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

Lennart Cockx
Guangming Luo

June 10, 2017

Contents

Abstract

This thesis explores enterprise Java development and the usage of micro-services to improve scalability and streamline application development. This is done by working out a case study prototype: An online cash register application as an alternative to existing cash registers at restaurants or events. This system should handle orders and payment from the mobile devices of customers and employees.

After a brief description of the project objectives, an overview of the current Java enterprise technology landscape is presented. From this listing, suitable technologies are selected which will form the base of this project. The case study will explain the design choices that must be made to create a functional application. Then, a chronological progression shows the requirements for micro-services development and details the steps taken to move from a monolithic application to a micro-service architecture. Finally, the completed prototype is presented and the benefits and drawbacks of the micro-services approach is discussed.

Index terms— JavaEE, Spring framework, Cloud, Micro-services

Technical Summary In this thesis we develop an online cash register application for restaurants and events using a case study proposed by Faros, a company part of the Cronos group. This application is a digital version of cash registers found in stores and restaurants. Customers can use their own mobile devices to get an overview of the products available at a location and place orders. Employees use computers or their own devices to receive and process these orders. This case study follows the standard procedures used in application development. We do market research, perform a use case analysis, determine the necessary and optional features, create navigational models and UML diagrams where needed. The application is developed with distribution in mind and functions as a web application running on multiple separate servers.

The core aspect of this project is exploring Java web technologies and analysing what features they offer. We research frameworks and libraries suitable for prototyping the case study, and dene constraints and other requirements with our coaches at Faros. This will include many technologies related to enterprise java development. A framework such as Spring or JSF is used to serve HTML pages from an application server to clients. For storing data, we look at both relational and document-based databases such as MySQL, OracleDB and MongoDB. Where beneficial, Object mapping tools are used. Security systems for authorisation and authentication are also discussed. Finally, we also look at front-end frameworks and libraries such as AngularJS and React and consider the advantages they offer.

The primary focus however, goes to the back-end architecture of the application. We look specifically at applications hosted on cloud platform like Amazon AWS, Microsoft Azure, IBM cloud and more; determining the appropriate way to develop applications which utilize the advantages of distributed platforms. For this we will look at the micro-services architecture and the advantages it offers for scaling a system to serve a large user base. Multinational companies such as Netflix, Amazon and eBay are already running on micro-services and it is becoming more popular in the web development world. By developing the online cash register application with this architecture, we analyse the advantages and disadvantages of using micro-services. First a single monolithic

application is created using development techniques familiar to us. This application will be a monolithic program that includes an application server, all the required mappings and classes, plus a connection to a single relational database. After successfully building a prototype application that implements the features specified in the case study, we restructure the project to a micro-service architecture. Business code for the services are extracted from existing modules and the application server will become a separate module. Each server gets its own database, using the technology most suited to data stored by the corresponding service. Additional services are created to support this micro-service architecture such as a Discovery service and a gateway. The result is an effective implementation of the API gateway design pattern.

At the end of this project we should have an excellent understanding of the technologies necessary to create an effective and scalable web application. Following the development process above, we will get insight into the advantages of a micro-services architecture, but also the problems that come paired with the development of such a system. Especially when migrating to a micro-services architecture from an existing application. By developing the online cash register case study, we have increased our general knowledge of web design and Java development. This should improve our ability to select the most appropriate tools in the development of future applications.

1 Master Thesis Objectives

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 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 regard 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: design of an online cash register that can be used in restaurants or at events. Existing cash register systems can be expensive. For smaller establishments or temporary events, this initial investment could be too high. 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 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. During the design and development of this project we want to note down 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 and targets, we define the general field we will work in and the features we should develop. See **section 1.3.1** for the constraints defined together with Faros.

After we have a good overview of the key aspects for this project, we can start searching for the appropriate technologies. The choices we make here are based on the aspects we think are needed for developing a web application meeting the aforementioned requirements. After doing this we should have a list of tools, systems, frameworks and libraries that could potentially be used during this project. This is done in **section 2.1** about the Java EE landscape. After defining the subject of our case study, this list can be shortened 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. In this case we will make a selection of items to keep the scope of the project realistic. Usually the most widely used frameworks will be selected. If a deviating set is chosen this will be elaborated on in the corresponding chapter.

The next step is to define the case study in more detail. The standard procedure for application development will be followed. Relevant documentation will be constructed including a use case analysis, feature description (nice-to-haves versus must-haves), navigational models, UML diagrams and market research. After this we can start working on getting familiar with the chosen enterprise Java frameworks. See **section 3** for the full case study. At this stage, we will also try out the different technologies we selected to be able to better compare their advantages and disadvantages. See **section 2.3** for a comparison of technologies.

As soon as we are sufficiently proficient with our chosen tools, we will start implementing the case study and use the knowledge we obtained to build a functional prototype for a realistic use case scenario. During the project, we might conclude 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. Apart from the implementation, there will be a separate chapter to showcase the resulting application. See **sections 4 and 5.1** for the implementation and results, respectively.

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 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 an 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 *service discovery*, *routing*, *load balancing* and *security*.

2 Research and Methodology

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 which provide a way to couple web resources to Java code. Persistence providers, relational or document-based 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. And any technologies related to the microservices architecture. Everything related to the case study is detailed separately in **section 3**

2.1 Java Enterprise Landscape

In this section, an overview of web technologies is presented and an indication is given as to how common they are in general use. 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. Skip to 2.3 if you already have a good knowledge about the available java technologies and their popularity.

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 user group. For more general statistics please refer to Stackoverflow's developer survey. (Stackoverflow, 2017)

Currently most 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 considered 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 an "Opinionated Defaults Configuration" approach. (Dinesh, 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 most popular frameworks. This should give us a good idea of the advantages that web frameworks offer, and how they compare. **Spring**

Boot looks promising as it provides autoconfiguration for many properties which will be advantageous for people new to the Java EE platform, such as ourselves. It is also worth mentioning that 17 percent replied that they do not use any framework. It can be concluded 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, as this build tool will be shared by all teammates in a project. In our project, the choice will be between **Maven or Gradle**, as they are better for dependency management in comparison with Ant+Ivy. Unless Ant's fine-grained control of the build process is needed, using Maven or Gradle will be much more convenient.

To deploy the application, an application server is needed. For production, *Tomcat* (42%) is the primary choice 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** (Included in the framework of choice) will get our project up and running faster.

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 consider as this will be the focus of this project. It seems that microservices have become quite popular in recent year for a variety of reasons, like separation of concerns, natural modular structure and excellence in continuous delivery platforms. (Gupta, 2017b) Although it differs from the usual approach we learned during our education, using this architecture in our application will 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 blocks 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 must be considered when developing our application. The choice for security will largely depend on which framework we work with.

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 enhance development even further. They offer tools and 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.

Finally, what types of databases are used to store information? The most used databases 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 developers 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 which features would be beneficial for this specific project.

2.2 Case study summary

In this project, a case study is used to define the functional requirements for development. During the research and implementation, we will keep these requirements in mind. The idea is to showcase a multi-services architecture using a relevant application prototype.

Here is a small summary of what this case study entails: We will develop a web application to be used by restaurant owners or event organizers. They can apply for an account, giving them access to a public webpage where they can manage their establishment and employees. They can also create and modify menus for their restaurants or events. Customers coming to these places can use their mobile devices to access the corresponding menus and order food and drinks directly from their device. Employees assigned by the manager can see and process these orders from their own computers or mobile devices.

We want to create an ecosystem for this purpose without the need for specialized devices, with the necessary infrastructure running on servers in the cloud. For a complete overview of the case study, see **section 3**.

2.3 Comparison of technologies

2.3.1 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 Java frameworks. There are several other advantages to using IDEA, but these were the most determining factors causing us to switch environment.

For the build tools, Maven is certainly 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 to explore the options

that were available. Gradle is currently the default build tool in Android development with android studio, and it is also utilized in Grails and Netflix OSS development. When comparing them directly, Gradle seems to offer a much richer feature set. (Gradle, 2017) When using this build tool in the project, we concluded 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 somewhat shorter and more readable build file in comparison to the XML notation that Maven uses.

We did stumble upon some issues while using Gradle. Because it also uses the Maven repositories for its dependencies, adding your own dependencies to the local repository is slightly more complex. To add a dependency, 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. See section 7.2 for more information. Based on our own preferences and experiences during the project, we decided to go for **Gradle** during development.

2.3.2 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. Based on our research in section 2.1, we have the following frameworks as a possible option: *JavaServer Faces, Spring/Spring Boot, Vaadin and Struts*. 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 out 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 packages were then combined into an Enterprise Application archive and deployed to a glassfish-based 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 file as well as in the application server itself. When changing database or server this configuration must be redone. In a prototyping stage, this can be a burden since we must reconfigure the project each time we want to make significant changes. Also, when we implement a micro-services architecture, it will be necessary to repeat this configuration for every service.

The other major problem is technology selection. Most technologies used for this project will be based on the specific requirements of the case study. For many other parts of the project we can't know what technologies will be most suitable in advance; like what application server do we use? what object mapping tool? and what security framework? And many other systems that we didn't know 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 implement these features for us in a production ready package. This way we can already start exploring and working with certain features without having to develop it from scratch. Most frameworks still allow changing the preconfigured defaults to different system where necessary. Based on these prerequisites, the focus will

go towards **Spring Boot**, **Vaadin** and **Struts2**.

From these 3 frameworks, Vaadin is the odd one out. Vaadin's primary purpose is creating Web interfaces for business applications. While it can be used as a general framework, the focus goes to fast development of dashboard-style applications with pre-made components. This is not in line with the custom, more customer-centric interface that we want to develop in this project. Additionally, Vaadin is considered a poor choice for building a scalable multi-services system. (Hauer, 2015) Vaadin is a server-side presentation framework, which means that the state and logic for the UI is stored on the server-side and a request will have to be done for each user interaction. This is well suited for creating applications which are used within a company, but less so for systems that would potentially serve millions of customers.

This leaves us with Struts2 and Spring Boot as potential candidates. Struts2 is considered to be simpler than Spring MVC. Due to its full JavaEE implementation, Spring is more flexible and allows other frameworks to be integrated with it. 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. (Mehta, 2015)

Luckily, the complexity of the Spring framework can be partially circumvented by using **Spring Boot**. This framework 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, as we can quickly get started and 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 tools that offer similar autoconfiguration for implementing microservices, most notably Dropwizard and JHipster (Which uses Spring Boot). The reason we decided to go for Spring Boot is the abundance of good online resources for learning the framework and how a micro-services architecture can be realized with it. 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 4.

2.3.3 Persistence providers

Data storage is utilized in practically all web applications. We will need at least a way to store information for authentication purposes. On top of this, 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. (MongoDB, 2017) When handling diverse data types or modification of existing schemas is required, a NoSQL database can be beneficial. E.g. If there are certain database entries that you want to update with an additional field, this field can be added to all new entries to the database while keeping the existing records intact. When using SQL, the entire database model would have to be updated and the existing records will need to be changed to fit this new structure.

Document-based databases allow for horizontal scaling, which means breaking

up the dataset into smaller parts and storing them on distributed servers. This does not mean that SQL does not allow for scaling however. In many cases a single large server is all that is needed. Load balancing can be done by replicating the server into multiple instances. Horizontal scaling via sharding is also possible with relational databases, but the process quickly becomes more complicated than with a NoSQL database and requires different strategies based on the type of data stored. (Narumoto, 2017)

Since we are going for a micro-services approach, it is possible to use a different persistence provider depending on the service. For example: 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 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 here is also possible, but requires a custom implementation of the *UserDetailsService* from Spring Security.

On the other side, we must store order information for customers. Each order entry can hold an unstructured, arbitrary number 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 in development this also becomes much easier. Our final project can have a **combination of SQL and NoSQL** providers, using the most suitable for each service separately.

It will take time to adjust if we want to use a NoSQL database as the principles behind designing NoSQL storage is completely different that relational data storage. During this project, we will make use of **MySQL** for storing relational data and **MongoDB** for storing non-relational data.

2.3.4 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 queries, which makes development faster and prevents having to write SQL manually. Since it is likely that we will make use of NoSQL database in certain services, ORM tools become less useful, since the traditional relational structure is not used here.

The usage of ORM tools will hinge on the data structure we decide on. If we use a relational database we will make use of **Hibernate**, the default ORM implementation in Spring Data JPA. If we use NoSQL in a micro-services system, using Object Mapping tools is less beneficial. There are few tools in existence for NoSQL and they often don't provide any significant benefits unless you are migrating from existing relational databases. (Singh, 2013)

2.3.5 Security systems

Proper security is expected in a web application that stores information or provides functionality which should not be accessible to anonymous users. Even when logged in, there are certain resources that should only be accessible to accounts with the correct privileges. This is referred to as authentication and authorization. Java Enterprise applications consist of components that can contain both protected and unprotected resources. Authorization provides controlled access to protected resources and is done after authentication. To authenticate, an identification process is required, which

verifies the identity of a user, device, or other entity in a computer system.

In our case, management of the restaurants and modifying the menus should only be permitted to authenticated users with a specific role of *Manager*. A role is an abstract name for the permission to access a specific set of resources in an application. Account information, restaurant statistics and the order overview are all sensitive information and should only be accessible by authorized users.

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 are to be implemented by the developers themselves. (Oracle, 2013) Java EEs security model specifies that the container must provide or support certain security features. You can implement these features in two ways: declarative and programmatic. Both approaches are based on the Java Authentication and Authorization Service (JAAS) API. However, certain aspects such as encryption of passwords and https authentication might need to be designed manually or external libraries must be used. Also, The Java EE Security documentation is not as extensive as some third-party frameworks, which can make it more difficult for beginners.

In comparison, the Apache Shiro framework is considered less complex and supports container independency. Additionally, the documentation for integrating it into an application is more extensive. Its core features are authentication, authorization, cryptography and session management and other web-specific features. By enabling Shiro for your application through a web.xml configuration file, you can secure any URL and can also specify filter chains for each of them. (Andrejtschik, 2016)

Like the frameworks above, Spring Securitys functionality is centred around authentication and authorization. Spring offers a complete system for securing web applications, available for use in any supporting framework. Spring Security is a system that can be added to an existing project and enables secure login and protection for specific pages and resources with minimal configuration. The documentation is also very comprehensive.

Because our project is Spring-based, we choose to make use of the included **Spring Security** system. The biggest hurdle we will likely face is implementing these security features when we switch to a multi-services architecture. We would have to implement a separate service that handles security for all other services. In the case of Spring, this could be achieved by using an OAuth2 server. (Pivotal, 2017)

2.3.6 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 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 try out the popular front-end modules **AngularJS** and **React**.

2.3.7 Application architecture and micro-services

During our research, there were several articles that inspired us to consider a **micro-services architecture** for developing this distributed web application. (Richardson, 2017) (Nommensen, 2015) (Delmas, 2015) The internet has become an integral part in the daily lives of many people. When creating a web application as a company, we must consider that this application could potentially be used by several thousand if not millions of people. At that scale, a lot of problems can occur due to the increased load on the servers used to run this application. Managing such an application can quickly become an expensive and complicated task.

The micro-services architecture is a potential solution to these issues. Applications are split up into more manageable chunks that increases the efficiency of development. It becomes much easier to refactor a single service, rather than refactoring part of a monolithic application and all modules related to it. Using micro-services improves scalability, as each service can be deployed to different physical servers, dividing the load. Each service can use the technology stack appropriate for the functionality required from that specific service. In a worst-case scenario one of the services might stop working. The rest of the application and services will remain functional, while with a monolithic application the entire system might stop functioning. (Badola, 2015)

Of course, this does not mean that micro services are the solution for all web development. Developing a micro-service system can be more complex than a similar monolithic application. More care should be given to exception handling in a microservice architecture, as you cannot know for certain if a service will respond. Usually these responses are also slower than communication within one integrated application. There is also more overhead when using micro-services and code-duplication is harder to avoid if you need certain features in multiple services. Implementations for security completely change when working with this kind of distributed model.

Even with the increased complexity and other disadvantages, micro-services is something worth considering. Large corporations like *Netflix*, *Amazon*, *Ebay* and *Uber* have already moved over to this architecture because of the advantages that it offers or simply because the scale of their userbase is so large that they cannot rely on more traditional development methods. It makes sense to explore the possibilities of micro-services when developing a distributed application.

There are several design patterns that can be applied when developing a micro-service system. (Gupta, 2017a) One of the most common designs involves using an API gateway. This is a central access point for clients that want to consume a service. The modular back-end structure is not visible for clients, it is similar to accessing a regular web application. To implement this pattern, we must create a REST API for each service. API calls to the gateway will be routed to the API of the corresponding services. The structure of this system would look like figure 1. We will try to achieve this architecture during the development of our case study. Instead of immediately starting out with a **micro-services application**, we will start from a **monolithic application** and separate the services afterwards. This will

give us an idea of the problems that we can expect during the migration of such an application.

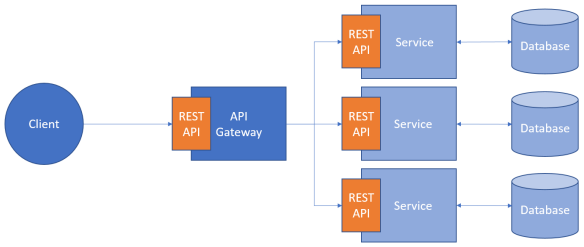

Figure 1: *Api gateway design pattern: The Api Gateway allows access to the various services running in the back-end via a central gateway.*

3 Case study: Online cash register

3.1 Introduction

The goal of this case study is to create a system 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 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 centralized system that manages all the required information. See figure 2 for a use case diagram.

3.2 Existing systems

Before creating our own online cash register application, we searched for similar systems that already exist. One of such systems is Gloriafood (Gloriafood, 2017). This is an online system that allows any restaurant owner to sign up and start receiving orders online. An extensive layout is available where you can add a menu and payment options.

Similarly, there is ChowNow (Chownow, 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 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.

3.3 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.

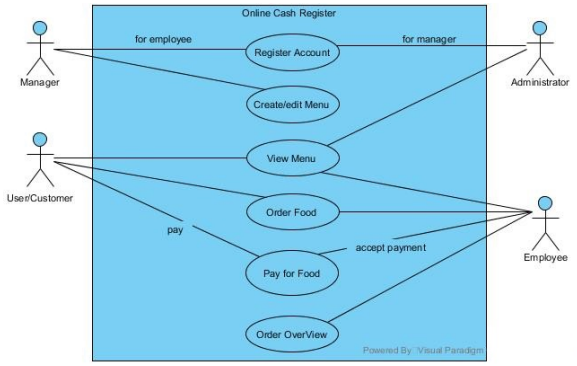

Figure 2: *Use Case Diagram: This diagram shows the actions each actor can undertake when interacting with the application. Customers can do basic actions like viewing the menu, ordering food and paying for it. Employees can see the order overview and modify the menu in addition to the basic actions. Managers can register employees and the Administrator can create accounts for managers and assign them to a restaurant.*

3.4 Must-haves and nice-to haves

As soon as we had the use cases defined, we can specify what features are 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 should 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 Statistics*)
- 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*)

3.5 Design Choices

3.5.1 Client

The customer can use this application to get an overview of the available menu items and prices. See figure 4. If the customers have decided, they can order the desired dishes straight from the application. Finally, they 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. 3.

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 3: *Example Order Overview: The order overview has the names and number of items ordered. It also specifies the table which made the order. Finally, it indicates the status of the order, making the distinction between orders just placed, and orders already processed in the kitchen*

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. 5.

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

Figure 4: *Customer Menu Layout: For customers, the layout should be intuitive and complete. Menu items are listed with their full name and price. A checkbox is used to select food for ordering. A field for filling in the amount would also be useful. This layout could also have an extended description of the item as well as pictures for showcasing.*

Veg Pzz	Haw Pzz	Blg Pzz	
Calz Pzz			

Figure 5: *Waiter Menu Layout: The menu layout for employees should be different because visuals and intuitiveness are not the most important factors here. Instead, this layout should have as much items as possible on one screen without affecting usability. The items can use shorthand names and should always load in the same position to allow for easy memorisation.*

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, e.g. 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.

4 Application Implementation

4.1 Development Process

In this section, 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.

4.1.1 Monolithic Application & Services

As mentioned in section 2.3.1, 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 setup a locally hosted Payara server which we used to deploy our enterprise application archive. A simple SQL data structure was running on an external MySQL server, which can be seen on figure 6

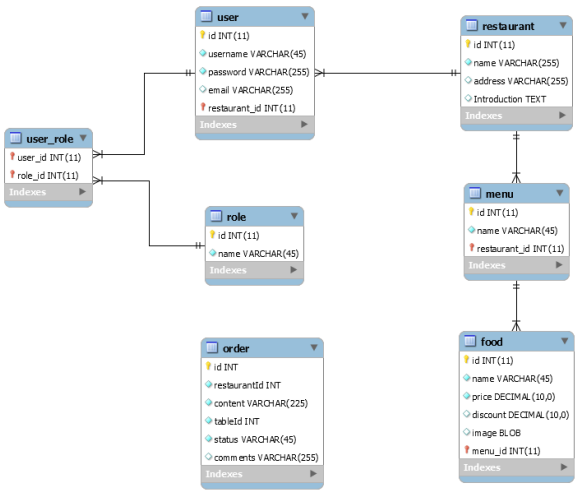

Figure 6: *Original MySQL data structure: This image shows the relational model we used while working with a single SQL database*

After our initial research, we switched over to **Spring Boot** and recreated the original project. Spring boot seemed like an appropriate choice based on the research done in section 2.3.2 and should make migrating to multi-services easier. At this stage, we kept the previous SQL 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 should just 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 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. 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. Figure 7 shows a simplified representation of the current application model.

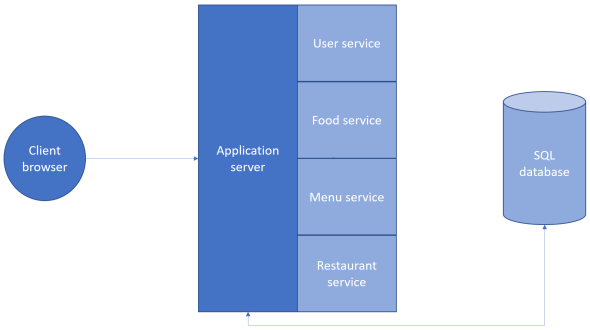

Figure 7: *Initial project structure: The application is accessed by a browser client. All the services are part of one big Java application and use a relational database to store data.*

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 web application.
- **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 creation and modification of user accounts.
- **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 auto-configuration 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 singleton beans. The reason the *@Service* annotation is used instead is to indicate what type of component we are creating following the guidelines from Domain-driven design. (Latest-tutorial, 2015) (Hoeller, 2017) We purposefully make a distinction between presentation, service and data access modules to obtain a clear project structure.

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 6. 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 must 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 consider 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 number 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.

4.1.2 REST & Micro-services

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 8.

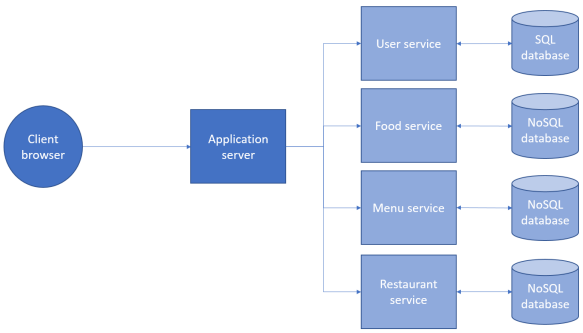

Figure 8: *Project structure after splitting into separate modules: Instead of the single monolithic application, the project now has a separate module for each service. Each module consists of an REST access point, a service that handles the REST calls and an individual database. This database can use 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 micro-services 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 service 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 must consider 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.

4.1.3 API Gateway & Discovery

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 must be known. For the application server accessed by browser clients these addresses can be hard-coded, although this is not recommended. (See the next paragraph about discovery.) For other clients, this is more difficult as 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. E.g. 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 1. The services are now accessible from the android application without the need for specific addresses or ports.

While we have a functional Api gateway system now, we are not yet making use of all the advantages the microservice architecture offers. For example, if we want to run multiple instances of the same service for redundancy or load balancing, how do we determine the addresses of these services? We cannot simply add more routes to the Zuul gateway, this is not scalable. Instead we created yet another service that works with the Eureka system, also part of the Netflix open source software. Eureka provides a discovery service on which other services can register themselves. The Eureka server holds a list of each services registered to it, as well as the number of instances available and on what addresses they are located. To have a service register itself to this discovery server they must be instantiated as a Eureka client. On start-up, they will connect to the server and register themselves with the corresponding information. The design we use for service discovery is shown on figure 9.

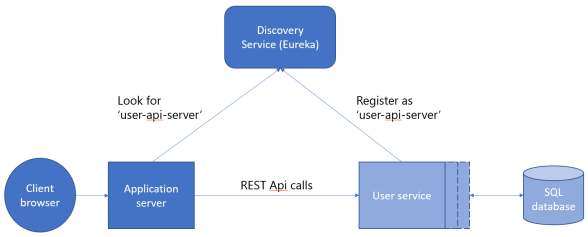

Figure 9: *Discovery design: Each time a service is started, it will register itself to the Eureka server. This can be done multiple times for the same service. The eureka server can then be consulted to get the addresses of the required services. After getting the information, the service can be addressed directly with REST Api calls. Services will need to send a heartbeat to the discovery server regularly, or they will be unregistered.*

This opens a lot of possibilities, we can know host each service multiple times and if one instance crashes or has connection issues the application will still function. Eureka also does basic round-robin load balancing, which means that incoming api calls from clients will be spread over the pool of available services, which helps with performance.

With the inclusion of a discovery service and Api gateway we now have a fully functional system that makes use of the advantages a microservices architecture brings to web development. The structure of this system is shown in figure 10

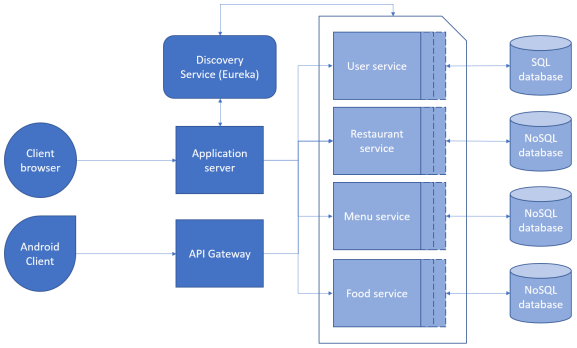

Figure 10: *Final project structure: The application server will now consult the Eureka server to find running instances of the services it needs. Each service can have multiple instances registered on the eureka service.*

5 Conclusion

5.1 Results

In this section, an overview of the final application prototype will be given, along with possible improvements that can be done in the future.

5.1.1 Final Prototype

The final version of our application is divided into several modules which can be seen on figure 11
The application can be accessed via a browser client on any device or employees can use our native application on android devices. When using a browser, the user will connect with an application server that serves html pages. Customers will be greeted by a menu page of the restaurant they browsed to. From this page, they can select items and place an order. These features do not require the user to be logged in. Employees and managers can log in to access more functionality. Once logged in, employees can view an overview of the currently placed orders and update their status. They can also modify the menu, creating new menu items or removing them. They can do this only for the restaurant they are assigned to. Managers will have the rights to create and manage employee accounts.

The HTML pages send by our application server are based on the data returned by the services for each aspect of the system. There is a separate service for storing food items, menu categories, orders, restaurant listing and information and user accounts. The application server finds an instance of these services by consulting the Eureka server, which registers all running service instances. If one of the services is not available, a status message will be shown to users to inform which service is unavailable. Any functionality of the web application that does not require this service will continue functioning.

Employees also have the option to use a native android application. This application provides a specialized interface to make taking orders faster. Instead of showing a scrollable list of food items, this interface displays the items in a grid, using a shorthand notation for the names of the drinks and meals. This application does

not need a running application server in the back-end, but instead relies on an API gateway to access the services it needs. Managers can also use this API to develop their own specialized applications, if so desired.

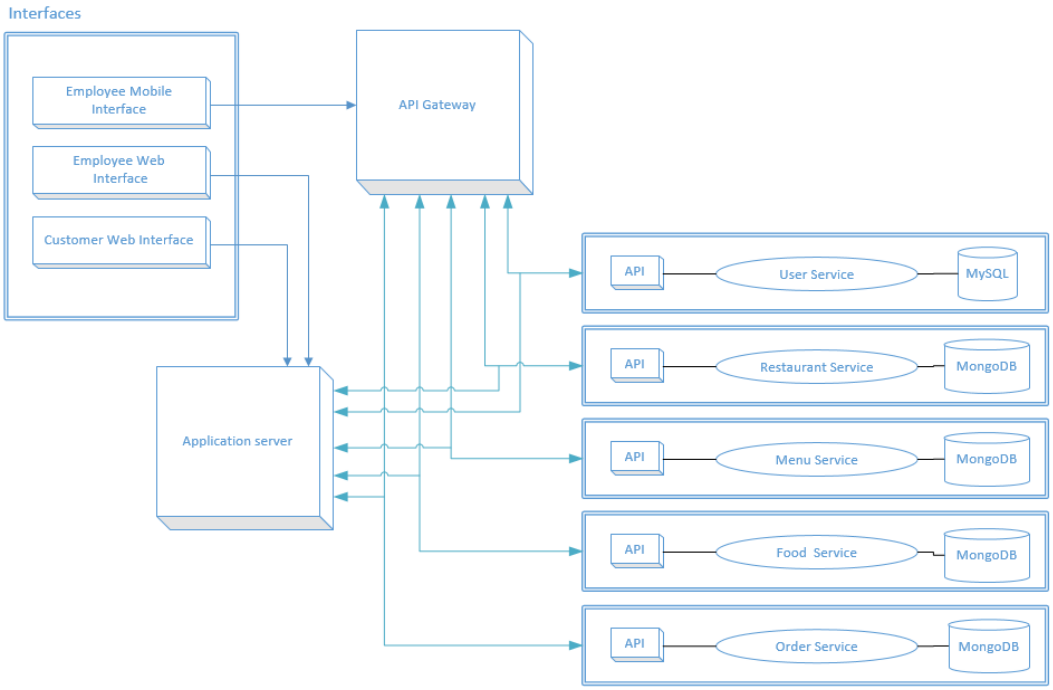

Figure 11: Application diagram

5.1.2 Possible Improvements

Improving Spring Security for micro-services

Configuring security in Micro-services architecture can be a very complex story. We do found some tutorials online, but they try to simply the process by using InMemoryAuthentication which is not what we wanted in this case study. We should use our OcrUser service application to do the JDBC authentication/ authorization and return a custom UserDetails as a token. This token can be further used for API calls to secure our API services. Currently, the JDBC authentication/authorization is implemented in OcrClient web application and the APIs from all the services are public, which means that the web pages are secure but all the APIs can be accessed by anyone.

Redesign existing models to make use of MongoDB's advantages

For current version, OcrRestaurant, OcrMenu, OcrFood and OcrOrder use MongoDB, and they all use corresponding entities (Restaurant, Menu, Food, Order) for mapping. When we add/remove/modify properties from those entities, there is no conflict in the database/document. A smarter way can be using JSONObject for mapping, however which needs refactoring code for the whole system. A big advantage is that you can choose whatever properties we want to store while creating the JSONObject. Specifically, for OcrMenu, currently the menus are randomly distributed according to the time when they are created. Menus from one restaurant can be found by looping though the whole document with findMenusByRestaurantId. A better way is to create an JSONObject inside

OcrMenu database immediately when a new restaurant is instantiated. All menus created for that restaurant can be put into that JSONObject with the right restaurant id. In this case, all menus from one restaurant are put in a concentrated big entity which can be easily exported for future usage.

Creating a multi-platform application with Apache Cordova

We started from researching and prototyping with PhoneGap, and we found that we can only make application for Android and Windows users due to the limitation of OS of our laptops. Hence, we decided to make an android application on Android Studio. In the future, if macOS can be used, making a native application with PhoneGap can be a better choice. In this case, you may build applications for mobile devices using CSS3, HTML5, and JavaScript instead of relying on platform-specific APIs like those in Android, iOS, or Windows Phone.

Implement missing nice-to-have features

- implement a real payment system with an existing payment API, so that actual payment and splitting up bills are possible.
- redesign the application so that hosting on a local network is possible.

5.2 Closing Statements

We succeeded to develop a working application in enterprise Java with Spring Boot and a micro-service architecture. We implemented all the proposed essential features (Must-haves) and most of optional features (Nice-to-have) mentioned from the case study in section 3.3. We now have an excellent understanding of all the technologies involved to make this Online Cash Register application. (listed in appendix 7) We gained a very clear understanding regarding the micro-services architecture and what advantages or disadvantages it has over more traditional web development. Additionally, we know how to implement a Eureka discovery server and Zuul gateway within microservices architecture. We also have more knowledge about Spring Security and how to implement it in different situations. Despite some minor drawbacks, the micro-service architecture is a good choice in this context where developers are building a server-side enterprise application. It should support a variety of different clients including desktop browsers, mobile browsers and native mobile applications. We believe with all the advantages, a micro-services system and the Spring Boot framework will be more widely used in web development and Enterprise Java in the future.

6 References

References

Andrejtschik, J. (2016). Comparison: Java ee security vs. apache shiro vs. spring security. http://blog.novatec-gmbh.de/java-ee-security-framework-comparison/. (Accessed on 26/05/2017).

Badola, V. (2015). Microservices architecture: advantages and drawbacks. http://cloudacademy.com/blog/microservices-architecture-challenge-advantage-drawback/. (Accessed on 26/05/2017).

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

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

Dinesh (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).

Faros (2016). Faros master thesis proposal.

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

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

Gupta, A. (2017a). Microservices design patterns. http://blog.arungupta.me/microservice-design-patterns/. (Accessed on 26/05/2017).

Gupta, K. (2017b). 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).

Hauer, P. (2015). Evaluating vaadin: Strengths and weaknesses. https://blog.philipphauer.de/evaluating-vaadin-strengths-weaknesses/#weaknesses. (Accessed on 26/05/2017).

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).

Latest-tutorial (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).

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) *note: this survey is specifically targeted at Java web developers.*

Mehta, M. (2015). Struts 2 vs springmvc: Know the difference and choose the best one based on your requirements. https://dzone.com/articles/struts-2-vs-springmvc-know-the-difference-choose-t. (Accessed on 26/05/2017) *note: several of the search results found when comparing these frameworks state the exact same points. We're sceptical of this source but our personal experience seems to confirm the aspects we discuss in the thesis .*

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

Narumoto, M. (2017). Sharding. https://docs.microsoft.com/en-us/azure/architecture/patterns/sharding. (Accessed on 26/05/2017).

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

Nommensen, P. (2015). Building microservices: Using an api gateway. https://dzone.com/articles/building-microservices-using. (Accessed on 26/05/2017).

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

Pivotal (2017). Spring cloud security. http://cloud.spring.io/spring-cloud-security/. (Accessed on 26/05/2017).

Richardson, C. (2017). Pattern: Api gateway / backend for front-end. http://microservices.io/patterns/apigateway.html. (Accessed on 26/05/2017).

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).

Singh, A. (2013). Sqlifying nosql are orm tools relevant to nosql? https://dzone.com/articles/sqlifying-nosql-%E2%80%93-are-orm. (Accessed on 26/05/2017).

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

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**.

- **Java + Java EE**
- **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** - Persistence APIs for database access
    - **Spring Data JPA** - Persistence API for database access and Object mapping
    - **Spring Data MongoDB** - Persistence API for NoSQL database access
  - **Spring Devtools** - Spring-specific tools useful during development. E.g. Hotswapping, auto-disable template caching and automatic restarts
  - **Spring Cloud** - Provides tools and patterns that assist in building distributed applications
    - **Netflix Zuul** - Reverse proxy implementation for API gateway pattern.
    - **Netflix Eureka** - Discovery server and clients for managing and load balancing services
- *Web Development*
  - **HTML** - Markup language used for web pages
  - **CSS** - Markup language for web pages styling
  - **JavaScript** - Scripting language to make web pages dynamic
  - **jQuery** - JavaScript library to simplify and extend JS development
  - **Material Design Lite** - CSS and JavaScript library for designing web pages using material design
  - **JSON** - lightweight data-exchange format
  - *React* - A component-based JavaScript library for building UIs
  - *AngularJS* - Structural framework for building dynamic web applications
- *Mobile Development*
  - **Android SDK** - Software development kit for android apps
  - *Apache Cordova* - Multi-platform Mobile application development framework
- *Development Tools*

- **Intellij IDEA** - Integrated development environment for the Java virtual machine.
- **Gradle** - Build automation system
- *Maven* - Build automation system
- **MongoDB CLI** - Command line interface for managing MongoDB
- **MySQL workbench** - Tool for managing MySQL database
- **Android Studio** - IDE for native android development
- **Git** - Version control system
- **Uwamp, Xampp** - Servers for hosting webpages and databases locally
- **Powershell** - task automation and configuration management framework for windows
- *Netbeans* - Development environment for Java Applications
- *Spring Tool Suite* - Specialized IDE for development with the Spring framework, based on Eclipse

7.2 Problems and their solutions

Problems with database model foreign keys

- When a foreign key is also included as part of primary key, you cannot remove this foreign key or drop the primary key.
- Remove the schema and regenerate all tables; if not necessary, do not include foreign keys as part of primary key.
- You cannot remove a table when one of the columns is used as foreign keys in other tables. Similarly, you cannot truncate a table when foreign keys in other tables have been instantiated.
- When instantiating an entity with a foreign key, errors usually appear when the foreign key doesn't have a valid value or empty.

Thymeleaf-related issues

- You cannot give two th:each properties in one div, which means you cannot loop through two lists at the same time.
- You can use Thymeleaf to check if the user is authenticated by adding *sec:authorize = "isAuthenticated()"* to a div, and the div will not show up if its false. Same for *sec:authorize = "hasAuthority('ADMIN')"*, the div will only show up when the user has authority of admin. E.g. *⟨a sec:authorize = "isAuthenticated()" th:href = "@/restaurant/name/" class="mdl-navigation link" href=""⟩Restaurant⟨/a⟩* (line 50 of navbar.html)
- If you want use model attributes from controller in JavaScript, th:inline = "javascript" has to be added to the script tag and use special format to use the attributes: *var url = /*[[$url]]*/'url';* (see line 33-41 of account.html)

HttpRequest POST problems

- Everything get post to the controller via HTTP is String, its up to the server to decide which type you want to decode. If you specify @Valid int tableId, the controller will try to decode the value to int and will introduce an error if it fails.
- Hidden input can be used to submit action where there is no input field. You can also put a hidden input field to normal form.
- RedirectAttributes can be used to pass data from post function after redirecting. This is specifically for Spring Framework. After initializing, you can use it like this: *redir.addFlashAttribute("message"," This username is already registered!");* and in the redirected GET method, you can get the information by specifying *@ModelAttribute("message")* final String message.
  - Those flash attributes are passed via the session
  - They will be destroyed immediately after being used
  - They are not visible in URL
  - You are not restricted to String, but may pass arbitrary objects.

Appending URLs
If you want to go to another base on the current page, you need add an "/"after. e.g. current url:"www.mybase.com/home/" *href="edit"* will lead you to "www.mybase.com /home/edit". If you current url is "www.mybase.com/home" *href="edit"* will lead you to www.mybase.com/edit.

Spring Security role checking on the server side
You may check if the logged in user has a role of admin in controller final: *Authentication auth = SecurityContextHolder .getContext().getAuthentication(); SimpleGrantedAuthority AUTHORITY_ADMIN = new SimpleGrantedAuthority("ADMIN"); org.springframework. security.core.userdetails.User user = auth.getPrincipal(); if(user.getAuthorities().contains (AUTHORITY_ADMIN)){ model.addAttribute("isAdmin",true); }*

Database issues
"table","order" and other similar keywords cannot be used as table name or column name, SQL will fail.

Creating a custom Maven/Gradle dependency
Initially, we want to make a jar file and include it in our project, but we failed to use/refer to it. Later, we choose to create a local maven dependency, and compile it in the gradle dependencies. How to make a maven dependency with your models? Create a maven project with groupId = com.xxx and Artificial Id, under java folder, create pakage: com.xxx.models, put all your java files under that folder. Then do maven build.

Combining microservices and Spring Security
According to the tutorial from spring.io/blog, the author disable component scan in the main application and create all beans inside main. However, this may cause an issue, because it will also disable the security configuration which is usually a separate config class. In this case, the whole project will use default spring security configuration (all pages need login).