ORDER FUNCTIONAL AREA

Part of Team Scrummy Bears

-By:
Avinash Arunachalam Arunachalachettiar Murugappan

With Input from the Scrummy Bears team:

Joe Schlupf

Sravishtta Kommineni

Lin Walton

Table of Contents

Overview

This component is responsible to create order for the end user when they place the order in this web application,

- It gets orer items and stores user’s shipping information and payment information in its independent database.
- Then order will be processed with the given user’s information.

Scrummy Bear’s Architecture

The Order Service in one of the four microservices running on AWS ec2 cloud.

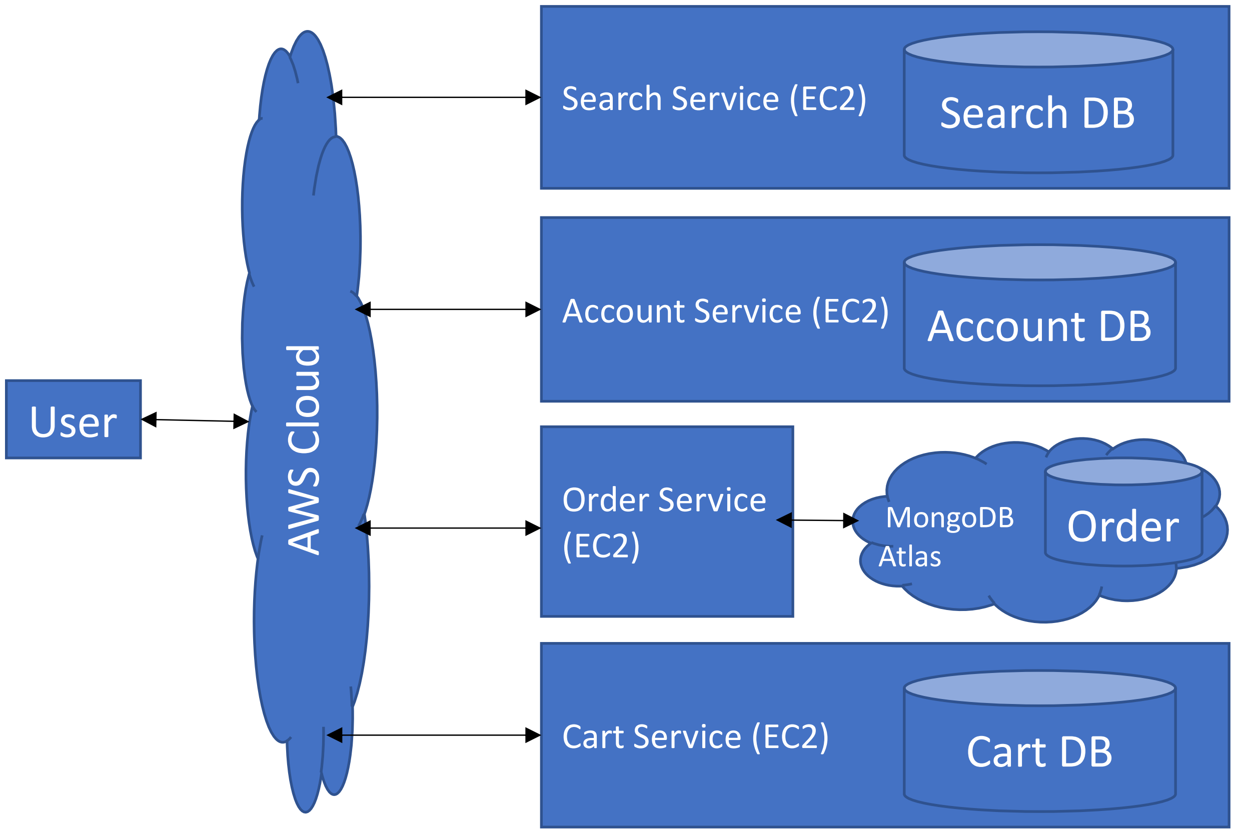


Figure 1: Scrummy Bears System Architecture

Order Component Service Architecture

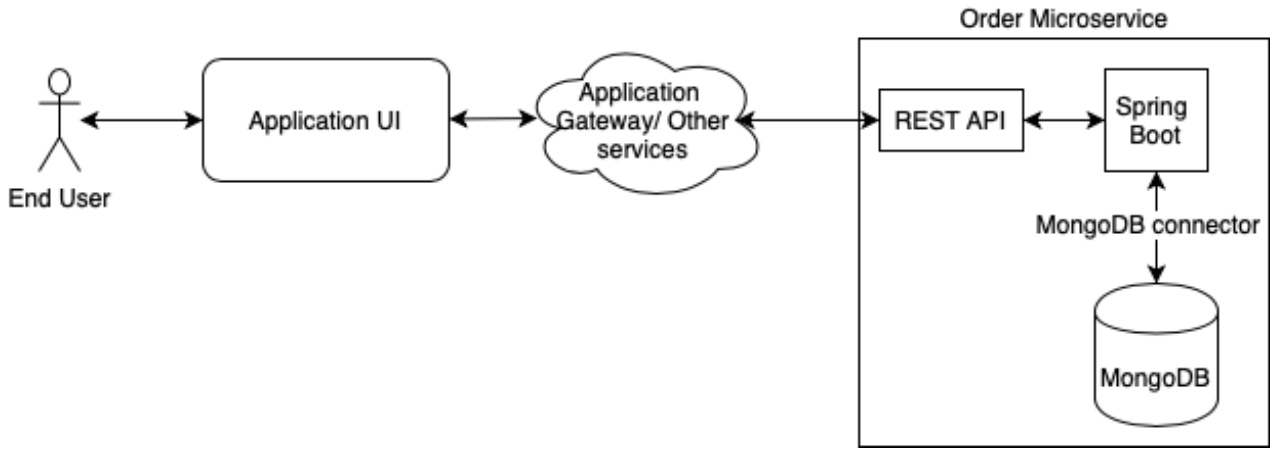


Figure 3: Order Component Service Architecture

This component will be implemented as a Spring-Boot micro-service with a REST API and utilizing a NoSQL: MongoDB. This micro-service will be implemented using an MVC architecture and will have its own UI components that will be incorporated into the complete web application. The diagram above shows the architecture of the Order micro-services component interfaced with other services.

- Front-End: HTML, CSS, JS, Bootstrap
- Rendering Engine: Thymeleaf
- Back-End: JAVA 1.8
- Database: MongoDB

Microservice Integration

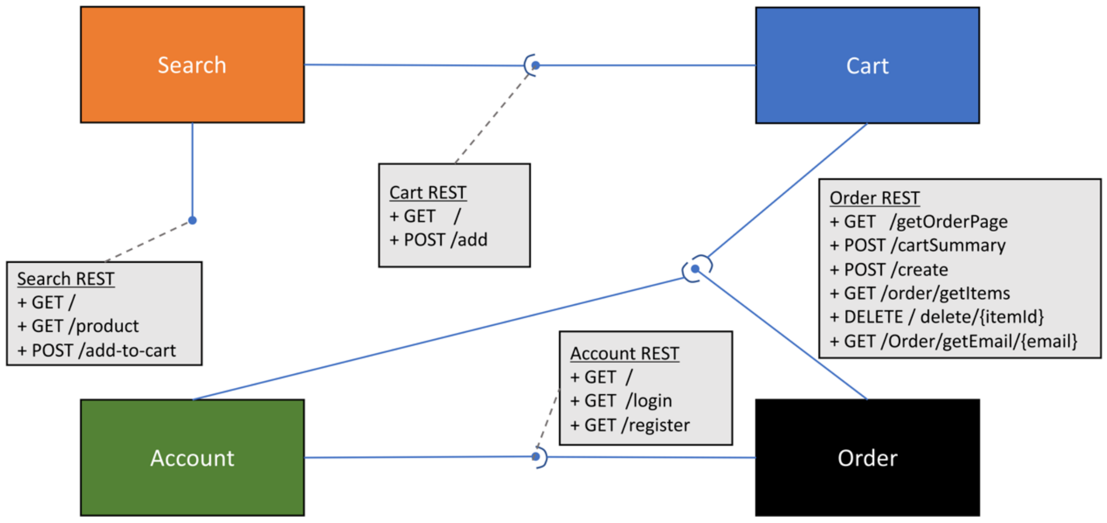


Figure 4: Microservices integration API

Once the cart service has the list of products a user is interested in, it coordinates with order microservice, where:

- POST /cartSummary/: Cart service posts cart items to order service
- DELETE /delete/{itemId}/: Cart deletes the item by itemId on order database if that item is deleted in cart
- GET /getOrderPage/: Returns Order page with Items that were in the cart as order summary

Once the order service receives items from cart, then order service coordinates with account service, where:

- GET /login/ : Order service calls the login page of account service
- - GET /getItems/ : Account service gets the items that was posted by cart from order service.

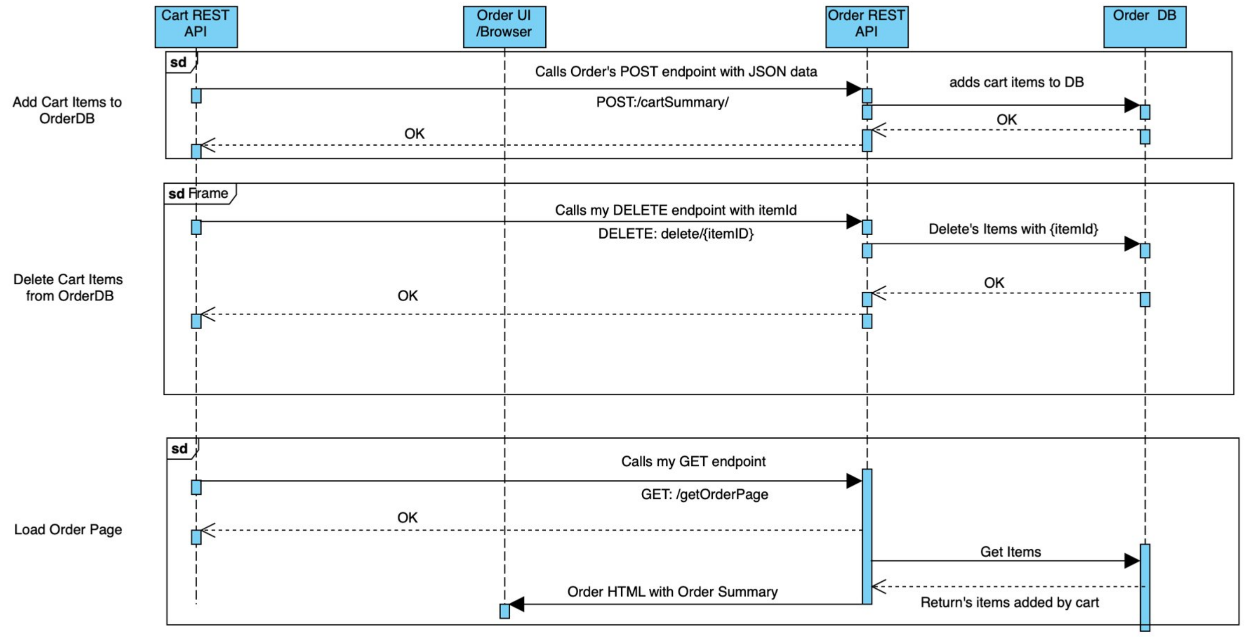
Order Component REST API

The Order Service has the following endpoints,

Endpoint	/create	/Order/getBy/{email}/	/cartSummary	/getOrderPage	/deleteItem/{itemId}	/Order/getItems
Purpose	Adds Order information to database	Other services can access order information by email	Adds cart items information to database	Displays item summary on order UI from database and returns order page	Deletes items on order db by itemId	Other services can access items that are ordered
Method	POST	GET	POST	GET	DELETE	GET
Data Format	JSON	JSON	JSON		URL Encoded	JSON
Expected Data	`fullName, email, address, city, state, zip, nameOnCard, CCNum, expMon, expYear, CVV`	`fullName, email, address, city, state, zip, nameOnCard, CCNum, expMon, expYear, CVV`	`itemId, title, description, price`		itemId to delete item from db	
Example Data	{“fullName” : “string”, “email”:”string”, “address”:“string“, “city” : “string”, “state”:”string”, “zip”: number, “nameOnCard“: number, “CCNum“: number, ”expMon”: number, “expYear“: number, “CVV“: number }	{ “fullName” : “string”, “email”:”string”, “address”:“string”, “city” : “string”, “state”:”string”, “zip”: number, “nameOnCard”: number, “CCNum”: number, ”expMon”: number, “expYear”: number, “CVV”: number }	{ “itemid”: “Long”, “name”: “string“, “description“: “string“, “prince“: “Double” }		/deleteItem/{itemId}	{ “ItemId”: “long”, “title”: “String”, “description”: “String”, ” price”: “Double” }

Table 1: Order Service RESR endpoints

Order & Cart Service Integration



Cart REST API
Order UI /Browser
Order REST API
Order DB
sd
Add Cart Items to OrderDB
Calls Order's POST endpoint with JSON data
POST:/cartSummary/
adds cart items to DB
OK
OK
sd Frame
Delete Cart Items from OrderDB
Calls my DELETE endpoint with itemId
DELETE: delete/{itemID}
Delete's Items with {itemId}
OK
OK
sd
Load Order Page
Calls my GET endpoint
GET: /getOrderPage
OK
Get Items
Order HTML with Order Summary
Return's items added by cart

Figure 5: Order/cart integration

The Order service works with cart service so the order summery on the Order UI is gotten from cart, So the customer can see the summary before placing the order.

1. Cart service adds items that are in the cart by using Order service's POST end point, once the items are passed to the order service and it is saved to the order database then order service returns ok response to cart service.
2. User clicks delete on cart, then cart service utilizes Order service's DELETE end point along with the itemId. Once the order service deletes that item from its database, it returns ok response to cart service.
3. When user clicks proceed to checkout, order service's GET end point is called and returns ok response to cart service.
4. Then the order service gets the items from order database and renders it on the Order UI and returns the Order UI along with the rendered information to the user.

Order & Account Service Integration

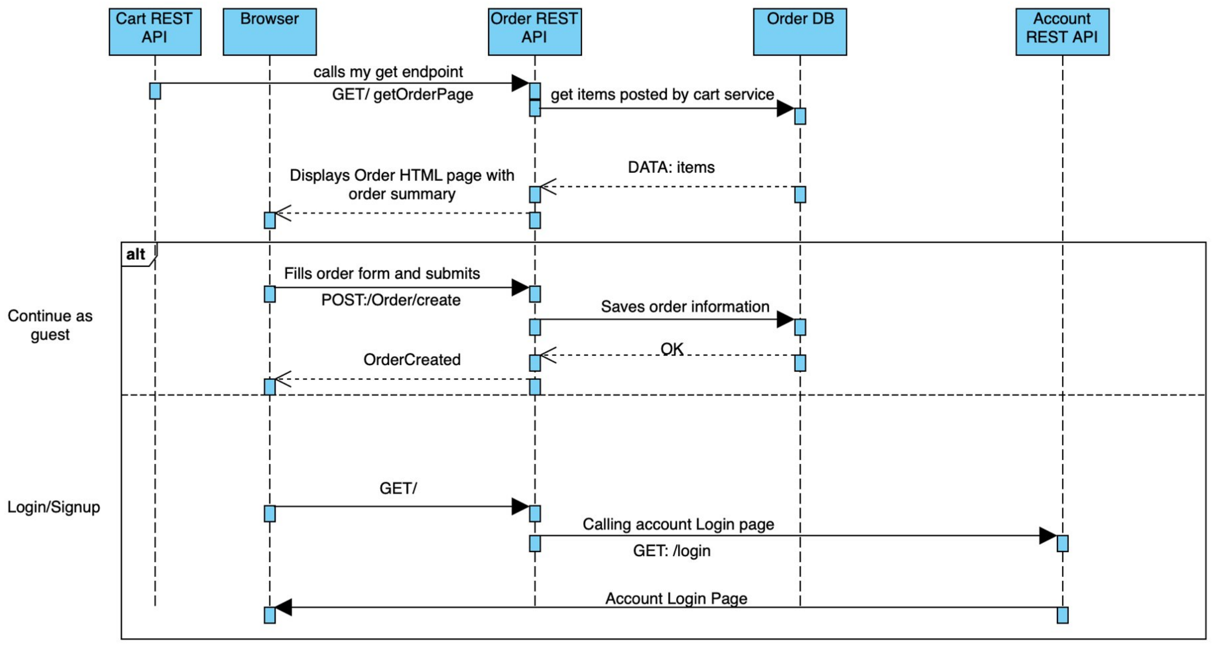


Figure 6: Order/Account integration

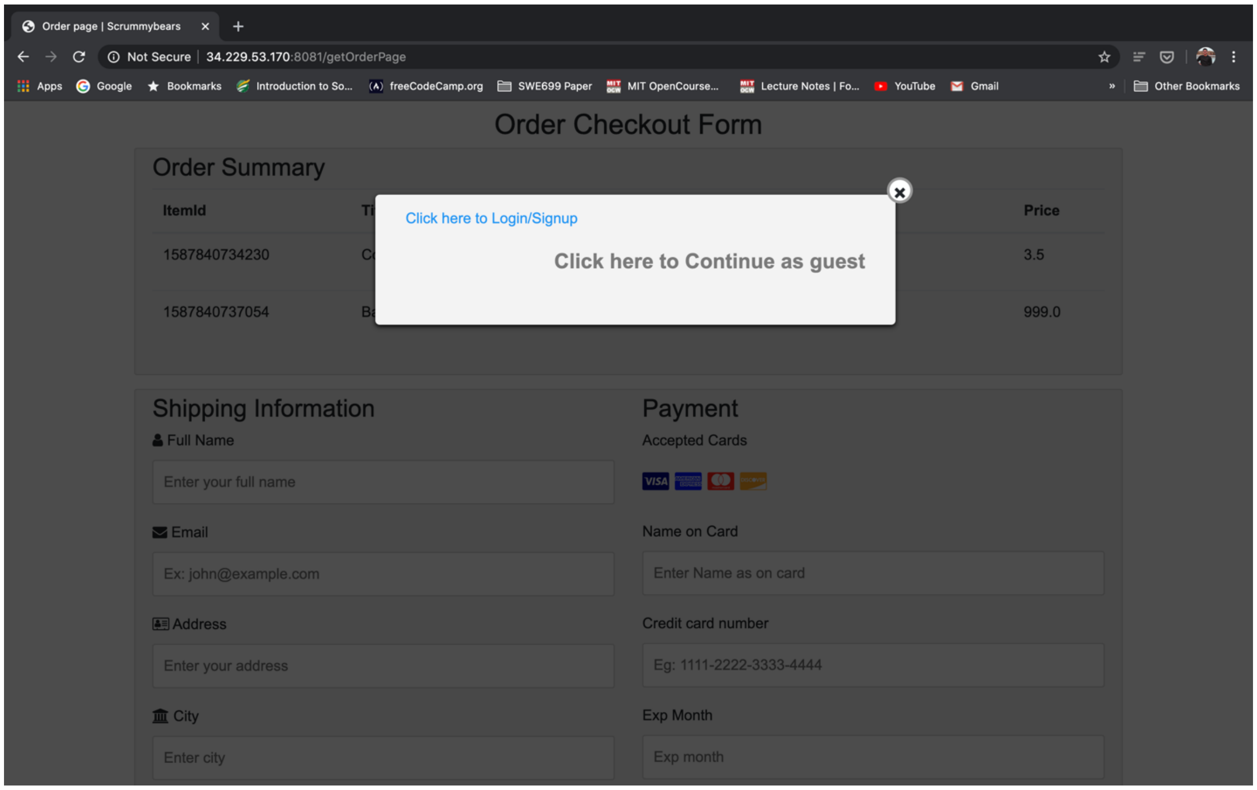
The order service works with account service to give end users the option to login, if not signup and then place the order. Order information and history can be viewed inside the user's account with the help of this integration also.

1. Thecart service has returned the order page with the order summary.
2. Then the user has two option,
   a. Continue as guest:
      i. The end user views the order summary and fill in the shipping information and payment information (ie, Order form).
      ii. When user submits the information, the order has been created as a guest check out and saved to the order database.
   b. Continue with Login/Signup:
      i. Then the Order service calls account service's get login page and displays the login screen of the account service.
      ii. Then the user can sign in or signup and place the order.

The account service will call order service's get endpoint to get the order summary.

Order Component UI

The below figures under this section are the order microservice's user interface.



Order Checkout Form

Order Summary

ItemId	Ti	Price
1587840734230	C	3.5
1587840737054	B	999.0

Click here to Login/Signup

Click here to Continue as guest

Shipping Information

Full Name: Enter your full name

Email: Ex: john@example.com

Address: Enter your address

City: Enter city

Payment

Accepted Cards

Name on Card: Enter Name as on card

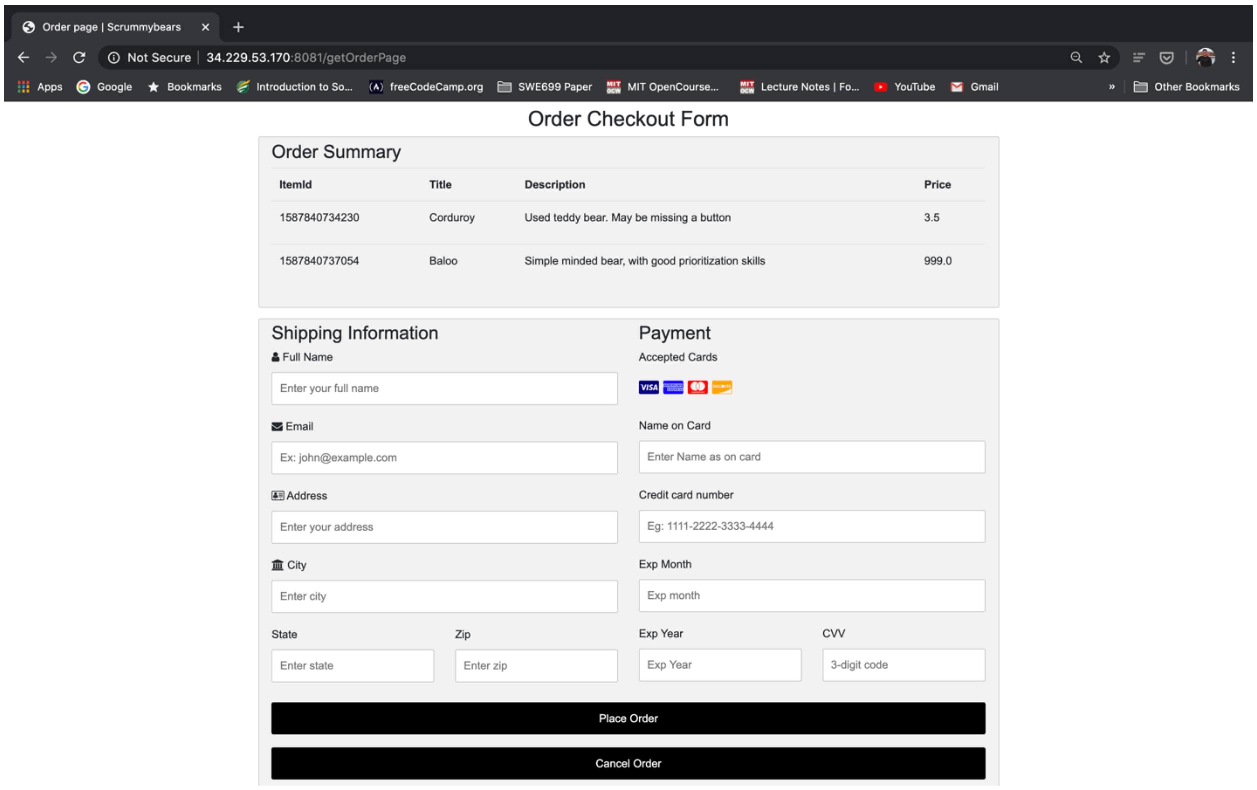
Credit card number: Eg: 1111-2222-3333-4444

Exp Month: Exp month

Figure 7: Order UI with popup

Figure.7 represents the user interface once you have landed on the order function page. There are two options to checkout.

- Continue and checkout as a guest.
  - If the user clicks checkout as guest, then he would be taken to the user interface shown on figure 8.
- Login/signup and then checkout.
  - If the user clicks login/signup, then, the order service will call the account service's GET API to get the account function's user interface.



Order Checkout Form

Order Summary

ItemId	Title	Description	Price
1587840734230	Corduroy	Used teddy bear. May be missing a button	3.5
1587840737054	Baloo	Simple minded bear, with good prioritization skills	999.0

Shipping Information

Full Name	Enter your full name
Email	Ex: john@example.com
Address	Enter your address
City	Enter city
State	Enter state
Zip	Enter zip

Payment

Accepted Cards

Name on Card	Enter Name as on card
Credit card number	Eg: 1111-2222-3333-4444
Exp Month	Exp month
Exp Year	Exp Year
CVV	3-digit code

Figure 8: Order UI / continue as guest

Figure.8 represents the order function's user interface when continued as guest.

- The order summary data is posted by calling order service's POST from cart service, the items are saved in order function's database.
- The items in the order summary can be deleted by calling order function's DELETE from the cart service.
- When the end user clicks proceed to check out in the cart service page, the cart service calls order's GET endpoint to render the dynamic information posted by the cart service.
- The user can fill in the shipping and payment information to place order. This calls order services POST endpoint to save it to the order's database.

CI/CD Pipeline

The pipeline for order service was implemented using Jenkins and SonarQube. It was demonstrated on sprint 3 review.

Pipeline Architecture

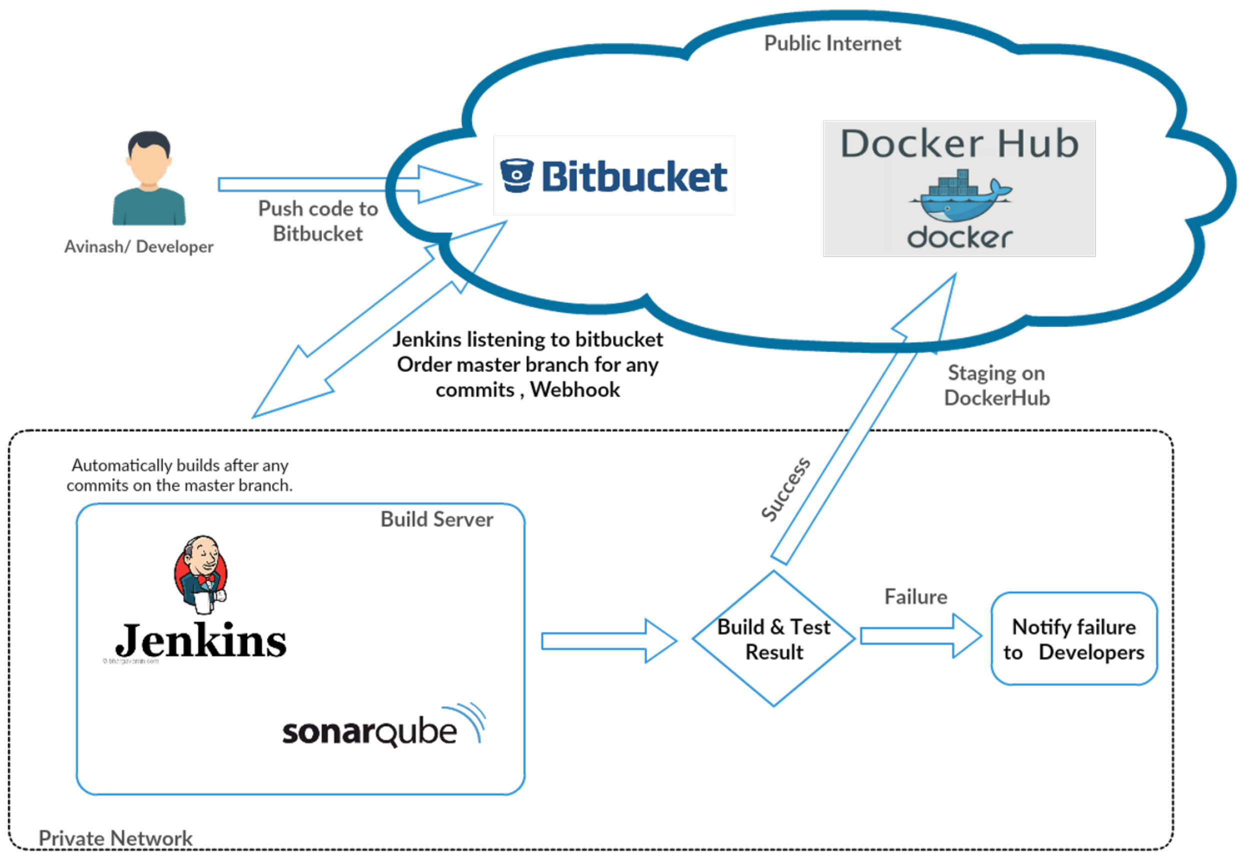


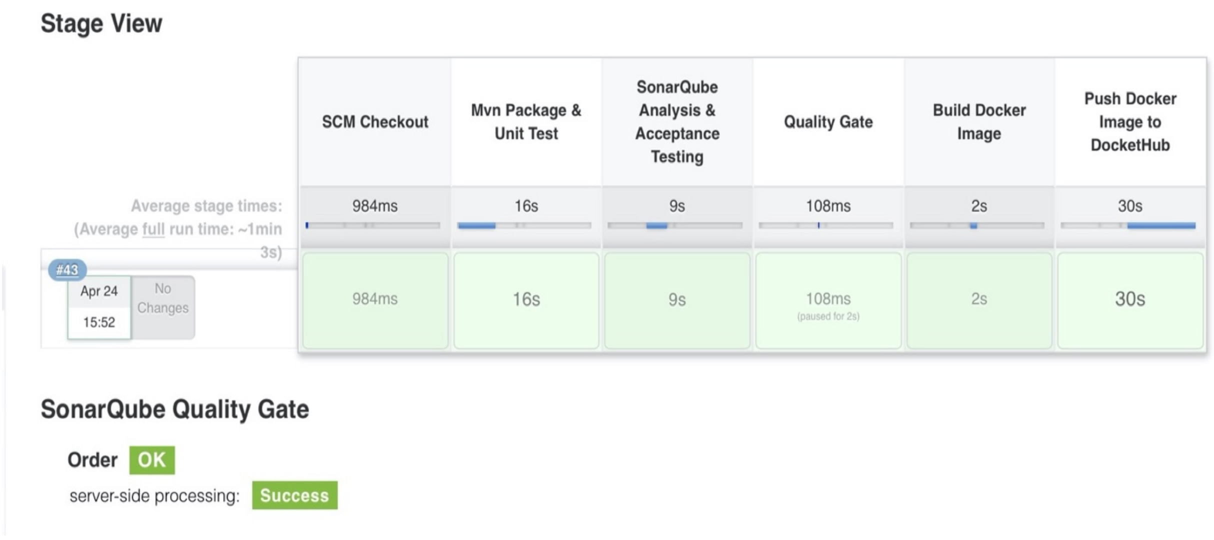
Figure 9: CI/CD Pipeline Architecture

A cloud repository of bitbucket was used to share code with the team. Since I didn’t have a way to run Jenkins and SonarQube on AWS cloud (the free-tier ec2 instance was not powerful enough to run both Jenkins and SonarQube), so I decided to create a private build server on my local machine. The build trigger was configured with webhook.

When the developer pushes code to Bitbucket (the code repo) it triggers Jenkins on my private network to start a build. If the build is success then it pushed the end result to a staging environment that is DockerHub, Else if the build fails it notifies the developer.

Jenkins

As mentioned before, Jenkins automatically runs the pipeline whenever code is pushed to the master branch in bitbucket repo. The pipeline has the following stages


Stage View

	SCM Checkout	Mvn Package & Unit Test	SonarQube Analysis & Acceptance Testing	Quality Gate	Build Docker Image	Push Docker Image to DocketHub
Average stage times: (Average full run time: ~1min 3s)	984ms	16s	9s	108ms	2s	30s
#43 Apr 24 15:52 No Changes	984ms	16s	9s	108ms (paused for 2s)	2s	30s

SonarQube Quality Gate

Order OK

server-side processing: Success

Figure 10: Jenkins Stage View

If either the unit tests fail, or the SonarQube analysis detects code quality is below a certain limit, the build will be aborted. This way, the quality can be assured while maintaining a fast delivery cycle. Once the image is pushed to DockerHub, it is considered staged for production, as the public EC2 instance can be updated to this image with two command line scripts.

The stages have been explained in the below table.

Stage	Description
SCM Checkout	Checks out the code from repo, so Jenkins can work with it.
MVN Package & Unit Test	Compiles the by running all the unit test and packs it into .jar using maven.
SonarQube Analysis & Acceptance testing	Analysis's the code coverage and performs static analysis using SonarQube with the default rules.
Quality Gate	This stage will abort the process if the results of the acceptance test is below acceptable limit.
Build Docker Image	Creates a new docker image with the compiled .jar file
Push Docker Image to DockerHub	Pushes the build docker image to DockerHub, staging it for production

Table 2: Pipeline Stage

SonarQube

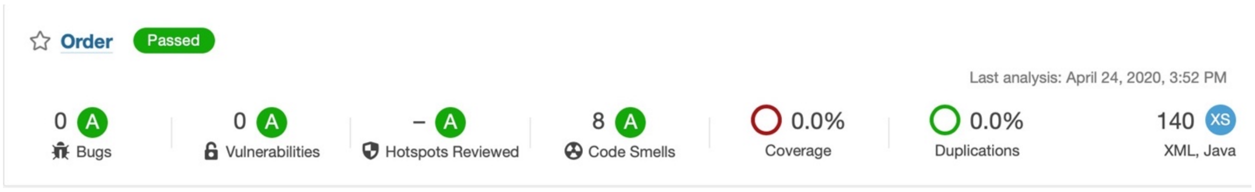


Figure 11: SonarQube Report

As mentioned before, SonarQube has been integrated into the Jenkins pipeline to perform code analysis and acceptance tests. The limit being used are standard thresholds provided by SonarQube which includes various code quality attributes such as bugs, vulnerabilities, hotspots reviewed, duplicates and code smells. From figure 9, you can see that the acceptance test and analysis has been passed. The reason coverage is 0.0 is because the code coverage rule was flexed. Since the unit tests are automatically being executed by Jenkins, and there was no code coverage requirement for this assignment, I did not pursue further in this matter.

Running the Service with Docker

The order service is designed to be run using docker to promote portability, since docker is known for its portability. The service is running using one container: The Spring MVC application. The order service is running in detach mode inside the instance on port 8081, So without any session the order service will be up always until the instance itself is terminated.

The Order application is ran using the following command after it has been pushed to the staging phase.

```
# docker run -it -d --rm -p 8081:80 --name OrderService
sai288/my-app:latest
```