Software Architecture

The software architecture for the project is the Microservice Architecture featuring three microservices with distinct responsibilities. There is the “main” microservice, the “auth” microservice, and the “fridge” microservice. These three interact with the user and each other to satisfy the requirements of the project.

1. Main Microservice

The Main microservice acts as a “hub” for the application, and serves as the brains of the operation. It is where the user will go after account creation if they want to login to the application. This is done by the user providing their username password pair, which is then sent to Auth through Main. If a valid JWT token is returned, and this is the first time the user has been seen, Main will create a new user in the Main database with an initial credit balance of 0. From here, the user has quite a few options to interact with the Main microservice.

Authenticated users can get a list of the users in their household, get a list of flagged users, reset the credit balance of the entire household, check their own flagged status, their own credit balance, and delete their account from both Main and Auth.

Main also acts as the calculation brain of the product. It will asynchronously handle events from Fridge by subscribing to a Redis pub/sub topic. There are various events that can occur, such as a product expiring, being used up, or being removed. These events all require different approaches as to how credits are added/removed, and this is all handled on the Main microservice.

We felt it was good to have a more “front-end” microservice that serves the most relevant information for the users, and that acts as the portal through which login occurs.

2. Auth Microservice

The Auth microservice is responsible for the security aspect of the application. It is where a user will initially go to create a new Account, and communicates with main for things like logging in, deleting an account, etc.

When creating a new user, Auth asks for a username and password in the form of a JSON object. Assuming the username is unique, the password is hashed and then stored in the database of Auth along with the username. The Auth microservice, upon successful login, will create and sign a valid JWT token using a private key for the user which is then sent back to main to be passed to the end user for future requests. This JWT token is validated locally by the other microservices by checking that the key was in fact signed by Auth by utilising a public key.

There are two endpoints on Auth that require authentication, one returns the users and their hashed passwords, and the other is used in the event you want to delete your account. The former being one that wouldn’t be exposed for public use, and the latter being one that is communicated via the main microservice.

We decided having an independent microservice dedicated to the security side of the application was smart. Keeping that separate minimises the chance of attack and isolates the storage of sensitive information like passwords on a separate database far from anything else. It also allows for one isolated microservice to be the only one with access to the all important private key for JWT token signing. This lowers the chance of an attacker getting a hold of it, and using it to create a token for themselves.

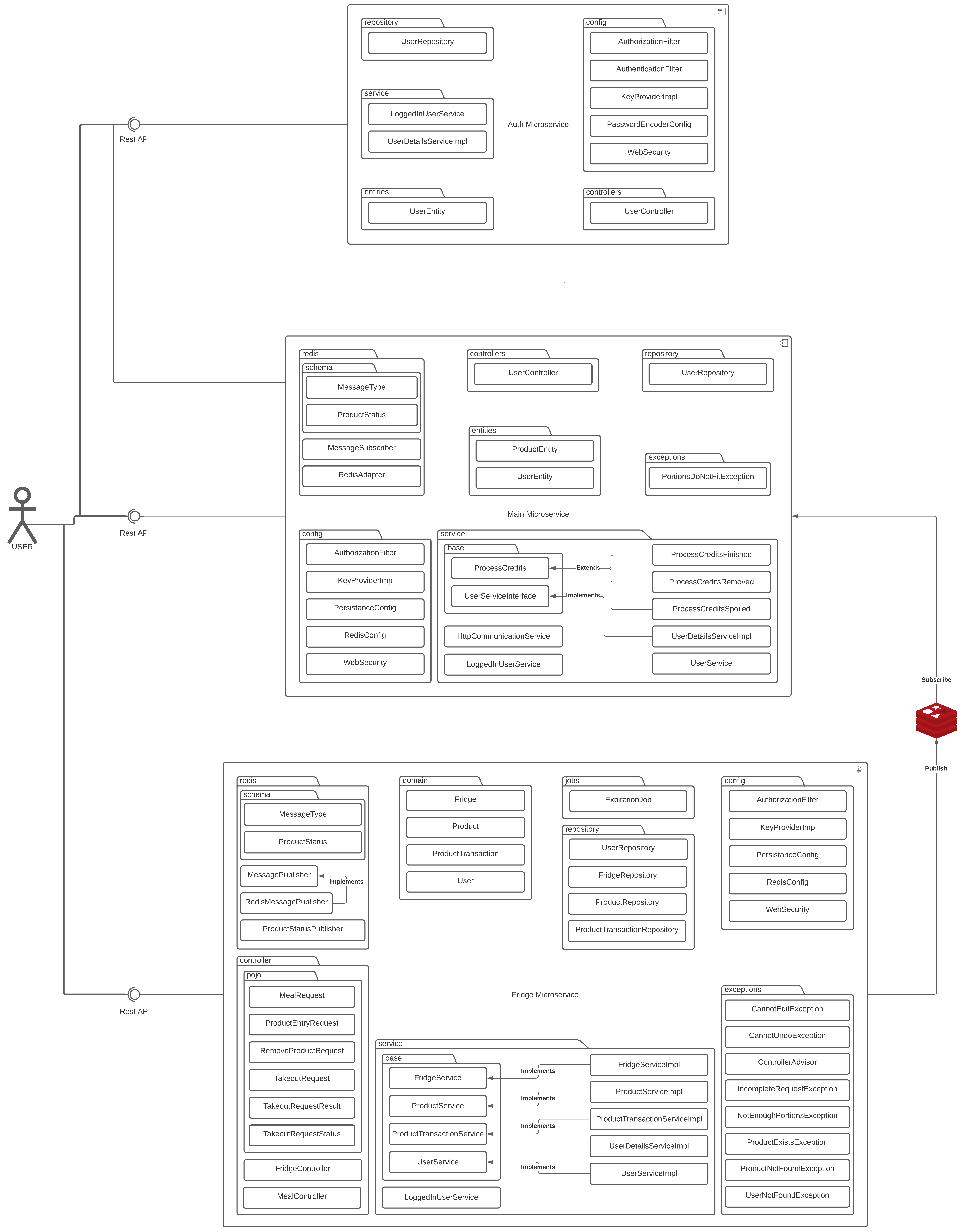
3. Fridge Microservice

The fridge microservice implements the desired functionality of the digital fridge that associates the set of users in the household and the set of products that are shared among the housemates. To interact with the fridge microservice, users must be authenticated and supply a valid JSON web token (JWT) in each request.

Users may interact with the fridge in different ways. Users may request the set of products contained in the fridge, along with the products' details such as the owner, the expiry date, the portions left and more. Based on that list, users may make a decision to take a portion of an existing product. If the user made a mistake, they may undo their last transaction. Users may add their own product in the fridge and the products may be edited and removed by their owner. There is another mechanism of consuming products and that is the meal. Housemates can also group together and eat together.

The fridge microservice does not calculate the credit balances after the product transactions. That is the responsibility of the main microservice. Fridge and main communicate with each other using Redis pub/sub. The Fridge publishes a product status message in the event that either a product expires, a product is removed by its owner, or the product is fully consumed. In turn, the main microservice, who is a subscriber to the topic, receives those messages and handles the credit calculation logic. We chose Redis because it's fast and offers simple to use message brokering. It is also easily integrated with our Spring Boot project, which was an important factor we considered when making the choice.

Products may expire at any given date. We elegantly solve the problem of handling product expiration by scheduling a cron job that runs every 24 hours. If the job finds an expired product in the fridge, it publishes the appropriate message on the topic.


Auth Microservice
repository
UserRepository
service
LoggedInUserService
UserDetailsServiceImpl
entities
UserEntity
config
AuthorizationFilter
AuthenticationFilter
KeyProviderImpl
PasswordEncoderConfig
WebSecurity
controllers
UserController
Rest API
USER
Rest API
Rest API
Main Microservice
redis
schema
MessageType
ProductStatus
MessageSubscriber
RedisAdapter
controllers
UserController
repository
UserRepository
entities
ProductEntity
UserEntity
exceptions
PortionsDoNotFitException
config
AuthorizationFilter
KeyProviderImp
PersistanceConfig
RedisConfig
WebSecurity
service
base
ProcessCredits
UserServiceInterface
Extends
Implements
ProcessCreditsFinished
ProcessCreditsRemoved
ProcessCreditsSpoiled
UserDetailsServiceImpl
UserService
HttpCommunicationService
LoggedInUserService
Subscribe
Publish
Fridge Microservice
redis
schema
MessageType
ProductStatus
MessagePublisher
Implements
RedisMessagePublisher
ProductStatusPublisher
domain
Fridge
Product
ProductTransaction
User
jobs
ExpirationJob
repository
UserRepository
FridgeRepository
ProductRepository
ProductTransactionRepository
config
AuthorizationFilter
KeyProviderImp
PersistanceConfig
RedisConfig
WebSecurity
controller
pojo
MealRequest
ProductEntryRequest
RemoveProductRequest
TakeoutRequest
TakeoutRequestResult
TakeoutRequestStatus
FridgeController
MealController
service
base
FridgeService
ProductService
ProductTransactionService
UserService
Implements
Implements
Implements
Implements
FridgeServiceImpl
ProductServiceImpl
ProductTransactionServiceImpl
UserDetailsServiceImpl
UserServiceImpl
LoggedInUserService
exceptions
CannotEditException
CannotUndoException
ControllerAdvisor
IncompleteRequestException
NotEnoughPortionsException
ProductExistsException
ProductNotFoundException
UserNotFoundException

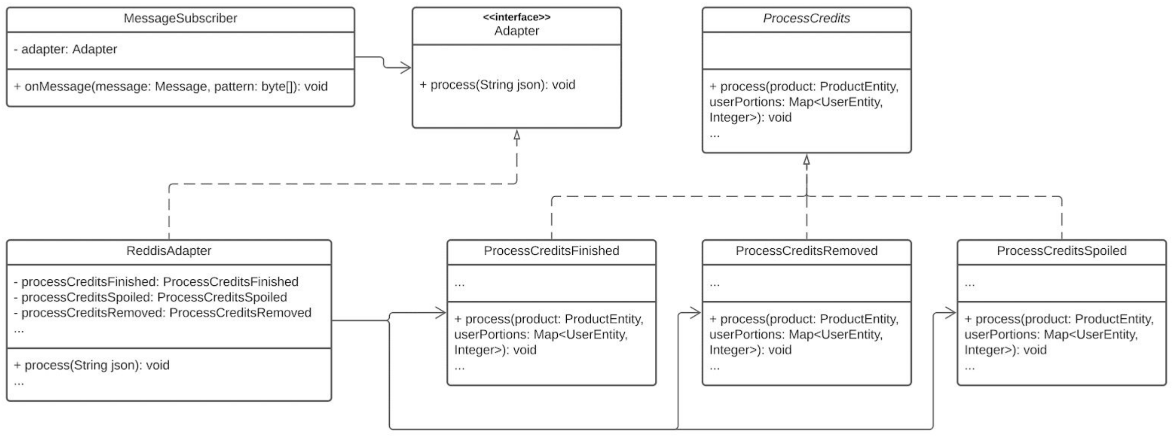
Task 2

Adapter

We use Redis to send messages from the fridge to the main microservice. Therefore, we need to connect our microservice to Redis. Redis provides messages in JSON, so this needs to be processed by a method like process(String json): void. The main microservice can process a message with this method: process(product: ProductEntity, userPortions: Map<UserEntity, Integer>): void. In addition to that, the correct child of ProcessCredits has to be chosen as can be seen in the UML diagram.

To solve this problem, an adapter design pattern would be a good fit.

To implement this, we implemented an interface that has the process(String json): void method. This implementation is called by the MessageSubscriber, which receives the Redis message. The implementation uses Gson to deserialize the JSON Based on the message, the ReddisAdapter then picks the right child of ProcessCredits to further process the message.


MessageSubscriber
- adapter: Adapter
+ onMessage(message: Message, pattern: byte[]): void
<<interface>>
Adapter
+ process(String json): void
ProcessCredits
+ process(product: ProductEntity, userPortions: Map<UserEntity, Integer>): void
...
ReddisAdapter
- processCreditsFinished: ProcessCreditsFinished
- processCreditsSpoiled: ProcessCreditsSpoiled
- processCreditsRemoved: ProcessCreditsRemoved
...
+ process(String json): void
...
ProcessCreditsFinished
...
+ process(product: ProductEntity, userPortions: Map<UserEntity, Integer>): void
...
ProcessCreditsRemoved
...
+ process(product: ProductEntity, userPortions: Map<UserEntity, Integer>): void
...
ProcessCreditsSpoiled
...
+ process(product: ProductEntity, userPortions: Map<UserEntity, Integer>): void
...

The pattern is implemented here:
https://gitlab.ewi.tudelft.nl/cse2115/2020-2010/7-student-house-food-management/op27-sem54/-/tree/master/mainService/src/main/java/nl/tudelft/sem/sem54/mainservice/redis

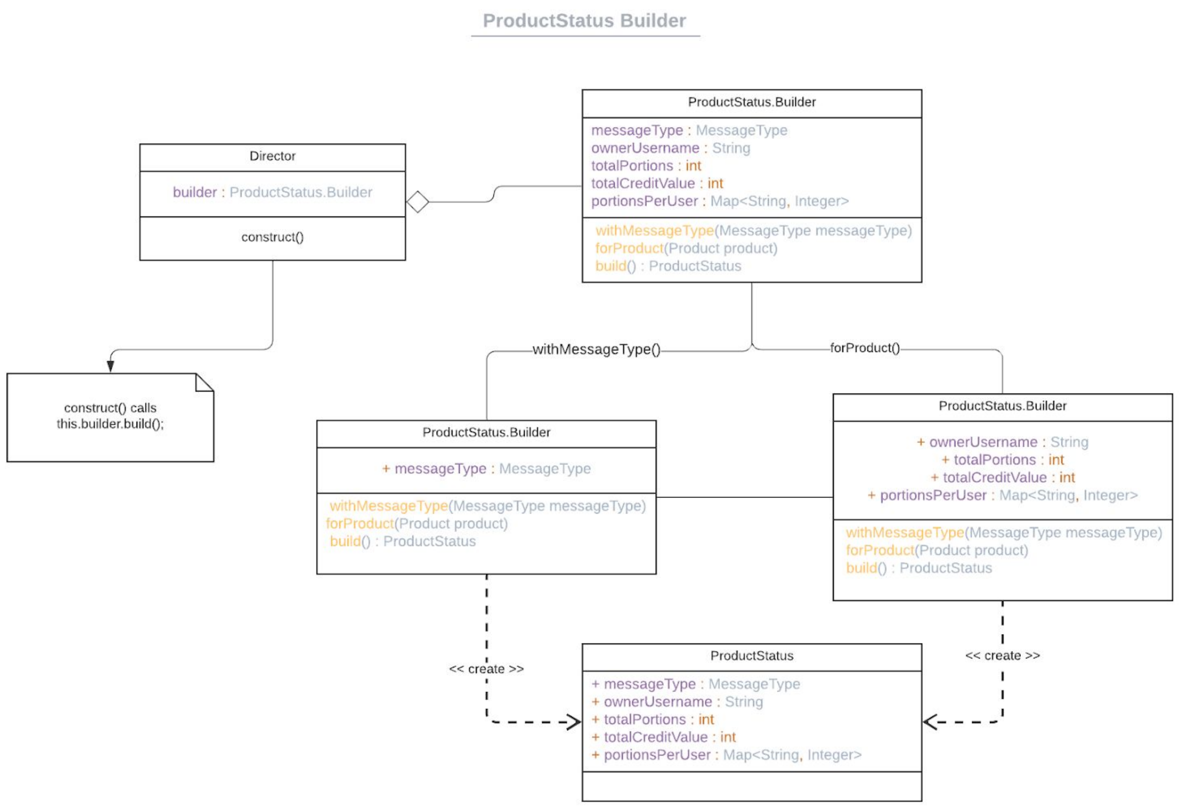
Usage of the Builder pattern

Surprisingly, the complexity of some of our classes grew beyond the ‘understand just by looking’ barrier even for such small requirements. This goes for their creation as well. We take advantage of the builder pattern in several classes we manually identified as having ‘fat constructors’. Here's a list of the refractored classes:

- TakeoutRequestResult.java
- Product.java
- ProductTransaction.java
- ProductStatus.java

Throughout our development of these classes we've tried to generally avoid inheritance wherever possible, and we've succeeded in doing that. Contrary to its representation in theory, we've actually found it quite avoidable in most of the cases for the better.

Here's a UML class diagram of one of the classes we've implemented the builder pattern on.



In the case of ProductStatus.java, we have a class which strictly depends on 2 others, the MessageType enum and the Product class. The builder allows us to create instances of our ProductStatus in a much cleaner and readable way.