An Online Auction System for Selling Fiber Products

Master Thesis
to obtain the academic degree of

Master of Science
in the Master's Program

INTERNATIONALER UNIVERSITÄTSLEHRGANG
“INFORMATICS: ENGINEERING & MANAGEMENT”
Abstract

The thesis presents the design and development of the Lenzing Global Auction Portal project. The Lenzing Global Auction Portal is a software system for running multiple simultaneous online auctions. The system implements the Dutch auction model, but provides the means to integrate different auction models as well.

The system is based on the Client/Server architecture model. The server-side follows the microservice based architecture model combined with the multi-layer architecture model. The client-side consists of two mobile-first web applications following the single-page application principle.
Acknowledgment

This thesis and the corresponding software project was realized in the frame of the Upper Austrian ISI Master Program\(^1\) in cooperation with an Upper Austrian company, Lenzing A.G.\(^2\)

Besides thanking the supervisor of the thesis, the author would like to thank the coordinators of the ISI program and the representatives of Lenzing for their help and contribution.

\(^1\) http://www.isi-hagenberg.at
\(^2\) https://www.lenzing.com
# Table of Contents

Table of Figures ........................................................................................................... 5
Introduction .................................................................................................................... 6

1. Auction Models ........................................................................................................ 9
   1.1. English Auction ................................................................................................. 10
       1.1.1. Candle Auction ............................................................................................ 11
       1.1.2. Japanese Auction ........................................................................................ 11
       1.1.3. Buyout Auction .......................................................................................... 11
   1.2. Dutch Auction .................................................................................................. 11
   1.3. First-price Sealed-bid Auction .......................................................................... 12
   1.4. Vickrey Auction ............................................................................................... 12

2. Project Description .................................................................................................. 15
   2.1. Requirement analysis ....................................................................................... 16
       2.1.1. LGAP Server ............................................................................................... 17
       2.1.2. LGAP Customer Applications ................................................................... 18
       2.1.3. LGAP Administrator Application ............................................................... 19
   2.2. System Architecture ......................................................................................... 19

3. Backend .................................................................................................................. 22
   3.1. Microservices .................................................................................................. 23
       3.1.1. Architecture ............................................................................................... 23
       3.1.2. Exception Handling and Logging ............................................................... 26
       3.1.3. Configuration ............................................................................................. 26
       3.1.4. Build and Deployment .............................................................................. 28
   3.2. Common Microservice Module ....................................................................... 29
       3.2.1. Common Data Model ............................................................................... 29
       3.2.2. Common Repository Components ............................................................ 30
       3.2.3. Common Service Components ................................................................ 31
       3.2.4. Common Verticles ..................................................................................... 31
       3.2.5. Common Dependency Injection Module ............................................... 32
       3.2.6. Common Exception Types ......................................................................... 33
       3.2.7. Common Configuration and Resource ..................................................... 34
   3.3. Product Microservice Module ........................................................................... 34
   3.4. Auction Microservice Module ......................................................................... 35
   3.5. Customer Microservice Module ....................................................................... 38
   3.6. Admin Microservice Module .......................................................................... 39
3.7. API Gateway Microservice Module ................................................................. 40
4. Frontend ............................................................................................................. 42
  4.1. Single Page Applications .............................................................................. 44
    4.1.1. Architecture ............................................................................................ 44
    4.1.2. Build and Deployment ............................................................................ 46
  4.2. Customer Single Page Application ............................................................ 46
    4.2.1. Auction Overview Page .......................................................................... 46
    4.2.2. Auction Page .......................................................................................... 47
    4.2.3. Login Page ............................................................................................. 47
  4.3. Administrator Single Page Application ...................................................... 48
    4.3.1. Product Overview Page .......................................................................... 48
    4.3.2. Auction Overview Page .......................................................................... 48
    4.3.3. Auction Creation Page ............................................................................ 49
    4.3.4. Login Page ............................................................................................. 49
5. User Guide .......................................................................................................... 50
  5.1. Customer Application ................................................................................... 50
  5.2. Administrator Application ........................................................................... 53
Conclusion and Further Development .............................................................. 56
References .............................................................................................................. 57
**Table of Figures**

Figure 1. Main auction model categories for the LGAP system ..................................................9
Figure 2. Having the highest valuation in the Vickrey auction model ........................................13
Figure 3. Not having the highest valuation in the Vickrey auction model ....................................13
Figure 4. Use-case diagram for the customer web application .......................................................18
Figure 5. Use-case diagram for the administrator web application ...............................................19
Figure 6. Architecture blueprint of the LGAP system ....................................................................21
Figure 7. Architecture blueprint of a microservice ..........................................................................24
Figure 8. LGAP microservice launcher type hierarchy diagram ......................................................28
Figure 9. LGAP base Verticle type hierarchy diagram .....................................................................31
Figure 10. LGAP base Guice module hierarchy diagram .................................................................32
Figure 11. LGAP base Exception type hierarchy diagram ...............................................................33
Figure 12. Product Microservice, data model diagram .....................................................................34
Figure 13. Auction Microservice, data model diagram .....................................................................36
Figure 14. Customer Microservice, data model diagram .................................................................38
Figure 15. Admin Microservice, data model diagram ......................................................................39
Figure 16. Architecture blueprint of a single-page application .........................................................44
Figure 17. Customer SPA, login screen ............................................................................................50
Figure 18. Customer SPA, auction overview screen .........................................................................50
Figure 19. Customer SPA, auction screen with ended auction .........................................................51
Figure 20. Customer SPA, auction screen with ongoing auction ......................................................51
Figure 21. Customer SPA, auction overview screen (mobile) ............................................................52
Figure 22. Customer SPA, auction screen with ended auction (mobile) ...........................................52
Figure 23. Administrator SPA, login screen ......................................................................................53
Figure 24. Administrator SPA, product overview screen .................................................................53
Figure 25. Administrator SPA, create new auction screen ...............................................................54
Figure 26. Administrator SPA, auction overview screen .................................................................55
Introduction

The problem approached in this thesis is the sale of fiber products via auctions through an online software system. The project is realized in collaboration with Lenzing A.G. (Lenzing), a company based in Upper Austria. Lenzing is the producer and world-wide distributor of high quality fiber products. The current sales approach utilized at the company to do business with their customers is the traditional, contract based bilateral negotiation. The amount of products sold and the final price is negotiated between the customer and the sales department of Lenzing. However, the company faces the problem of being completely oversold, therefore they are looking for another approach to increase profits.

One of the approaches the company wishes to experiment with is the auction based sale of products. This approach provides the benefit of letting the customers drive the price higher while competing against each other, instead of competing against the sales department of Lenzing. The auction based sales approach is utilized effectively in several markets, spanning several industries. With online providers such as eBay, the concept of auctions became increasingly popular in the past years. An important goal for Lenzing is to maintain the good relations they have built with their customers over the year. One of the difficulties Lenzing’s customers have to face due to Lenzing being completely oversold is the long delivery time for newly placed orders. Currently, it takes several months for Lenzing to provide the products following an order placed by the customer. To make the auction based sale desirable for the customers, Lenzing provides the benefit of quick product delivery for the customers participating in the auctions. In order to achieve this, the amount of auctioned products are reserved and delivered to the target region of the auction, even before the auction starts. Via this approach, the winner of the auction can receive the desired products within weeks, instead of having to wait several months. Another benefit Lenzing provides for customers participating in the auctions is discounts. The discounts are assigned per auction basis for selected Lenzing customers. Information regarding the discount remains private between Lenzing and the customer. The customers are still going to compete against each other by the bids they place, however, once a customer won the auction, the discount is going to be applied to the final price determined in the auction.

The idea of the auction based sales approach is not yet fully developed at the company, therefore the requirements for this project are vague. The company is open for ideas and suggestions; however, they have expressed certain requirements, wishes and limitations. The first requirements set by Lenzing is finding an appropriate auction model to experiment with. Another requirement is a software system running in an online environment that makes it possible for Lenzing to provide the auctions for selected customers, while also allows the customers to interact with the auctions.

An important direction in which Lenzing wishes to move forward regarding their customer relation management is a digital platform for their customers. Due to Lenzing’s global presence, their customer base is also global, therefore the online auction system should be accessible on as many platforms as possible. Another desire of Lenzing is to bring their digital presence and interaction as close to their customers as possible. Therefore, targeting mobile platforms is a priority.
The system designed and developed in the frame of this thesis is not part of the Lenzing Digital Customer Portal (LDCP). However, both systems are designed and developed simultaneously in the frame of the Upper Austrian ISI Master Program in collaboration with Lenzing. Certain functionality implemented and provided by the LDCP system must be integrated into the system developed in the frame of this thesis.

Lenzing did not provide any requirements regarding the user interface. However, they have expressed that using the system should be enjoyable for the customer and should not expose the end user to any additional burden, not related to the functionality they wish to use. This is too vague to be treated as a requirement, however, discussions with the company revealed that they value simple user interfaces which are not loaded with distracting or too complex functionality. They have also revealed that the design choices made in modern smart phone applications can serve as guidelines for designing the user interface of the auction system.

The company set certain requirements regarding the software system. The system should be designed with security in mind including data privacy in terms of stored data and storage service location. The system should be scalable to adopt to increased number of customers, and it should be deployable to cloud services.

The auction system designed and developed in this project is meant to be a proof-of-concept. The main role of the system is to aid Lenzing in future internal discussions regarding the topic of online, auction based sales approach and to generate feedback from a selected set of customers. Based on this, flexibility and extensibility of the system on all levels is treated as an important requirement as well, since a software system of this nature is subject to frequent changes. Besides the requirements discussed above, the company set certain limitations as well. The first limitation is set in the context of auctions. Lenzing maintains certain regions or region groups into which their customers are classified. Since one of the main ideas behind the auction based sales approach proposed by Lenzing is speeding up the delivery time, auctions are also region locked. Only customers residing in a given location shall access auctions running in that region. The second limitation is that the system must not implement its own user account management, but consume the related services provided by the LDCP system. The LDCP system manages customer user accounts, performs authentication, and maintains the list of regions a customer is associated with. The third limitation is regarding the data storage. Lenzing systems (including the LDCP system) make use of relational databases for persisting data, therefore a relational database system must be used here as well. The final limitation is regarding the integration with other Lenzing systems. Lenzing’s main system for operation and customer relation management is the SAP system, therefore the system developed in the frame of this thesis must be capable for potential future integration with the SAP system.

Based on the requirements set and various discussion sessions with the company, the thesis can be divided into two main topics:

1. Auction Theory: a survey of available auction models with the goal of finding the best fitting model in terms of the requirements of the company.
2. Enterprise Software Design and Development: design and development of an online, mobile-first software application that is capable of executing several auction processes in parallel, based on the auction model chosen by the company, and the additional requirements introduced by the company.
The first topic covers the most commonly used auction models, providing examples for use cases and recommended bidding strategies. A straightforward way is provided to categorize auction models based on features they share. The categories are used to highlight the main differences between the various models. Several auction models are introduced that are based on common models, but introduce new features or significantly alter the base model.

The second topic covers the design and development of the server-side and client-side components of the Lenzing Global Auction Portal (LGAP) system. Since the idea of an online auction system is not new, several design choices were inspired by other systems providing similar functionality. Such systems include eBay and Amazon. Besides the requirements set by Lenzing, architectural features recommended by the developers of said systems are incorporated into the design of the LGAP system.

The thesis presents a microservice oriented software system on the server-side (backend), and two separate web applications following the single-page application principle on the client-side (frontend). The backend satisfies the requirements and limitations set by Lenzing regarding infrastructure, security, scalability and cloud deployment options following state-of-the-art practices and technologies. The frontend applications (including the customer facing user interfaces) also satisfies the requirements set by Lenzing. The customer facing part of the project is delivered as a responsive, mobile-first web application. A web based approach was chosen in order to support as many platforms as possible with the same development and maintenance effort. The single-page application principle along with the mobile-first design pattern was used to meet the mobile platform related priority set by Lenzing. The incorporated design choices also reflect state-of-the-art practices and technologies. The communication between the server-side components and client-side applications are realized with a single layer of interaction. Therefore, supporting client-side applications or devices other than those presented in this thesis (such as native Android or iOS applications) remain a straightforward task.

The thesis is divided into five main chapters. The first chapter presents the most common auction models used today. The second chapter presents the framework of the project, the requirements and requirement analysis of the system, use cases and interaction with other Lenzing platforms. An architecture overview of the entire system is presented in this chapter as well. The third chapter presents the server-side architecture and components in details. The chapter includes technology stack, data model, and component interaction. In case multiple approaches are possible for a given task, the decisions made during the development are elaborated. The fourth chapter presents the client-side part of the project. The client-side consists of two separate mobile-first Single Page Applications (SPAs). One SPA is for the customers, while the other is for the administrators of the online auction system. Similarly to the previous chapter, this chapter goes into details regarding technology stack, architecture, component interaction, and the interaction between the client-side components and the server-side components. The final, fifth chapter serves as a user guide for the system. This chapter includes an illustrated walkthrough of both client-side web applications, describing the functionality offered by them.
1. Auction Models

An auction in a very broad sense is a process where items are offered by a seller, and buyers can place bids to acquire the offered items. Bidders can potentially compete by raising their bid. From the seller’s perspective, there are two main goals regarding the outcome of the auction. The first goal is to gather information about the buyer’s real valuation of the sold items, while the second goal is to receive the highest profit (potentially above market value) on every item sold. From the buyer’s perspective, the primary goal is to acquire the sold items at, or below their true valuation. The secondary goal is to acquire the items for as cheap as possible, achieving a high payoff by maximizing the gap between their true valuation of the items and the final price.

The process of an auction is governed by the auction model selected for the auction. There are several existing auction models for different use cases. This chapter provides a brief overview of the most commonly used auction models. Through this analysis, the following features of an auction model are considered:

- Bid visibility, i.e. the submitted bids are published for the participants or not.
- Increasing or decreasing price, i.e. the price of the items is progressing in the favor of the bidders or the seller.
- Single or multi-turn, i.e. the auction process is separated into turns, where the outcome of each turn determines the progress of the auction, or the outcome of the auction is only evaluated at the end.
- Single or multi-unit, i.e. the number of items being sold in the auction.
- Synchronous or asynchronous, i.e. the participants are required to be present during the auction, or the participants can interact with the auction independently.

Bid visibility divides the auction models into two main categories, namely open-bid auction models and sealed-bid auction models. Open-bid auction models are usually synchronous, and can be further divided into ascending-price or descending-price auction models. The auction process determined by these models are multi-turn auctions. Several auction models in this category can be used for single-unit as well as multi-unit scenarios. The sealed-bid auction models are usually asynchronous and ascending-price. The most common sealed-bid auction models can also be divided into two main categories, namely first-price and second-price auction models. Sealed-bid auctions are usually evaluated after the auction ends, therefore they are considered single-turn. Figure 1. illustrates the auction model categories elaborated in this chapter.

![Figure 1. Main auction model categories for the LGAP system](image-url)
The auction model features mentioned above can be combined in several ways, where the resulting auction model does not fit into the categorization followed in this chapter. For example, an auction might consist of several sealed-bid turns, where the result of every sealed-bid turn is revealed and used as a basis for the next turn. This chapter focuses on the most common auction models based on the categorization portrayed by Figure 1.

1.1. English Auction

The English [1] [2] auction model is the most widely known and used auction model. It is used most often in situations where the seller is not aware of the potential buyer’s valuation, or the value of the sold items cannot be determined (e.g. selling art).

This auction model defines an open-bid, ascending-price, multi-turn and single-unit auction process. The auction process is synchronous, it is carried out in an action-reaction manner. Bidders place bids publicly, and other bidders react to them by placing their own bids, therefore this auction model results in an active competition. Information disclosed before the auction starts includes the start time of the auction, the item being auctioned and the starting price. Bidders might join at any time during the auction process. The seller can optionally set a reserve price and a minimum increment per bid.

The standing bid is available for all the participants at any time of the auction process. A bidder can enter the auction by placing a bid higher than the standing bid, thus acquiring the new standing bid. If a minimal increment is defined by the seller, the bid placed by a bidder must be higher than the standing bid with at least this amount. However, English auctions also allow bid jumping, meaning that a bidder might decide to raise the standing bid with an amount greater than the minimal increment.

Every valid bid determines the end of a turn. The end of a turn can also be tied to a fixed time period following the last bid. The auction ends when no higher bids are placed or at a fixed time determined by the seller. The winner of the auction is the bidder with the highest bid, and the price to be paid is the highest bid. However, if a reserve price was set by the seller, and it was not reached by the end of the auction, the seller can decide to not sell the item.

The optimal bidding strategy in the English auction is to always bid under one’s true valuation of the auctioned item [3]. Bidding under one’s true valuation is called bid shading. The amount of which a bidder can shade their bid can be determined by several factors, like the number of participants and the approximate valuation of the competition.

The general formula for bid shading is the following:

\[
\text{shaded bid} = \frac{\text{number of participants} - 1}{\text{number of participants}} \times \text{valuation}
\]

If only two bidders are participating in the auction, the shaded bid based on the general formula is the half of the bidder’s valuation. However, the more bidders are present, the closer the shaded bid gets to the true valuation.

Several auction models exist which are based on the English auction, or can be used along with the English auction to introduce new features. Some of these models are introduced below.
1.1.1. Candle Auction

English auctions, by their nature, have a side-effect that might not be desirable for the seller or the bidders. Since it is possible that the exact end of an auction is fixed by the seller, and it is known to all participants, and participants can enter at any time, it is also possible to wait until the very last moment of the auction to submit the highest bid, thus winning the auction without competition.

The Candle auction (also referred to as auction by the candle) introduces a new feature to the English auction to eliminate this side-effect. Originally, when this method was invented, the end of the auction was determined by the flame of a burning candle. When the flame went out, the auction ended. This introduced an uncertainty regarding the end of the auction.

Therefore, Candle auction refers to an English auction where the exact ending of the auction is not known to the participants.

1.1.2. Japanese Auction

The Japanese auction [4] is very similar to the English auction, however, it introduces two significant differences.

1. After the auction process started, no more bidders can join the auction.
2. The price is not driven by the bidders. Instead, it is periodically incremented by the seller. Following each increment, the bidders must decide to remain in the auction or drop out. Bidders who decide to drop out cannot reenter the auction.

The auction ends when only one bidder remains after an increment. The winner of the auction pays the last price announced by the seller.

1.1.3. Buyout Auction

The Buyout auction model does not alter the core mechanism of the base auction model it is used along with; however, it introduces a new feature regarding the end of the auction.

The added feature is a buyout price, at which any participant can acquire the auctioned item, independently of the progress of the auction.

1.2. Dutch Auction

The Dutch auction [2] [1] model is mostly used when the results of the auctions are required fast (e.g. when selling perishable goods). Its name originates from the flower markets of the Netherlands, where it is commonly used to sell flowers.

This model defines an open-bid, descending-price, multi-turn, single-item auction process. The auction process is carried out synchronously in real-time. The start time of the auction, item being auctioned, reserve price and price decrement per turn are all announced by the seller before the auction starts. Bidders might join at any time during the auction process.

The auction starts with a very high starting price, which is automatically decremented each turn by the value announced by the seller. The auction ends when any of the participants signals that they are willing to accept the last announced price, or the reserve price is reached. In case the price was accepted by a participant, the final price to be paid is the last
announced price. Otherwise, if the reserve price is reached the seller can decide to not sell the item.

The recommended bidding strategy involves bid shading, just like in the English auction. The general formula for bid shading is introduced in section 1.1. However, in the Dutch auction model, the pressure is higher on the participants to bid sooner, since once somebody submitted a bid, the auction is over.

1.3. First-price Sealed-bid Auction

The First-price Sealed-bid auction model [2] [1] is comparable to the English auction model. However, there are two main differences.

1. Participants may only submit one bid during the auction.
2. The bid placed by an individual participant is not visible to the other participants.

Therefore, this auction model defines a sealed-bid, single-turn auction process. Since the participants cannot see the bids placed by the others and the bids are only evaluated at the end of the auction by the seller, this auction process runs asynchronously.

The seller announces the starting time and ending time, with an optional reserve price. Bidders might bid any amount they wish; however, each participant may only submit one bid before the end of the auction. Following the end of the auction, the seller evaluates all the bids, and selects the highest bid. The bidder with the highest bid is the winner of the action. The final price to be paid by the winner is the highest bid. If a reserve price was set by the seller, and the highest bid did not reach the reserve price, the seller can decide to not sell the item.

The first-price sealed-bid auction model is strategically equivalent to the Dutch auction model; therefore, the recommended strategy involves bid shading. In both cases, participants have to make a decision without knowing anything about the other bids. However, using this auction model, the time related pressure is removed from the participants.

1.4. Vickrey Auction

The Vickrey auction [5], also known as second-price sealed-bid auction [2] [1] is very similar to the first-price sealed-bid auction model. The key difference is that the winner determined by the highest bid only has to pay the amount submitted by the second highest bidder.

This auction model is common in digital environments, for example a customized version of this auction model is used by Google to sell ad slots. The customization in Google’s case means that the criteria for determining the winner of the auction is not simply the highest bidder. Google factors in various attributes they associate with the bidder.

The recommended strategy in a second-price sealed-bid auction is to always bid one’s true valuation of the auctioned item. In order to understand this strategy, it is important to realize that there are only two cases, with each case having four scenarios.
The first case is when a bidder’s valuation is the highest. The corresponding four bidding scenarios are illustrated by Figure 2.

![Figure 2](image)

*Figure 2. Having the highest valuation in the Vickrey auction model*

Figure 2. shows that if a bidder’s valuation is the highest (1st price), said bidder can make the following four bids:

1. Bidding over their valuation. (A)
2. Bidding under their valuation, but over the bid of the 2nd highest bidder. (B)
3. Bidding the same as the 2nd highest bidder. (C)
4. Bidding under the 2nd highest bidder. (D)

Scenarios A and B are essentially the same, since both will result in the bidder with the highest valuation winning the auction. Scenario C is risky, because it is up to the seller to handle a tie and decide the winner. Scenario D will result in losing the auction. Therefore, bidding the true valuation will always result in winning the auction, given that the bidder has the highest valuation. Bidding anything else can only lead to losing the auction.

The second case is when a bidder’s valuation is not the highest. The corresponding four bidding scenarios are illustrated by Figure 3.

![Figure 3](image)

*Figure 3. Not having the highest valuation in the Vickrey auction model*
Figure 3. shows that any bidder not having the highest valuation also has four possible bids:

1. Bidding over their valuation, surpassing the 1\textsuperscript{st} price. (A)
2. Bidding over their valuation, but not surpassing the 1\textsuperscript{st} price. (B)
3. Bidding their own valuation. (C)
4. Bidding under their valuation. (D)

Scenario A will lead to winning the auction, however the price to be paid might be higher than the bidder’s valuation of the item. Scenarios B, C and D are essentially the same with B being risky, since it might lead to the same result as scenario A. Therefore, bidding the true valuation will result in losing the auction without the risk of having to pay more than the bidder’s valuation if the item.

In both cases, in each scenario, it is proven for every bidder that when bidding the true valuation, the payoff is at least as big as when bidding another value.
2. Project Description

The Lenzing Global Auction Portal (LGAP) project presented in this thesis involves the design and development of a mobile-first, online software system, capable of executing multiple simultaneous auctions for selected Lenzing customers.

The design and development of the LGAP system was realized in the framework of the Upper Austrian ISI Master Program (ISI) in cooperation with Lenzing A.G (Lenzing). The cooperative work between ISI and Lenzing in the year 2016-2017 involves the design and development of two software systems, one being the LGAP system, while the other one is the Lenzing Digital Customer Portal (LDCP). The LDCP system is designed and developed by Mahmoud Waleed and Mohamed Diab, both students of the ISI program.

The LGAP system and the LDCP system are designed and developed independently from each other. Among several other features provided by the LDCP system, it is responsible for customer user account management. This functionality is partially required by the LGAP system as well, however, the functionality is not duplicated here. The LDCP system provides the means for the LGAP system to consume the required customer user account related services, therefore the LGAP system relies on the LDCP system to some extent.

Lenzing operates in several countries and regions around the world. The auctions handled by the LGAP system are bound to one or more regions defined by Lenzing. Therefore, only customers who have the permission for a given region shall be able to access the auction running in that area. The customer and region relations are managed within the LDCP system, and queried by the LGAP system for each customer. The regions defined by Lenzing are the following:

- Europe and the United States
- North Asia
- Africa, Middle East and China

Customer operations are automatically filtered by regions, therefore customers can only access data about an auction given that they belong to the same region as the auction.

In addition to the region based access control, the LGAP system provides role based access control as well. There are two roles defined in the system:

- Administrator
- Customer

Following the sign-in, customers are automatically granted the customer role, while administrators are granted the administrator role. Resources and operations require certain role based permissions, meaning that an operation might only be accessible for administrators (e.g. canceling an auction), while other operations are only available for customers (e.g. placing a bid). The LGAP system can also handle multiple permissions for a single resource or operation.
2.1. Requirement analysis

This chapter introduces the requirements set by, and determined along with Lenzing regarding the LGAP system. The design patterns and principles utilized to meet said requirements are elaborated here.

Since the idea behind the LGAP system is in an experimental phase, and it is not yet fully developed by the company, the initial description of the system in terms of requirements was very vague. During the design and development of the LGAP system, the company provided the freedom of selecting the core design principles and technologies for the system. However, through various discussion sessions with the company, the following requirements were determined:

- The system must be accessible online;
- The system must target as many platforms as possible;
- The system must treat mobile platforms and mobile users with priority;
- The system must provide as simple and as clean user interface as possible; user experience through the system must be consistent and pleasant. Therefore, no disruptive or too complex functionality should be included.
- The system must include security principles by design;
- The system must ensure data privacy in terms of the stored data and data storage location.
- The system must be scalable to account for increasing demand;
- The system must be deployable to cloud services;
- The system must isolate auctions by regions. Therefore, a given auction must only be accessible for customers in the given region or regions where the auction is running;
- The system must use a relational database management system;
- The system must integration customer user account management related functionality from the LDCP system. Therefore, no such functionality should be replicated by the system;
- The system must be capable to be integrated with Lenzing SAP systems.
- The system must be flexible and easily extensible with different auction models in order to reflect new approaches, ideas or requirements.

Based on the last requirement in the above list, it should be noted that the LGAP system is designed in a modular fashion which is not tied to a particular auction model. However, the company selected the Dutch auction model for the prototype of the system, therefore the implementation contains only the Dutch auction model.

The system designed and developed within the frame of this thesis is divided into three main sub-systems:

- LGAP server (backend);
- Customer application;
- Administrator application.

In the following sections, each sub-system is elaborated individually, in terms of the requirements and provided functionality.
2.1.1. LGAP Server

The server-side components of the LGAP system are responsible for the data storage and persistence operations, for the execution of business logic and for serving the requests of client-side applications.

Security measures were taken into account while designing the server-side components. The data access operations and business logic operations are private, isolated parts of components or related group of components. However, the services provided by a component or group of components are tied to authentication and authorization. Therefore, security is addressed on the architecture level. The access control based on user regions and user roles is an integral part of the services provided by server-side components. However, it remains mostly transparent for the majority of the systems.

The persistence operations defined by server-side components are designed to be flexible and independent of the underlying persistence system. To meet the requirements of the company, persistence operations are currently implemented to use relational database management systems. However, changing the implementation to support other persistence management systems does not require the refactoring of the system. The data privacy issue regarding data storage location is implemented by completely decoupling the LGAP system from the data storage system. Therefore, from the LGAP system’s perspective, the location of the data storage services is irrelevant. Pointing the LGAP server-side components to a supported data storage service is a matter of configuration. Data privacy in terms of the stored data is implemented by encrypting sensitive information (e.g. passwords).

Scalability and the cloud based deployment were also handled as priority from the beginning of the design. They are both implemented on the architecture level.

The interaction between the LGAP system and the LDPC system is implemented in the server-side components as well. The LDPC system provides a web API for the LGAP system to perform the following operations:

1. Authenticate a customer user account by user identifier and password. The user identifier and password are transmitted to the LDPC server via HTTPS. Upon successful authentication, the LDPC system emits a token for authorizing further request in the name of the logged in customer.
2. Request the regions assigned to a customer. To request region data through the web API, the LGAP system must provide the authorization token previously emitted by the LDPC system.
3. Request customer user profile information. This request must also include the authorization token emitted by the LDPC system.

The core technology used by the server-side components is compatible with the SAP system, therefore future integration is possible. Furthermore, due to the architecture of the LGAP system, integration can be achieved by adding new components or modules to the system. The integration process does not require the refactoring of the system.
2.1.2. LGAP Customer Applications

The customer application provides access to the online auction system to selected Lenzing customers.

The customer application must be accessible for all of Lenzing’s global customer base. As of 2016, the majority of website visits are coming from mobile devices. Therefore, taking into account the requirements from Lenzing to support as many platforms as possible, but threat mobile platforms as a priority, the customer application is developed as a web application following the mobile-first design pattern [6] and the single-page application (SPA) principle. With the mobile-first approach, desktop computers, tablets and smart phones, as well as devices running supported web browsers are covered. The advantage of the mobile-first web applications is that only one single web application is developed and maintained. This web application will then automatically adopt to the environment in which it is being utilized.

The reason behind adopting the SPA principle in this project is also somewhat related to the mobile-first approach. A SPA is usually downloaded from the server once with the first request and then it runs on the client (in this case the web browser) utilizing the resources of the client device. The advantage of this design principle is that it minimizes the number of request-response interactions between the client and the server, thus reducing the load on the server as well as reducing the generated web traffic. A single page web application does not necessarily consist of one single page. The core idea behind this design principle is to not reload the whole web page following every single user interaction with the web application. Only reload certain parts of the web application that need to be changed.

The customer application consumes services provided by the backend through a web API. Authentication and authorization is completely transparent for the customer application. In order for Lenzing customer to gain access to the auction system, they must possess an active LDCP user account, as well as a privilege set by the LDCP system which allows the access of the auction system. Following a successful authentication through the LDCP system, the customer interface provides an overview of ongoing, upcoming and past auctions in the customer’s region. The customers can participate and bid in the ongoing auctions through the customer interface.

Figure 4. illustrates the use-cases of customers.

![Use-case diagram for the customer web application](image-url)
2.1.3. LGAP Administrator Application

The administration application provides an overview of the system, as well as operations for adding new products to the system, and starting new auctions based on an already existing product. Administrator accounts are treated differently from customer accounts. While the customer user accounts and customer authentication are managed by the LDCP system, the administrator user accounts and authentication are managed by the LGAP system.

Administrators can access a broader range of functionality compared to customers, but they do not have permission to perform all operations provided by the LGAP system.

The administrator application follows the same design principles and architecture determined during the design of the customer application. Therefore, it is also a mobile-first, single-page web application. However, it is developed as a separate web application.

Figure 5. illustrates the use-cases of administrators.

![Use-case diagram for the administrator web application](image)

2.2. System Architecture

The requirement to support as many platforms as possible is addressed by separating the LGAP system to client and server-side based on the Client/Server architecture model. The business logic of the system is provided by the server-side components, while implementing the user interface and user interaction handling remains the task of the platform specific client-side components. The LGAP system includes a client-side application for the web platform, which is based on the component-oriented architecture model.
During the design of the LGAP system, the architectural features of eBay and Amazon were taken into consideration, since both systems are global, and provide similar functionality. Both eBay and Amazon started out following the monolithic software architecture model [7]. However, as the user base and the daily load grew, both had to transition towards more scalable architectural approaches. The direction they chose is the service-oriented architecture model [8]. Based on the single responsibility principle [9], the functionality they provide is divided into independent services, where each service is scalable on its own. In addition to the functional decomposition and horizontal scaling at every tier, their architectural transition included the move to asynchronous execution environments. Based on these findings, the LGAP system on the server-side follows the microservice based architecture model [10] [11] [12]. In addition, the server-side is built on top of an asynchronous environment.

The functionality provided by the server-side LGAP components is divided into microservices. A microservices is a relatively small, lightweight, independent encapsulation of a service provided by the system. Every microservice in the LGAP system is deployable on its own, even to different cloud hosting services.

In addition to the vertical separation of the system into microservices, every microservice is horizontally separated into different layers, following the multi-layer software architecture model, which is popular in the monolithic software architectures. Therefore, the LGAP system on the server-side combines the microservice architecture model with the monolithic architecture model. This allows for horizontal scalability of the system on the level of microservices, as well as flexibility and scalability within the microservices. Every microservice maintains its own data model, data storage system, persistence operations and business logic implementation. The microservices publish their services into a service discovery infrastructure within the LGAP system. Services published by microservices are protected by JWT based authentication and authorization.

The LGAP system follows the message-oriented communication model among microservices. The message-oriented middleware is an event bus, accessible from all the microservices running in the system. The event bus can reach as far as the client-side applications. However, to support virtually any client-side application, the main communication channel between the server-side and the client-side is HTTP.

Services provided by individual microservices are collected from the service discovery infrastructure and published to the outside world by an API Gateway [13]. The web API provided by the API Gateway is the single layer of communication between the client-side applications and the microservices. Client-side applications are not aware of the individual microservices running in the system. Besides delegating client-side requests to the corresponding microservice, the API Gateway acts as a load balancer for the system. The API Gateway is implemented as a microservice, therefore all the general properties of microservices mentioned above apply to the API Gateway.

The architecture of the LGAP system is illustrated by Figure 6.

To sum up, the LGAP system is based on the Client/Server architecture model. On the server-side, the microservice and multi-layer architecture models are combined.
The LGAP system at its core consists of a public facing API Gateway, a service discovery infrastructure and an event bus. Services provided by the system are implemented as isolated, individually scalable and deployable microservices. Microservices interact with each other via the event bus, and publish their services to the service discovery infrastructure. The API Gateway provides a JSON based web API and acts as a Façade [14] for the system, delegating client-side request to the corresponding microservice registered in the service discovery infrastructure.

Figure 6. Architecture blueprint of the LGAP system
3. Backend

The server-side components of the LGAP systems (backend) are based on an open source
technology called Vert.x [15], maintained by the Eclipse Foundation. Vert.x defines itself
as a tool-kit for building reactive applications running on the Java Virtual Machine (JVM).

Vert.x is a polyglot technology, meaning that it supports several programing languages, not
only Java. The officially supported languages as of the time of this writing are Java,
JavaScript, Groovy, Ruby, Ceylon, Scala and Kotlin. In addition, several other languages
are supported (e.g. Python) through 3rd party extensions. In a polyglot environment, certain
parts of the system can be built using different programming languages, frameworks and
tools that fit best to the task. Meanwhile, the various parts can interact with each other as if
they were written in the same language. This fits very well to the microservice architecture,
as well as provides the benefit of having multiple developers with different technical
knowledge contribute to the system as a whole.

Among the out-of-the-box features of Vert.x, it supports the full range of enterprise
technologies and frameworks available for the JVM. This is a huge advantage regarding the
future of the LGAP system. Vert.x also provides a very good base framework for high
scalability and for implementing the microservice architecture.

Vert.x was chosen as the base framework for the LGAP backend for several reasons. The
key points are the following:

- Supports relational database management systems;
- Supports web requests over HTTP;
- Supports the asynchronous approach;
- Supports the event-driven and message-oriented approach;
- Supports the decoupling of components into individual, isolated modules;
- Supports service discovery;
- Supports the delegation of events among components, running on different hosts;
- Supports token based authentication and authorization.

Vert.x at its core is event-driven, asynchronous and non-blocking. The Vert.x core API does
not block the caller thread, therefore it can deal with many concurrent requests. This
corresponds to the recommendations of eBay developers [16], since the non-blocking
approach allows for higher scalability of the system. With every request, the caller can
specify a handler which will be called by Vert.x once the request is completed. This is also
the recommended way of developing applications on top of Vert.x. Vert.x implements a
custom, enhanced version of the Reactor design pattern, which is call the Multi Reactor
pattern. The main difference between the two patterns is the number of event loops they
define. The Reactor pattern defines one event loop thread to deliver events to the handlers
specified in the requests. Vert.x took this approach one step further, and assigns several
event loop threads for every Vert.x instance. By default, the number of event loop threads
is determined by the available CPU cores in the system. This results in Vert.x being scalable
with hardware upgrades. Vert.x implements a deployment and concurrency model based on
the Actor Model. The actors based on the Actor Model are called Verticles. A Verticle is a
unit of code that is deployed to, and executed by Vert.x. A Vert.x instance is a JVM process
that is running the Vert.x Engine. Every Verticle deployed to the Vert.x Engine is assigned
to an event loop thread, guaranteeing that it is always going to be executed on the same
thread. Via this approach, Vert.x removes the burden of multi-threaded programming from
the developer. Verticles can be designed in a single-threaded fashion without involving
synchronization. The multi-threaded execution of Verticles is handled in the background by
Vert.x.

Verticles deployed to the Vert.x Engine can interact with each other via the Vert.x Event
Bus. Each Vert.x instance has an event bus instance, however, a single event bus can span
several Vert.x instances, let them be local or remote instances. Message delivery on the
Vert.x event bus is based on the Publish-Subscribe design pattern. Publishers send their
messages to a string based address, while subscribers listen to messages on the same string
based address. A message delivered by the Vert.x event bus can support any arbitrary
formats through codecs. However, the Vert.x API heavily uses the JSON format, therefore
the recommended messaging format is also JSON.

The Vert.x core API provides a small and lightweight framework for creating asynchronous,
reactive applications. In addition to the core API, Vert.x provides API for various common
tasks through extensions. The LGAP backend uses three such extensions, namely the Vert.x
Web [17] extension for serving HTTP requests, the Vert.x Auth [18] [19] extension for
authentication and authorization, and finally the Vert.x JDBC Client [20] extension for
communicating with relational database management systems.

The server-side components of the LGAP system are implemented in Java, version 8.
Therefore, the remaining parts of this thesis are meant to be interpreted in the context of
Java 8, and are based on the Java API of Vert.x.

3.1. Microservices

Based on the features Vert.x provides through its core API and extensions, it is suitable for
developing the microservice oriented LGAP backend.

3.1.1. Architecture

The architecture of a microservice in the LGAP system follows the multi-layer architecture
model. Every microservice defines its own data model, repository layer, service layer and
web API layer. In addition to the API layer, a microservice might define a separate web
layer for serving arbitrary HTTP requests not mapped to the “/api/” URI prefix. The layering
is strict. Every layer specifies an interface through which its components can be accessed.
Components in a layer might only access the components in the layer directly below them.
This approach provides flexibility in terms of extensibility and refactoring.

The architecture of a microservice is illustrated by Figure 7.
There are several methods for decoupling a system into independent modules, or in this case microservices. One key point is the decoupling of the data model of the system. There are two basic approaches for this, one being a shared data model among microservices, while the other approach is an independent, private data model per microservice. The shared data model approach makes it easier to implement the communication among microservices, because every microservice has a uniform knowledge about the data represented by the data model objects. However, in case the data model objects are shared, microservices become dependent on each other. Any change on a given microservice’s data model requires the adaptation, rebuilding and redeployment of the entire system. The private data model per microservice approach makes it more difficult to share data among microservices, since a common layer of knowledge about the data needs to be introduced. However, the microservices remain flexible and independent.

The LGAP system slightly combine the two approaches. The microservices are determined by decoupling the data model of the system. Various parts of the data model and the related operations and services are assigned to a microservice. Several parts of the domain model are strongly related, therefore this process is not trivial. A microservice must satisfy the single responsibility principle, while avoiding inter-microservice transactions. In the LGAP system, every microservice maintains its own private data model. Certain data models that must not be redefined by every microservice are shared. These data models represent an abstract concept, and are not subject to frequent changes, but if they do change, the entire system must be aware of the changes. In case a data model object must be referenced from a different microservice, only the identifier of the data model object is shared. The data
structure or the state of the data is not shared. It is the duty of the client-side applications to handle the embedding and the fetching of the external objects based on the identifier.

Data model objects are defined as Plain Old Java Objects (POJO) within a microservice. The public services provided by a microservice publish the data in JSON format. The structure of the published data does not necessarily reflect the data structure defined by the POJO. Vert.x provides a code generator for automatically generating JSON to POJO and POJO to JSON converters. Unfortunately, the functionality of the generated converters is very narrow, therefore the LGAP system defines its own converters.

The **repository layer** is responsible for the data storage system and the persistence operations. Both are determined on a per microservice basis. The persistence operations are defined through repository interfaces. The data storage system specific persistence operations are implemented separately, conforming to the declarations in the repository interfaces. Therefore, a microservice is capable of supporting several data storage systems.

The **service layer** implements the business logic of the microservice. The services are defined via service interfaces. A service can be implemented as a local service, as an event bus service or as a Verticle. Local services are meant to be used by the other components of the microservice only. Service interfaces for event bus services which are meant to be used by other microservices are shared. In this case, the shared service interface contains the address where the services are published. A service is implemented as a Verticle when it needs to be scalable on its own.

The **API layer** publishes the public services of a microservice via the HTTP protocol. Since the API is open to the client-side request, the authorization of requests is implemented in this layer. The LGAP API components are using the JSON Web Token (JWT) mechanism for authentication and authorization. The JWT implementation is provided by the Vert.x Auth JWT extension. The authorization on the API layer of microservices is implemented in order to limit the access on them to the API Gateway. Each microservice publishes its API services to the service discovery infrastructure of the LGAP system. The published service record includes the name of the service, the HTTP address and port where the service is available, and the event bus address where a short-lived authorization token can be requested from within the system. The API Gateway can look up the individual service records in the service discovery infrastructure, and can request the authorization token through the event bus. The request must include the user role and regions. The microservices in the LGAP backend are running in a cluster. Normally, the API of a microservice should not be visible from outside the network defined by the cluster. Since the authorization token can only be requested from within the cluster, in the case of a bad network configuration (e.g. wrong port forwarding), direct requests made to a microservice will always fail due to the lack of the authorization token. The API layer components are implemented as Verticles.

Vert.x does not provide an out-of-the-box dependency injection mechanism. To use an existing framework, it must be compatible with the asynchronous, non-blocking nature of Vert.x. One such framework recommended by the Vert.x community is Guice [22]. Guice is a lightweight dependency injection framework developed at Google. Guice supports the annotation based dependency resolution and injection, therefore there is no need for external configuration files to specify mappings. Despite being lightweight, Guice provides several
mechanisms, including the direct mapping of interfaces to implementations, dependency providers for customizing the dependency resolution, named dependencies, constructor injection, field injection, lazy loading and singletons.

The LGAP system takes advantage of almost all the features provided by Guice to decouple the internal working of the components from the dependency resolution. The different layers specify only the interface of the components they want to communicate with. Resolving the dependency and creating the instance is handled by Guice. The disadvantage of Guice compared to an Inversion of Control container is that the bindings must be specified manually, either by annotations or via direct mappings. The LGAP system uses the direct mapping mechanism by creating a custom Guice Module for each microservice. The advantage of direct mapping is that all the mappings are configured in a central location. However, component specific dependency resolution behavior is either specified by providers or via annotations.

3.1.2. Exception Handling and Logging

The recommended way of delegating exceptions within the framework of Vert.x is through the handlers passed to the asynchronous methods. Handlers are notified about failed operations by passing a failed future object to the handler. A failed future object contains a failure message and an optional chained exception. In case an exception occurs in a layer, the exception is logged, wrapped into a layer specific exception type and it is passed to the next layer via a failed future object.

3.1.3. Configuration

Every Vert.x instance maintains a JSON based configuration. The LGAP system follows this approach. Each microservice has a configuration file in the JSON format. Microservices do not specify default configuration parameters, therefore the configuration file must always be present and up-to-date. The configuration file is loaded by the LGAP Launcher at startup time. The LGAP Launcher is an extension of the Vert.x Launcher. It is used to provide a common startup environment for all the microservices. The launcher is responsible for initializing and starting the Vert.x instance and configuring the cluster. The cluster configuration is part of the LGAP Launcher, therefore microservices launched with the same launcher are automatically added to the same cluster, no matter where they are running.

The LGAP system uses the Hazelcast [23] cluster manager implementation provided by Vert.x [24] for managing the clusters and the nodes in the cluster. This is the default cluster manager for Vert.x, however several others are also supported. The LGAP Launcher assumes that the configuration file is located in the config directory at the root of the class path, and the file itself is called config.json. The content of the configuration file is a valid JSON object, where every key-value pair is a configuration element. The configuration element is also a JSON object, containing the configuration parameters. Every microservice follows the same structure for the configuration file. The configuration element related to the relational database management system is slightly different compared to the other configuration elements. The keys within this JSON object are in the format expected by the Vert.x JDBC extension, therefore the configuration can be passed directly to the extension. In the following section, the common configuration elements are described. The description
contains the configuration element key as well as the key, expected Java type and the description for every parameter in the configuration element.

The configuration element identified by the key “data.source” specifies the configuration of the relational database management system, via the following parameters:

- “driver_class” (java.lang.String): the fully specified Java class name of the JDBC driver corresponding to the specified relational database management system.
- “user” (java.lang.String): the user name used to connect to the relational database management system.
- “password” (java.lang.String): the password of the previously specified database user.
- “max_pool_size” (java.lang.Integer): the maximum number of concurrent database connections maintained by the driver.

The configuration element identified by the key “api” specifies the configuration of the API component and the service record published to the service discovery infrastructure. For any other non-API web service, the same parameters are specified under the “web” configuration element. The parameters are the following:

- “service.name” (java.lang.String): the fill name of the service publishing the API.
- “http.address” (java.lang.String): the host name where the API server is running.
- “http.port” (java.lang.Integer): the port on which the API server is listening.
- “http.ssl” (java.lang.Boolean): a switch indicating whether the API server is running on ssl mode or not. In case this parameter is set to true, the configuration element must contain the parameter “http.ssl.certificate” (io.vertx.core.json.JsonObject) with the following parameters:
  - “jks.file” (java.lang.String): path to a JKS type key store file used to store the SSL certificate.
- “metadata” (io.vertx.core.json.JsonObject): contains meta information about the service for the service discovery infrastructure. This JSON object contains the following parameters:
  - “endpoint.name” (java.lang.String): the part of the service name which appears in the URL.
  - “endpoint.type” (java.lang.String): type of the HTTP endpoint, which is used to distinguish between API services and any other web services provided by the microservice.

The configuration element identified by the key “jwt.auth.provider” specifies the configuration of the JWT authorization mechanism protecting the API. The parameters are the following:

- “keystore.file” (java.lang.String): path to the key store file used to store the JWT certificate.
- “keystore.type” (java.lang.String): type of the key store file supported by the Vert.x JWT implementation (e.g. JKS or JCEKS).


- “keystore.algorithm” (java.lang.String): the token signing and verification algorithm used by the Vert.x JWT implementation (e.g. HS256).

Every microservice can add its own configuration elements to the configuration file, following the same structure described above.

3.1.4. Build and Deployment

The LGAP backend is organized into Apache Maven [25] modules. Every microservice is wrapped into a Maven module, therefore it is possible to build them individually. The output of the build process is called *fat-jar* or *uber-jar*. A fat-jar is an executable Java Archive (JAR) file with all its dependencies packaged into the JAR. Therefore, microservices can be deployed individually to any machine running a Java Virtual Machine. The only requirement other than a JVM is a network connection.

Packaging a microservice into a fat-jar is handled by the Apache Maven Shade plugin [26]. The Shade plugin fetches, organizes and packages all the indicated dependencies of a microservice along with the microservice. The code generated by the Vert.x code generator is also handled by the Shade plugin. The Shade plugin requires the Main-Class manifest entry in order to create an executable JAR file. The Main-Class property of a microservice must be set to the class name of the Vert.x Launcher used to launch the Vert.x instance. The Vert.x Launcher provides a way for specifying command line arguments and custom hooks for different lifecycle phases of the system startup. The LGAP backend defines two customized Vert.x Launchers, in the lgap-launcher module. The custom launchers are inserted into the Vert.x Launcher hierarchy, and are meant to handle LGAP specific startup operations. The complete launcher hierarchy is illustrated by Figure 8.

![Figure 8. LGAP microservice launcher type hierarchy diagram](image)

The *edu.lenzing.lgap.launcher.Launcher* class extends the *io.vertx.core.launcher.Launcher* class and provides all the base configuration and custom lifecycle operations needed to initialize and launch a microservice. The *edu.lenzing.lgap.launcher.ClusteredLauncher* further extends this launcher class to provide the cluster configuration for microservices. Therefore, the Main-Class manifest entry in the Shade plugin configuration is set to point to the *edu.lenzing.lgap.launcher.ClusteredLauncher* class. The Main-Class manifest entry is a common configuration of the whole LGAP backend, therefore it applies to all the microservices.

Besides the Main-Class manifest entry, every microservice must specify its main entry point in the form of a Verticle class. This Verticle is launched by the launcher, and it is responsible
for initializing and starting up other Verticles needed by the microservice. The main Verticle is specified via the Main-Verticle manifest entry of the Shade plugin. Every microservice must specify this individually by declaring a property in the Maven POM descriptor with the name main.verticle, and setting it to the class name of the main Verticle of the module.

3.2. Common Microservice Module

The LGAP backend defines common functionality and configuration for microservices in the lgap-microservice-common module. This module does not contain a microservice on its own, however, it is embedded into every microservice.

This module defines abstract base classes for Verticles, dependency injection, the data model and repository layers as well as custom exception types. Shared service interfaces are also defined in this module.

Building this module results in simple JAR file (Maven artifact), which is referenced by the microservice modules as a dependency.

3.2.1. Common Data Model

The common data model related elements of the LGAP system are organized in the edu.lenzing.lgap.microservice.common.model package.

The LGAP system adds the edu.lenzing.lgap.microservice.common.model.AbstractModel abstract base class to encapsulate shared attributes of data model classes, also referred to as entities. As per the request of Lenzing, the LGAP system uses relational databases for persisting data. The AbstractModel class includes a 64-bit numeric field (java.lang.Long) for storing the id generated by the relational database system. Thus, every entity class in the LGAP system automatically inherits this field.

Besides the common id field, the AbstractModel class declares a constructor which takes a io.vertx.core.json.JsonObject as parameter and a toJson method, which returns the same JsonObject type. The constructor is used to create an entity instance based on a JSON object, while the toJson method converts the entity instance into a JSON object. However, these are not merely for utility purposes. Vert.x uses the JSON format heavily in its internal workings for polyglot support and message-based communication. In order to make a new type compatible with the Vert.x API, it must be annotated with the io.vertx.codegen.annotations.DataObject annotation and must provide the means for converting to, and from JSON object via the constructor and method mentioned above. Therefore, this extra constructor and method is provided for every entity type.

Furthermore, the common package defines two enumerations for specifying the user roles and user regions. The edu.lenzing.lgap.microservice.common.model UserRole type is the first enumeration, and it lists all the possible roles the system can handle. The second enumeration is the edu.lenzing.lgap.microservice.common.model.UserRegion type, which contains all the regions specified by Lenzing that are used for access control.
3.2.2. Common Repository Components

The `edu.lenzing.lgap.microservice.common.repository` package includes the common components of the repository layer available for the microservices.

Unfortunately, Vert.x does not provide out-of-the-box support for sophisticated Object Relational Mapping (ORM) frameworks, and the Java Persistence API (JPA) [27] implementations are synchronous and blocking. Using a blocking framework is possible in the frame of Vert.x, but integrating a sophisticated ORM system or a JPA provider completely undermines the concept of lightweight, asynchronous microservices.

One option could have been the JOOQ framework [28], which provides Vert.x support, however it follows a database first approach. JOOQ generates POJOs from database schemes, and provides persistence operations on the generated POJOs. This approach is not desirable in the frame of this project, because Vert.x also sets certain requirements for the data objects, therefore the generated POJOs must be modified after every code generation to meet these requirements.

The LGAP system uses the Vert.x JDBC Client extension for connecting to, and interacting with relational database management systems (RDMS). The disadvantage of using this extension is that SQL queries must be written and maintained along with the codebase. However, the microservices remain lightweight. The Vert.x JDBC Client provides an asynchronous API wrapped around the inherently synchronous and blocking JDBC API [29]. To achieve this, the database operations are executed on worker threads. Worker threads are separate from the event loop threads, therefore executing blocking code on them does not cause disruptions in the system. When a blocking operation completes, the Vert.x JDBC API calls the handler associated with the operation.

Acquiring a database connection, maintaining a connection pool and delegating persistence operations are implemented by the `io.vertx.ext.jdbc.JDBCClient`. The LGAP backend provides the `edu.lenzing.lgap.microservice.common.repository.BaseJDBCRepository` class as an abstract layer over the `JDBCClient`. The methods provided by the `BaseJDBCRepository` hide most of the logic required by the `JDBCClient`, thus allowing the concrete repository implementations to focus on their own persistence logic. The `BaseJDBCRepository` expects incoming parameters, and provides the query results in the JSON format. It is the responsibility of the implementing repository component to build the entity object from the JSON object.

The `edu.lenzing.lgap.microservice.common.repository.BaseJDBCEntityRepository` class provides another layer of abstraction on top of the `BaseJDBCRepository`. The `BaseJDBCEntityRepository` class provides a generic way to perform persistence operations on entity objects extending the `AbstractModel` class. However, this is still not an entity manager, therefore references among entities must be handled manually.

The `edu.lenzing.lgap.microservice.common.repository.BaseRepository` interface declares basic repository operations to be implemented by every concrete repository component.
3.2.3. Common Service Components

The `edu.lenzing.lgap.microservice.common.service` package contains shared service interfaces and abstract base implementations of services.

Shared service interfaces are used for publishing and consuming event bus services. The service publisher specifies the implementation of the service, while the consumer can access the service via an event bus service proxy. The shared service interfaces usually contain the event bus address where the service is published. If this is not the case, the service address is published through the service discovery infrastructure, or it is part of the private configuration of the involved microservices. Furthermore, shared service interfaces specify the `io.vertx.codegen.annotations.ProxyGen` and `io.vertx.codegen.annotations.VertxGen` annotations. The `ProxyGen` annotation tells Vert.x to generate event bus service proxies for the given service. The `VertxGen` annotation enables polyglot support. The service interfaces conform to the fluent nature of the Vert.x API. Every declared method is annotated with the `io.vertx.codegen.annotations.Fluent` annotation, and returns the same type as the service itself.

The `edu.lenzing.lgap.microservice.common.service.ApiJWTAuthService` interface defines the means to acquire an API authorization token from the microservices. This service interface does not contain the event bus address of the service, instead the address can be found in the service discovery infrastructure in the service record published by the microservice.

The `edu.lenzing.lgap.microservice.common.service.APIGatewayJWTAuthService` interface extends the `ApiJWTAuthService`, and adds the event bus service address of the service. The `edu.lenzing.lgap.microservice.common.service.impl.BaseApiJWTAuthServiceImpl` abstract base class provides the basic implementation for generating the JWT token.

3.2.4. Common Verticles

The `edu.lenzing.lgap.microservice.common.vertx` package contains abstract base classes for the different type of Verticles used in the system. These Verticles are organized into a hierarchy, gradually adding new functionality. The hierarchy of Verticles is illustrated by Figure 9.

---

**Figure 9. LGAP base Verticle type hierarchy diagram**
To create a Verticle, the defining class must implement the `io.vertx.core.Verticle interface`. This interface provides the functionality required to initialize and deploy a Verticle to a Vert.x instance. Vert.x provides an abstract base implementation for the Verticle interface in the `io.vertx.core.AbstractVerticle` class. The functionality of a microservice in the LGAP system is divided into several Verticle instances deployed to a Vert.x instance.

The `edu.lenzing.lgap.microservice.common.vertx.BaseMicroserviceVerticle` is the first component of the Verticle hierarchy defined by the LGAP system. This class extends the `AbstractVerticle` directly. The `BaseMicroserviceVerticle` class is the base class for every Verticle representing a microservice. It provides the service discovery infrastructure as well as the service and service proxy registries. It provides methods for registering different kind of services, as well as unregistering them when the microservice is stopped. Every microservice in the LGAP system contains at least one Verticle extending the `BaseMicroserviceVerticle`. This Verticle is used as the entry point of the microservice, and it is responsible for initializing and deploying other type of Verticle instances. Verticles extending the `BaseMicroserviceVerticle` must call the start and stop methods of the base class in case they overwrite them.

The `edu.lenzing.lgap.microservice.common.vertx.BaseHttpEndpointVerticle` component is the next component in the hierarchy. It extends the `BaseMicroserviceVerticle` and provides the means to create and start a HTTP server on a given host, port and router configuration. Besides that, it provides shortcut methods to respond to HTTP requests.

The `edu.lenzing.lgap.microservice.common.vertx.BaseApiVerticle` is the final component of the hierarchy. It extends the `BaseHttpEndpointVerticle`, adding base functionality for Verticles providing web API. Such functionality incudes the authentication and authorization of API requests.

3.2.5. Common Dependency Injection Module

The `edu.lenzing.lgap.microservice.common.guice` package contains the abstract base class for custom Guice modules. The hierarchy of base classes is illustrated by Figure 10.

To use the non-annotation based dependency resolution mechanism of Guice and specify custom dependency providers, the Guice instance must be initialized with a class implementing the `com.google.inject.Module` interface.

Guice provides the `com.google.inject.AbstractModule` class as a base implementation for the Module interface. The LGAP system also provides an LGAP specific base Guice module.

![Figure 10. LGAP base Guice module hierarchy diagram](image-url)
The `edu.lenzing.lgap.microservice.common.guice.BaseVertxGuiceModule` class extends the `AbstractModule`, and adds custom dependency providers for the Vert.x instance and configuration object associated with the Vert.x instance.

A custom dependency provider in this case is any public method of the `BaseVertxGuiceModule` class annotated with the `com.google.inject.Provides` annotation. Guice selects the appropriate provider via the return type of the method.

For example, the dependency provider method responsible for resolving the Vert.x instance returns the `io.vertx.core.Vertx` type. To inject the Vert.x instance into a class, the class must declare a field or a constructor with the same `Vertx` type parameter, and the field or the constructor must be annotated with the `com.google.inject.Inject` annotation.

The Vert.x configuration object can be injected as a `JsonObject` type via named dependency resolution. A named dependency includes the `javax.inject.Named` annotation along with the Provider and Inject annotations. The Named annotation takes a String parameter to specify the name of the dependency. The name of the named configuration object dependency is “VertxConfig”. Injecting the configuration object always resolves to the instance maintained by Vert.x, therefore modifications to the injected object are reflected through the Vert.x instance.

The `BaseVertxGuiceModule` class provides another named dependency for injecting the `io.vertx.ext.auth.jwt.JWTAuth` instance into API Verticles. The JWTAuth provides the JWT based authentication and authorization services. The name of the named dependency is “ApiJWTAuthProvider”. The JWTAuth instance is created only once, therefore injecting it will always resolve to the same instance. This is achieved by adding the `com.google.inject.Singleton` annotation to the dependency provider method.

### 3.2.6. Common Exception Types

The `edu.lenzing.lgap.microservice.common.exception` package defines the base exception types for the LGAP system. Layer specific packages add further, concrete exception types. The base exception type hierarchy of the LGAP system is illustrated by Figure 11.

Every exception type defined in the LGAP system provides static factory methods for creating failed Future objects based on the implementing exception type.
3.2.7. Common Configuration and Resource

The `lgap-microservice-common` module does not include shared configuration files, however it does include various shared resources. The shared resources are stored in a directory called “config”. This directory is copied to the root of the fat-jar created by building a microservice module which includes the `lgap-microservice-common` module.

To avoid confusion and potential collision with resources defined by the individual microservice modules, the shared resources are stored in the “common” subdirectory of the “config” directory.

With the base configuration of the system, the `config/common/https/keystore.jks` file contains the SSL certificate used for HTTPS transmissions, while the `config/common/jwt/keystore.jceks` file contains the certificate used for signing tokens used by the APIs. However, these resources can be provided by the individual microservice modules, therefore it is not mandatory to use the shared key store files.

3.3. Product Microservice Module

The `lgap-microservice-product` module contains the `edu.lenzing.lgap.microservice.product` package. The purpose of the Product microservice is to provide access to Lenzing products for the rest of the LGAP system.

The data model of the Product microservice consist of only one data model object, namely the `edu.lenzing.lgap.microservice.product.model.Product`. The data model hierarchy is illustrated by Figure 12.

![Figure 12. Product Microservice, data model diagram](image)

The Product model directly extends the `AbstractModel`, and it is used to represent information about Lenzing products. The Product model to JSON and JSON to Product model conversion is realized with converters automatically generated by Vert.x during the build time.

The Product model related repository operations are specified in the `edu.lenzing.lgap.microservice.product.repository.ProductRepository` interface, which extends the `BaseRepository` interface. The JDBC specific implementation of the `ProductRepository` is provided by the `edu.lenzing.lgap.microservice.product.repository.jdbc.JDBCProductRepository` class. The `JDBCProductRepository` class extends the `BaseJDBCEntityRepository` class.

The `edu.lenzing.lgap.microservice.product.service.ProductApiJWTAuthService` interface extends the `ApiJWTAuthService` and specifies the event bus service address where the API authorization token can be requested.
The `edu.lenzing.lgap.microservice.product.service.impl.ProductApiJWTAuthServiceImpl` class provides the default token generation mechanism for the `ProductApiJWTAuthService` by extending the `BaseApiJWTAuthServiceImpl` class.

The `edu.lenzing.lgap.microservice.product.guice.ProductGuiceModule` Guice module extends the `BaseVertxGuiceModule` class and adds the Product microservice specific dependency resolution bindings.

The `edu.lenzing.lgap.microservice.product.ProductMicroserviceVerticle` Verticle is the main entry point of the Product microservice. Upon startup, it injects the event bus service address of the product API JWT service from the `ProductApiJWTAuthService` interface to the Vert.x configuration object. It creates the Guice injector based on the `ProductGuiceModule` class, and deploys the product API Verticle.

The `edu.lenzing.lgap.microservice.product.ProductApiVerticle` extends the `BaseApiVerticle`, creates the HTTP endpoint for the API, registers the URL handlers and provides the API request handler methods.

The Product microservice does not specify any additional configuration elements in its configuration file.

Currently, product information is uploaded through the LGAP administrator application. However, it is possible that in the future the LGAP system has to fetch information about products directly from Lenzing SAP systems. Such tasks should be implemented within the Product microservice, and therefore the change should remain transparent for the rest of the system.

This description of the Product microservice module introduced the layering and modularization of a microservice in the LGAP system. The remaining microservice modules follow the same structure and naming conventions. Therefore, the following sections describe only their unique parts.

### 3.4. Auction Microservice Module

The `lgap-microservice-auction` module contains the `edu.lenzing.lgap.microservice.auction` package. Besides persistence and service operations, the Auction microservice is responsible for the management of the auction processes.

The `edu.lenzing.lgap.microservice.auction.model` package contains the data model the Auction microservice, which consists of several data model objects. The data model hierarchy is illustrated by Figure 13.
The `edu.lenzing.lgap.microservice.auction.model.Cargo` model object is used to specify the product that is being sold in an auction. The Cargo model object contains the identifier of the product that is associated with the cargo. Besides the product reference, the geographical location of the cargo, the amount of auctioned products and the expected delivery date in the region is stored.

The `edu.lenzing.lgap.microservice.auction.model.Auction` model object encapsulates information that is common for every auction, independent of auction model. As described earlier, the LGAP system prototype only implements the Dutch auction model. To add support for different auction models, specific auction classes extend this class and add properties required by the auction model they represent.

The `edu.lenzing.lgap.microservice.auction.model.AuctionPhase` enumeration contains all the possible states in which an auction process can be.

The `edu.lenzing.lgap.microservice.auction.model.AuctionType` enumeration contains the supported auction model types.

The `edu.lenzing.lgap.microservice.auction.model.DutchAuction` model object extends the Auction model object, and adds the Dutch auction model specific properties, such as price modifier and turns.

The `edu.lenzing.lgap.microservice.auction.model.DutchAuctionTurn` model object represents a turn in the Dutch auction, specifying all the details for managing the turn based Dutch auction process.
The `edu.lenzing.lgap.microservice.auction.model.Bid` model object is responsible for storing the placed bids, and linking the customers to auctions.

To handle data model to JSON and JSON to data model conversion, the Auction microservice defines its own converters. The converters are located in the `edu.lenzing.lgap.microservice.auction.model.converter` package. The automatic converter generation capability of Vert.x is not used in this microservice module, because of the complex relations of data model objects within this module. The code generation mechanism of Vert.x is not customizable, and the generated converters cannot handle cyclic relations.

The repository layer, service layer, dependency injection module, API Verticle and JWT authentication and authorization mechanism is implemented in a similar fashion as described in the 3.3 section.

The persistence strategy concerning the Java type inheritance between the base Auction model and the specific auction models extending the Auction class is the same as the Joined Inheritance Type defined by the JPA specification. Every model object has its own corresponding database table. The base Auction class specifies a discriminator value in its table. The discriminator value is the `AuctionType`, and it is used to join the base table with the extension table. The ID of every auction is generated in the base Auction model’s table, and it is stored in the extension tables as well.

Each auction type has its own repository and service components, however the `edu.lenzing.lgap.microservice.auction.repository.AuctionRepository` interface along with the `edu.lenzing.lgap.microservice.auction.service.AuctionService` interface specify operations that are auction model independent. For example, searching and filtering is defined in these components.

The `edu.lenzing.lgap.microservice.auction.service.AuctionBroadcastService` defines operations for broadcasting auction process related information within the LGAP system. The event bus is used for broadcasting information, however instead of the point-to-point communication model used for the JWT mechanism, the publish-subscribe communication model is used. Subscribers can listen to the general broadcasting channel, or auction specific channels. To listen to messages only about a selected auction, the ID of the selected auction is appended to the event bus address of the broadcast service.

The `edu.lenzing.lgap.microservice.auction.AuctionManagerVerticle` is responsible for starting auction processes managers when the Auction microservice is started.

Since the auction processes are time based, the current progress of auctions is verified and adjusted at every startup of this microservice. This means that if an auction process should have been ended while the system or this microservice was offline, the auction is not continued when the system becomes online again. Instead, it is updated to the appropriate status. This behavior applies to individual turns of turn based auction processes as well.

The actual management of auction processes is further divided into various Verticles. The `edu.lenzing.lgap.microservice.auction.manager` package contains the interfaces and implementations of the various auction managers.
The `edu.lenzing.lgap.microservice.auction.manager.DutchAuctionManager` interface extends the `edu.lenzing.lgap.microservice.auction.manager.AuctionManager` and `Verticle` interfaces and specifies the operations related to managing the Dutch auction. The `Verticle` interface is added to the hierarchy to be able to deploy the manager as a Verticle. The `edu.lenzing.lgap.microservice.auction.manager.verticle.DutchAuctionManagerVerticle` class implements the `DutchAuctionManager`. However, it is not required to implement auction managers as Verticles. To add support for additional auction models, the model specific manager only has to implement the `AuctionManager` interface.

The `edu.lenzing.lgap.microservice.auction.AuctionMicroserviceVerticle` is the main entry point of this microservice module. Besides deploying the API Verticle, it is also responsible for deploying the `AuctionManagerVerticle`.

### 3.5. Customer Microservice Module

The `lgap-microservice-customer` module contains the `edu.lenzing.lgap.microservice.customer` package. This microservice module handles the interaction between the LGAP and LDCP systems, manages the customer data within the LGAP system and functions as a webserver for the LGAP customer web application. The customer web application is embedded into the `/webroot/` directory of the fat-jar of this microservice module.

The data model of this microservice is in the `edu.lenzing.lgap.microservice.customer.model` package. The data model hierarchy is illustrated by Figure 14.

![Customer Microservice, data model diagram](image)

The `edu.lenzing.lgap.microservice.customer.model.Customer` model object does not contain all the data associated with a Customer. This information is queried from the LDCP system every time it is required. The LGAP system stores only the permanent customer identifier provided by the LDCP system.

The `edu.lenzing.lgap.microservice.customer.model.Discount` model object represents the discount assigned to a customer for an auction.

The repository layer, service layer, dependency injection module, API Verticle and JWT authentication and authorization mechanism is implemented in a similar fashion as described in the 3.3 section.

The `edu.lenzing.lgap.microservice.customer.service.LenzingCustomerPortalService` interface specifies the various LDCP system operations that are available in the LGAP system. The `LenzingCustomerPortalService` is implemented by the `edu.lenzing.lgap.microservice.customer.service.impl.LenzingCustomerPortalServiceImpl`
class. The communication between the LDCP and LGAP systems is through HTTP, and it is implemented using the Vert.x Web extension.

The `edu.lenzing.lgap.microservice.customer.service.CustomerAuthenticationService` interface defines the customer authentication mechanism. This is implemented by the `edu.lenzing.lgap.microservice.customer.service.impl.CustomerAuthenticationServiceImpl` class. The customer authentication includes several steps, many of which is provided by the LDCP system. The `CustomerAuthenticationServiceImpl` class performs all of these steps, and only authenticates the user when all the steps are successfully completed.

The `edu.lenzing.lgap.microservice.customer.CustomerWebVerticle` extends the `BaseHttpEndpointVerticle`, and it is responsible for providing access to the LGAP customer web application. This Verticle maintains the routes and route handlers for accessing the web application. However, it is not necessary to deploy the customer web application to this microservice. The `CustomerWebVerticle` acts as a very simple web server with no specific functionality tied into the customer web application, therefore the customer web application can be deployed to any web server.

The `edu.lenzing.lgap.microservice.customer.CustomerMicroserviceVerticle` is the main entry point of this microservice module. The `CustomerMicroserviceVerticle` is responsible for deploying the `CustomerWebVerticle` and the API Verticle.

### 3.6. Admin Microservice Module

The `lgap-microservice-admin` module contains the `edu.lenzing.lgap.microservice.admin` package. The Admin microservice is responsible for administrator user account management and authentication. This module also functions as a webserver for the LGAP administrator web application.

The data model of this microservice is in the `edu.lenzing.lgap.microservice.admin.model` package. The data model hierarchy is illustrated by Figure 15.

![Figure 15. Admin Microservice, data model diagram](image)

The `edu.lenzing.lgap.microservice.customer.model.Admin` is the single data model object of this microservice. However, the Admin model object contains the authentication data of admin users. The administrator passwords stored by this microservice module are encrypted via the `edu.lenzing.lgap.microservice.admin.util.PasswordEncryptor` component. To make the encryption process configurable, the Admin microservice adds the “encryption” configuration element to its configuration file. The content of this configuration element is the following:

- “encoding” (`java.lang.String`): the character encoding used when encrypting the password (e.g. UTF-8).
• “algorithm” (java.lang.String): the algorithm used to perform the encryption (e.g. SHA-256)
• “salt” (java.lang.String): the salt value used for the encryption.

The `edu.lenzing.lgap.microservice.admin.service.AdminAuthenticationService` interface defines the authentication mechanism. The `AdminAuthenticationService` is implemented by the `edu.lenzing.lgap.microservice.admin.service.impl.AdminAuthenticationServiceImpl` class. Unlike the customer authentication, the administrator user account management and authentication is handled completely by the LGAP system.

The `edu.lenzing.lgap.microservice.admin.AdminWebVerticle` extends the `BaseHttpEndpointVerticle`, and it is responsible for providing access to the LGAP administrator web application in a similar fashion as described in section 3.5.

The `edu.lenzing.lgap.microservice.admin.AdminMicroserviceVerticle` is the main entry point of this microservice module. The `AdminMicroserviceVerticle` is responsible for deploying the `AdminWebVerticle` and the API Verticle.

### 3.7. API Gateway Microservice Module

The `lgap-microservice-api-gateway` module the `edu.lenzing.lgap.microservice.apigateway` package. This microservice is the public front of the LGAP backend. All requests towards the LGAP backend are addressed to this microservice, and all the traffic is routed through it.

The API Gateway microservice is responsible for determining the appropriate microservice for handling a client-side request. Once the appropriate microservice is found, the request is delegated to it. When the selected microservice finishes processing the request, and the response is ready, the API Gateway delegates the response back to the client. Non-api request are also routed through the API Gateway.

An API request is distinguished from non-api requests via the “/api/” prefix in the request URI. API requests are bound to JWT based authentication and authorization, while non-api requests are not.

The request URI contains the name of the targeted service. The service name is the second element of the request URI in case of API request (e.g. `/api/admin/`), while it is the first element of non-api requests (e.g. `/admin/`). Individual microservices publish their service name to the service discovery infrastructure.

The API Gateway uses the service name specified in the client-side request to look up the appropriate microservice from the service discovery infrastructure. Since more than one microservice can be registered in the service discovery infrastructure to handle the same request, the API Gateway acts as a very simple load balancer for the LGAP backend.

The API Gateway microservice implements the Circuit Breaker design pattern. The purpose of a circuit breaker is to monitor calls to critical resources, and if the call fails, remove the critical resource from the system.

A circuit breaker in electronics has two basic states, open or closed. If the circuit is closed, the requests flow to the monitored resource. If the resource fails, or does not respond within
a configured threshold, the circuit becomes open and all future requests to the resource are going to fail immediately.

The Circuit Breaker design pattern defines a third, semi-open state. After a given threshold following the opening of the circuit, the circuit breaker becomes semi-open. This means that the next request is allowed to the critical resource, and if it is completed successfully, the circuit becomes closed again. However, if the request fails, the circuit becomes open.

Vert.x provides an out-of-the-box implementation of the Circuit Breaker design pattern, which is used by this microservice.

The API Gateway microservice differs from the rest of the microservices in the LGAP system. It does not have a data model or repository components. However, it provides an authentication service for the rest of the LGAB backend.

The APIGatewayJWTAuthService declared in the common module is implemented by the edu.lenzing.lgap.microservice.apigateway.service.impl.APIGatewayJWTAuthServiceImpl class. This service is published to the rest of the LGAP backend as an event bus service.

Components performing authentication can acquire the JWT based authentication and authorization token. This token is required by the API Gateway to access the API provided by the LGAP backend.

The API Gateway adds the “circuit.breaker” configuration element to its configuration file. This configuration element is used to initialize the circuit breaker. The content of this configuration element is the following:

- “name” (java.lang.String): the name of the circuit breaker instance.
- “max.failures” (java.lang.Integer): maximum number of allowed failures before opening the circuit breaker.
- “timeout” (java.lang.Long): timeout in milliseconds before a non-responding call is considered to be failed.
- “reset.timeout” (java.lang.Long): timeout in milliseconds before switching the state of the circuit breaker to semi-open.
- “fallback.on.failure” (java.lang.Boolean): switch indicating whether the fallback function (if specified) should be executed upon failure.

The edu.lenzing.lgap.microservice.apigateway.APIGatewayVerticle is the entry point of this microservice. The APIGatewayVerticle extends the BaseHttpEndpointVerticle, and it is responsible for registering the appropriate routes and route handlers, creating the circuit breaker and registering the APIGatewayJWTAuthService. This Verticle also acts as the mediator between client-side applications and the rest of the LGAP Backend.

The API Gateway microservice is a single point of failure for the LGAP system. Future updates should keep this microservice module as simple as possible to avoid creating failure options.
4. Frontend

The client-side web applications provided by the LGAP system (fronted) follow the single-page application (SPA) principle and the mobile-first design pattern.

The LGAP frontend applications are based on the open source Vue.js framework [30], developed by Google engineer Evan You, and the Bootstrap web framework [31] developed at Twitter.

Vue.js is a lightweight, component based, reactive, progressively adaptable JavaScript framework for developing interactive web user interfaces as well as complete web applications. Vue.js provides the core framework for the reactive single-page application, while Bootstrap adds all the necessary elements to make the SPA responsive and mobile-first. Several other client-side JavaScript frameworks were also considered, such as Angular.js or React.js, but Vue.js outperforms both in terms of size, speed and simplicity. The Vue.js and Bootstrap duo was chosen as the base framework for the LGAP frontend for several reasons. The key points are the following:

- Supports responsive and reactive web design;
- Supports the single-page application and model-view-controller principles;
- Supports the decoupling of the application into components;
- Supports dynamic component loading and reloading;
- Supports routing within the web application;
- Supports ECMA 5 compliant web browsers;
- Supports HTML based templates;
- Supports the virtual DOM;
- Supports seamless integration with JSON based web API;
- Supports integration with a wide variety of JavaScript frameworks;
- Supports token based authentication and authorization.

Vue.js is a progressive framework, which means that it is possible to integrate it into web applications that are based on different frameworks. It is not mandatory to use the Vue.js framework on its own. However, the LGAP frontend systems are based entirely on Vue.js and its extensions.

Vue.js implements the virtual Document Object Model (DOM), popularized by the React.js framework developed by Facebook. The virtual DOM is the in-memory representation of the component tree used to render the web pages. Maintaining a virtual copy of the DOM increases the memory footprint of the application, however it provides several significant benefits. The most resource intensive part of loading a website is rendering the web pages. The virtual DOM is the core mechanism behind the dynamic component loading and reloading provided by Vue.js. Vue.js always updates the virtual DOM first, and then applies the changes to the actual DOM if, and only if the virtual DOM effectively changed. This means that only those parts of the website are re-rendered that need to change. This behavior is used by the LGAP frontend applications to implement the SPA principle. The web applications are loaded only once from the server. Every update to the rendered content is realized by re-rendering the specific parts of the web application that are affected by the update. Another benefit of the virtual DOM is that it allows Vue.js to create a two-way
binding between the data model and the visual representation of the data. Updating the data model will automatically update the visual representation of the data, while updating the visual representation (e.g. via an input field) will automatically update the data model. Vue.js provides one and two-way data binding via directives.

Vue.js at its core is based on the model-view-controller (MVC) principle. Every Vue instance or component can define its own data model and provides the means for creating views using a HTML based template syntax. Vue templates are valid HTML with added custom directives defined by Vue. Directives are custom HTML attributes beginning with the v- prefix. After Vue renders a view based on a template, the directives are removed and the content remains pure HTML.

A common pattern with microservice based systems is that individual microservices provide the user interface fragments along with their services. The various UI fragments are assembled in a common layer. The LGAP frontend systems use exclusively the client-side rendering capabilities of Vue.js, however, Vue.js also support server-side rendering.

A Vue application can be composed of several components. There are several ways to create Vue components, however the LGAP frontend systems use exclusively the single file component approach. Every component is defined and implemented in a separate file with .vue extension. The content of these files can be divided into three parts:

1. The view template for defining the view of the component.
2. The Vue specific JavaScript part for implementing the component logic.
3. Optional CSS styles.

The view template is defined within the <template/> tag. The template of a component can only contain one single root element, and must be valid HTML. Component logic is enclosed within the <script/> tag. Components can import other components or JavaScript modules, and they can export functionality. If a component defines a template, the visual representation of a component can be embedded into other components. Single file components can define CSS styles within the <style/> tag. Component exclusive styles can be defined by adding the scoped attribute to the style tag.

Vue.js makes it possible to integrate arbitrary client-side JavaScript frameworks, however, it also defines its own extension mechanism. By installing and loading these extensions, they integrate into the Vue API, therefore they can be used through the Vue instance. The Vue instance is available from every component. The LGAP frontend applications are using three such extensions:

1. The vue-router extension;
2. The vue-resource extensions;
3. The vuex extension.

The vue-router extension [32] provides routing within the application. Binding components to routes makes it possible to define component hierarchies and dynamically change the displayed content without reloading the whole page. The vue-resource extension [33] is a simple HTTP client. It is used provide a JavaScript API for backend services. The vuex extension [34] provides a centralized state management for the Vue application.
The Bootstrap framework focuses on the view layer of web applications. It provides a grid based layout management, and custom CSS classes for responsive user interface design. The responsive design of LGAP frontend applications are based on the layout management and row/column classes provided by Bootstrap.

4.1. Single Page Applications

The core API and extensions of Vue.js provides a solid base for single-page web application development. Equipped with the Bootstrap framework, the mobile-first and responsive requirements are also met for the development of LGAP frontend applications.

Both frontend applications provided by the LGAP system follow the same architecture model and design principles.

4.1.1. Architecture

The architecture of the LGAP single-page applications is based on the component oriented architecture model. The components are either Vue components or JavaScript modules. The general architecture of a SPA is illustrated by Figure 16.

![Figure 16. Architecture blueprint of a single-page application](image)

The SPA consists of one single HTML page, the _index.html_. The Vue instance binds to an element in this page, and renders the rest of the content at the location of this element. The root Vue component is called App. There is one layer between the index.html page and the App component. The common layer is a JavaScript module. This module is responsible for loading the necessary Vue extensions, creating the Vue instance and performing any additional configuration. The render function of the Vue instance is pointed to the App
component. The *App component* defines the main layout of the SPA. The main layout consists of a header, the dynamic content and an optional footer. The dynamic content is represented by a *router view* element. This element is identified by the vue-router extension, and it is used to render dynamic content at its location, based on the current route.

The *Header component* of the SPA contains the available navigation options. It is implemented as a Vue component in a form of a horizontal menu bar, with the navigation options taking place on the right side. In case of a smaller screen side, the navigation options are available in the form of a collapsible menu. The navigation options contain router links. The router links are used to specify the target of the navigation. When an element containing a router link is clicked or otherwise triggered, the vue-router navigates to the selected page by reloading the content of the router view. Router links do not explicitly declare the route associated with the content they redirect to. Instead, every route has a name that can be used to decouple the route information from the UI elements triggering the router.

The *Router Module* is implemented as a JavaScript module in the *modules/router.js* file. It is responsible for loading and configuring the vue-router extension and for protecting routes that require authentication. The routes and the mapping of components to specific routes are created and configured in this module. All the components that can be accessed via different routes are known imported by this module. Therefore, this module is a centralized place for every route specific configuration. The router module exports the configured vue-router instance.

The *API Module* implements an API access layer for the SPA in the *modules/api.js* file. This module is responsible for loading and configuring the vue-resource extension. The main purpose of this module is to decouple the API service providers from the API service consumers. This is achieved by wrapping API services into vue-resource specific resources. A resource, as defined by the vue-resource extension, contains the URI and the HTTP method of the API service. This information is associated with JavaScript functions, providing a JavaScript API for the various backend services. The location of the LGAP API Gateway is configured in this module, therefore this module acts as a centralized location for API related configuration. The various resources defined in the frame of vue-resource are wrapped into a JavaScript object. This object is pushed into the prototype of the Vue, therefore the API defined by vue-resource is available for every component. The vue-resource extension provides the means to intercept outgoing HTTP requests. This mechanism is used to inject the API token into every API call.

The *Store Module* defines a central data storage for shared application state in the *modules/store.js* file. This module defines the shared state and provides getter and setter functions for manipulating the state. Updating the state is achieved in a commit like fashion, therefore the shared data remains consistent.

Vue.js provides *filters* for formatting or modifying values embedded into the template of components. Filters are JavaScript functions that can be used with a special syntax in the Vue templates. Several filters can be chained together to achieve the final representation of the data. Filters are available through the Vue instance, and are used to provide commonly used formatting and display options. The custom filters are defined in the *Filter Module* in the *modules/filters.js* file. This module registers the filters in the Vue instance, therefore components can use the defined filters without importing them.
4.1.2. Build and Deployment

The frontend applications provided by the LGAP system are built and packaged with Webpack [35]. Webpack is also responsible for dependency management. It maintains the list of installed dependencies with their respective version numbers, and fetches them from a central repository. During the build process, Webpack takes all the different JavaScript modules and Vue components defined in the system, and bundles them into a single JavaScript file. This JavaScript file is referenced in the index.html page.

The LGAP frontend applications use the latest syntax and features defined by the ECMAScript 6. Not all these features are supported by every browser. Before building the frontend systems, the components are processed by the Babel.js compiler [36]. Babel is a JavaScript to JavaScript compiler. It translates the ECMAScript 6 code to a more widely supported ECMAScript 5 code.

The LGAP frontend application projects are generated by the Vue-CLI tool, using the webpack-simple template.

The final content bundled by Webpack is deployed to the microservice responsible for delivering the SPA to the client-side.

4.2. Customer Single Page Application

The Customer SPA is responsible for providing all the functionality for customer. This includes displaying information about auctions as well as providing the means for the customers to interact with the auctions. Besides the responsive and mobile-first requirements, the Customer SPA aims to remain as small as possible regarding the size of the web application.

The Customer SPA is divided into two main pages, the overview page and the auction page. Besides these two content pages, the Customer SPA provides a separate login page for authenticating the user.

4.2.1. Auction Overview Page

The overview page is accessible from the header menu, and it provides general information about the ongoing, upcoming and past auctions in the customer’s region(s). This page provides the means for searching and filtering auctions. By selecting an auction in the overview page, the customer is redirected to the auction page.

The root component of the overview page is the components/overview/OverviewMain.vue component. This component is responsible for loading the auctions from the backend. The content of this component is divided into two main parts:

1. Searching and filtering auctions;
2. Displaying the loaded auctions.

The auction searching and filtering functionality is implemented in the components/overview/OverviewFilter.vue component. Displaying the loaded auctions is managed by the components/overview/OverviewGroup.vue component by wrapping every auction into a components/overview/OverviewGroupItem.vue component.
4.2.2. Auction Page

The auction page provides all the information about a selected auction, but it is not directly reachable from the header menu. The auction page is divided into four main sections following the single responsibility principle. The four main sections are the following:

1. Product details;
2. Auction details;
3. Auction state;

Every section is implemented by a Vue component. Sections share certain common parts defined by the `components/auction/BaseComponent.vue` component. Components implementing a section embed their content into the default slot defined by the `BaseComponent`. The product details section is auction model independent. However, auction models supported by the system must implement the remaining three sections. The Customer SPA provides Dutch auction specific implementations for these sections. Furthermore, general implementations for the auction state and bidding sections are provided for auction phases where these sections are not available.

The root component of the auction page is the `components/auction/AuctionMain.vue` component. This component is responsible for loading the data corresponding to the selected auction from the backend. As the requested data is backend loaded from the backend, the appropriate section implementations are dynamically loaded to handle the data.

The product details component is implemented in the `components/auction/product/ProductDetails.vue` component. This component provides information about the product being auctioned.

The Dutch auction model specific auction details component is implemented by the `components/auction/details/DutchAuctionDetails.vue` component, and it provides general information about the selected auction.

The content of the product and auction details components is static, meaning that it is not updated by the Customer SPA during the auction process. The content of these components is available for upcoming, ongoing and past auctions.

The Dutch auction model specific auction state component is implemented by the `components/auction/state/DutchAuctionState.vue` component, and it provides information about the current state of the auction. This content is live, meaning that it is constantly updated to reflect the state of the auction process.

The final, bidding component is the only part of the auction page where the customer can interact with the system by placing a bid. The Dutch auction model specific bidding component is implemented by the `components/auction/bidding/DutchAuctionBidding.vue` component.

4.2.3. Login Page

The login page is implemented in the `components/login/LoginMain.vue` component. This page is the only page that is not protected by the router, meaning that it is reachable for
unauthenticated users. When the Customer SPA is loaded by an unauthenticated user, the user is automatically redirected to this page. The main content of the Customer SPA cannot be loaded until the user is not authenticated.

4.3. Administrator Single Page Application

The Administrator SPA offers the administration related functionality of the LGAP system. This is where products can be added to the system and auctions can be created, started or canceled.

The main idea behind completely separating the Customer and Administrator SPAs is to allow the two to grow independently. The Administrator SPA can implement all the overview and maintenance functionality of the system that might be required later, without causing downtime in the Customer SPA. Meanwhile, the Customer SPA can remain lightweight and focused on serving the needs of the customers.

The Administrator SPA is designed and implemented in the same fashion as introduce in the section 4.2. The content is divided into several main pages, each page containing its own hierarchy of components. The most important pages are described below.

4.3.1. Product Overview Page

The product overview page provides information about the products currently available in the system. New products can be created here, or existing products can be updated.

New auctions are also created from this page, by selecting a product. This behavior was introduced in order to simplify the auction creation process by preselecting the product, and to avoid having the same product in the system multiple times. In order to start an auction, the administrator must find the required product first, or create it if it is not yet present in the system.

The root component of the product overview page is implemented in the components/product/ProductMain.vue component. This component is responsible for loading the available products from the backend. It also provides a control panel for searching and creating products.

The control panel is implemented by the components/product/ProductControls.vue component. Loaded products are wrapped into a components/product/ProductDetails.vue component. This component is responsible for displaying the product data, and for redirecting to the auction creation page with the selected product.

4.3.2. Auction Overview Page

The auction overview page is similar to the one in the Customer SPA introduced in section 4.2.1. It provides an auction model independent general overview of auctions available in the system. However, the auctions displayed here are not filtered by region and are displayed in more details for the administrators.

This page provides advanced search and filtering options for finding auctions in the system. The functionality to cancel an auction is also provided in this page.
The root component of this page is the `components/auction/AuctionMain.vue` component. The implementation is very similar to the one described in section 4.2.1. This component is responsible for loading auction data from the backend.

The `components/auction/filter/AuctionFilter.vue` component implements the advanced search and filter functionality. The auctions loaded from the backend are wrapped into a `components/auction/AuctionDetails.vue` component, which is responsible for displaying the auction data.

### 4.3.3. Auction Creation Page

The auction creation page is divided into two sections. The first section provides input fields for the general, auction model independent properties of the auction. The form for the second part is dynamically loaded based on the selected auction model.

The root component of this page is the `components/auction/new/NewAuction.vue` component. This component implements the form for the general auction data, and loads two other components implementing a form.

The first form is used to collect data about the cargo associated with the auction and it is implemented by the `components/auction/new/NewCargo.vue` component. This component also displays the preselected product.

Based on the selected auction model, the component implementing the second, auction model specific form is loaded dynamically. The Administrator SPA provides the Dutch auction model specific form in the `components/auction/new/NewDutchAuction.vue` component. To support additional auction models, the auction model specific form should be added here.

### 4.3.4. Login Page

The login page is implemented in the `components/login/LoginMain.vue` component, and it is identical in its functionality to the component described in section 4.2.3.
5. User Guide

This chapter provides an illustrated overview of the client-side web applications. The illustrations are snapshots from the LGAP system.

5.1. Customer Application

The first screen the user encounters upon loading the Customer SPA is the login screen, illustrated by Figure 17. If the user enters correct authentication data, the next screen is loaded. Otherwise the appropriate error message is displayed.

![Figure 17. Customer SPA, login screen]

Following the successful authentication, the menu bar in the top-right corner is loaded, and the user is redirected to the auction overview screen, illustrated by Figure 18, on a computer, and by Figure 21, on a mobile device. This screen contains the search options and the loaded auctions separated into three categories. In order to navigate away, the user can select an auction or log out.

![Figure 18. Customer SPA, auction overview screen]
Selecting an auction will load the auction screen. The data displayed in the auction screen depends on the phase of the selected auction. An ended auction screen is illustrated by Figure 19, on a computer, and by Figure 22, on a mobile device. An ongoing auction screen is illustrated by Figure 20. The user can navigate back to the previous screen using the menu bar or the back button of the browser. The user can also log out any time using the menu bar.

![Figure 19. Customer SPA, auction screen with ended auction](image1)

![Figure 20. Customer SPA, auction screen with ongoing auction](image2)
Figure 21. Customer SPA, auction overview screen (mobile)

Figure 22. Customer SPA, auction screen with ended auction (mobile)
5.2. Administrator Application

The first screen of the Administrator SPA is also the login screen, illustrated by Figure 23. Upon successful authentication, the menu bar and the home screen is loaded. The home screen is a simple landing page, therefore navigation options can be chosen from the menu bar.

![Figure 23. Administrator SPA, login screen](image)

By selecting the Product option from the menu bar, the product overview screen is loaded. This screen is illustrated by Figure 24. The user can filter products based on name and description; create new or update existing products using the provided buttons; or start a new auction based on a selected product. Selecting the New auction button will load the auction creation screen.

![Figure 24. Administrator SPA, product overview screen](image)
The auction creation page, illustrated by Figure 25, is essentially a form that has to be filled by the user in order to create a new auction. Once the form is filled, creating the auction is a matter of submitting the form.

Once the form is submitted, the user is redirected back to the product overview screen.

By selecting the Auction option from the menu bar, the auction overview screen is loaded. The auction overview screen is illustrated by Figure 26. This screen provides advanced search options for finding and filtering the loaded auctions. Every upcoming or ongoing auction displays a button for canceling the auction.
Figure 26. Administrator SPA, auction overview screen
Conclusion and Further Development

The design and development of an online software system with the goal of selling fiber products via auctions was successfully realized in the frame of the LGAP project.

The surveyed auction models provide a good basis for Lenzing to further develop the idea of the online, auction based sales approach.

The microservice architecture model on the server-side along with the mobile-first single-page web applications on the client-side proved to be a good combination for designing a modern software system and for meeting the requirements set by Lenzing. The project was presented to the company on the 16th of June 2017 via a WebEx conference for final approval.

There are several options for further developing the LGAP system:

- Common functionality like logging and configuration can be extracted to separate microservices. A logging microservice would provide a centralized location for monitoring log files and system events. A configuration microservice would allow the system to be centrally configurable. Some of the configuration details could be retrieved upon starting a microservice, therefore changing the configuration would not require the redeployment of the affected microservices.
- The LGAP system is prepared to support various auction models by the early architectural choices. Supporting new auction models could result in valuable feedback from customers about the system and the idea of online auctions.
- Furthermore, the scaling of individual microservices could be made automatic based on load or various other factors.
References


[14] E. Gamma, R. Helm, R. Johnson and J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1994.


Curriculum Vitae

First Name: Csaba
Last Name: Ilonka
Birthdate: February 17, 1993
Phone Number: +40 747 261464
Email Address: icsaba.zilik@gmail.com

Work Experience

  Software Developer, Softech SRL, Cluj-Napoca, Romania

Jul 2014 – Sep 2014
  Intern, Codespring SRL, Cluj-Napoca, Romania

Jul 2013 – Sep 2013
  Intern, Nokia Solutions and Networks KFT, Budapest, Hungary

Education

2016 – present
  Johannes Kepler University, Linz – M.Sc.

2015 – present
  Babeș-Bolyai University, Cluj-Napoca – M.Sc. (Enterprise Software Design and Dev.)

2012 – 2015
  Babeș-Bolyai University, Cluj-Napoca – B.Sc. (Computer Science)
STATUTORY DECLARATION

I hereby declare that the thesis submitted is my own unaided work, that I have not used other than the sources indicated, and that all direct and indirect sources are acknowledged as references.

This printed thesis is identical with the electronic version submitted.

Place, Date

Signature