Krispy Koders

EECS 4413: Building E-Commerce Systems

Team Contributions

Arjit Johar

Contributions
Worked on implementing REST for the backend testing with postman alongside Melvin.

Learned about the project from other member's work
I was able to learn from Varuhn and Sharujan how to construct the application that was used to connect the frontend with the backend of our application. They were able to construct and create each java and html file so that they were connected to operate as a system. I was also able to learn from Melvin how to further implement the REST framework for backend testing.

Melvin Gagarao

Contributions
Worked on implementing REST for the backend testing with postman alongside Arjit. Also helped with the construction of the diagrams used in the final document.

Learned about the project from other member's work
I was able to learn how to construct the application from Varuhn and Sharujan that was used to connect the frontend with the backend of our application. They were able to construct and create each java and html file so that they were connected to operate as a system. I was also able to learn from Arjit how to further implement the REST framework for backend testing.

Sharujan Rajakumar

Contributions
I worked on creating and modifying the majority of the java files along with Varuhn. I also helped on setting up the connection between the MySQL database and spring by incorporating a MySQL driver to the applications.properties file. I helped with creating a database schema to manage our products, orders, order_details in the shopping cart. I helped on designing and improving the frontend system of the web application. I helped

implement the searching by brand or type functionality for the productList.html file. I attempted to deploy our web application on AWS Elastic Beanstalk, Google Cloud, and Heroku.

Learned about the project from other member's work
I was able to learn from Varuhn how to implement the shopping cart details into a table and how to view the total transactions each customer has made. I was also able to learn from Melvins how to organise our coding styles into a diagram and how he and Arjit were able to implement REST for backend testing.

Varuhn Ruthirakuhan

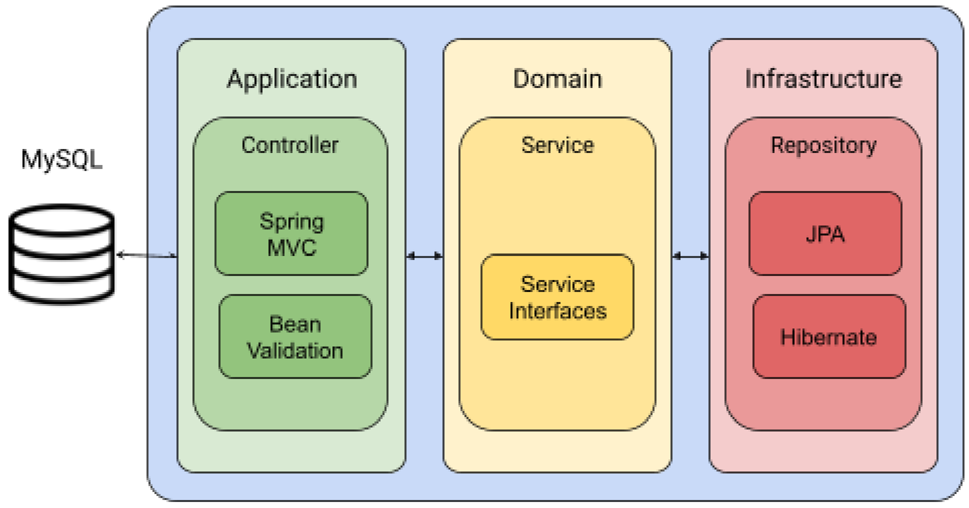
Contributions
Work on deciding which architecture framework to follow and implement for the project. Also worked on creating and modifying the majority of the java files along with Sharujan. Helped with the connection between the MySQL database and our java files so the application could access the database and display required contents for each product. Additionally worked on designing the security config that implemented the bCryptPassword encoder to protect the admin and managers profiles. Finally, worked on the implementation of the front end system with the html files as well as the final design of the front end application. Also attempted to deploy the application on many systems such as Google Cloud, AWS, Heroku, and Railway.

Learned about the project from other member's work
I was able to learn from Sharujan how to implement the shopping cart details into a table and how to view the total transactions each customer has made. I was also able to learn from Melvins how to organise our coding styles into a diagram and how he and Arjit were able to implement REST for backend testing.

Table Of Contents

Architecture

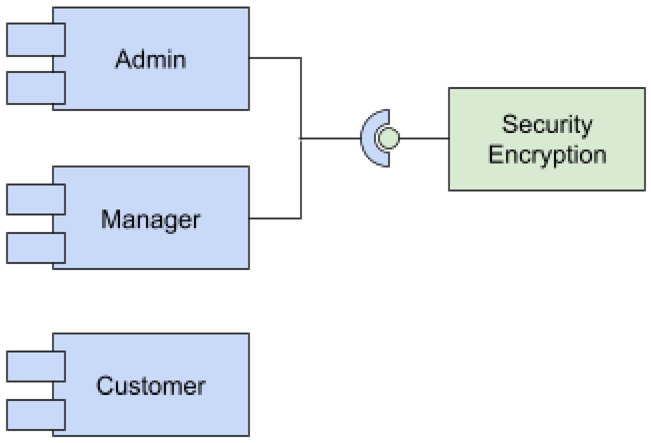
Overall System



System Components

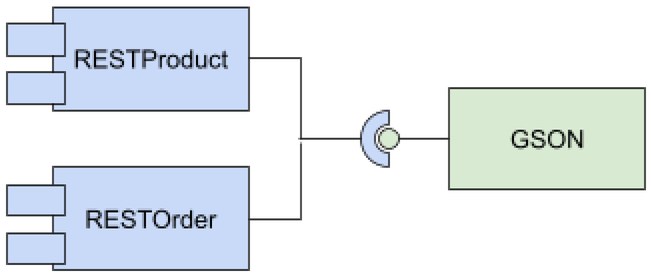
Login Verification System

We first stored the user name and encrypted password into the database. Then, we created a WebSecurityConfig.java file that has a BCryptPasswordEncoder method implemented to decrypt the password we encoded. This works by having the user log into the application, where the password entered is compared to the password encrypted in the database.



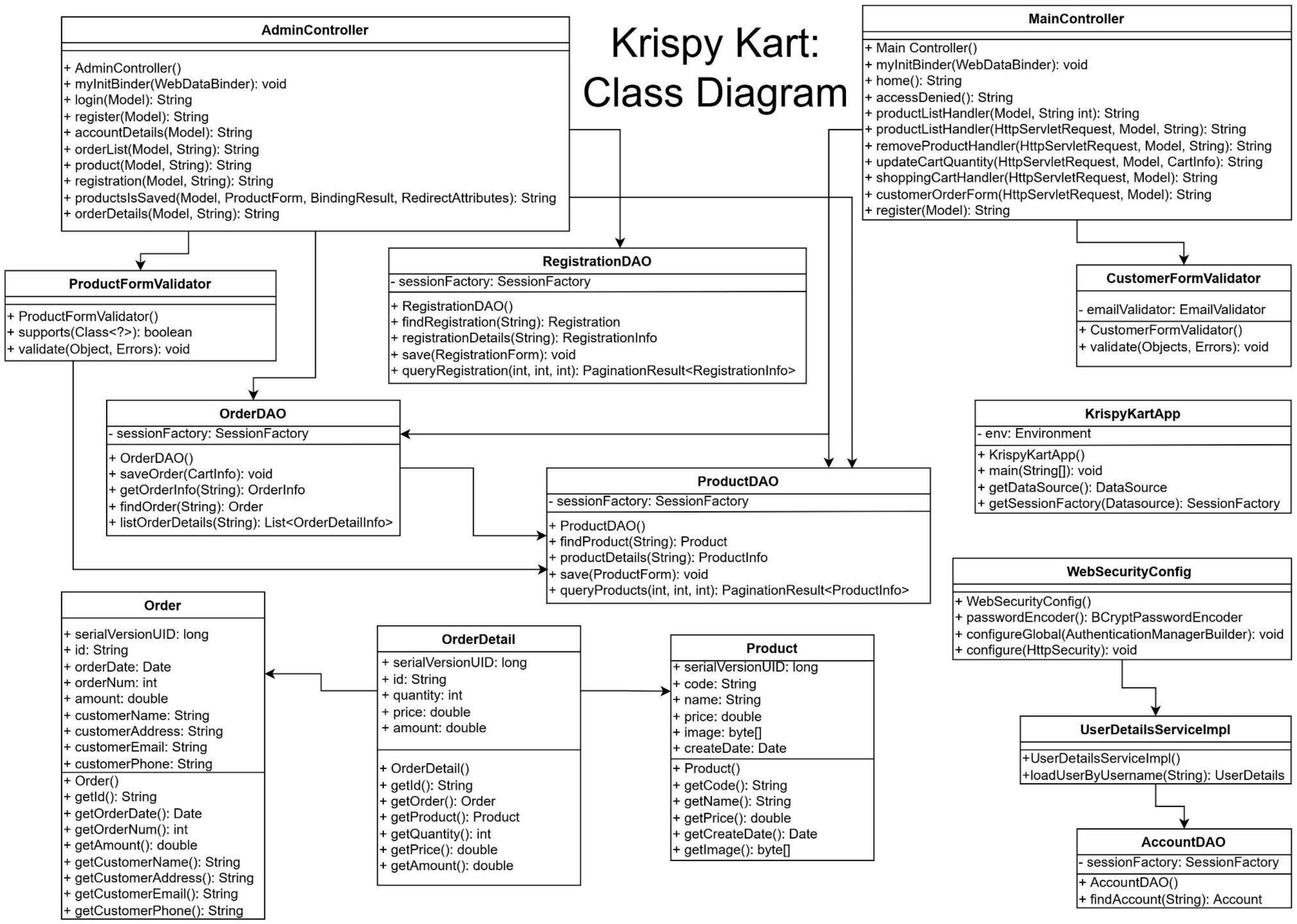
REST + GSON

The REST and GSON were coordinated in the project to ease the production and processing of JSON. toString methods in @Entity files classes made use of GSON's implementation.



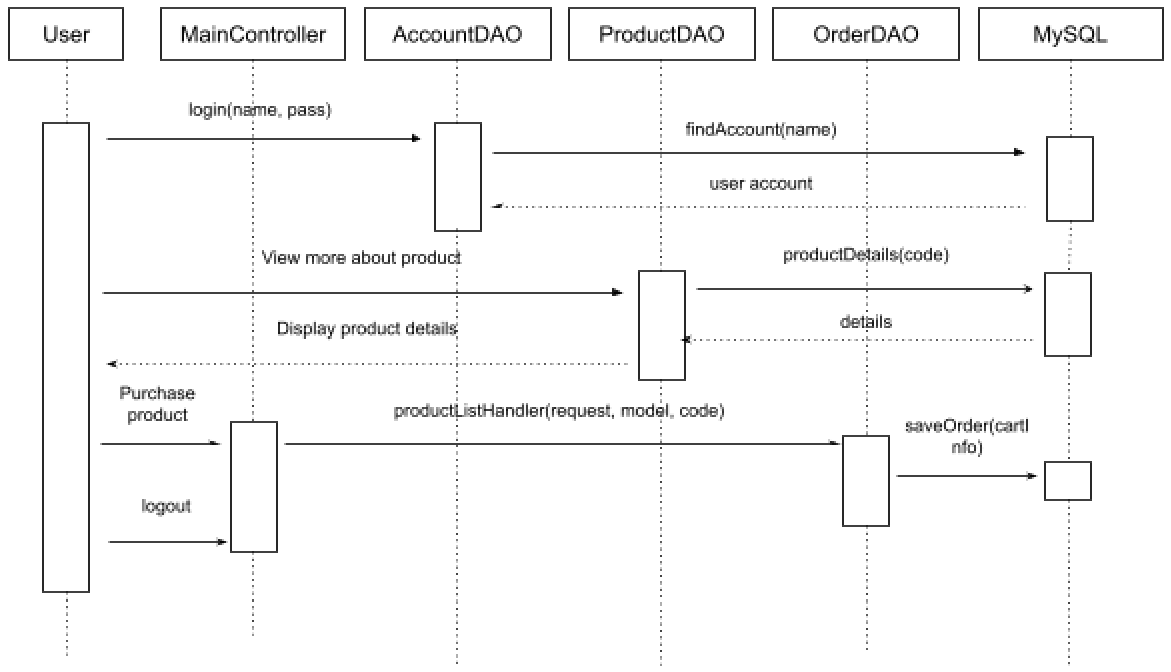
DAO
The DAO was used in our application to make it easier to access the data technologies Hibernate and JPA. The Exception Translation mechanism was enabled and active for all our DAO’s by using the @Repository annotation preceding the classes.

Class Diagram

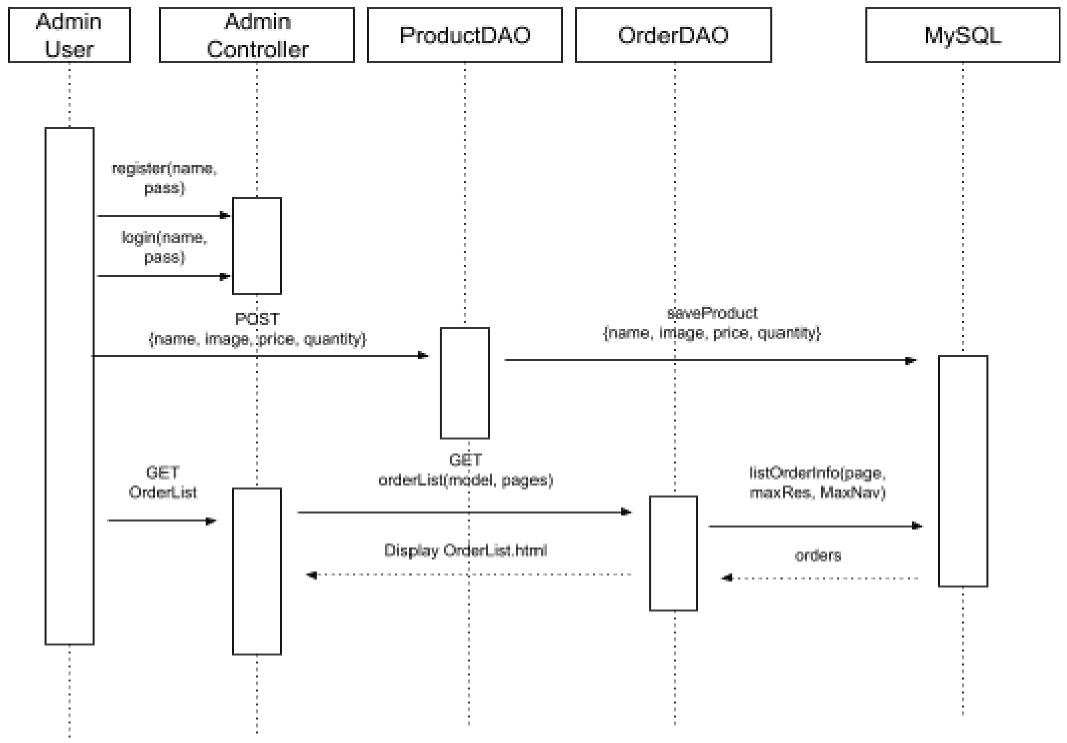

Krispy Kart: Class Diagram
AdminController
+ AdminController()
+ myInitBinder(WebDataBinder): void
+ login(Model): String
+ register(Model): String
+ accountDetails(Model): String
+ orderList(Model, String): String
+ product(Model, String): String
+ registration(Model, String): String
+ productsIsSaved(Model, ProductForm, BindingResult, RedirectAttributes): String
+ orderDetails(Model, String): String
MainController
+ Main Controller()
+ myInitBinder(WebDataBinder): void
+ home(): String
+ accessDenied(): String
+ productListHandler(Model, String int): String
+ productListHandler(HttpServletRequest, Model, String): String
+ removeProductHandler(HttpServletRequest, Model, String): String
+ updateCartQuantity(HttpServletRequest, Model, CartInfo): String
+ shoppingCartHandler(HttpServletRequest, Model): String
+ customerOrderForm(HttpServletRequest, Model): String
+ register(Model): String
ProductFormValidator
+ ProductFormValidator()
+ supports(Class<?>): boolean
+ validate(Object, Errors): void
RegistrationDAO
- sessionFactory: SessionFactory
+ RegistrationDAO()
+ findRegistration(String): Registration
+ registrationDetails(String): RegistrationInfo
+ save(RegistrationForm): void
+ queryRegistration(int, int, int): PaginationResult<RegistrationInfo>
CustomerFormValidator
- emailValidator: EmailValidator
+ CustomerFormValidator()
+ validate(Objects, Errors): void
OrderDAO
- sessionFactory: SessionFactory
+ OrderDAO()
+ saveOrder(CartInfo): void
+ getOrderInfo(String): OrderInfo
+ findOrder(String): Order
+ listOrderDetails(String): List<OrderDetailInfo>
KrispyKartApp
- env: Environment
+ KrispyKartApp()
+ main(String[]): void
+ getDataSource(): DataSource
+ getSessionFactory(Datasource): SessionFactory
ProductDAO
- sessionFactory: SessionFactory
+ ProductDAO()
+ findProduct(String): Product
+ productDetails(String): ProductInfo
+ save(ProductForm): void
+ queryProducts(int, int, int): PaginationResult<ProductInfo>
WebSecurityConfig
+ WebSecurityConfig()
+ passwordEncoder(): BCryptPasswordEncoder
+ configureGlobal(AuthenticationManagerBuilder): void
+ configure(HttpSecurity): void
Order
+ serialVersionUID: long
+ id: String
+ orderDate: Date
+ orderNum: int
+ amount: double
+ customerName: String
+ customerAddress: String
+ customerEmail: String
+ customerPhone: String
+ Order()
+ getId(): String
+ getOrderDate(): Date
+ getOrderNum(): int
+ getAmount(): double
+ getCustomerName(): String
+ getCustomerAddress(): String
+ getCustomerEmail(): String
+ getCustomerPhone(): String
OrderDetail
+ serialVersionUID: long
+ id: String
+ quantity: int
+ price: double
+ amount: double
+ OrderDetail()
+ getId(): String
+ getOrder(): Order
+ getProduct(): Product
+ getQuantity(): int
+ getPrice(): double
+ getAmount(): double
Product
+ serialVersionUID: long
+ code: String
+ name: String
+ price: double
+ image: byte[]
+ createDate: Date
+ Product()
+ getCode(): String
+ getName(): String
+ getPrice(): double
+ getCreateDate(): Date
+ getImage(): byte[]
UserDetailsServiceImpl
+UserDetailsServiceImpl()
+loadUserByUsername(String): UserDetails
AccountDAO
- sessionFactory: SessionFactory
+ AccountDAO()
+ findAccount(String): Account

Sequence Diagrams

User Login → Browse products → Enter customer info → Update order list → log out



Use Case 2 (Original Use Case)
Register → login → update products → view all orderlists


Admin User
Admin Controller
ProductDAO
OrderDAO
MySQL
register(name, pass)
login(name, pass)
POST {name, image, price, quantity}
saveProduct {name, image, price, quantity}
GET OrderList
GET orderList(model, pages)
listOrderInfo(page, maxRes, MaxNav)
Display OrderList.html
orders

Design Patterns

Singleton

The first design pattern that we used in our project was the singleton pattern. The singleton pattern guarantees that only one instance of a class is instantiated. By default, Spring creates all beans as singletons. For example, in our KrispyKartApp.java we used the @Autowired notation, which makes them singleton by default. The disadvantages of using a singleton pattern is that singleton patterns make unit testing very hard since you cannot completely isolate classes dependent on singletons. Also, the disadvantages of using a singleton pattern is that they create hidden dependencies. The advantages of using a singleton pattern is that singleton prevents other objects from instantiating their own copies of the singleton object, ensuring that all objects access the single instance.

Model View Controller (MVC)

The second design pattern that we used in our project was the MVC design pattern. The MVC pattern is a software design pattern that separates an application into three main logical components: the model (input logic), the view (business logic), and the controller (UI logic). The advantages of using the MVC is that it organizes large-size web applications and is easily modifiable. The disadvantage of using the MVC is that it can't be suitable for small applications.

Function as a Service (Fass)

The third design pattern that we used in our project was the Function as a Service (Faas) design pattern. The Faas design pattern is a serverless type of cloud computing that allows users to deploy applications in the cloud. The advantages of using the Fass is that there is increased developer productivity and faster development time. The disadvantage of using Faas is that it may be vulnerable to DoS.

Implementation

Implementation 1: Spring framework + Hibernate + Thymeleaf + MySQL

We chose the Spring framework as part of our stack because it provides a MVC architecture and ready components that can be used to develop flexible and loosely coupled web applications. The spring framework is a java framework that focuses on improving the speed, simplicity and productivity of the web application. We chose Hibernate, an object-relational mapping tool, because it helps reduce lines of code by maintaining object-table mapping itself and returns results to applications in the form of Java objects. We chose Thymeleaf, a Java library, because it provides an elegant and well-formed way of creating templates. We chose MySQL database because it's easy to use and is a high-performance but relatively simple relational-database system, and it's much less complex to set up and administer than larger systems.

The advantages of choosing this stack is that it provides a comprehensive infrastructure support for developing Java based applications. However, its disadvantages are that working with Spring is more complex and requires a lot of expertise.

Implementation 2: Main dependencies used to configure our project

Certain packaging and imports were required in order to make our application accessible. For example, the war packaging was required so our application could produce a war file to push to the cloud after running the maven build. Additional dependencies were also included to help with the functionality with our app. One example includes the use of the mysql-connector-java dependency so that we could seamlessly collect data stored in our backend database and display it on our front end application.

Limitations

Limit 1: Testing for accuracy of images

One primary limitation with our testing was testing for the accuracy of images on certain products on our web application. Since the images were downloaded onto harddrive and then stored on the database, we were unable to check for duplicates. It was also difficult to test that the images were stored in the correct products within the product list.

Limit 2 : Testing for registration

Another limitation with our testing was that the user can only register with a normal username and password. The admin and manager have a bcrypt password because their profiles were previously made and stored into the database. This leaves possible security flaws since the password stored in the database for newly registered users does not get encrypted.

Security

Vulnerability Testing

SQL Injection

URL queries are String parsed for sql keywords and operators before they are concatenated and passed to the MySQL database.

Bcrypt Password Encoder
Admin and manager password are stored using a bcrypt encoder. This was done to protect the passwords if a threat were to go through and access the backend of our database. Once the bcrypt password is stored in the database, the application will use a bcrypt encoder to read the password as a normal string to check that the user's password is valid.

Untested Threats
These threats are deemed as unlikely scenarios with difficult preventative measures beyond the scale of the e-commerce website.

Cross Site Scripting
Inserting malicious scripts into html components to gain access to unintended data

Path traversal
Inserting patterns into the web server hierarchy to access databases, configuration files, and other information stored on hard drives.

Distributed Denial of Service
Attacker bombards the server with an excessive amount of requests.

Performance Testing
REST API Error Testing
GET requests were placed for nonexistent data
POST requests were made for existing data

Load Testing
100 simultaneous virtual users periodically made randomly generated GET requests to ../products and ../orders. Server response times were under 100 ms.

1…N simultaneous virtual users periodically making GET requests caused a significant lag spike around 10,000 users onwards.

Appendix

REST API
The HTTP methods: POST, GET, PUT, DELETE correspond to the Create, Read, Update and Delete (CRUD) operations. REST endpoints were created for products and orders.

CREATE: used for creation of a new resource on the server.
READ: used for fetching details from the server (read only operation)
UPDATE: used to update the old/existing resource on the server or to replace the resource.
DELETE: used to delete the resource on the server.

Products
GET ../products → [{"code": "Apple2", "image": "4AAQS", "name": "Apple iPhone 13 Pro", "price": 1399.0, "date": "2022-04-15"}, {"code": "BlackBerry", "image": "Ih2dWVkAAA", "name": "BlackBerry KEY2", "price": 1189.0, "date": "2022-04-16"}
}, ...]

GET ../products{Google} → [{"code": "Google", "image": "9KWVIgABA", "name": "Pixel 2", "price": 1129.0, "date": "2022-04-17"}]

POST ../products{"code": "Fake", "image": "9KWVIgABA", "name": "Pixel 2", "price": 1129.0, "date": "2022-04-17"} → Adds to MySQL database if not already present

DELETE ../products{"code": "Google", "image": "9KWVIgABA", "name": "Pixel 2", "price": 1129.0, "date": "2022-04-17"} → Removes matching product

Orders
GET ../orders → [{"id": "0ed3166a-5309","amount": 989.0, "customer_address": "94 Headquarters Road", "customer_email": "shield@email.com", "customer_name": "Steve Rogers", "customer_phone": "9687590143", "order_date": "2022-04-17", "order_num": 5}, {"id": "1163bfcb", "amount": 4730.389999999999, "customer_address": "77 Crossbow Road", "customer_email": "dixon@email.com", "customer_name": "Daryl Dixon", "customer_phone": "2345678901", "order_date": "2022-04-16", "order_num": 2} }, ...]

GET ../orders{0ed3166a} → [{"id": "0ed3166a","amount": 989.0, "customer_address": "94 Headquarters Road", "customer_email": "shield@email.com", "customer_name": "Steve Rogers", "customer_phone": "9687590143", "order_date": "2022-04-17", "order_num": 5}]

POST ../orders{"id": "iu7d78","amount": 989.0, "customer_address": "94 Headquarters Road", "customer_email": "shield@email.com", "customer_name": "Steve Rogers", "customer_phone": "9687590143", "order_date": "2022-04-17", "order_num": 5} → Adds to MySQL database if not already present

DELETE ../orders{"id": "iu7d78","amount": 989.0, "customer_address": "94 Headquarters Road", "customer_email": "shield@email.com", "customer_name": "Steve Rogers", "customer_phone": "9687590143", "order_date": "2022-04-17", "order_num": 5} → Removes matching product