Application development with Microservice Architecture in Enterprise Java
A Spring Boot Case Study

Lennart Cockx
Guangming Luo

May 23, 2017

Contents

Abstract

In this report we will explore various Java technologies and analyse what features they offer. The focus of this project is to find which frameworks are most suitable for quick prototyping of a business application. Constraints and other requirements will be defined and are then applied to a specific case study. At the end of this project we should have an excellent understanding of the technologies involved to make a online cash register application that handles orders and payment in restaurants. [WIP]

Index terms— JavaEE, Spring framework, Cloud, Microservices

1 Project Objective

1.1 Proposal

This master thesis is commissioned by Faros, a Cronos group company specialized in the development of Java web applications and primarily focused on designing Rich Internet Applications. Development within Faros is done mainly with Vaadin, Spring, HTML5, ExtJS, JavaScript, JSP, JSF, NoSQL and more.(Faros, 2016) This means that during this thesis, special attention is given to these technologies in regards to design choices. Naturally, comparisons with other possible frameworks will still be done and valid arguments will be set forward to confirm these decisions.

The following case study was proposed by Faros: designing an online cash register that can be used in restaurants or at events. Existing cash register systems are expensive and for smaller establishments or temporary events, this investment is not worth it. The idea is to use existing devices from employees and customers to handle the ordering and payment process.

This project is an opportunity for us to learn about the frameworks and technologies used for the development of rich, interactive web applications. Additionally, we want to explore the best methods to quickly build a web application from scratch. Some key aspects that we will look out for are: Documentation availability, speed of new project setup, customizability, integration with other frameworks and security features. When given the corresponding documentation, it should be possible to reproduce this application. During the design and development of this project we want to describe the problems we faced and how we solved them. This will give other students the option to learn from our mistakes and hopefully help them increase their understanding of Java EE concepts.

1.2 Concept

At the start of the project we discuss the key aspects of this project with our coaches at Faros. Based on the requirements provided by them, combined with our own interests, we will define the general field we will be working in.

After we have a good overview of what aspects must be included in this project (or should be avoided), we can start deciding core features that will determine which technologies will be appropriate for this project. These choices are based on the aspects we think are needed for developing a web application based on the aforementioned requirements. This should give us a list of possible systems and frameworks that could potentially be

used during this project.

Now that we have a list of possible systems, we can start to think about the case study we want to develop. Given the general outline of the project, we can begin selecting the appropriate tools and frameworks that fit this case study. This will shorten the list to technologies specifically applicable to this project. For certain frameworks it will be difficult to obtain a shorter list, as they might all comply with the requirements we set. In this case we will make a selection of items to keep the scope of the project realistic. Usually the most popular frameworks will be selected. If a deviating set is chosen this will be elaborated on in the corresponding chapter.

Now that the tools are known, the standard procedure for application development will be followed. Relevant documentation will be constructed including a use case analysis, feature description (Nice-to-have versus Must-have), navigational models, hierarchical task analysis, UML diagrams and market research. After this we can start working on getting familiar with the chosen enterprise Java frameworks.

As soon as we are sufficiently proficient with our chosen tools, we will start working on a case study where we use the knowledge we obtained to build a functional prototype for a realistic use case scenario. During the project we might come to the conclusion that certain choices we made were suboptimal or simply do not allow further progression. In such cases, we will adjust our trajectory and change to the appropriate tools. Any time such a decision is made we will mention it in this report and provide the matching argumentation as to why we made the switch.

1.3 Project Requirements

1.3.1 Constraints

As this project is appointed by Faros, there are certain guidelines we will follow during this study. The primary constraint is the usage of Java. As Faros is a Java consultancy company, the technology stack we use should match what the employees there utilize. Within Java we are still free to choose what frameworks will be used. Another constraint is that we should not aim to create a perfectly working application with extensive styling and fail-safes, but rather focus more on experimenting with various different systems to get a better overview of the big picture and compare these systems with each other. There are also certain project-specific constraints that will be detailed in the case study.

1.3.2 Design focus

During this thesis we will develop a application based on a case study as mentioned in section 1.2. This case study is a device used for guiding development, but it is not the primary goal of this thesis. Instead, it is a way that allows us to explore the technologies used in application development on an enterprise level. In particular, we want to look at the microservices architecture and what advantages or disadvantages this has over more traditional web development. We will explore the microservices architecture in detail and research all related technologies and principles such as discovery, load balancing and security.

1.4 Existing systems

Before creating our own Online cash register application we searched for similar systems that already exist. One of such systems is Gloriafood (Glo, 2017). This is an online system that allows any restaurant

owner to sign up and start receiving orders online. A extensive layout is available where you can add a menu and payment options.

Similarly, there is ChowNow (Cho, 2017). This service also handles online ordering and payment with no extra input required from the restaurant manager. ChowNow will also create a custom branded mobile application to showcase the brand and food of the restaurant.

Both of these systems are focused on the online ordering aspect and not so much on allowing people to use their devices to order food on location. In this project we want to put the emphasis on the latter part, creating an application that is meant to be used at the location itself.

2 Research and Methods

2.1 Approach

We will take a quantitative approach in this paper, exploring and comparing frameworks for each of the features required for this project. This includes: General frameworks providing a way to couple web resources to Java code. Persistence providers, SQL or noSQL databases to store customer information. Object relational mapping tools to generate mappings from existing data-stores. Security systems to allow authentication and authorization in a safe way. Front-end frameworks, providing styling options and client side scripting. Everything related to the case study is detailed separately in section 3

2.2 Java EE Landscape

We will not discuss every existing tool in the Java environment, but it might be interesting to have a look at the current state of Java EE in the professional world. We will use the numbers from a survey published by Zeroturnaround to do our analysis.(Maple, 2016) This website has a blog for Java web developers and will display a certain bias towards this usergroup. For more general statistics please refer to Stackoverflow's developer survey.(Dev, 2017)

Currently the majority of Java developers work on full-stack web applications (67%), a smaller group works on back-end code only (18%) with the rest of the developers divided into batch scripting, mobile apps, desktop applications, etc... This shows us that only in a fragment of the projects specialization is beneficial. Usually developers know and work with both front-end and back-end technologies. This is also what we will do during this project, implementing both the front-end and back-end of the application.

So which front-end and back-end frameworks are used by developers? The most commonly used web framework is Spring MVC (43%), followed closely by Spring Boot (29%). Practically, this means that over 70 percent of respondents use the Spring framework. Spring Boot itself is not really a framework, but rather a streamlined entry point to the Spring ecosystem with a "Opinionated Defaults Configuration" approach. (Spr, 2016)(Webb, 2013) Other common web frameworks are JavaServer Faces (19%), Vaadin (13%) and Struts (7%). Other frameworks exist but are too numerous to list. For this project we will limit ourselves to analysing the 3 most popular frameworks. This should give us a good idea of the advantages that web frameworks offer, and how they compare. Spring Boot

in particular looks promising as it provides autoconfiguration for many properties which will be advantageous for people new to the Java EE platform. It is also worth mentioning that 17 percent replied that they do not use any framework. It seems that for certain projects using a framework is not worthwhile.

When working with frameworks, developers will make use of build tools to avoid having to manually invoke the steps in the compilation process. The most popular build tools for Java are Apache Maven (68%), Gradle (16%) and Apache Ant(11%). While using a different build tool won't directly affect the result of a project, it is important to make an appropriate choice from the start. This build tool will be shared by all teammates in a project.

To deploy the application, an application server is needed. For production, Tomcat (42%) is the primary choices for running Java web applications. The rest of the users are quite evenly distributed over the other application servers with the most popular being Jetty (12%), Wildfly (10%) and JBoss EAP (7%). Developers have the choice of running their own application server and deploying their apps to it or making an application with an embedded server. For us, using an embedded server will get our project up and running faster. We will also run an external server to learn how this changes the development cycle.

While not directly related to specific technologies, the survey also asked participants to indicate whether they have adopted a microservices architecture. This is interesting for us to look into as it was mentioned by our coaches at the company. It seems that microservices have become quite popular in recent year for a variety of reasons.(Gupta, 2017) Although it differs from the usual approach we learned during our education, trying to use this architecture in our application might provide valuable insights if we succeed.

Something that is not mentioned in the survey is security. Basic security is expected in any web application, but implementing a secure system is difficult. Java EE already provides the building block for securing a web application, but part of the design is still up to the developers themselves.(Oracle, 2013) Modern Java web frameworks go a step further however, giving developers a production ready security system out of the box. E.g. with Spring boot using Spring Security offers a system with built-in authentication and authorization. Safety and resilience are key factors that have to be taken into account when developing our application.

Another aspect not discussed in the survey is front-end frameworks. Commonly, the interface of a web application is designed using HTML, CSS and Javascript. On top of this CSS and javascript libraries like Bootstrap and JQuery can be used to speed up development by giving access to pre-made modules for frequently used interface elements. Front-end frameworks such as AngularJS and libraries like React can be used to speed up development even further. They offer tools and design guidelines to make designing interactive interfaces efficient and fast. Although we will also discuss these frameworks in this paper, priority will be given to technologies related to the back-end while styling will take a back seat.

Finally, what types of databases are used to store information? The most used database are OracleDB (39%), MySQL (38%) and PostgreSQL (29%). All three are relational databases. The most used NoSQL databases are MongoDB (15%) and Redis (8%). Note that these results are multiple choice, as some projects use

multiple databases simultaneously. For the case study, the selection of database will depend on the type of data we want to store. Most of our experience is with relational databases, so trying out a NoSQL database will be valuable experience.

Hopefully this section gave some insight into the technologies used in Enterprise Java Development and how we will approach them in our project. We want to try out various technologies and determine those most suitable for the use case and our development style. In the next section a more detailed explanation of the technologies will be given. During the exploration of these frameworks we will already create a prototype for the case study. This will give us a better idea of which features would be beneficial for this specific project.

2.3 Case study summary

2.4 Comparison of technologies

Even before we specify the case study for our project we can already try out the different technologies discussed in the previous subsection and make a selection of those we deem appropriate for this master thesis. Any choices directly related to our specific case study will be discussed in section 3.

2.4.1 IDE's and build tools

While the development environment and build tools are essential in application development. The choice itself is based on the preferences of the individual programmer. This project started out with the Netbeans environment and Maven for dependency management. When we started using Spring we tried out the Spring Tool Suite based on Eclipse, but we finally settled on IntelliJ IDEA and Gradle. IDEA offers many tools often needed during development such as a database editor, gradle/maven task execution, ORM modules, bitbucket plugin support, REST client for testing, superior auto-completion and built-in support for many frameworks. There are several other advantages to using IDEA, but these were the most determining factors causing us to switch environment. For the build tool, Maven is actually not inferior to Gradle. Both support the dependency injection that we need for this project. The primary reason we decided to try it out was curiosity. Gradle is currently the default build tool in Android development with android studio, and it is also utilized in Grails and Netflix OSS. When comparing them directly, Gradle seems to offer a much richer feature set. (Gradle, 2017) When using this build tool in the project, we came to the conclusion that these extra features were not really a necessity for our specific application. We did stick to Gradle in the end though, as the DSL syntax that it uses makes for a much more readable build file in comparison to the XML notation that Maven uses. We did stumble upon some issues while using Gradle. It also uses the Maven repositories for its dependencies, which means that if you want to add your own dependency to the local Maven repository, you will need to enable the maven-publish plugin within Gradle and add the relevant build code to bundle it correctly. Compare this to Maven, where executing the install task achieves the same result. In general, the choices we made for this section are based on our own preferences and experiences during the project.

2.4.2 General framework

The general framework will be the decisive element in the case study we develop. It will determine what modules we can use during development and what the general architecture looks like. Once we choose a certain framework, it will be difficult to convert to another. Our first iteration of

this project started off as a plain Java EE project which served pages with the JSF framework. Apart from the configuration issues we had when first starting out, this worked quite well for simple servlets and beans. We created the necessary entity beans and business logic in an EJB package. Servlets, JSP files and static web content were packaged into a web archive. Both these packages were then combined into an Enterprise Application archive and deployed to a glassfish server. There were two big problems with this. First, configuring the application and server during the first setup takes a significant amount of time. The data sources must be configured in both the .ear as well as in the application server itself. When changing database or server this configuration must be redone. During development this can be a burden since we have to reconfigure should we want to make significant changes. Also, when we implement a microservices architecture, it will be necessary to repeat this configuration for every service. The other major problem is technology selection. Most technologies we use will be based on the specific requirements of the case study. For many other parts of the project there are still technologies that are used for any type of project, like what application server do we use?, what object mapping tool? and what security framework? plus many other systems that we didn't know the existed yet. Figuring out what systems we need and how to configure them will take a significant amount of time and they won't necessarily be the optimal choices. To avoid spending too much time here, we can use a framework which already implements these features for us in a production ready package. This way we can already start exploring and working with certain features without having to implement it all ourselves. Most frameworks still allow changing the preconfigured defaults to different system where necessary. Searching for these prerequisites led us to the Spring framework. Or rather, the Spring Boot ecosystem.

Like mentioned before, Spring Boot provides an opinionated toolset which gets you up and running in no time. All the necessary components are automatically configured and ready for deployment. For us this is ideal, we can quickly get started and we can learn about the necessary modules for deploying a Java Enterprise application by examining the code generated by the Spring Boot Initializer. Another interesting aspect about Spring is that it is considered an excellent base for designing applications following a microservices architecture. (Shelajev, 2016) (Delmas, 2015) Of course there are other frameworks that offer similar autoconfiguration and tools for implementing microservices, most notably Dropwizard and Vertx. The reason we decided to go for Spring Boot first is that, due to its popularity, it has great documentation and is very comprehensive, containing an extensive set of modules for all things required for modern web applications. Additionally, it is compatible with a wide range of other frameworks, so if we would need specific features from another framework this is still possible. This choice proved to be very good, we managed to reimplement the existing project we had in a matter of days and continued working in this framework. More details about our further steps in section 2.5.

2.4.3 Persistence providers

Data storage is practically utilized in any web application. We will need at least a way to store information for authentication purposes. In addition, we will likely need to store data related to the restaurant and menus. When storing relational data, a SQL database is typically used. Because we are taking a microservices approach

and want to explore scalability, a NoSQL database might be more suitable.(Mon,) When handling diverse data types, improve scaling and modifying existing schemas, a NoSQL database can be beneficial. If we want to use this technology, it will take time to adjust. The principles behind designing NoSQL storage is completely different that relational data storage. Since we are going for microservices, it is actually possible to use a different persistence provider depending on the service. E.g. The default configuration for Spring Security makes use of a relational database. One table contains the usernames and passwords. This is connected with a one-to-many relationship to another table containing the role-user mapping, which in itself is connected to a table containing all the possible roles. This is a typical structure that would require a relational database. Using a NoSQL database is also possible, but requires a custom implementation of the UserDetailsService from Spring Security. On the other side we have to store order information from customers. Each order entry can hold an unstructured, arbitrary amount of items. It could also contain comments by the customer, but this field is not necessary. For this kind of data NoSQL might be more appropriate. If there are certain fields that we want to add or remove later on in development this also becomes much easier. Our final project can have a combination of MySQL and NoSQL providers, using the most suitable for each service separately.

2.4.4 Object relational mapping tools

ORM tools are useful utilities that offer automatic creation of Java entities from a database or vice-versa. It also makes an abstraction layer on top of SQL queries, which makes development much faster and prevents having to write SQL manually. Since it is likely that we will make use of NoSQL database in some of the services, an ORM is not useful anymore, since the traditional relational structure is not used in a system using NoSQL. The usage of ORM tools will hinge on the data structure we decide on.

2.4.5 Security systems

Security is something that is expected in a web application that stores information or provides functionality which should not be publicly accessible. In our case management of the restaurants and modifying the menus should only be permitted to employees of the corresponding restaurant. Account information, restaurant statistics and the order overview are all sensitive information and, again, should only be accessible by employees.

Java EE provides various modules to secure a web application, but many of these modules still require quite a bit of configuration and certain features need to be implemented by the developers themselves. Things like encryption of passwords and https authentication might need to be created manually. Spring offers more extensive systems for securing a web application, available for use in any supporting framework. Spring security is a system that can be added to an existing project and enables a secure login and protection for specific pages and information with minimal configuration. The biggest hurdle that we will likely face is implementing these security features in a multiservices architecture. We will likely have to implement a separate service that handles security for all other services. In the case of Spring, this could be achieved by making an OAuth2 server.

2.4.6 Front-end and Styling frameworks

For styling our web application, we can make use of several Front-end frameworks

and libraries. Usually the general web frameworks allow the serving of static web pages in the form of HTML and CSS pages. While making a beautiful interface is not the main focus of this project, we can still make use of a styling library such as Bootstrap or MDL to speed up and improve the design of our pages.

Another trend in web development are responsive and reactive web pages. With the increase in mobile devices around the world, it cannot be assumed that people will visit a website on a large pc screen. Rather, most users will access web services from their phones or tablet devices. This means that when building a website we should try to make the interface scale properly for any size of device. Going even further than that, it is possible to improve performance by only reloading separate elements of the page and not redirecting to a new page each click. For this we will look at popular front-end modules like AngularJS and React.

2.5 Implementation

In this subsection, the actual process for the implementation of the project will be laid out. We will present a chronological progression of the project and discuss what we implemented and what roadblocks we encountered.

As mentioned in section 2.4.1, we initially started out with a project made in Netbeans using Java EE serving JSP pages. This was a monolithic application that included client, business logic and database. This was our first experience with Enterprise java development. We had a locally hosted Payara server which we used to deploy our enterprise application archive. A simple SQL datastructure was running on an external MySQL server, which can been seen on figure 1

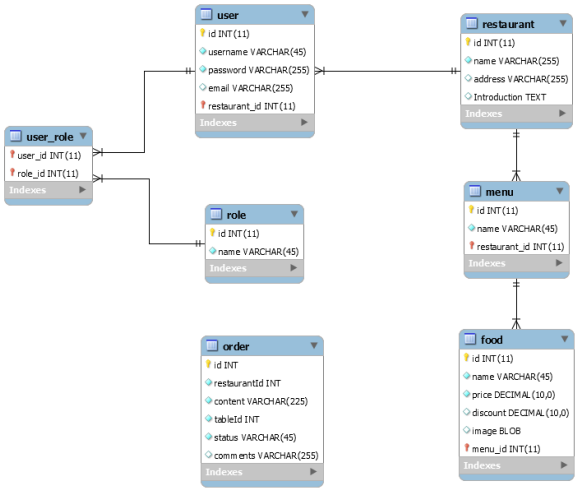

Figure 1: Original MySQL datastructure: This image shows the prototype relational model we used while working with a single SQL database

After our initial research we switched over to Spring Boot and recreated the original project. We still kept the previous database. The Payara server was replaced by the internal server that Spring Boot uses. This is a tomcat server that is packaged together with the resources and java code into a single executable .jar. This is great for testing as we just have to execute this one file to start our application server, deploy the project files to it and launch the server on the port we specified.

For this application we used hibernate to generate Java entities from the database model we designed. There was also a basic version of Spring security running in this project that only allowed access to certain pages based on the role assigned to the account you logged in with. When logged in, users could access the restaurant page they were assigned to. At this time the page was a blank template on which the corresponding restaurant's name was shown. From the start we chose to split up the functionality of the program into different modules, we hoped that this would enable us to more easily divide the project when changing to a microservices architecture.

Now that we have a functional Spring Boot application, we can start adding the services we need for our case study. The final application should have the following services:

- **Restaurant service**
  Provides the name, address and a description of each restaurant registered with our webapplication.
- **Menu service**
  Holds the menu categories for a specific restaurant.
- **Food service**
  Has a list of food items associated with a category and registers their price.
- **User service**
  Manages everything related to the login and authorization of employees and managers.
- **Order service**
  Takes and stores orders made by customers.

Implementing these services went smoothly as we can avoid writing a lot of boilerplate code by using the autoconfiguration from Spring Boot. When creating the various services, they are configured with the *@Service* annotation. Functionally this annotation works the same as *@Component*, and allows this class to be used with automatic bean detection on the application classpath. When used, these classes will be instantiated as a singleton beans. The primary reason the *@Service* annotation is used instead is to indicate what type of component we are creating as defined in Domain-driven design.(Com, 2015)(Hoeller, 2017) We purposefully make a distinction between presentation, service and data access modules to make classification easier.

Currently, the services use entities to access relevant data from one MySQL database. The process used to generate these entities is as follows: First, a UML diagram was created including all tables and relations required, seen on figure 1. This diagram was then forward-engineered into a local database and filled manually with sample data. Then, the Hibernate provider for Java EE persistence was used to generate Java entities based on this database. This gives us entities which can be retrieved and persisted on the SQL server. To do queries on the database, e.g. to search for users or restaurants, we would have to write our own queries. Luckily Spring Data gives us a *JpaRepository* class that we can extend to generate the queries automatically. By extending this class, we immediately have access to all basic operations to create, modify and remove entries from the database. A simplified notation can be used to create the necessary queries. For example, we need a way to find restaurants based on ID or name. The only thing we have to do is create a class that extends *JpaRepository* and declare 2 methods: *Restaurant findRestaurantByID(int id);* and *Restaurant findRestaurantByName(String name);*. Spring Data will configure the rest.

Once we specified these services with the correct annotation, they can be used in the business logic. Using *@Autowired* allows us to refer to a registered component or service. At this stage the services are utilized from within the controllers that we use to map the webpages served by our application. *@RequestMapping* is used to map certain addresses to their corresponding HTML pages. These pages are parsed with the Thymeleaf templating engine to include the data we receive from the services. At the start we only used Thymeleaf to pass database information to the web pages. We soon had multiple pages, often with duplicate code. This made us look into Thymeleaf fragments.

All commonly used interface elements were added on a general file, where the content was filled in with fragments depending on the purpose of that page. A good example of this is the navigation bar, which should be visible on every page.

At this point we have a functional web application that has pages to display restaurant information from the database. These pages are protected by a basic Spring Security system. Users can only access the restaurant they are assigned to. An administrator account can create new users and bind them to a specific restaurant. To limit the amount of similar pages, we wanted to show or hide certain parts of a page depending on the authorization of the current user. While this is easily implemented by using the controller model parsing from Spring MVC, this would require us to pass security configuration in all the methods of each controller. A more streamlined solution would be preferable. This lead us to the Spring Security dialect for Thymeleaf. Including this dependency allows the usage of Spring security entities directly in the HTML code, which prevents a significant amount of redundancy in the controllers. This can be used, for example, to show the name and privileges of the logged in user.

When certain errors occur during authentication or on our pages in general, it can be useful to have an error page to display what went wrong. We used the Spring Boot Actuator package to create a default error page. This page does not use any styling and has a limited amount of information, so we implemented a custom *ErrorController* that can show stack trace information and matches the design of our other pages. During this time we finished much of the business logic and static pages for the case study. However, It was still a monolithic application running on a single SQL server. The next step would be to start creating REST controllers to enable REST requests to the services. This is needed to build a REST API, allowing access to the services after splitting them up in separate modules.

Changing the output of a controller to is relatively straightforward. We make new controllers for each service using a *@RestController* annotation instead of *@Controller*. Instead of returning String templates, the mappings should be changed to return the entity classes. In practice this will return a JSON object to the client doing a REST call. Having these REST controllers allows for effortless communication between the various services and the application server. Now we are in a good position to start separating the services into different modules. The result of this separation is shown in figure 2.

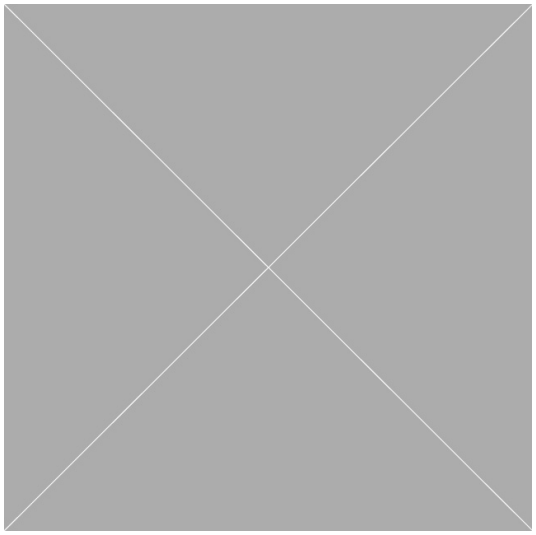

Figure 2: Project structure after splitting into separate modules: Instead of the single monolithic application, the project now has a separate module for each service. Each module consists of an REST access point, a service that handles the REST calls and an individual database. This database can use SQL or NoSQL depending on the data it contains.

Splitting the existing project into modules that can run and function proved to be

quite a difficult task. One of the major problems was the database structure. It is possible to have all services connect to one central database, but this conflicts with the microservices concept. There would still be a central point of failure and scaling would still be limited to replicating the database on multiple servers or using application specific sharding. Instead of that we created separate databases for each service. We had to rethink the way we approached our data, as direct relations are not possible any more. Another disadvantage is that the initial migration from a monolithic application takes a significant amount of time. For us it took around one week to translate the single large project into a functioning microservices system. After getting the first few services to work, this process does speed up. Because the services are simplified to their most basic form, implementing additional services after the first one is only a change in the business logic. During migration, we also noticed that certain existing systems break when moving to a microservice ecosystem, Spring security in particular. Instead of using the default configuration, an additional services that provides security for all services is required. Similarly, many of the technologies which are easily used in a monolithic application become much more difficult it is split into individually functioning parts. We have to take into account that these services are running on different ports and could be running on completely independent servers.

Although there were several issues changing over to microservices, it did also bring noticeable advantages for development. The different services now have a REST API that returns a JSON object instead of Java entities. This is very convenient when making additional clients for accessing our application. We can do a http *GET* or *POST* to the corresponding services and we will get JSON back that can be immediately used to access the necessary information. We used this to create a native android application to be used by employees at a restaurant. More detail about this here: Another interesting aspect of this structure is that we can build modular web pages that get their information from multiple services. If a certain service is down or is experiencing connection problems the page can still be loaded. The missing content is simply not shown and we show a message on screen to indicate that a certain service is having issues. Thanks to the microservices architecture, only basic exception handling is needed to implement this.

We are now at the point where we have a functional application build upon a microservices foundation, but there are still some practical issues with the current structure. Currently, there is one instance of each service running in their own container, each bound to a specific port number. To access an API, the address and port of the service have to be known. For the application server accessed by browser clients these adresses can be hard-coded, although this is not recommended.(See the next paragraph about discovery.) For other clients this is more difficult, they do not know which addresses they can use the access the APIs. Hard-coding this would be detrimental as it would require an update of the application each time infrastructure changes are done. For example when changing the host of a certain service. The solution we used was creating an extra service that functions as a gateway for all the API endpoints. The Spring cloud dependency offers the foundation for this gateway under the form of Zuul, developed by Netflix. (Netflix, 2017) Implementing this gateway allows us to accept requests that include the service name and route these requests to the corresponding service by specifying their addresses in the Zuul gateway configuration. The API gateway functions as a reverse proxy, illustrated in figure **??**.

3 Case study: Online cash register

3.1 Introduction

The goal of this case study is to create a service that can be used at events and in restaurants to process orders made by customers, relay these orders to the waiters and chefs and handle the payment. This service should include an application that clients can use to view the menu and order items from it. Waiters also have their own application that can place and acknowledge orders. These orders will be send to a central system that manages all the required information. See fig. 3 for a use case diagram.

3.2 Use Cases

In this case study we can identify several distinct actors that will interact differently with the application.

Customer The customer is the primary user of this application. They access the menu of a specified restaurant or event. They can place orders, either via the application itself or from a waiter with their own device. Finally, they can pay for the orders that they placed. Again, either via the application or a waiter.

Waiter Like the customers, the waiter can access the menu of their restaurant and place orders for their clients. In addition to this they can display an overview of the currently queued orders, with the corresponding table numbers or location. They will also handle payments from customers through the application, or accept payment in other forms of currency.

Chef The chef can access similar functions as the waiter. Additionally, they can modify the status of an order and mark it as completed or in progress.

Manager The manager can modify the menu and change prices. They can also create accounts for employees and give them the corresponding privileges.

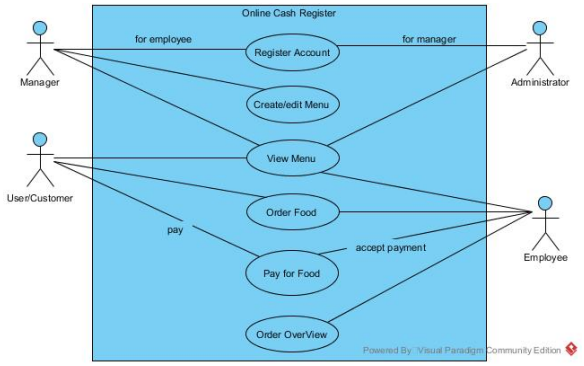

Figure 3: Use Case Diagram

3.3 Must-haves and nice-to haves

As soon as we had the use cases defined, we can specify what features are absolutely essential and which are optional. This was done in collaboration with our coaches at Faros and gave us the following specifications.

Must-Haves

- The system can be used by multiple managers or event-organizers simultaneously. (*Multi-tenancy*)
- A central application will handle orders and payment. (*Central application*)
- Waiters can use an application to send the orders of customers to the kitchen and accept payment. (*Employee interface + Payment system*)
- Customers can view the menu of the restaurant/event with their mobile device. (*Customer interface*)

- An order overview is available for the waiters and chefs. It shows the completed orders and orders which have to be made. (*Order overview*)

- Customers can add comments to certain orders. E.g. no tomato (*Special requests*)

- Customers can place their own orders with their mobile device. (*Customer Mobile ordering*)

- Prevent customers from making ridiculous orders or misusing the system. (*Misuse prevention*)

Nice-to-Haves

- Event organizers can run the central application on their own local network. (*Local network hosting*)

- After an order is finished, the completed order number is shown on a big screen. (*Take-a-number/Now serving System*)

- The manager can find statistics on his restaurant and orders via a web app. (*Order Statistic*s)

- Customers can split the bill or combine orders (*Bill splitting*)

- Provide an easy way for customers to use our system at an event/restaurant. Even if they dont know it yet. (*Easy access*)

- Customers can order additional food or drinks. This will be added to the total bill. (*Multiple orders*)

3.4 Technology choices

3.4.1 Application type

3.4.2 Framework choices

3.5 Design Choices

3.5.1 Client

We want to make an application for customers of restaurants and events. The customer can use this application to get an overview of the available menu items and prices. If the customers have decided, they can order the desired dishes straight from the application. Finally, the customers can pay for their meal with this application. We can differentiate a couple of different scenarios when the customer goes to a restaurant or event that supports this system. We assume that a web app is used. If a native or hybrid application is used, installing the app will be an additional step before using the system.

After (or before) the meal is finished, customers should be able to pay directly from within the application or pay a waiter. When using the application to place an order for multiple people, the customers should have the option to order and pay separately or order together and split the cost. For example: Three people want to place an order, two of which have access to the application.

- One person can place the order for the entire group.

- One person can order for himself and the person without the application. While the third person makes his own choice with his own device.

- The two persons with the application order for themselves, while the other person orders something from a waiter.

In all three cases the order will be combined into one order for that table. After the meal the bill can be paid for.

- One person decides to pay for the entire bill. This can be paid to a waiter or with his application.
- Each person pays for the part he ordered separately, with the price corresponding to what they ordered.
- Everyone decides to split up the bill equally, dividing the total cost over three people.

Typically, when people eat at a restaurant, they will place multiple orders. During the meal additional drinks might be ordered and at the end of the meal some people might want a dessert or coffee. We must design the application so it can accept multiple orders and display the sum of these orders at the end of the meal.

3.5.2 Employee

To be able to interact with customers using this system, the waiters and chefs will need their own version of this application. They will need to get an overview of the current orders placed and they should be able to modify and confirm these orders.

The order overview will have all the relevant information of the order: The different dishes and drinks ordered by the customer, the table from which the order was placed, the price and the status of the order. Additional information such as a time and date and an order number can be added for statistical purposes. The order overview could look something like displayed in fig. 4.

Order	Table	Status
2x Mac 'n cheese – 1x Coke – 1x Sprite	2	Ready to serve
1x Java coffee - 1x Nougat pie	5	Processing
1x Chicken Spicy - 1x Beef Indi - 1x Red wine hs bottle	13	Processing
1x Mussel menu 2per	7	In queue
1x Spr water bottle	13	In queue

Figure 4: Example Order Overview

Whenever an order is served, it will be removed from the overview and the next item will be moved up. When the bill is requested, all orders from the corresponding table will be added together. After payment the orders for that table are reset to allow the next customers to place their order.

In some cases, an order might be wrong or a mistake with the number of dishes is made. The application should be flexible and allow a waiter to make small modifications to the order to correct a mistake. Additionally, a customer might make a special request for his order. This can be due to allergies or plainly because of a certain preference. It might be useful to provide a bit of space to the order overview for extra remarks.

An order might contain a lot of different items. The full names of these items will take a lot of space in the order overview. Typically, a shorthand notation is used by waiters to more efficiently note down and read orders. A similar system could be used for the order overview. Also, for showing the menu to the customer we will display a vertical list with the name of every menu item. This list will contain detailed information like the name of the dish, the ingredients, pricing and possibly even a picture. These menu items can be sorted into categories for easy navigation. On the other side, the waiter does not need all the detailed information. Instead a compact and efficient layout is preferred to be able to easily take an order. This could be a grid layout with simple icons or abbreviations for each dish. See fig. 6.

☐	Vegetarian pizza	9.00 €
☐	Pizza Hawai	12.00 €
☐	Pizza Bolognaise	13.50 €
☐	Calzone	11.80 €

Figure 5: Customer Menu Layout

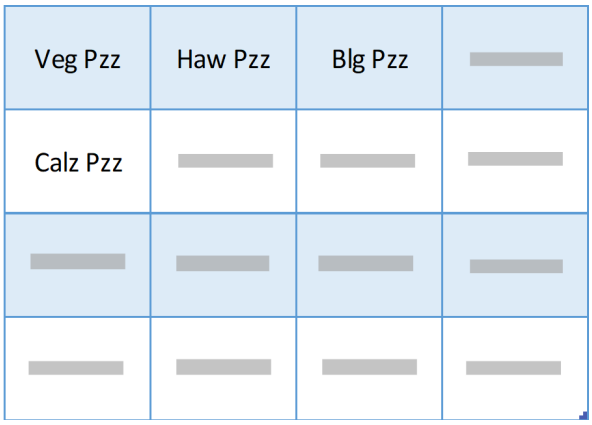

Figure 6: Waiter Menu Layout

It is possible that certain people with bad intent might try to place orders for ridiculous amounts of food, repeatedly place small orders to fill the system or even place orders for another table. Some of these situations can be prevented with proper programming like limiting the amount of orders one table can make each minute. Another way to prevent some of these issues is to have waiters confirm orders on their device before adding them to the kitchen. In day to day use, these scenarios will be rare, but its still important to think about the possible problems that might occur. While a user account for the customer might be desirable to track certain statistics and prevent misuse of the system (like mentioned above), requiring an account might not be interesting, as this can make the process of using the application slow and unintuitive for the customer.

3.5.3 Manager

The manager will be able to create new accounts for employees that work in their restaurant. They can modify properties of the restaurant and alter the menu and prices.

One of the most important aspects to consider here is multi-tenancy. The application should not only provide an interface for a certain restaurant but should be usable by different managers who own different restaurants. In other words the application should be generic and allow for customization for each restaurant separately.

3.5.4 Remarks and challenges

3.6 Major issues and their solutions

4 Results

4.1 Final Prototype

4.2 User tests

5 Conclusion

6 References

References

Mongodb and mysql compared. https://www.mongodb.com/compare/mongodb-mysql. (Accessed on 15/05/2017).

(2015). Difference between spring @component, @service, @repository, @controller. http://latest-tutorial.com/2015/01/19/difference-spring-component-service-repository-controller/#ixzz3PFFZtOsj. (Accessed on 17/05/2017).

(2016). Spring boot tutorial best complete introduction — dinesh on java. http://www.dineshonjava.com/2016/06/introduction-to-spring-boot-a-spring-boot-complete-guide.html#5. (Accessed on 28/04/2017).

(2017). Gloriafood - free online ordering system for restaurants. https://www.gloriafood.com/. (Accessed on 28/04/2017).

(2017). Online food ordering system & app - chownow. https://www.chownow.com/. (Accessed on 28/04/2017).

(2017). Stackoverflow developer survey. http://stackoverflow.com/insights/survey/2017. (Accessed on 03/05/2017).

Delmas, C. (2015). Why spring boot? https://cdelmas.github.io/2015/11/01/A-comparison-of-Microservices-Frameworks.html. (Accessed on 15/05/2017).

Faros (2016). Faros master thesis proposal.

Gradle (2017). Gradle vs maven, feature comparison. https://gradle.org/maven-vs-gradle. (Accessed on 15/05/2017).

Gupta, K. (2017). Why are microservices getting popular now-a-days? https://www.freelancinggig.com/blog/2017/01/25/microservices-getting-popular-now-days/. (Accessed on 03/05/2017).

Hoeller, J. (2017). @service annotation - spring api. http://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/stereotype/Service.html. (Accessed on 17/05/2017).

Maple, S. (2016). Java tools and technologies landscape report 2016. https://zeroturnaround.com/rebellabs/java-tools-and-technologies-landscape-2016/. (Accessed on 03/05/2017).

Netflix (2017). Github - netflix zuul. https://github.com/Netflix/zuul. (Accessed on 23/05/2017).

Oracle (2013). Security mechanics in java ee 6. http://docs.oracle.com/javaee/6/tutorial/doc/bnbwy.html. (Accessed on 15/05/2017).

Shelajev, O. (2016). Why spring is winning the microservices game. https://zeroturnaround.com/rebellabs/why-spring-is-winning-the-microservices-game/. (Accessed on 15/05/2017).

Webb, P. (2013). Spring blog - spring boot introduction. https://spring.io/blog/2013/08/06/spring-boot-simplifying-spring-for-everyone. (Accessed on 03/05/2017).

7 Appendix

7.1 List of technologies used

Underneath, a list of technologies used during this project can be found. Technologies present in the final application will be indicated in **bold**.

- **Spring Boot** - Bootstrapper used on top of Spring MVC
  - **Spring MVC** - Java web development framework
  - **Spring Security** - Java web security dependency
  - **Thymeleaf** - Templating engine for web pages.
    - **Thymeleaf Security Dialect** - integration module for Spring Security
  - **Spring Actuator** - Application endpoints for monitoring and management
  - **Spring Data JPA** - Persistence API for database access and Object mapping
  - **Spring Devtools** - Spring-specific tools useful during development. E.g. Hotswapping, auto-disable template caching and automatic restarts.
- **HTML** - Markup language used for web pages
- **CSS** - Markup language for web pages styling
- **JavaScript** - Scripting language to make web pages dynamic

7.2 Problems and their solutions

7.2.1 Thymeleaf security dialect

Index terms— Thymeleaf