Karlsruhe Institute of Technology

The CoCoME Platform for Collaborative Empirical Research on Information System Evolution

Evolution scenario in the second founding period of SPP 1593

Robert Heinrich, Sandro Koch, Ralf Reussner

August 9, 2018

Contents

List of Figures

Acknowledgement

We would like to thank our student assistants Niko Benkler and Tobias Haßberg for contributing to this Technical Report.
This work was supported by the DFG (German Research Foundation) under the Priority Programme SPP1593: Design For Future – Managed Software Evolution.

1 Introduction

This technical report describes the evolution scenarios for the community case study Common Component Modeling Example (CoCoME) [4, 2] in the second funding periode of the DFG Priority Programme Design For Future – Managed Software Evolution (SPP1593)[1]. It extends the technical report on the evolution scenarios in the first funding periode of SPP 1593 [3].

Chapter 2 introduces the new evolution scenarios. Design details for the scenarios are described in Chapter 3 and the implementation is explained in Chapter 4.

[1] http://www.dfg-spp1593.de

2 Evolution Scenarios

This chapter introduces the evolution scenarios of the second funding period of the DFG Priority Programme Design For Future – Managed Software Evolution. The evolution scenarios cover the categories adaptive and perfective evolution. Corrective evolution is not considered in the scenarios as this merely refers to fixing design or implementation issues.

2.1 Evolution Scenarios of the Hybrid Cloud-based Variant

This section introduces ~~the~~ two evolution scenarios for CoCoME . In the first scenario a container virtualization is introduced. The second scenario presents a mobile application which can be used with CoCoME. The goal is to enable ~~of the~~a hybrid cloud-based variation of CoCoME.

2.1.1 New Scenario 1 – Container Virtualization~~Setting up a Docker environment~~

This scenario aims to facilitate the deployment and operation of the CoCoME system. Thus, container-based virtualization with Docker is introduced. Docker eases the integration of CoCoME into build and deployment pipelines. The functionality of CoCoME remains the same although the technology stack must be extended as visualized in Fig. 3.5.

The CoCoME company identified the IT administration as a significant cost factor. ~~The CoCoME company mustwants to reduce IT administration costs.~~ Nevertheless, it is required to continuously improve the entire system in order to stay competitive. Thus, frequent updates to the enterprise and store software are necessary. ~~Nevertheless, frequent updates to the enterprise and store software are necessary to continuously improve the entire system.~~ As a consequence of the frequent update process, the IT staff ~~needs to~~must update the system components. Updating is ~~necessary~~required as soon as a new software version is released.

The old update process is as follows: After the new version of CoCoME is built a~~A~~n operations team member has to get access to the actual server. The old version ~~has to~~must be undeployed and replaced with the new version. The whole update process is time consuming and expensive as the updates have to be done manually.

Therefore, ~~a~~ Docker ~~version~~ is elaborated to simplify the administration process. As soon as a new release~~software~~ version of CoCoME is ready for delivery, the d~~D~~evelopment t~~T~~eam starts the rebuilding process of the CoCoME Docker Image.~~wraps it into a Docker Image~~. CoCoME is build in the Docker Container. This Docker Image can be automatically deployed to the destination server according to the principle of Continuous Deployment (CD) [5].

2.1.2 New Scenario 2 – Adding a Mobile App

After successfully adding a Pick-up Shop , the CoCoME company stays competitive with other online shop vendors (such as Amazon). However, in the smartphone era customers do not only want to buy goods exclusively from their homes or local stores. Purchasing goods anywhere and anytime has become a demanding requirement in order to stay competitive on the market. This raises the idea to create a ~~second~~third sales channel next to the existing Pick-up Shop and local stores in the CoCoME system. As a consequence, more customers can be acquired~~attracted~~ to increase the companies market share. ~~gain a larger share of the market.~~

The customer can order and pay by using the CoCoME Mobile App~~app~~. The delivery process is similar to the Pick-up Shop: The goods are delivered to a pick-up place (~~i.e.~~e.g., a store) of the customers~~his/hers~~ choice.~~for example in the neighborhood or the way to work.~~ By introducing the Mobile App as a multi OS application, the CoCoME system has to face various quality issues such as privacy, security and reliability. Also the performance of the whole application can be affected if many customers order via the app.

2.2 Evolution Scenarios of the Microservice Architecture

This section introduces the evolution scenario of the microservice-based variation of CoCoME.

2.2.1 New Scenario 3 – Defining different microservices

After a year of economical stagnation, the CoCoME company decides to restructure its infrastructure. Global players like Amazon or Netflix demonstrated that using a microservice ~~A~~architecture makes them more flexible regarding new functionality. When adding the Pick-up Shop, the CoCoME company realized that they have to break open the existing system. It ~~was~~is necessary to modify the *WebService::Inventory* and the *TradingSystem::Inventory* component [3]. Furthermore, adding a *MobileAppClient* demonstrated that the SOAP/WS*-based web services provided by CoCoME are not compatible with REST-based App development.

Inspired by the flexibility and reusability of microservices, the CoCoME company decided to invest money into a restructuring process. The current system is divided into a collection of loosely coupled services. Each of them covers a specific part of the former CoCoME system. The aim is to preserve the functionality of the current system and solely change its architecture.

This enables the company to tap into~~develop~~ new markets with fewer difficulties. ~~much easier.~~ Therefore, the future competitiveness is secured. For example, the CoCoME company wants to extend their ~~product~~service range by offering movie streaming.This requires a vastly different system. Nevertheless, customer management like login and means of payment are identical to the former CoCoME system. Those components already exists as a microservice and therefore can be reused~~taken over~~. ~~The management is certain that this will soon result in economical growth.~~

3 Design Details for Evolution Scenarios

In this chapter we provide the detailed design documentation for each of the evolution scenarios introduced in the previous section. Sec. 3.1 sketches the design decision for the Mobile App that provides a second sales channel next to the existing Pick-up Shop. Sec. 3.2 describes the adaptive changes of using a Docker environment to simplify the update process. They are both based on, or at least use the Hybrid Cloud-based Variant of CoCoME [3]. In contrast, Sec. 3.3 provides a detailed design documentation of a new architectural version of CoCoME. This perfective evolution scenario is realized based on the ~~M~~microservice idea.

3.1 Adding a Mobile App Client

Developing the Mobile App Client as an extension of CoCoME requires additional use cases. They are described in Sec. 3.1.1. Further, Sec. 3.1.2 describes extensions on design level. The content of this chapter mainly originates from [6].

3.1.1 Use Cases of the Mobile App

UC 14 - ProcessAppSale

Brief Description A Customer selects the product items s/he wants to buy and the payment by credit card is performed.
Involved Actors AppCustomer, Bank
Precondition The App is ready to process a new sale and the Customer already has an account registered in the System.
Trigger The Customer opens the app and wants to buy product items.
Postcondition The Customer has paid and the sale is registered in the inventory.
Standard Process

1. The AppCustomer searches products provided by the App.
2. The AppCustomer can see details for each product on a separate site.
3. The AppCustomer adds the product items s/he wants to purchase to the Shopping Cart. Step 1-3 is repeated until all items are added to the cart.
4. The AppCustomer gets an overview of the items in the cart, their price and the running total.
5. The AppCustomer proceed to the Checkout
6. The AppCustomer selects the Store where s/he wants to pick up his/her purchased product items.

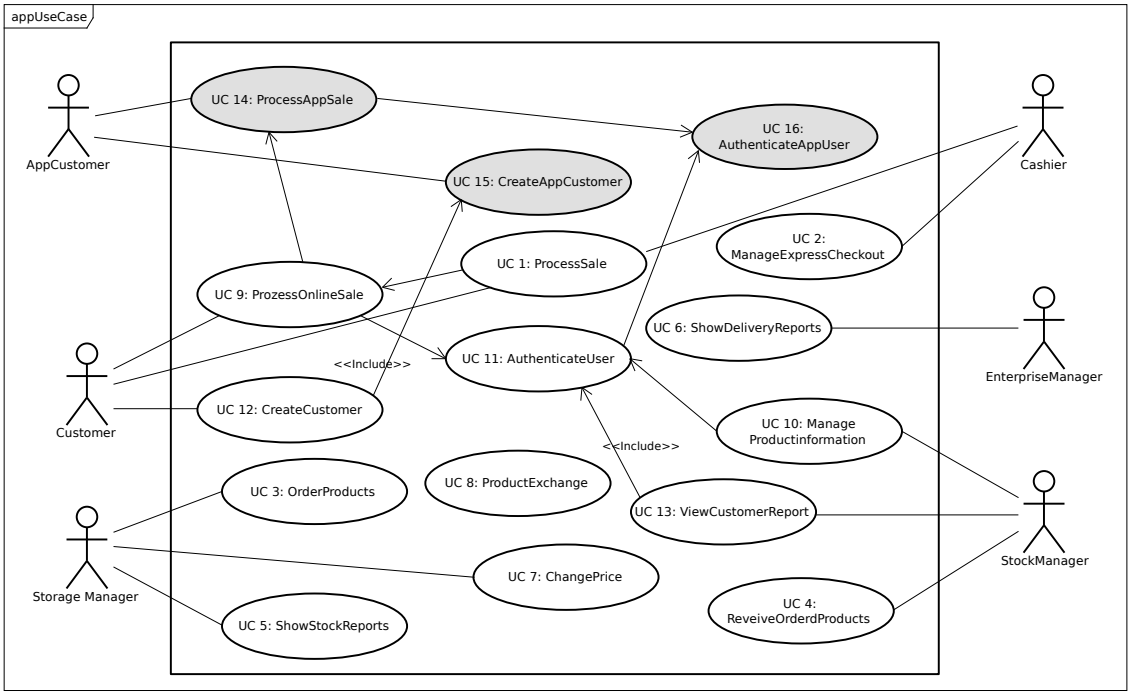

Figure 3.1: Use Case Diagram CoCoME Mobile App

7. The AppCustomer is presented with a login form and is required to complete the "Authenticate user" use case.
8. In order to initiate card payment, the customer selects a credit card used for the purchase.
9. The AppCustomer enters his/her PIN in the designated field presented by the System.
10. The System presents the Customer with an overview of the purchase, the AppCustomer confirms the purchase and waits for validation. Step 9 is repeated until the validation is successful or the Customer decides to cancel the purchase.
11. Completed sales are logged by the System and sale information are sent to the Inventory in order to update the stock.
12. A success message is presented to the AppCustomer and the product items are being prepared to be picked up by the customer.
13. The AppCustomer closes the app.

Alternative or Exceptional Processes

- *In step 8: No Card available*
  1. In order to add a new credit card the Customer clicks the Add Card button.
  2. The Customer enters the card number of the new credit card and saves the card.
- *In step 10: Card validation fails*

1. The Customer tries again and again.
2. Otherwise, the Customer can decide to cancel the purchase.

UC 15 - CreateAppCustomer

Brief Description The app offers a possibility to create a new Customer account.
Involved Actors AppCustomer
Precondition The Customer does not have a Customer account yet and the app is started.
Trigger A new AppCustomer wants to create an account.
Postcondition The User is authenticated.
Standard Process

1. The AppCustomer has to fill out forms, requesting all necessary information to create a new AppCustomer account.
   a) Form for name, email and password
   b) Form for address
   c) Summery of the information
2. The Customer fills out the forms, verifies and submits the information.
3. The app verifies the given information and creates a new Customer account in the Inventory.

Alternative or Exceptional Processes

- *In step 3 : Provided information is incorrect or not valid.*
  The Customer is notified of the problem and enters the information again until it passes the check.

UC 16 - AuthenticateAppUser

Brief Description The app provides the possibility to authenticate a User.
Involved Actors AppCustomer
Precondition The app is started.
Trigger An AppCustomer wants to authenticate his/herself.
Postcondition The AppCustomer is authenticated.
Standard Process

1. The AppCustomer gets displayed a login form. S/he is asked to enter email and password.
2. The App checks the provided credentials. If correct, the AppCustomer is logged in.

Alternative or Exceptional Processes

- *In step 2: Wrong credentials*
  1. An error message is displayed.
  2. The User may try again until the authentication succeeds.

3.1.2 Design of the Mobile App

Fig. 3.2 sketches the component diagram of ~~this~~the evolution scenario SC1. When adding the Mobile App client, the hybrid cloud-based variant of CoCoME did not have to be modified. ~~Simply the~~ We focus on the three web services *WebService::Inventory::*, *WebService::Inventory::Store* and *WebService::Inventory::Enterprise* of the App Client. ~~are emphasized.~~ The entire component diagram for the hybrid cloud-based variant is available in the previous Technical Report [3].

The AppShop requires an adapter (i.e., *AppShopAdapter*) to access the web services provided by CoCoME. This is because CoCoME uses SOAP/WS*-based web services which are not compatible with the technology used to implement the AppShop Client. A more detailed introduction about the technology used to implement the Mobile App Client can be found in [6].

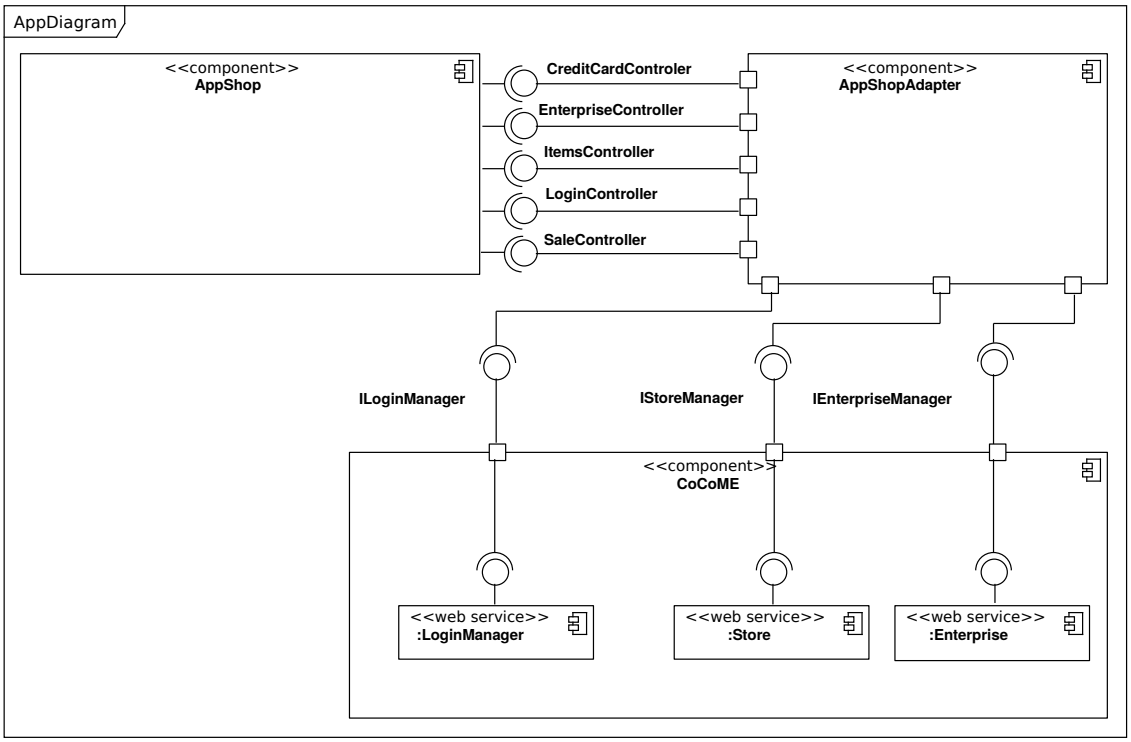

Figure 3.2: Component Diagram of the CoCoME Ecosystem ~~A~~after ~~A~~adding the Mobile App Client

The *AppShopAdapter* consumes the three web services *WebService::Inventory::LoginManager*, *WebService::Inventory::Store*, and *WebService::Inventory::Enterprise*3 Additionally, the *AppShopAdatper* provides a ~~Rest~~REST ~~Api~~Interface which is used by the ~~actual~~ *AppShop*. The ~~Rest~~REST

InterfaceApi contains provides endpoints to retrieve and process Credit Card, Enterprise and StockItem information. To implement the use cases *UC14-16* the Api RestREST Interface also provides endpoints for user management and processing sales.

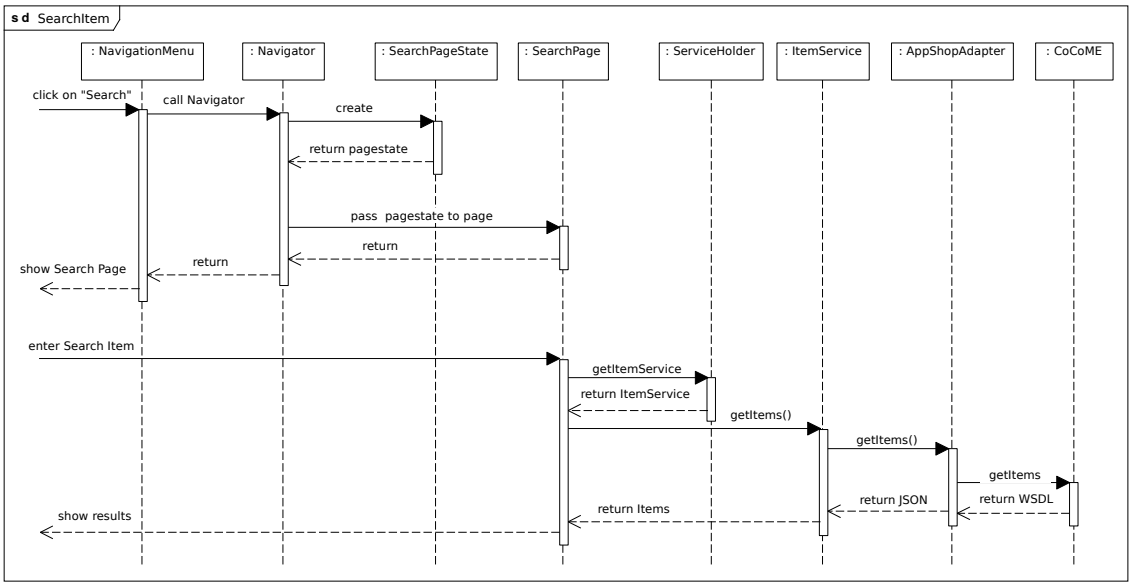

Figure 3.3: Sequence Diagram of Searching an Item in the Mobile App Client

Fig. 3.3 shows the process of opening a page to search for an Item. The customer opens the *WebShopClient* and triggers the "Search" function to search for an item. To open the page the *NavigatorMenu* must call the *Navigator* which creates a pagestate object and passes the object to the page. This HTML page is nowthen presented to the customer. To fill the page with information (i.e., when searching for a *ProductItem*) the page uses services provided by the *ServiceHolder*. In this case, the *ItemService* calls the responsible REST-Service of *AppShopAdapter* which in turn retrieves the necessary information from the WSDL services provided by CoCoME.

Fig. 3.4 demonstrates how the Mobile App Client processes sales. For the sake of clarity, the diagram is simplified and only contains the most important calls. First, the customer searches for items (according to Fig. 3.3). By clicking on the desired Iitem, the according *ItemPage* is shown. ThisThe *ItemPage* page carries information about the Iitem. Here, the customer decides whether the Iitem should be added to the Sshopping Ccart or not. The last steps

TODO: welche genau?

are repeated until the customer decides to proceed to the checkout. If not logged in the customer gets forwarded to the *LoginPage*. When successfully logged in the customer clicks the

BuyNow-Button. The ~~i~~sale process is finished as soon as the backend (CoCoME) has processed the sale.

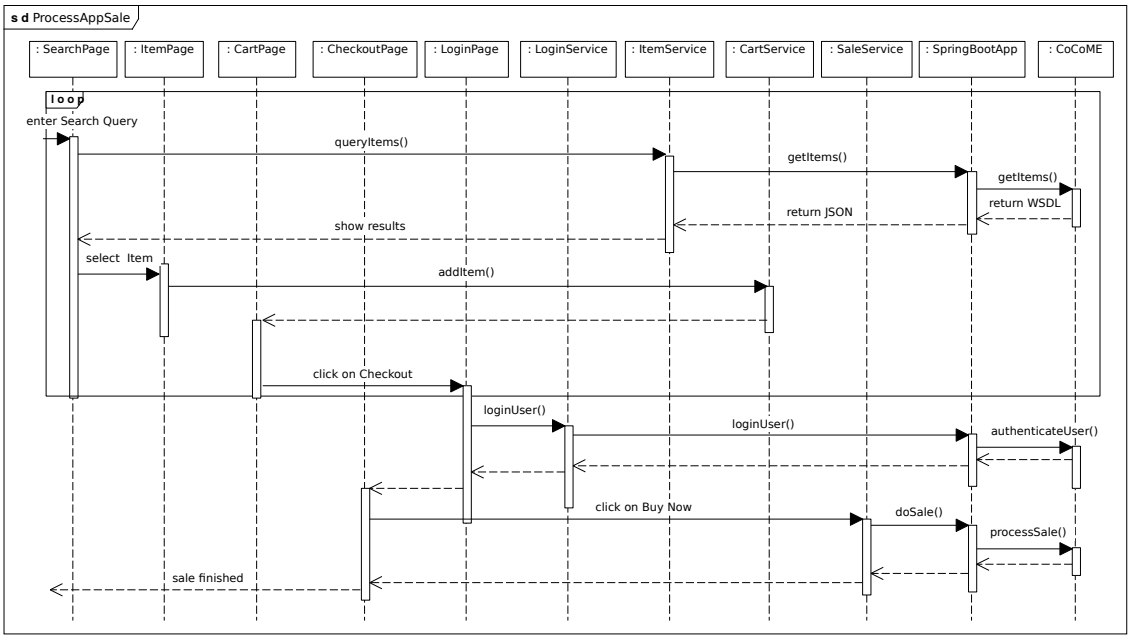

Figure 3.4: Sequence Diagram of ~~p~~processing a ~~s~~sale

3.2 Using a Docker Environment

As shown in Fig. 3.5, using a Docker Environment affects the technology stack by adding additional layers. The CoCoME stack consists of Glassfish, Java Virtual Machine (JVM) and Maven. More detailed, the given CoCoME Sstack is moved into thea Docker DeamonContainer which runs a Linux distribution. The original parts of the stack, like Glassfish and the , Java Virtual MachineJVM are still a part of the stackfunctioning as before.

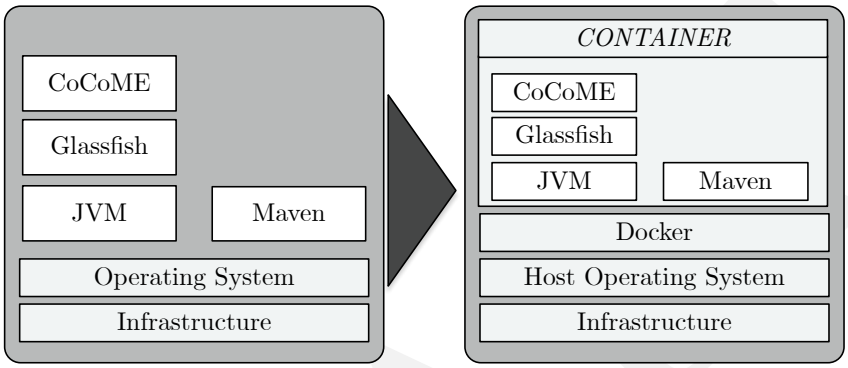

Figure 3.5: Extended technology stack CoCoME

The Dockerfile defines an environment based on the latest version of the Linux distribution Ubuntu. Maven, Git and Java are also installed using the standard Ubuntu package manager. Git hasserves two purposes: On the one hand it is used to download the most recent version of the CoCoME source code or a precompiled CoCoME version. On the other hand, it is used to download a prefabricated version of Glassfish that is already tailored to the needs of the CoCoME applicationincludes domains and other adjustments required for CoCoME . Java is required by Glassfish and CoCoME as they need the Java Virtual MachineJVM. Maven is needed to build and deploy the latest version of CoCoME onto the provided Glassfish servers.

During the development phase, it was we decided to implement and provide two different versions of the CoCoME Docker Container. The first version always pulls the most recent CoCoME source code from GitHub, downloads the entire dependencies with Maven, compiles and builds the project and finally deploys CoCoME on the Glassfish servers. As a consequence, creating and starting the Docker Container for CoCoME takes about one hour.

In contrast, the second version of the CoCoME Docker Containeronly pulls a prefabricated version of CoCoME from GitHub. Therefore, pulling the source code up to building the CoCoME project is skipped. Maven does not have to be included in the technology stack. Solely, deploying CoCoME on the Glassfish server is necessary. This reduces the deployment time to a few minutes. but has a disadvantage: Nevertheless, the prefabricated version of the Glassfish Servers and CoCoME has to be updated manually. Thus, it is sometimes not the most recent version. By providing both, a fast deploying version and a current version, the user can choose what's suits best for his/her needs situation.

3.3 Microservices Technology

In this section we provide a brief design documentation of the use cases that are defined in the hybrid cloud-based variant of CoCoME [4](p.4-10). The following subsections are divided into the ~~M~~microservices and the corresponding use cases. Sec.3.3.4 describes the general architectural overview of the ~~M~~microservice variant of CoCoME.

3.3.1 Orders

This section describes the design of the use cases implemented in the *Orders* ~~M~~microservice. T~~t~~hat service provides main parts of the functionality for UC 3 and UC 4.

Behaviou~~u~~ral View on UC 3 - Order Products

For a better understanding, UC 3 is divided in two steps. The first part is described in Fig. 3.6: A user chooses the product items and amount to order. ~~and the corresponding amount.~~ Each *ProductOrder* is stored in the *ProductOrderRepository* as a *OrderEntry*. In the second part (Fig. 3.7), the *OrderEntries* are wrapped in a single *ProductOrder* element ~~that a~~. Additionally, the *ProductOrder* element contains information about the store and the date of the order. When the user presses the button *Order*, the *OrderManagement* iterates over the collection of *OrderEntries* and sets a reference to the actual *ProductOrder*.

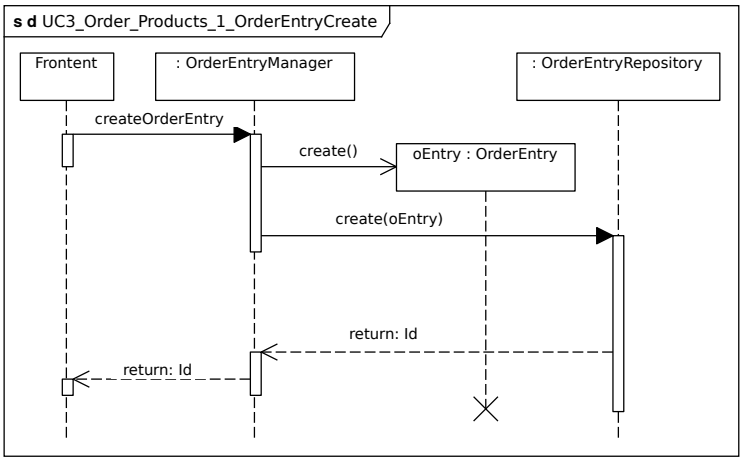

Figure 3.6: UC 3: *Order Products* (part 1)

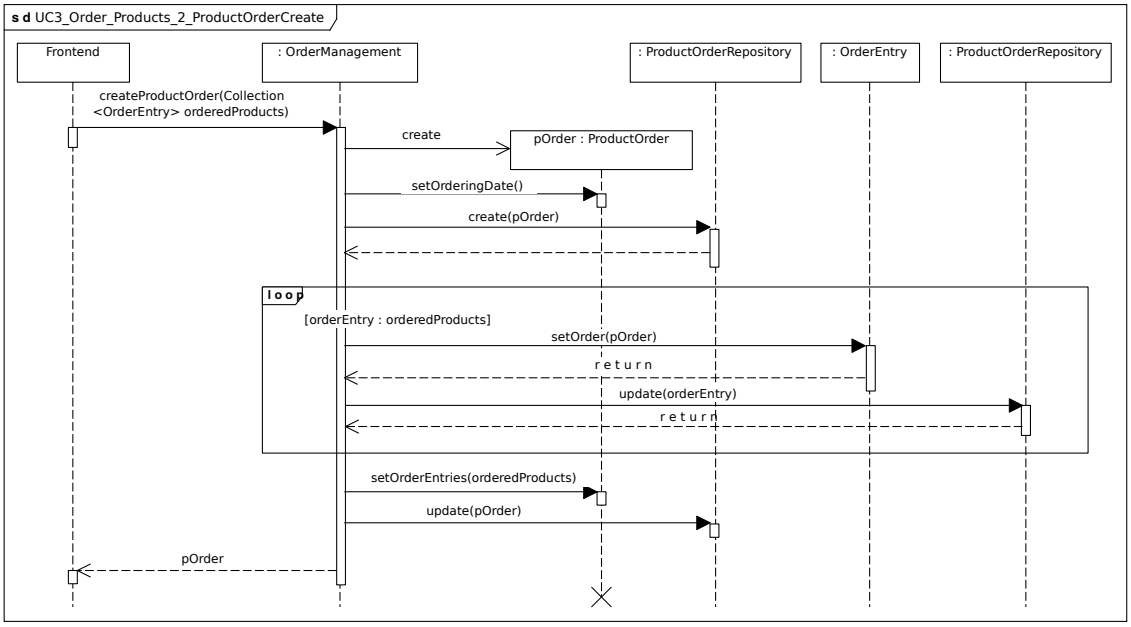

Figure 3.7: UC 3: *Order Products* (part 2)

Behavioural View on UC 4 - Receive Ordered Products

Fig. 3.8 shows~~, that first~~ the *ProductOrder* element. It~~, which handles the order with the passed id*Id*,~~ is refreshed and the delivery date is set to the passed date. The *ProductOrder* element handles the order regarding the containing *Id*. After this, it performs for each ~~OrderEntry~~*OrderEntry* a rest call to the store microservice. With that REST~~rest~~ call the number of ~~StockItem~~*StockItems*~~, which is representing the corresponding product in the store,~~ is increased by the amount of delivered products. *StockItems* representing a concrete product within the system.

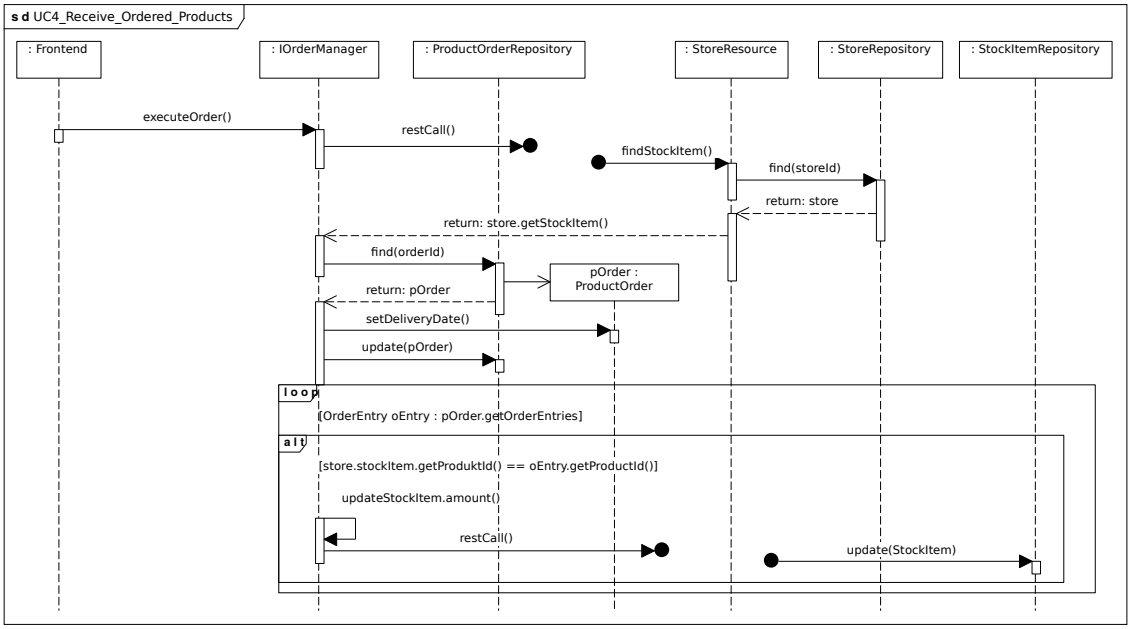

Figure 3.8: UC 4: *Receiver Ordered Products*

3.3.2 Stores

This section describes the design of the use cases implemented in the *Store* ~~M~~microservice that provides main parts of the functionality for UC 1, UC 2, UC 7 and UC 8.

Behaviou~~u~~ral View on UC 1 - Process Sale

Again, UC 1 is divided in two parts (Fig. 3.9 and Fig. 3.10). The first part describes how a user can add products to the sale process. When the *startSale* action is executed, the sale mode is activated and the *CashDesk* is resetted from a previous, probably cancel~~l~~led sale processes. To add products to the sale process, the user can either enter the barcode ~~digit by digit~~manually using the keyboard or scan the barcode using a *Scanner*. Further, the user can choose how many products s/he wants to purchase. This process is depicted in the inner loop.

Several checks are executed when successfully entering the barcode: The scanned item must exist in the system. If an item with the same barcode was already added to the sale, then the ~~actual~~ total *inventoryAmount* is increased by the second purchase amount. If ~~not,~~no item with the same barcode is present, the scanned item is added to the sale. ~~provided that an item with this barcode exists.~~In both cases, the availability and the amount of the item ~~amount~~ in the stock is checked and reduced. If one of the conditions is violated, the attempt of adding a product with the provided barcode and amount is ~~quit~~aborted. Subsequently, the display is updated and the product information is added to the printer output.

The second part of this use case, shown in Fig. 3.10, handles the end of the sale process. The *finishSale* routine is called when the *FinishSale* button is pressed.~~By calling the *finishSale* routine (when pressing the button *FinishSale*), the display is updated.~~ Thus, the display gets updated. Now, the user needs to choose between paying by card or cash and the *CashDesk* is set to the corresponding paying mode. In case the user wants to pay by credit card, s/he needs to enter the credit card details. In the other case, the cash amount is entered. In both cases, the information is checked ~~to~~for accuracy. After successfully ending the payment, the printer and display are updated and the *CashDesk* ends the sale process.

Behavioral View on UC 2 - Manage Express Checkout

Changing the express mode is triggered on two occasions (*externalCall*). First, when finishing a sale and if some conditions are fulfilled

TODO: welche bedingungen?

, the *CashDesk* switches into express mode automatically. Second, the Cashier switches manually back to normal mode.~~is able to switch back to normal mode.~~ In both cases, the *updateExpressLight* routine checks the current *ExpressLight* state and performs an update in accordance with the kind of call.

Behaviou~~u~~ral View on UC 7 - Change Price

The *StoreManager* is able to change a price for *StockItems* that are available in his/her store. As depicted in Fig. 3.12, the *StoreAdminManager* selects the right *Store*, based on the *StoreManager* that is logged in. To find the correct *StockItem*, the available items are filtered by a given product id. When the correct *StockItem* is found, the sale price can be simply updated by entering the new price. Finally, the *StockItem* in the database is updated.

Behaviou~~u~~ral View on UC 8 - Product Exchange

In case a Store is going to run out of a certain *StockItem*, products from a different *Store* within the same Enterprise can be exchanged. The process is triggered at the end of a sale. It is shown in Fig. 3.13. The ~~M~~microservice *Store* checks if the stock amount of the sold items have passed the minimal stock amount. If ~~not, nothing happens. Otherwise,~~the minimal stock amount is passed, the ~~S~~system calls the *shiftItem* routine. First, all *Stores* within the same *Enterprise* of the *Store* that is running out of stock are collected. For each *Store*, the system checks if the desired *StockItem* is available. The *findOptimum* routine decides ~~(using heuristics)~~ whether the transportation is meaningful. Therefore, heuristics are used. After a successful query, the *StockItem* is shipped from one Store to another, decreasing the amount at the first *Store* and increasing it at the second.

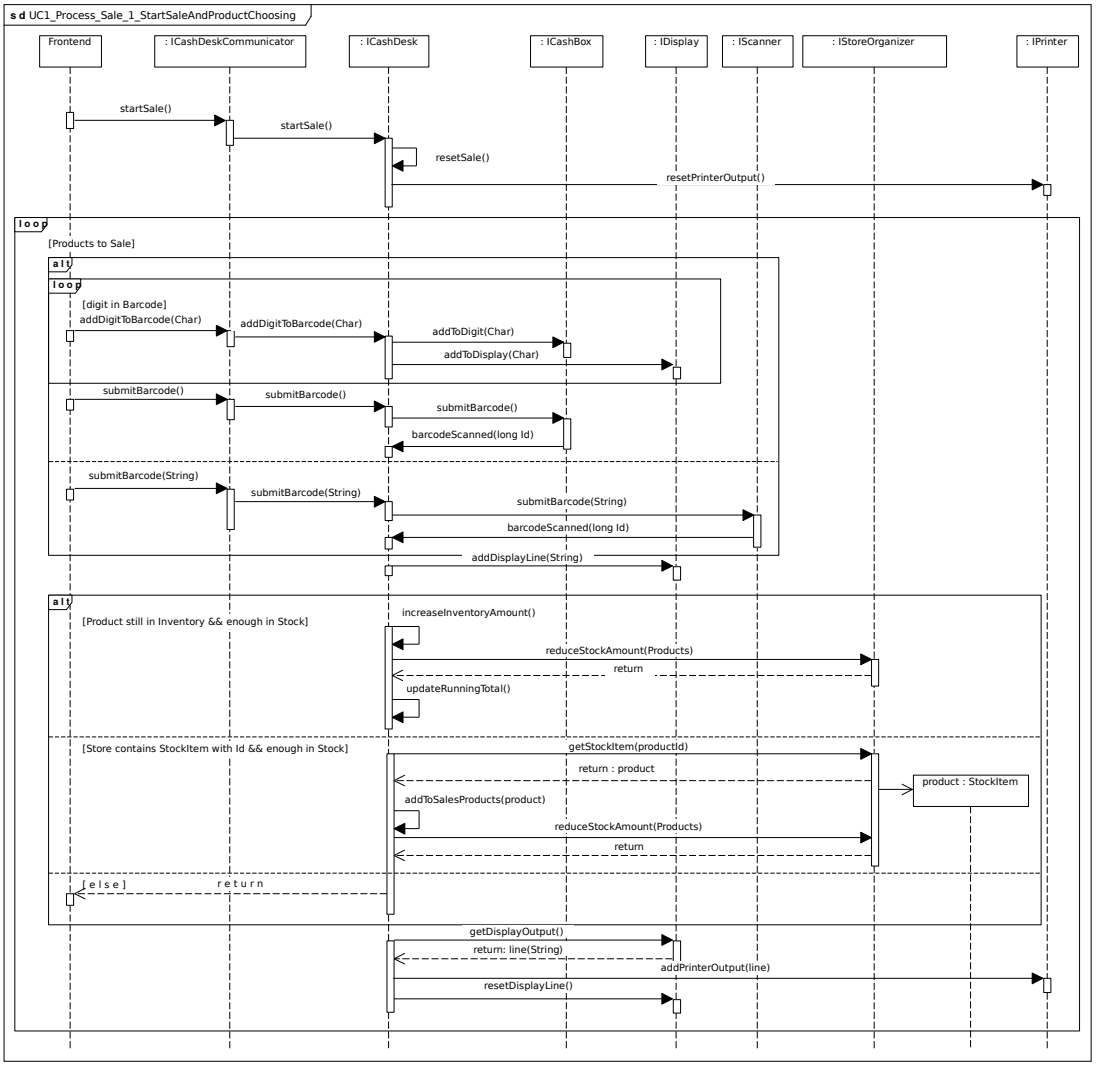

Figure 3.9: UC 1: *Process Sale* (part 1)

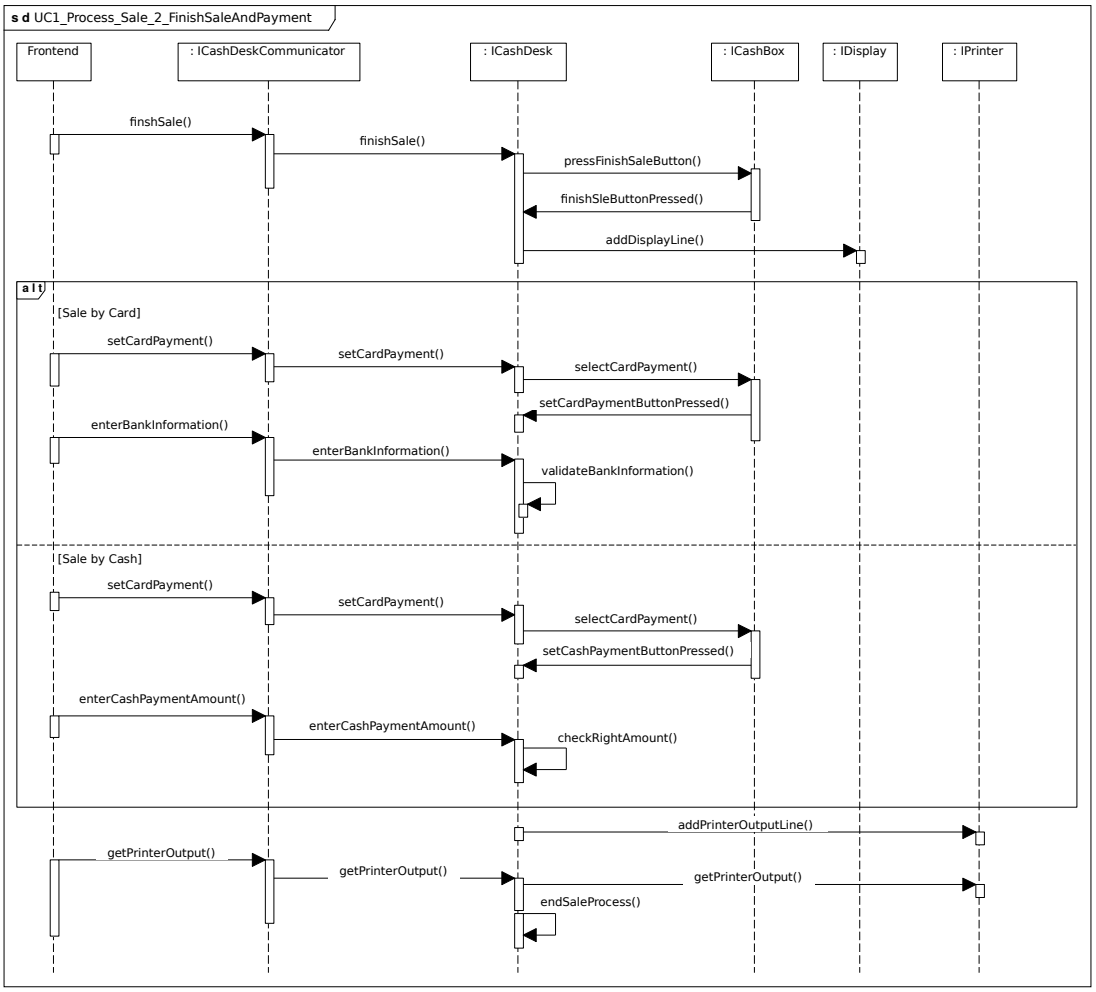

Figure 3.10: UC 1: *Process Sale* (part 2)

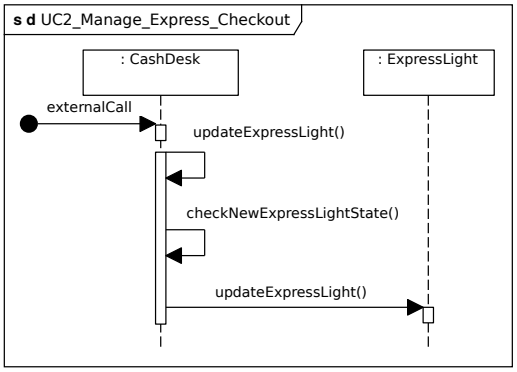

Figure 3.11: UC 2: *Manage Express Mode*

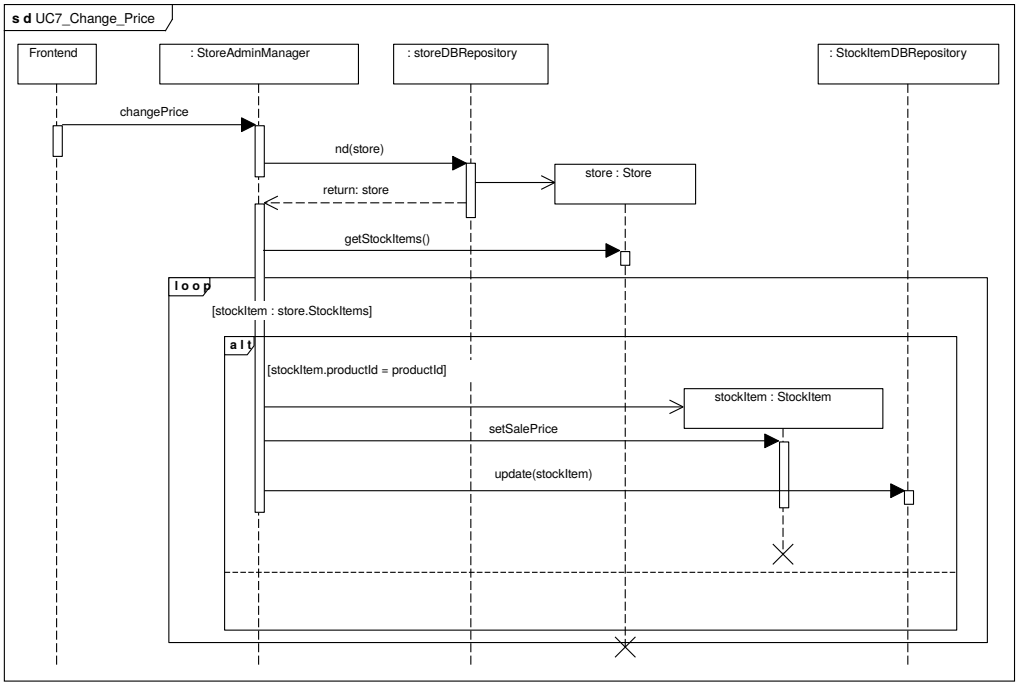

Figure 3.12: UC 7: *Change Price*

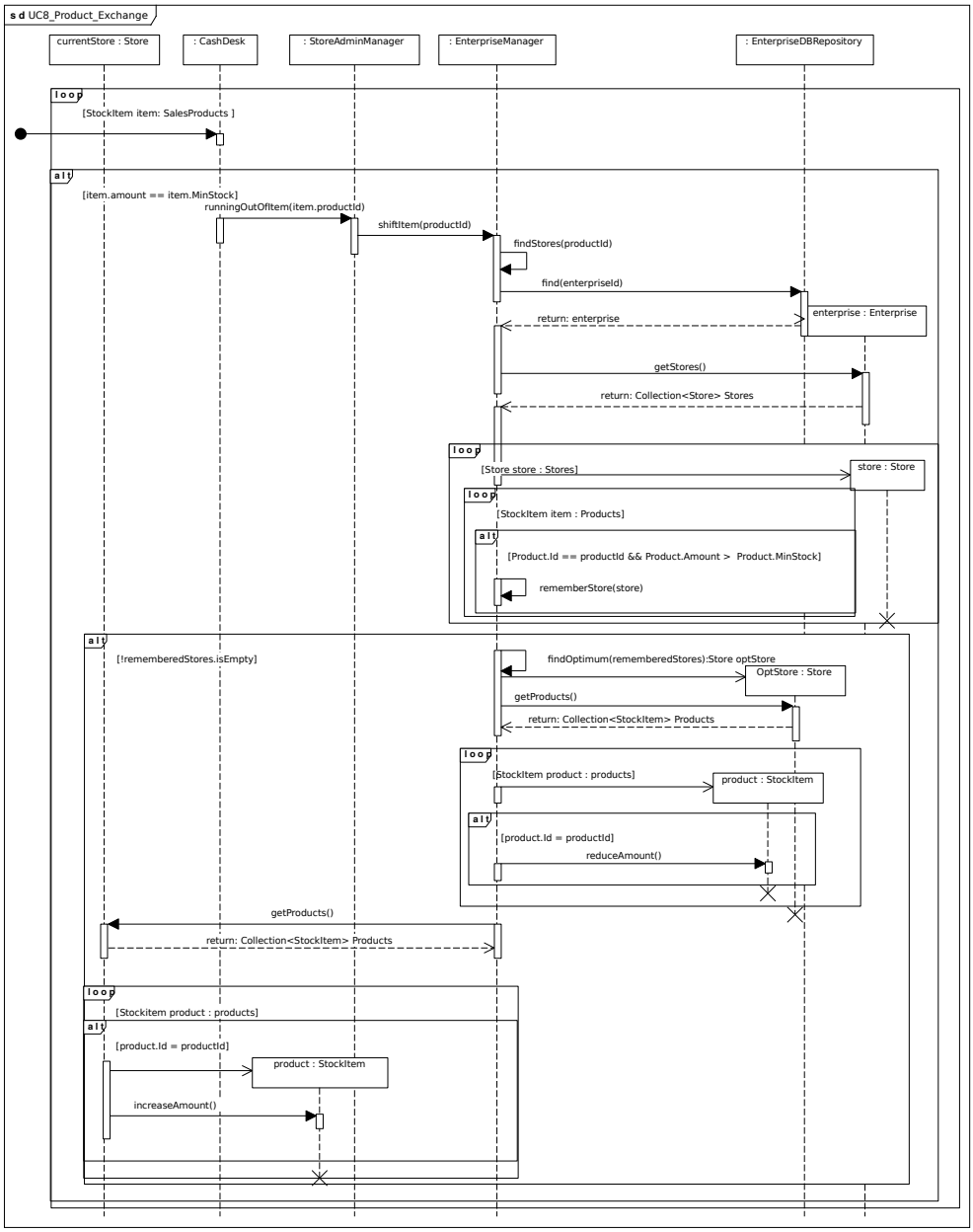

Figure 3.13: UC 8: *Product Exchange*

3.3.3 Reports

This section describes the design of the *Reports* ~~M~~microservice, which provides the functionality of the UC 5 and 6.

Behavio~~u~~ral View on UC 5 and UC 6 - Show Stock Report and Show Delivery Report
The *StoreManager* can request a full *StockReport* that includes all available stock items in the store. This process is described ~~a~~in the use case UC 5 in 3.14. The *StoreManager* enters the store identifier and the *StoreCommunicator* within the *Reports* ~~M~~microservice requests the desired information via a ~~rest~~REST call.

Besides, the *Reports* ~~M~~microservice provides the ~~opportunity~~functionality to calculate the mean time~~s~~ a delivery takes from each supplier to a considered enterprise ~~takes~~ (UC 6). The process is described in Fig. 3.15. The *EnterpriseManager* enters the order ~~id~~*Id* and the *OrderCommunicator* requests the information as a delivery report via ~~rest~~REST call. The report is displayed to the *EnterpriseManager*.

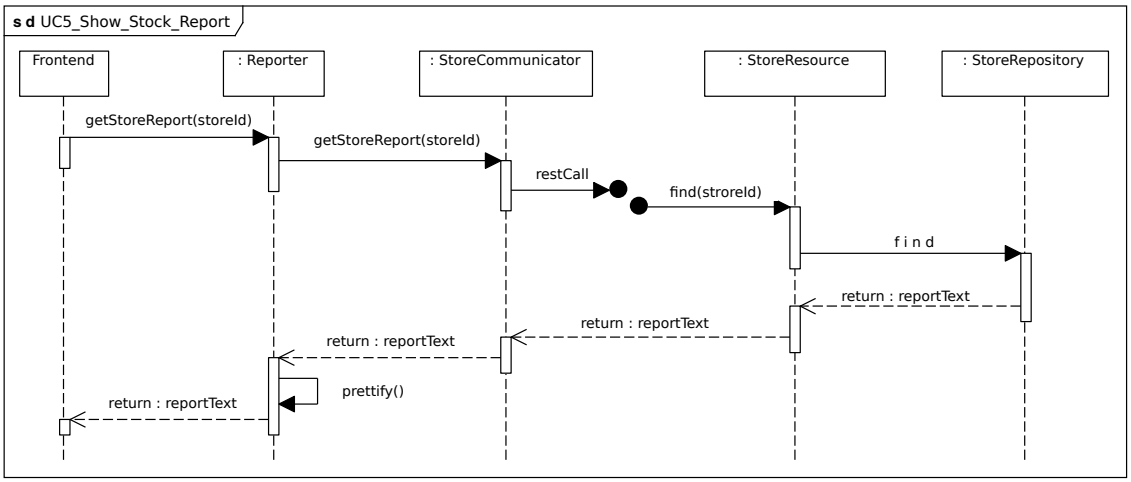

Figure 3.14: UC 5: *Show Stock Report*

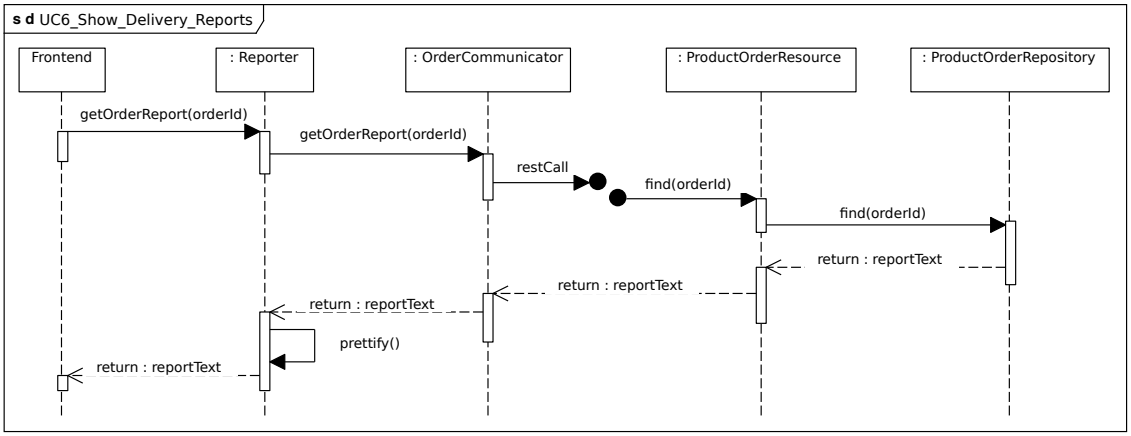

Figure 3.15: UC 6: *Show Delivery Report*

3.3.4 Architectural overview

This section describes the architectural design of the ~~M~~microservices. CoCoME is divided into four different ~~M~~microservices: *Orders*, *Reports*, *Stores* and *Products* (Fig. 3.16). Each ~~M~~microservice provides its own graphical user interface. The graphical user interface ~~that~~ can be loaded dynamically. Further, each service provides its core functionality that sometimes requires a connection to other ~~M~~microservices via REST. Each service apart from *Reports*, has its own Database.

The *Store* service provides functionality for Store- and Enterprise Managers. They can create stores, change sale prices for goods or order products. For the last two functionalities, the *Order* and *Products* ~~S~~service is needed. Further, the *Store* service handles the sale process.

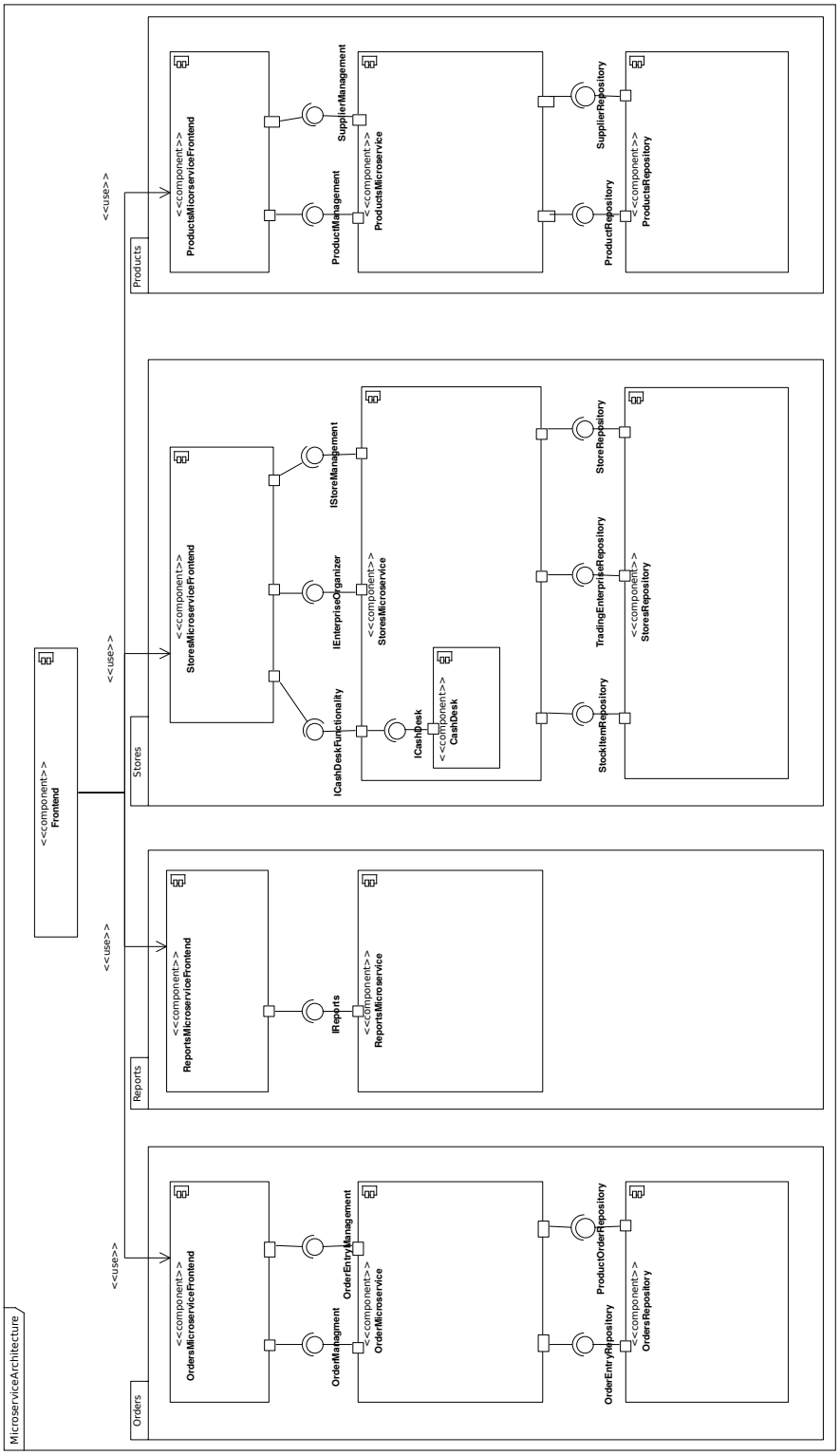

Figure 3.16: Microservice architecture

4 Implementation of Evolution Scenarios

This chapter describes implementation details of the Docker environment in Sec. 4.1~~,~~. ~~t~~The Mobile App Client for the existing hybrid cloud-based variant of CoCoME is described in Sec. ~~(~~4.2~~)~~ and the Microservice-based variant is described in Sec. ~~(~~4.3~~)~~.

4.1 Using a Docker Environment

As shown in Fig. 4.1, the Docker Container contains five different Glassfish servers. In particular they are called *WEB*, *ENTERPRISE*, *STORE*, *REGISTRY* and *ADAPTER*. By default, Glassfish provides a Derby ~~DB~~Database. The Derby Database~~that~~ is connected to the Service Adapter using the Java Database Connectivity (JDBS) interface. The deployment assignment within the Docker environment is identical to the one specified in CoCoME deployment guide. This means the maven generated archive files *cloud-web-frontend*, *enterprise-logic-ear*, *store-logic-ear*, *cloud-registry-sevice*, and *service-adapter-ear* are deployed on the servers by using the following assignment:

Server	Deployment file
WEB	cloud-web-frontend
ENTERPRISE	enterprise-logic-ear
STORE	store-logic-ear
REGISTRY	cloud-registry-service
ADAPTER	service-adapter-ear

Figure 4.2: Assignment of archive files to the Glassfish Servers

As mentioned earlier, there are two versions of this Docker project. The fast version can be extended by the Pick-Up shop[1]. This Pick-Up shop runs inside a separate Docker container which is shown in Fig. 4.3. As shown in Fig. 4.3, this container provides ~~only~~ one Glassfish server.

Server	Deployment file
PICKUP_SHOP	cocome-pickup-war

Figure 4.4: Assignment of archive files to the Glassfish Server~~s~~

[1] `https://github.com/cocome-community-case-study/cocome-cloud-jee-web-shop`

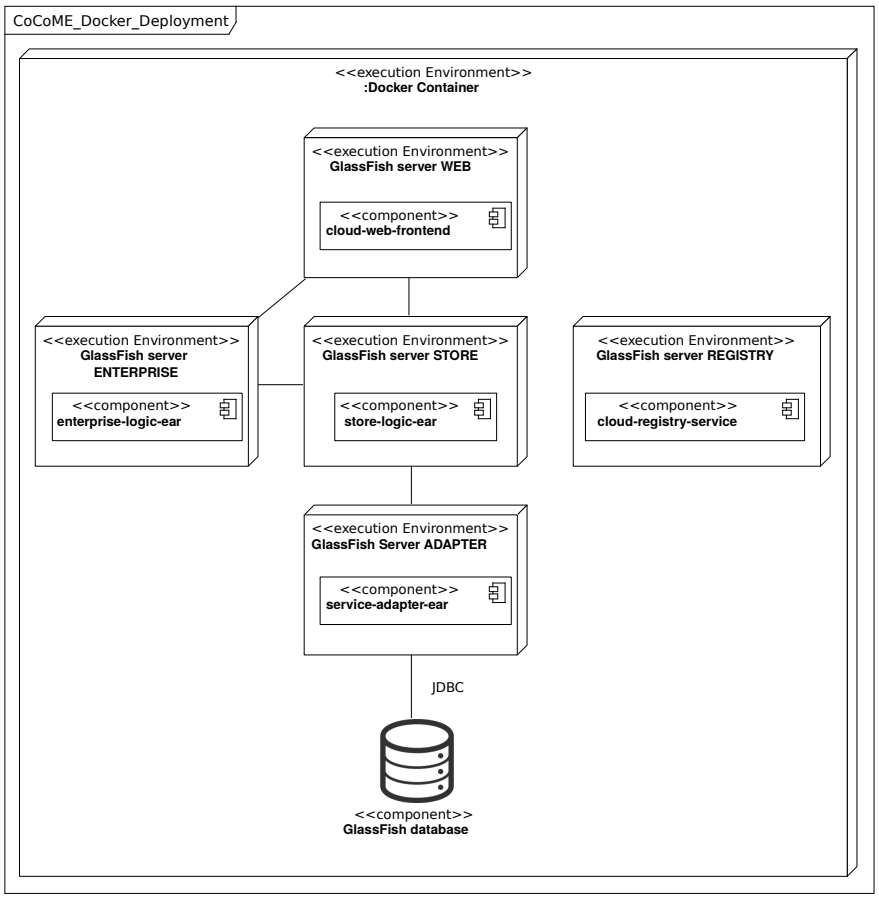

Figure 4.1: Deployment diagram CoCoME

To control the start of both containers, precisely the CoCoME and the Pick-Up Shop, another ~~specific~~ file is needed: the Docker Compose file. It ensures that the CoCoME Container is active, before the Pick-Up Shop container starts. This is necessary as the Pickup Shop requires a running instance of CoCoME to register itself.
Whereas CoCoME does not require the Pick-Up Shop, the ~~inversion~~Pick-Up Shop requires CoCoME. ~~is not correct.~~ Both containers need to communicate with each other. By default, ~~d~~Docker prohibits any outgoing and ingoing communication from and in a container. This is solved by opening specific ports through which the communication is possible. Which ports the containers can use is specified in the Docker Compose file as well.

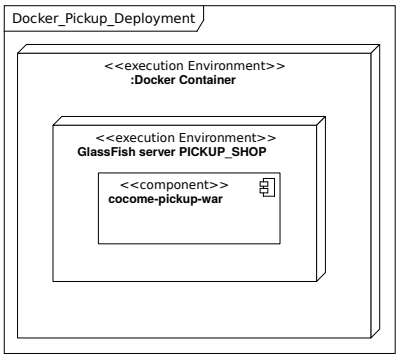

Figure 4.3: Deployment diagram CoCoME Pickup Shop

4.2 Adding a Mobile App Client

Adding a Mobile App Client ~~did~~addeddoes not require a modification within the hybrid cloud-based variant of CoCoME. The implementation was done using the Cordova framework and OnsenUI to provide a multi OS compatible Backend and UI [6]. The App itself is written in Typescript/Javascript. Fig. 4.5 shows the ~~principal~~primary classes and their relationships.

The *Navigator* is the primary class that manages the pages. The pages consist of two components: The *Page* itself and its *PageState*. The *PageState* is used to store and transfer the current status of a page. There are currently six different pages available: *IndexPage*, *SearchPage*, *ItemPage*, *CheckoutPage*, *CartPage* and *LoginPage*. For the sake of clarity, they are subsumed under the generic terms *ConcretePage* and *ConcretePageState*.

Pages use components. Such components are ~~i.e.~~for instance the *Navbar* or the *Searchbar*. These components are abstract descriptions of UI elements that are connected to the actual *HTML-elements* via Knockout.js. By using Knockout.js, changing values of a component results in an immediate change of the UI. Besides, the App Client retrieves information of the CoCoME system. As mentioned in 3.1.2, the Client is not able to access the CoCoME system directly. Therefore, the pages use *Services* provided by a *ServiceHolder* to call the *AppController's* ~~Rest~~REST-API. The *AppController* is written in Java using the SpringBoot framework and converts the ~~Rest~~REST-requests of the App Client to SOAP-Requests in order to match the CoCoME-API.

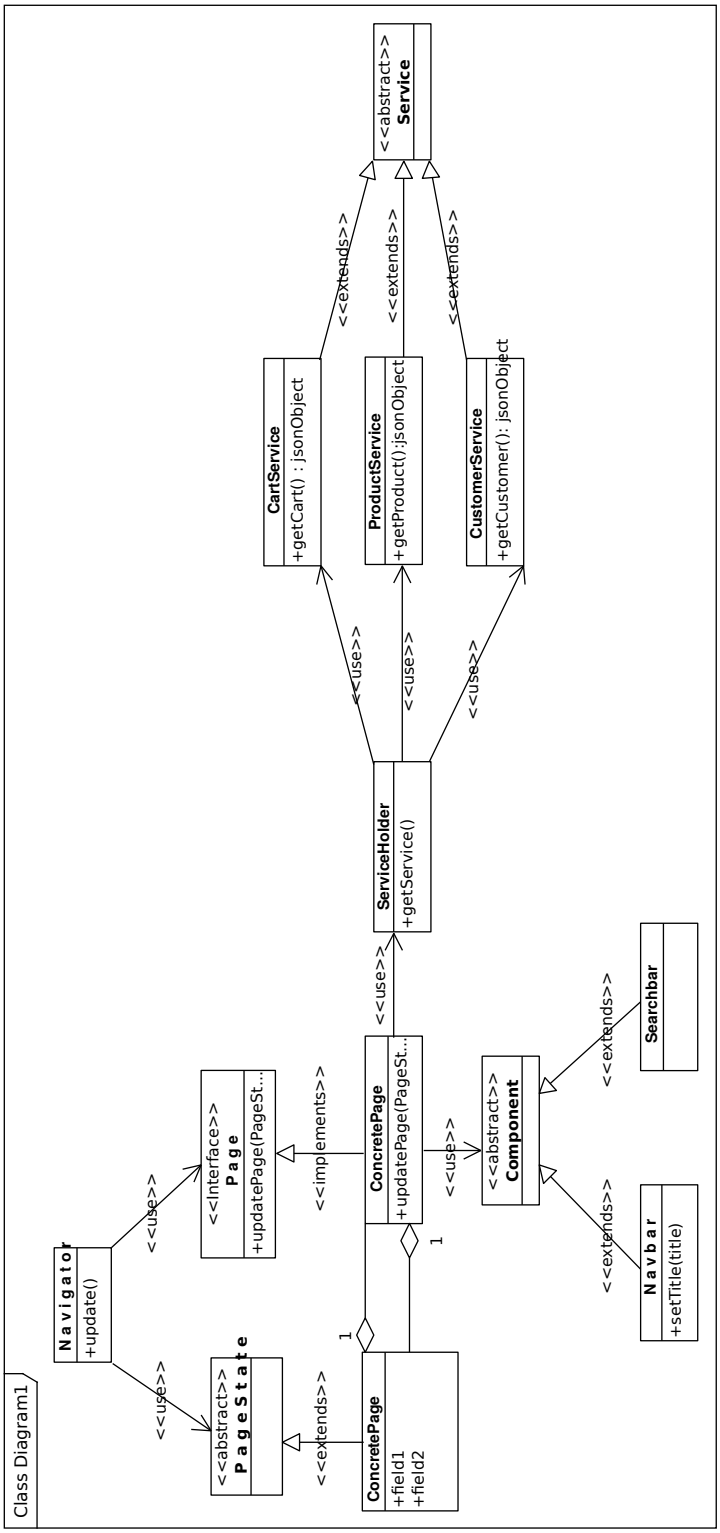

Figure 4.5: Primary Classes of the App

4.3 Using Microservice Technology

The Microservice evolution scenario of CoCoME transfers the functionality of the hybrid cloud-based variant of CoCoME into a Microservice architecture. The service~~that~~ is using REST communication between the different services. The implementation is done using Java EE as programming language and Glassfish as deployment server. The Java Persistence API is used to store elements in a relational DB. The Java API for RESTful Web Services (JAX-RS) is used to standardize the REST-communication between the different ~~M~~microservices. To serialize and deserialize Java-classes to XML- or JSON-files, the Java Architecture for XML Binding (JAXB) is used.

Fig. 4.6 demonstrates the project structure of the *Store* ~~M~~microservice. The other services have the same structure. Therefore, they are not shown explicitly. Each service project contains three sub-projects, namely *name*-service-ejb, *name*-service-rest and *name*-service-ear. The first sub-project contains the core logic as well as the persistence functionality. The second one provides the RESTful webservice. The last one is used for packaging the projects to ear files.

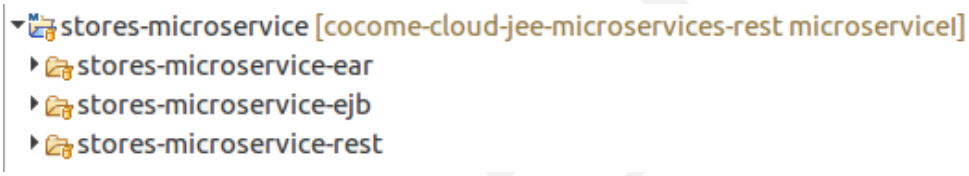

Figure 4.6: Project structure Microservices

Bibliography

[1] U. Goltz, R. H. Reussner, M. Goedicke, W. Hasselbring, L. Märtin, and B. Vogel-Heuser. Design for future: managed software evolution. *Computer Science - Research and Development*, 30(3):321–331, Aug 2015.

[2] R. Heinrich, S. Gärtner, T.-M. Hesse, T. Ruhroth, R. Reussner, K. Schneider, B. Paech, and J. Jürjens. A platform for empirical research on information system evolution. In *27th International Conference on Software Engineering and Knowledge Engineering*, pages 415–420, 2015.

[3] R. Heinrich, K. Rostami, and R. Reussner. The cocome platform for collaborative empirical research on information system evolution. Technical Report 2, 2016.

[4] S. Herold et al. CoCoME – the common component modeling example. In *The Common Component Modeling Example*, pages 16–53. Springer, 2008.

[5] H. H. Olsson, H. Alahyari, and J. Bosch. Climbing the" stairway to heaven"–a mulitiple-case study exploring barriers in the transition from agile development towards continuous deployment of software. In *Software Engineering and Advanced Applications (SEAA), 2012 38th EUROMICRO Conference on*, pages 392–399. IEEE, 2012.

[6] J. Schnabel. Mobile application client for a cloud based software system – practical course report, Karlsruhe Institute of Technology, WS16/17. `https://github.com/cocome-community-case-study/cocome-cloud-jee-app-shop/blob/master/doc/SQEE1617_MobileApp_Schnabel.pdf`.

[7] N. Sommer. Erweiterung und wartung einer cloud-basierten jee-architektur – practical course report, Karlsruhe Institute of Technology, SoSe17. `https://github.com/cocome-community-case-study/cocome-cloud-jee-microservices-rest/blob/master/doc/report.pdf`.