COMP30220 Online Shopping

Saoirse Houlihan, James Kirwan, Martynas Jagutis, Caoimhe Tiernan, Adam Conway

University College Dublin

January 5, 2021

Table of Contents

Chapter 1: **Introduction**

The aim of our distributed application was to employ a number of technologies (both learned throughout the COMP30220 course, and self-taught) in order to develop a multi-module shopping website. *Yeet Shop Swedish* is the name of the website, where sellers can create accounts and list items for sale, whereas users can create accounts and purchase items that are listed for sale from other sellers. The project is split into several modules, wherein the whole system can still function even if certain, non-core modules are removed. Below is a list of all modules and their port numbers.

Module	**Port number**
Aggregator	8080
Consumer	8081
Stock	8082
Seller	8083
Cart	8084
Web	8085
Eureka	8761

Chapter 2: Setup

2.1 With Docker

To run with Docker, enter the project root and do the following:

1. mvn clean compile
2. mvn install
3. docker-compose up –build

Warning: the services may take a long time to propogate and register with the Eureka server, so please wait to ensure that all services are connected and registered before continuing to interact with any of them.

2.2 Without Docker

To run without Docker, enter the project root and do the following:

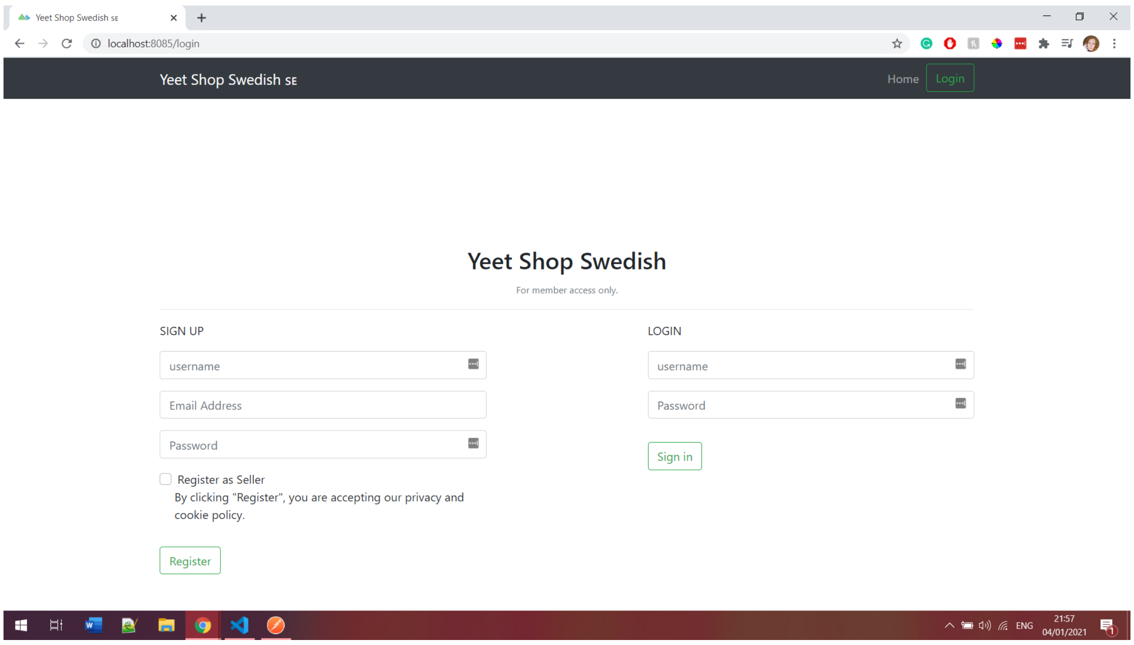
1. mvn clean compile
2. mvn install
3. mvn spring-boot:run -pl eureka
4. mvn spring-boot:run -pl aggregator
5. Then, start the other services in any order using the same steps as above, but naming the other services instead.

Chapter 3: Aggregator

3.1 Overview

The aggregator module effectively acts as an API gateway for an entire application, and is made up of a Zuul server which connects to our Eureka server. It configures the dynamic routing between each of our microservices, so that they can all communicate with each other.

Chapter 4: **Consumer**



Yeet Shop Swedish

For member access only.

By clicking "Register", you are accepting our privacy and cookie policy.

http://localhost:8081

4.1 Overview

The consumer module manages all consumers, and keeps a database of registered users. It does this by creating consumer objects (*Consumer.java*) which are stored in our consumer repository. The consumer repository extends a JPA repository for most of its functionality. The Consumer object contains the following information:

- UserID, of type integer
- Username, of type String
- Password, of type String
- Role, of type string

The role identifies whether a user is a shopkeeper or a consumer. While we are reading from the consumer module (and so, all users are expected to be a consumer), we keep track of the role so that once this consumer object leaves the consumer module, other modules can still understand what type of user this is.

All registrations for consumers go through this module. A controller exists which maps the following endpoints:

1. **/all-consumers**

   This endpoint will return all registered consumers in the repository.

2. **/exists**

   This endpoint will check if a consumer with the same username and password already exist in the repository.

3. **/username-exists**

   This endpoint will check if a consumer with the same username already exists in the repository.

4. **/add-consumer**

   This endpoint saves a signed up user to the repository.

5. **/get-consumer/{id}**

   This endpoint can be used to view an existing consumer manually by its ID, and to return the object so that it can be parsed and data can be viewed.

4.2 Technologies used

In the consumer module, we made use of a JPA repository for storing user data. Users stored in the JPA repository are of the Consumer type. This module operates through Spring Boot.

Chapter 5: Stock

`http://localhost:8082`

5.1 Overview

The stock module handles all stock, keeping a repository of the stock, and the quantity of the stock available. When an item is purchased, the stock is decreased by 1 and saved back to the repository again. This module also allows the modification of the description of items, and handles the creation of new item objects. Items are created in the core, and they contain the following information:

- ItemID, of type integer
- Name, of type String
- Category, of type String
- Description, of type String
- Price, of type double
- Quantity, of type integer

Item is a core object, as items are crucial to the entire system, and are used in some way, shape, or form in all modules. Therefore, we need to ensure that they are included in the building of each module, and are referenced correctly in each. In this module, we have the following endpoints:

1. **/items**

   This endpoint will return all items in the repository.

2. **/items/create**

   This endpoint can create a new item, saving the item to the repository.

3. **/remove/{id}**

   This endpoint removes an item from the stock database.

4. **/item/{id}**

   This endpoint can be used to view an existing item manually by its ID, and to return the object so that it can be parsed and data can be viewed.

5. **/items/{category}**

   This endpoint can be used to view all items in a particular category.

6. **/items/categories**

   This endpoint is used to determine all the unique categories in our database.

7. **/items/modify/{id}**

   This endpoint can be used to modify the description of any given item.

8. **/bought/{id}**

   This endpoint is used to decrease the quantity of the available stock, once an item is purchased.

9. **/return/{id}**

   This endpoint is used to increase the quantity of available stock if a consumer decides to remove it from their cart.

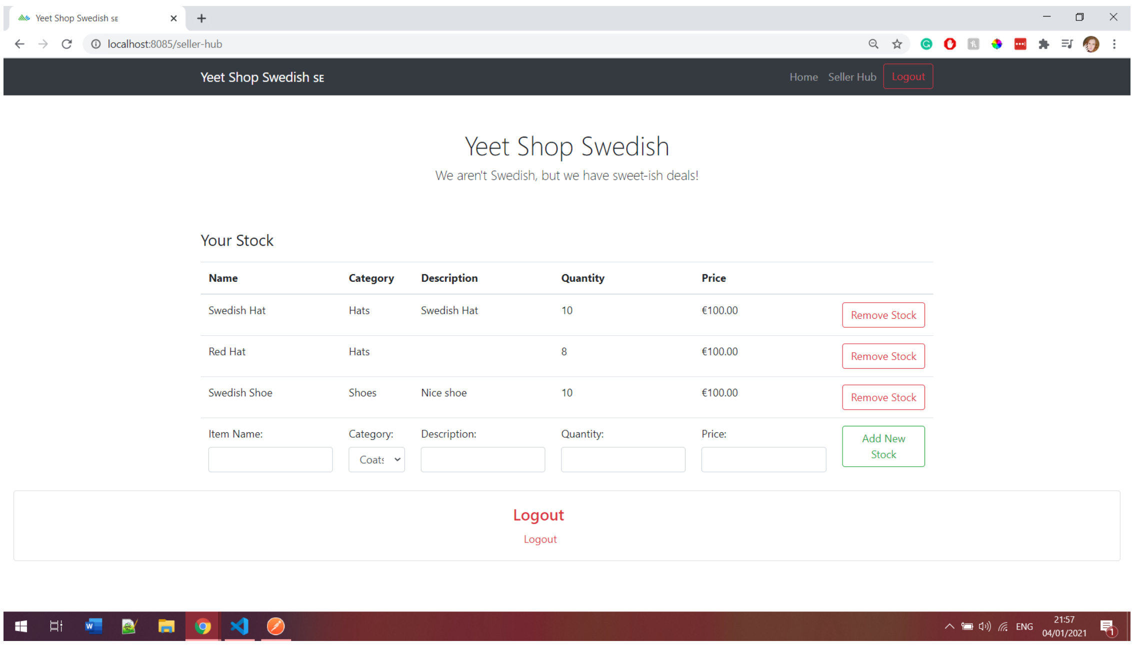
10. **/seller-stock/{id}**

    This endpoint finds all the stock belonging to a seller.

5.2 Technologies used

In the stock module, we made use of a JPA repository for storing item data. All data is stored in a H2 SQL database.

Chapter 6: **Seller**



Yeet Shop Swedish SE

localhost:8085/seller-hub

Yeet Shop Swedish SE

Home Seller Hub Logout

Yeet Shop Swedish

We aren't Swedish, but we have sweet-ish deals!

Your Stock

Name	Category	Description	Quantity	Price	
Swedish Hat	Hats	Swedish Hat	10	€100.00	Remove Stock
Red Hat	Hats		8	€100.00	Remove Stock
Swedish Shoe	Shoes	Nice shoe	10	€100.00	Remove Stock
Item Name:	Category: Coats	Description:	Quantity:	Price:	Add New Stock

Logout

Logout

21:57 04/01/2021

http://localhost:8083

6.1 Overview

The seller module handles the management of sellers, with a repository filled with seller objects. A seller contains the following information.

- SellerID, of type integer
- Username, of type String
- Password, of type String
- Role, of type string

The role identifies whether a user is a shopkeeper or a consumer. While we are reading from the seller module (and so, all users are expected to be a seller), we keep track of the role so that once this seller object leaves the seller module, other modules can still understand what type of user this is.

1. **/all-sellers**

   This endpoint will return all sellers in the repository.

2. **/exists**

This endpoint will check if a seller with the same username and password already exists in the repository.

3. **/username-exists**

   This endpoint will check if a seller with the same username already exists in the repository.

4. **/add-sellers**

   This endpoint will add a seller object to the repository.

5. **/get-seller/{id}**

   This endpoint can be used to view an existing seller manually by its ID, and to return the object so that it can be parsed and data can be viewed.

6.2 Technologies used

In the seller module, we made use of a JPA repository for storing user data. Users stored in the JPA repository are of the Consumer type. This module operates through Spring Boot.

Chapter 7: Cart

http://localhost:8084

7.1 Overview

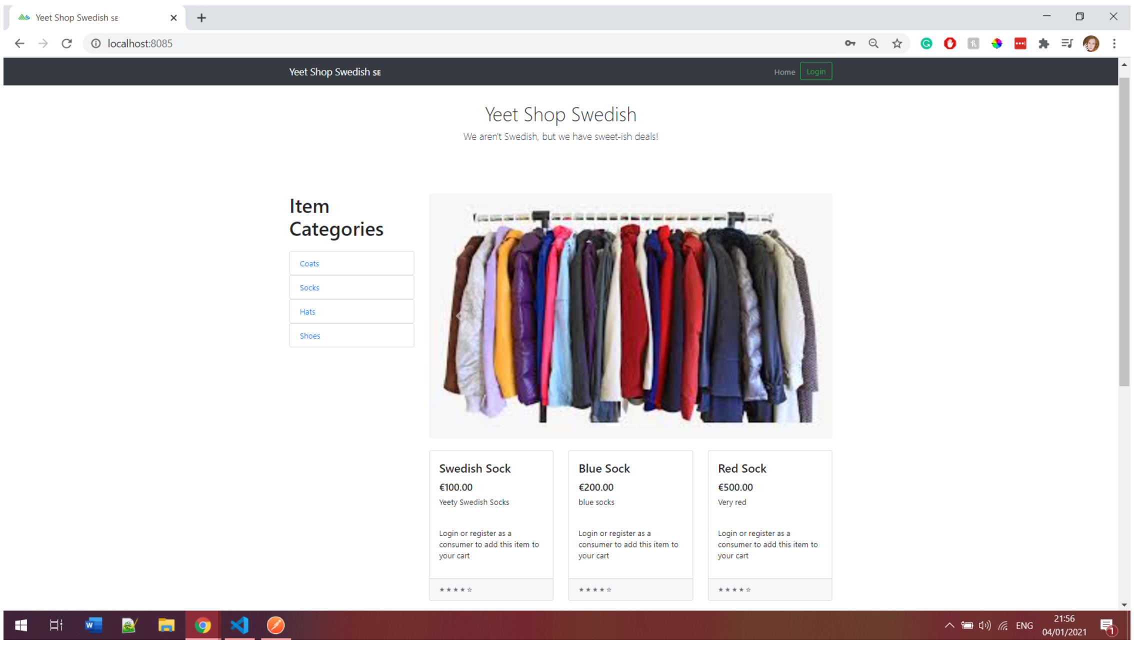
The cart module handles the management of carts, with a repository filled with item objects and user ID references. The cart module can check whether an individual item exists in a user's cart, and can also return every item in the user's cart.

1. **/cart**
   This endpoint will return the cart of a user.
2. **/already-in-cart/{user}/{item}**
   This endpoint will check if an item is already in the cart of a specific user.
3. **/get-cart/{user}**
   This endpoint will return the cart of a particular user.
4. **/get-cart/{user}/{item}**
   This endpoint will return one specific item from a user's cart.
5. **/add-to-cart**
   This endpoint will add an item to a user's cart.
6. **/remove-from-cart**
   This endpoint will remove an item from a user's cart.
7. **/update-cart**
   This endpoint will update a cart object.

7.2 Technologies used

In the seller module, we made use of a JPA repository for storing cart data. This module operates through Spring Boot.

Chapter 8: **Web**



Yeet Shop Swedish

We aren't Swedish, but we have sweet-ish deals!

Item Categories

Swedish Sock
€100.00
Yeety Swedish Socks
Login or register as a consumer to add this item to your cart

Blue Sock
€200.00
blue socks
Login or register as a consumer to add this item to your cart

Red Sock
€500.00
Very red
Login or register as a consumer to add this item to your cart

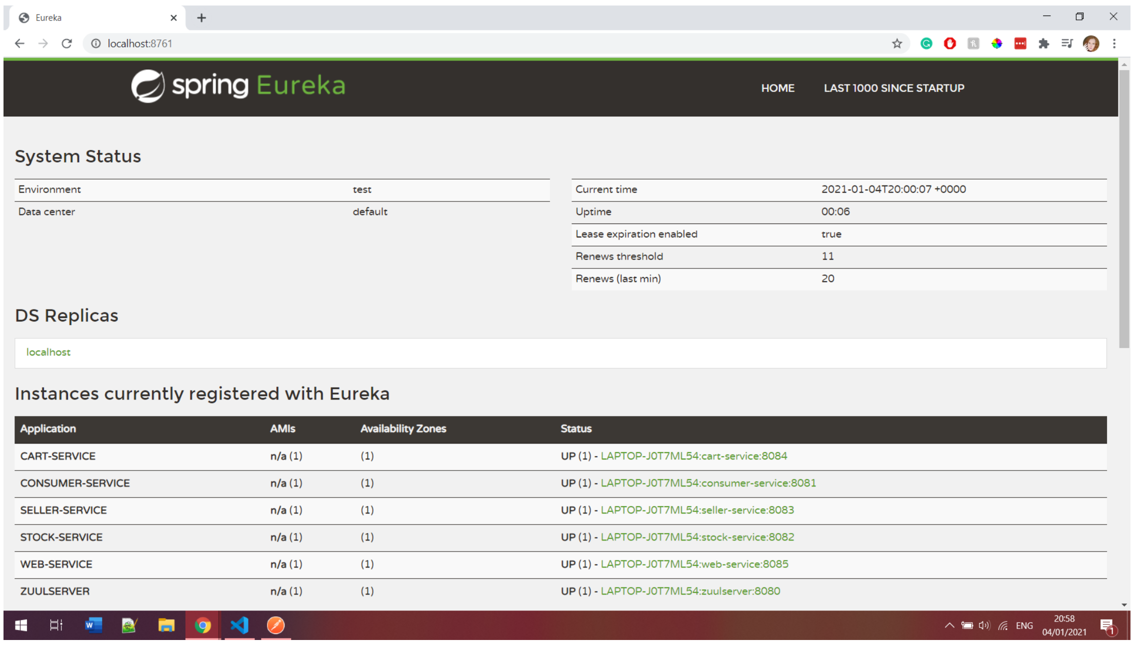
http://localhost:8085

8.1 Overview

The web module handles the delivery of the web site, containing all of the website's HTML pages and controllers. Pages are designed using a Bootstrap template, and data is read into the pages from each of the various databases and displayed using Thymeleaf. Through the website, we can handle logins and admin controls, wherein the admin controls effectively act as a GUI for the back-end. For consumers, the front-end can be used for purchasing items. Users can view items, add them to their cart, and then purchase them. Purchasing them through the website will decrease the stock of the item.

Through the web module, we access pretty much every other module, either directly or indirectly.

Chapter 9: **Eureka**



System Status

Environment	test
Data center	default

Current time	2021-01-04T20:00:07 +0000
Uptime	00:06
Lease expiration enabled	true
Renews threshold	11
Renews (last min)	20

DS Replicas

localhost

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
CART-SERVICE	n/a (1)	(1)	UP (1) - LAPTOP-J0T7ML54:cart-service:8084
CONSUMER-SERVICE	n/a (1)	(1)	UP (1) - LAPTOP-J0T7ML54:consumer-service:8081
SELLER-SERVICE	n/a (1)	(1)	UP (1) - LAPTOP-J0T7ML54:seller-service:8083
STOCK-SERVICE	n/a (1)	(1)	UP (1) - LAPTOP-J0T7ML54:stock-service:8082
WEB-SERVICE	n/a (1)	(1)	UP (1) - LAPTOP-J0T7ML54:web-service:8085
ZUULSERVER	n/a (1)	(1)	UP (1) - LAPTOP-J0T7ML54:zuulserver:8080

`http://localhost:8761`

9.1 Overview

Eureka is a microservice used for registering all other microservices. This allows us to have all services communicate with each other *without* the need to hardcode an IP address and port in each service. The Eureka service is the only fixed point which all of our microservices will register with, and Eureka will handle the connections between each thereafter. This is client-side server discovery, and allows our client microservices to find each other without any direct user input.

Appendix

Unit tests

	Class %	**Method %**	**Line %**
All classes	100% (4/4)	77.4% (24/31)	68.8% (53/77)
cart.controller	100% (1/1)	75% (6/8)	71.4% (10/14)
consumer.controller	100% (1/1)	83.3% (5/6)	77.8% (7/9)
seller.controller	100% (1/1)	83.3% (5/6)	77.8% (7/9)
stock.controller	100% (1/1)	72.7% (8/11)	64.4% (29/45)