active content objects with *count* elements starting from the *offset* element.

GET */content/:content_id*

Action: Returns the content with the given id, error otherwise

PUT */content/:content_id*

Parameters: (optional) type, (optional) text, (optional) parameters
Action: Updates the object, *content_id* with the given values, and returns the updated object.

DELETE */content/:content_id*

Action: Deactivates the object, *content_id* and returns 'success' otherwise an error.

Receipt Interface

POST */send*

Parameters: recipient_id, sender_email, content_id, (optional) values
Action: queues a new message with the given *content_id* to the *recipient_id*. Optional values should correspond to the parameters required by the referenced content.

GET */receipt?count=<max>&offset=<amount>*

Parameters: In addition to count and offset, recipient_id, content_id, and status filter the returned receipts.
Action: Returns an array of receipts with *count* elements (default 20) starting from the *offset* element.

GET */receipt/:receipt_id*

Action: Returns the receipt with the given id, error otherwise

Job Interface

POST */job*

Parameters: receipts[], (optional) timeSend
Action: Creates a new job which will send a message according to the receipt_id's in receipts list at the provided time (UNIX). If no time is provided, it'll start queuing messages now.

GET */job?count=<max>&offset=<amount>*

Action: Returns an array of job objects with *count* elements starting from the *offset* element. Objects will have limited receipt info, and the state filters jobs according to their status.

GET */job/:job_id*

Action: Returns the job with the given id, error otherwise

PUT */job/:job_id*

Parameters: (optional) type, (optional) text, (optional) parameters
Action: Updates the job, *job_id* with the given values, and returns the updated object. Only available to jobs in the READY state.

DELETE */job/:job_id*

Action: Stops the job, *job_id* and returns 'success' otherwise an error. This only affects jobs in the READY state.

Preference Interface

POST */preference*

Parameters: user_id, (optional) email, (optional) sms
Action: Creates new preference setting for the user_id. Defaults to email=true and sms=true.

GET */preference/:user_id*

Action: Returns the preference settings for the given user_id, error otherwise

PATCH */preference/:user_id*

Parameters: (optional) email, (optional) sms
Action: Updates the preference settings of the user to match the provided values. A value being absent means it won't be altered.

DELETE */preference/:user_id*

Action: Removes the preference settings for the user designated by *user_id*.

Trigger Interface

POST */trigger*

Parameters: content_id, key, (optional) parameters
Action: Creates new trigger where the given key references the given content_id. The optional parameters will be the attempted fallback each time a trigger is started. Returns the created trigger object.

GET */trigger?count=<max>&offset=<count>*

Action: Returns a list of triggers with count elements starting from the offset.

GET */trigger/:trigger_key*

Action: Returns the trigger for the given key, error otherwise

PUT */trigger/:trigger_key*

Action: Updates the trigger with the new object information and returns the updated object.

DELETE */trigger/:trigger_key*

Action: Removes the trigger.

POST */trigger/:trigger_key/activate*

Parameters: trigger_key (the string which references the trigger), (optional) parameters

Action: This will send a message with the content and parameters of the given trigger_key. The provided parameters will override any saved parameters of the trigger. If the given parameters combined with the saved trigger parameters don't fulfill the content's required parameters, this will error, and no message will be sent.

4.6 Technical Stack

4.6.1 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.

4.6.2 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.

4.6.3 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 will use Node for an entry point to our frontend and use the node package manager (npm) to download and install client javascript libraries.

4.6.4 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.

4.6.5 RabbitMQ

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

4.2.6 Redis

Redis is a data structure server, which allows different processes to query and modify the same data.

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.

4.2.9 SparkPost

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

4.7 Testing

4.7.1 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.

4.7.2 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.

4.7.3 Process

To implement and maintain the testing criterion within our methodology, we plan on using Travis CI, a continuous integration tool, to constantly test code on commits

5 Conclusions

5.1 Completed Work

We have completed version 1 of our project plan. This plan introduces our application, main deliverables, design concepts, requirements and specifications, challenges, and a timeline for completion. We have traced though both Consumer and Provider workflows which identified Exresso's key functionalities. We then split these functionalities into separate microservices: User, Billing, Inventory, Subscription, and Communication services. We each took responsibility of a micro-service and introduced data objects and REST API endpoints. Jonny Krysh created the Expresso website, consolidating all Expresso related documents in one place and displaying information about the team.

5.2 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.

5.3 Solution

Given the assessed advantages and disadvantages above, 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.