Design patterns

Builder

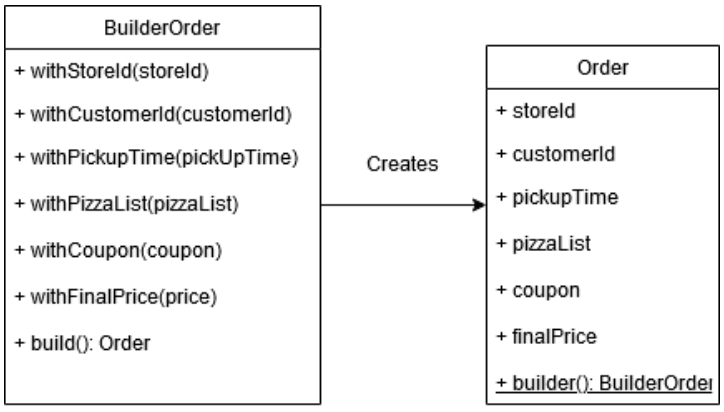
The intent of the Builder pattern is to separate the construction of a complex object from its representation. We are using the builder design pattern for the Order class, in order to provide a convenient and flexible way to create an ‘Order’ object. An ‘OrderBuilder’ class is implemented which contains a set of methods that allow the attributes of the ‘Order’ object class to be set in a way that is different to that of a traditional constructor. This set of methods involves a method for each attribute that performs the setting of the attribute when used in conjunction with the ‘build()’ method. This ‘build()’ method allows us, the developers, to create these objects in one single step instead of having to first create a new object and then set each of its attributes manually using setters.The attributes can be set in any order, and not all of them have to necessarily be set manually at object creation. This allows us to create ‘Order’ objects by only setting the fields that are necessary, without having to provide all attribute values at once, making object creation more flexible and also greatly simplifying testing. Overall, the use of the Builder pattern on our Order class makes the creation and configuration of ‘Order’ objects simpler and more convenient.

An example of how the builder pattern is used in object creation can be seen below:

```
order = Order.builder()
        .withStoreId(1L)
        .withCustomerId("Matt")
        .withPickupTime(ldt)
        .withPizzaList(List.of(pizza1, pizza2))
        .withCoupon("ABCD12")
        .build();
```

As shown on the image, each attribute is set using ‘with[attributeName]’ methods in combination with the ‘build()’ method at the end.
This way of object creation is visibly more concise, flexible, and convenient.

The UML diagram for the Builder pattern is as shown below:



Facade

The facade design pattern that we use is represented by our RequestHelper class. The reason we use the facade pattern is to greatly simplify microservice interactions, by hiding the complex functionality behind this class. The client will call a microservice and indirectly call the RequestHelper. At the same time, a microservice requesting from another microservice could be considered a client on its own.
Because we are using this RequestHelper we are not interested in the implementation detail of how the actual HTTP messages are sent. For instance spring provides RestTemplate but also HttpClient classes to make the actual requests. Right now we are using RestTemplate, but we can easily switch to HttpClient without the other microservices noticing.
The RequestHelper uses a RestTemplate in order to return ResponseEntities from one microservice to another. The RequestHelper provides methods for requests that can be called and by specifying the port of the microservice to interact with, the path of the request, an object to be sent in case of a POST request, and the class of the expected result we will obtain only the necessary information from another microservice.

An example of a GET request:

```
Does a GET request to another microservice.
Params: port – the port that the other microservice is running on. Check application.properties of the
        other microservice to see the port number
        path – the path of the other microservice /store for instance
        responseClass – the class type of the response we expect. If the other microservice sends String
        then it String.class. If the other microservice send List then it is List.class
Returns: a spring Response entity
public <T> T getRequest(int port, String path, Class<T> responseClass) {
    return doRequest(HttpMethod.GET, port, path, toSend: null, responseClass);
}
```

The doRequest(...) method is private and returns the body of the ResponseEntity generated by requesting from another microservice. This class also uses a Logger as well to log the requests made as well as an Authentication Manager to obtain the JWT Token.
By using the RequestHelper, we not only make interacting with other microservices easier, but we also avoid a lot of code duplication. The RequestHelper also hides away the dependency of the RestTemplate which can be replaced in the future by any other HTTP client that makes requests.

The functionality that is abstracted is available for get, post and delete requests. The method in the image below shows the functionality which is hidden from the controllers. This code is lengthy and complex and having the Facade provided by the Request Helper is a great way to simplify our workload, avoid code duplication and most importantly have a consistent system of achieving proper interaction between each microservice. The doRequestWithResponse() method returns an object of type ResponseEntity<T>. However, we have a secondary method as can be seen in the image

above, doRequest(), which only returns the object of type <T>, which is ultimately what we want to get from other microservices.

```java
private <T> ResponseEntity<T> doRequestWithResponse(HttpMethod httpMethod, int port, String path, Object toSend,
                                                    Class<T> responseClass) {
    URI url = createUrl(port, path);
    logger.info("Doing a " + httpMethod.toString() + " on " + url);

    try {
        RequestEntity.HeadersBuilder<?> request =
            RequestEntity.method(httpMethod, url).accept(MediaType.APPLICATION_JSON);
        var requestWithToken : RequestEntity<...>.HeadersBuilder<...> = request.header( headerName: "Authorization", ...headerValues: "Bearer " + getAuthenticationToken());
        if (toSend != null) {
            // add to post request the object
            var postRequest : RequestEntity<Object> = ((RequestEntity.BodyBuilder) requestWithToken).body(toSend);
            return new RestTemplate().exchange(postRequest, responseClass);
        }
        return new RestTemplate().exchange(request.build(), responseClass);
    } catch (ResourceAccessException connectException) {
        logger.error("The other microservice can't be reached. Check if port is ok or the path is ok.");
        throw new IllegalArgumentException("The url " + url
            +
            " is not valid,copy url to postman to check. Maybe the other server is not running or the path is not good");
    }
}
```