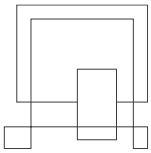


Project Report

Airline Reservation System

Distributed and Heterogeneous System

Names of Students:

Jan Vasilcenko – 293098

Karrtiigehyen Veerappa - 293076

Nicolas Popal - 279190

Patrik Horny – 293112

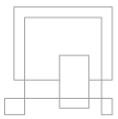
Supervisors:

Jakob Knop Rasmussen

Ole Ildsgaard Hougaard

VIA University College

Bring ideas to life
VIA University College



Number of characters: 81655 including spaces

Software Technology Engineering

Semester 3

09 December 2020

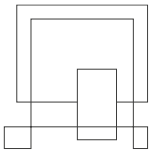
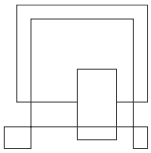
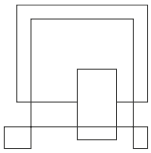
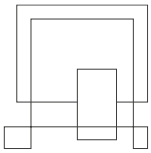


Table of Contents

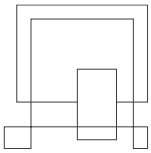






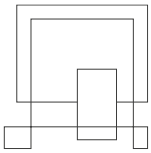
List of Diagrams

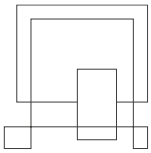
List of Tables



List of Figures

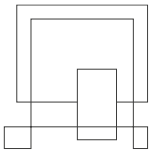
List of Code Snippets





Abstract

Aviation is an important industry as it allows people to connect in a matter of hours when it took days in the old days, and software are made where people can buy flight tickets with ease. This project is concerning the production of a distributed and heterogeneous airline reservation system. This paper shows how this distributed system is constructed using a three-tier architecture, along with the usage of design patterns and SOLID principles. Different types of testing are used for testing the requirements, where the results are discussed on how to fix or improve the quality of the system.



1 **Introduction** (Karrtiigehyen and Nicolas Popal)

1.1 **Background Description** (Karrtiigehyen)

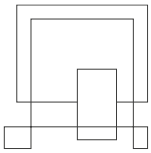
Aviation has allowed mankind to visit other countries and continents in a matter of hours. Once, aviation was only available to people of power and money, but nowadays, commercial airlines provide flights to the public. Not only that, but airlines also provide extra services such as hotels, car rentals and insurance to the public as well. The real insurgence of aviation came when people were able to buy flight tickets online from the comfort of their home. This not only benefited the people buying the flight tickets, but also for the airlines since there is no need for travel agents (Altexsoft, 2019).

Due to the Covid-19, the aviation industry has seen unprecedented drop in travels nowadays (ICAO, 2020). Plus, it does not help that people find that online flight reservation systems are quite arduous and lengthy processes relative to the modern age (Blessing, Umar and Opeyemi A., 2017). When the travel embargos do eventual open worldwide, people will want to travel. This means that the market where customers can book flight tickets using an intuitive reservation system with ease and time-efficiently is ripe for the taking.

Not only that, but the management of the reservation system is often a complicated software to control (Aviation Job Search, 2012). So, improving the management side of the reservation system to be more efficient and easier would be a good idea since the operators managing the reservation system can make less mistakes with the flight managements.

1.2 **Definition of Purpose** (Karrtiigehyen and Nicolas Popal)

The purpose of the airline reservation system is to help customers book flight tickets and operators to manage the flight booking system with ease and efficiently.

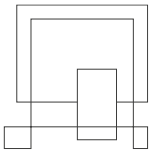


1.3 **Delimitations** (Karrtiigehyen and Nicolas Popal)

There are some delimitations set to define the scope of the project, so that the project can be more focused and refined.

1. Transactions with money will not be included.
2. Round-trip flights will not be considered, just one-way flights.
3. Refunding services will not be included.
4. Anything regarding insurance will not be accounted for.
5. Any additional services beyond the flight, for example cars and hotels at the destination, will not be included.
6. Rebooking services will not be included.

The methodology used, time schedule, and risk assessment can be found in the Project Description, which can be found in Appendix A.



2 **Analysis** (Everybody)

In the analysis, the problem domain will be further defined by elaborating upon the contents from the previous section. Firstly, the requirements will be explored in the shape of user stories. Then, use case diagram and descriptions are presented to show the different actors and scenarios involved, from which System Sequence Diagrams can be erected. A section will also be devoted to security, where a threat model and a risk assessment table will be made. The conclusion of this section will be with the presentation of the domain model of the system, which will be used as the groundwork for the 3rd section, Design.

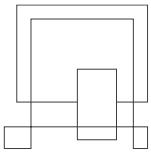
2.1 **Requirements** (Everybody)

SMART principles were used as the framework for the requirements, so that the requirements are testable.

2.1.1 **Functional Requirements** (Everybody)

Critical priority

1. As a customer, I want to book flights, so that I can fly to my destination.
2. As a customer, I want reserve seats on flights, so that I can sit on the favoured seat in the flight.
3. As a customer, I want to view available flights, so that I can book my preferred flights.
4. As a customer, I want to choose the date of the flights, so that I can reach my destination in my desired date.
5. As a customer, I want to be notified when my booked flights get delayed or cancelled, so that I will be aware of it.
6. As a customer, I want to be able to view the status of my booked flights, so that I will know the status of my booked flights.
7. As a customer, I want to register myself as a customer, so that I can use the airline reservation system.



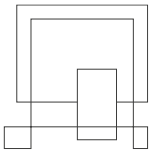
8. As a customer, I want to be able to log in as a customer, so that I can access the airline reservation system.
9. As a customer, I want to view my booked flights, so that I know which flights I have booked.
10. As a customer, I want to cancel a booked flight, so that the flight is not booked to me anymore.
11. As a customer, I want to get a code from booked flights, so that I can use that information to check in at the airport.
12. As an operator, I want to create flights, so that the customers can view them.
13. As an operator, I want to cancel flights, so that the cancelled flights will be removed and customers who have booked the cancelled flights will be notified.
14. As an operator, I want to log in as an operator, so that I can access the airline reservation system.

High priority

15. As a customer, I want to search for specific flights by their destination, so that I can narrow my flight search.
16. As a customer, I want to view the details of the flights such as the destination, time taken and date, so that I know the details of the flight I am about to book.
17. As a customer, I want the option to buy additional luggage, so that I can bring more luggage to the flight.
18. As an operator, I want to delay flights, so that customers will be able to check whether their flights are delayed.

Low priority

19. As a customer, I want to edit my user info, so that it is updated.
20. As a customer, I want to view my name when I log in, so that I can identify if I have logged into my account.
21. As a customer, I want to view the previous flights that I have taken, so that I can look up which flights I have taken.



2.1.2 **Non-Functional Requirements** (Everybody)

1. The airline reservation system should be a heterogeneous system, using Java and C#.
2. The airline reservation system should be a distributed system.
3. The airline reservation system should use sockets and web services.
4. Customers cannot modify the flight information.
5. Customers cannot view other users' information.
6. The system must be usability tested by end users.
7. The airline reservation system should work on separate computers over the internet.
8. There will only one type of seat offered by the system to customers.

2.2 **Use Case Diagram** (Everybody)

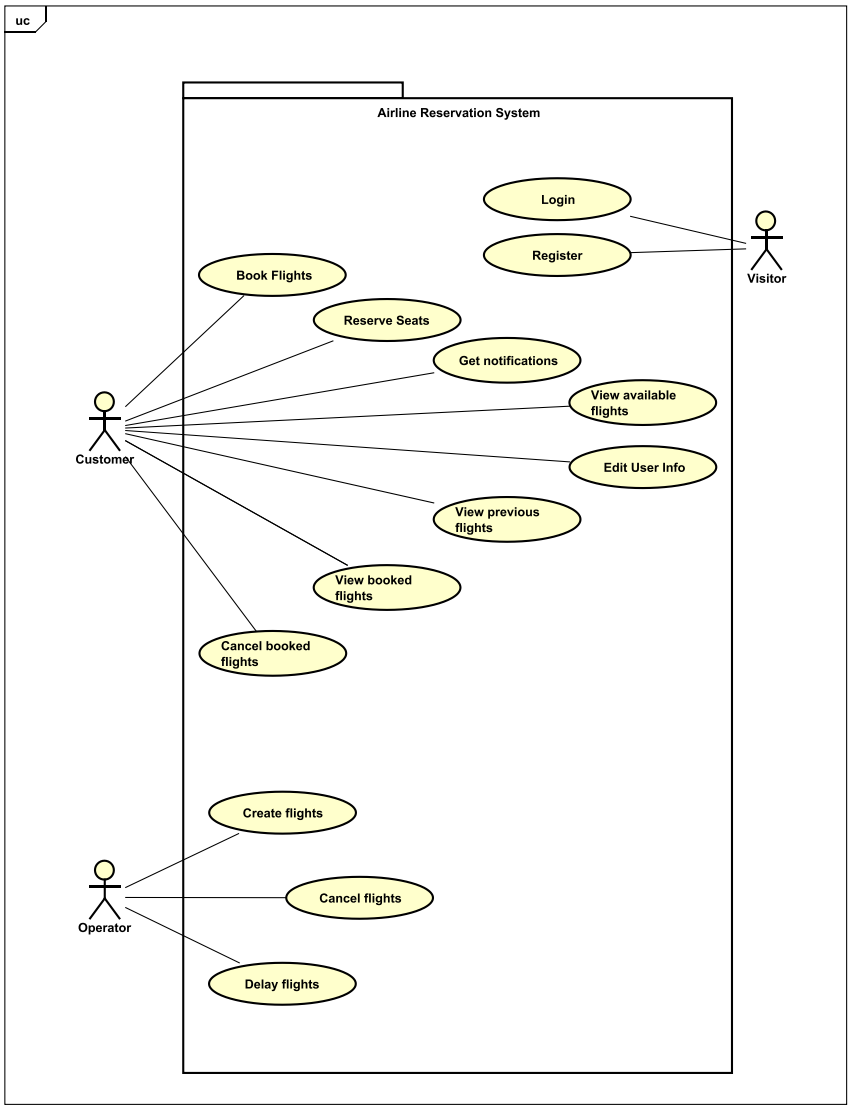
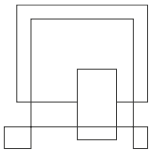


Diagram 1: Use case diagram of airline reservation system

As seen from Diagram 1, there are three actors – *Visitor, Customer* and *Operator*. When a user has not performed login or register, the actor is then a *Visitor.* Once the *Visitor* has logged in or registered, the actor is then either *Customer* or an *Operator* based on their account type.

Customer can book flights, which are not fully booked. While booking flights, *Customer* can reserve their seat position, provided that the chosen seat is not already reserved by



another *Customer*. The *Customer* can view available flights to book and view already booked flights. The *Customer* can also cancel flights that are booked, and edit their account information. The *Customer* will receive notifications if any of their booked flights are delayed or cancelled.

The use cases of *Operator* are create flights, cancel flights, and delay flights.

2.3 **Use Case Descriptions** (Everybody)

The use case descriptions show how each scenario will play out. The use case descriptions are based on the use case diagram that was just shown.

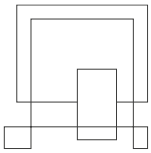
The use case descriptions below are for Create Flights and View Available Flights. These two cases will serve as the main examples throughout this report and will be referred to frequently. These use cases are chosen because the view available flights use case is from the *Customer's* point of view, and the create flights use case is from the *Operator's* point of view.

Rest of the use case descriptions can be found in Appendix B.

Create Flights

Use case	Create flights
Summary	The operator creates a new flight
Actor	Operator
Precondition	Operator must have all details about new flight
Postcondition	The flight is created
Main scenarios	1. The operator chooses to create new flight 2. The operator needs to insert information about flight 3. The system validates data of the new flight 4. The flight is created and stored in the database
Alternative scenarios	2a The flight information is invalid 1. System informs operator, that information is invalid

Table 1: Use case description for Create Flights



View Available Flights

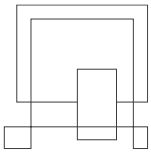
Use case	View available flights
Summary	View flights available for customer
Actor	Customer
Precondition	Flights must be added by operator into the system and dates of the flights are in the future
Postcondition	Flights are viewed to the customer
Main scenario	1. Customer inputs departure from 2. Customer inputs flight destination 3. Customer inputs departure date 4. System shows a list of all available flights with those constraints
Alternate Scenario	At any time during step 1 customer cancels 1. The use case ends If there are no data stored 1. Continue from Step 1 4a If the system cannot find item with specific attribute 1. System will show an error message.

Table 2: Use case description for View Available Flights

2.4 **System Sequence Diagram** (Nicolas Popal and Patrik Horny)

The examples shown for the System Sequence Diagrams are for create flights and view available flights. All the system sequence diagrams can be found in Appendix C.

System sequence diagram (SSD) is based on use case description. In SSD the system is treated as a black box, which means that the inner workings of the system are not displayed or known. For an example, the system sequence diagrams used for creating



and viewing flights in the airline reservation system shows the input and output events of the system.

Create Flights

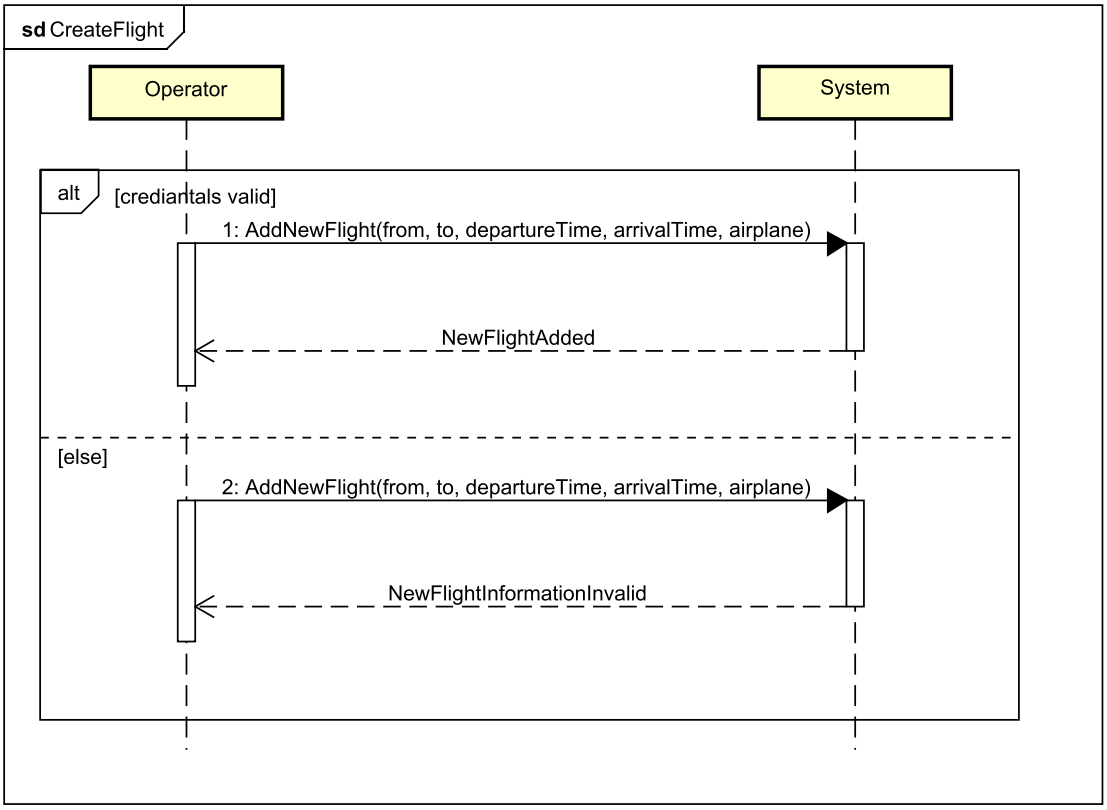
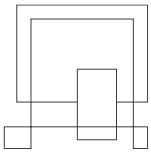


Diagram 2: System sequence diagram for creating flights

This system sequence diagram is from the *Operator's* point of view. There are two lifelines, *Operator* and *System.*

If the credentials of the *Operator* are correct, then the *Operator* can create flights with parameters origin airport, destination airport, departure and arrival times, dates, and the airplane. Else, the *Operator* would be informed that a new flight cannot be created.



View Available Flights

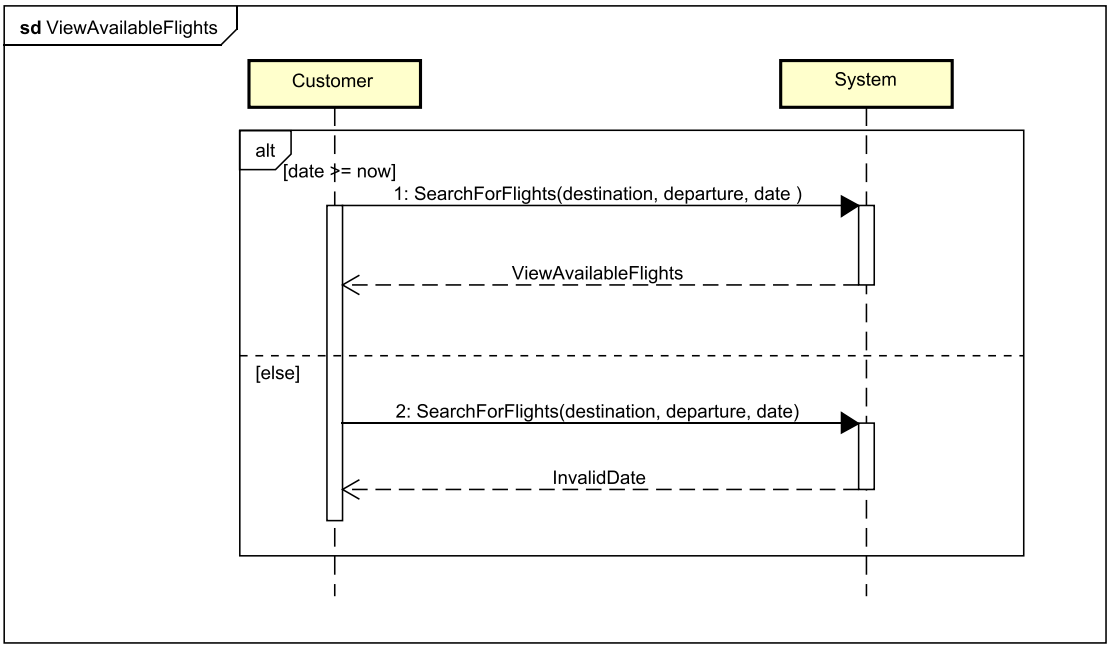


Diagram 3: System sequence diagram for viewing available flights

In this sequence diagram, the actor is *Customer,* and the lifelines are *Customer* and *System.*

The *Customer* can search for flights with the arguments *destination, departure* and *date.* If the date inputted by the *Customer* is bigger than or equal to the current date, the available flights are returned to the *Customer*. If the date is in the past, an error will occur.

2.5 **Security** (Karrtiigehyen and Nicolas Popal)

In this subsection, potential threats to the airline reservation system will be discussed. A threat model for the airline reservation system will be made, using the STRIDE methodology to categorize attacks. The information garnered from the threat model can be summarized in a risk assessment table, which will conclude this subsection.

Security mechanisms that can be constructed to prevent the threats discussed in this subsection can be found in the Design section.

2.5.1 **Threat Model** (Karrtiigehyen and Nicolas Popal)

Only one threat will be discussed in this subsubsection. The rest of the threats will be included in the risk assessment table, which can be found in the next subsubsection.

The potential threats are categorized by using the STRIDE methodology. One example will be shown here on how the threats are analyzed.

In the airline reservation system, there are two types of users (actors): *Customer* and *Operator*. *Operator* has special privileges that the *Customer* does not possess such as creating, delaying, and cancelling flights (for more details, check the use case diagram and descriptions in subsections 2.2 and 2.3). So, it is important that the *Customer* is not given the privileges that the *Operator* possesses. But this can be overcome by using an SQL injection. To showcase this attack, a data flow diagram for a *Customer* performing login is shown below.

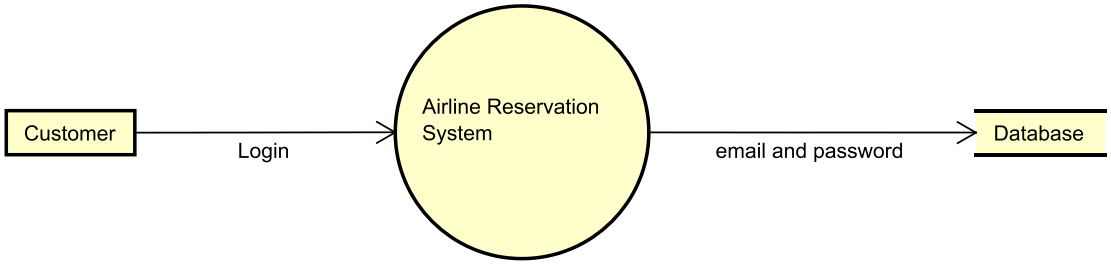


Diagram 4: Data flow diagram for login

While the diagram above is not an accurate representation of the architecture of the airline reservation system, it does provide a visualization of the data flow for the login operation.

The *Customer* (threat agent) performs login with their email and password, and it is checked whether the inputted credentials match in the database. However, an SQL injection can be performed in order to elevate one's privileges. The data flow diagram below shows this.

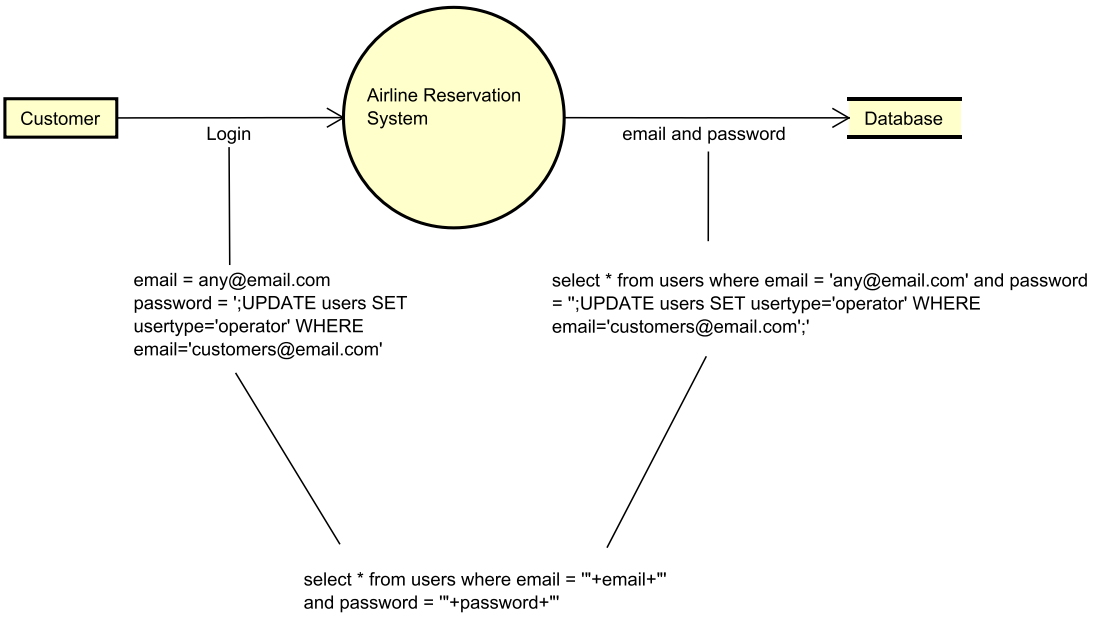
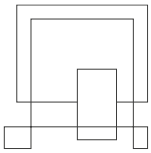


Diagram 5: Data flow diagram for login with SQL Injection

By doing this, the threat agent will now have the privileges of the *Operator.* This can be devastating for a myriad of reasons. Some of them being that the threat agent can now cancel flights, delay flights, and create non existing flights. (This example does make an assumption that the threat agent is aware of the construction and terms used in the database though).

While this example only shows the threat agent elevating their status from *Customer* to *Operator*, the SQL injection can be used for different causes as well such as deleting tables from the database and so on.



This particular SQL injection attack can be categorized as Tampering, Information Disclosure and Elevation of Privilege according to STRIDE. The SQL injection itself is Tampering, but the result of the SQL injection gives the threat agent *Operator* level privileges and access to data that the threat agent should not meddle with, which leads to Information Disclosure and Elevation of Privilege. The type of privilege escalation in this attack is horizontal, meaning that the threat agent desires to escalate their privileges to that of an *Operator.* This type of an attack is an active attack, namely modification of messages.

The security objectives compromised because of this SQL injection are Integrity, Confidentiality, and Authorization. Data integrity is compromised since the SQL injection manipulates data in the database. The threat agent having *Operator* privileges leads to an authorization problem.

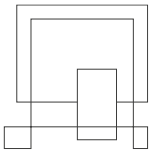
Where the attack is coming from, and by whom is useful to know. For this, the EINOO method is used. SQL injections can be performed by external (*Visitor*) and internal *(Customer* and *Operator)* attackers, but this particular type of SQL injection can only be performed by *Customer*, which means that the threat agent must have already registered into the system, making them an internal attacker. It is an online attack, as opposed to offline or network attacks.

To prevent SQL injections, the query will be performed using a prepared statement separate from the database. This means that the additional query that the threat agent performs will not be executed.

There are other potential threats such as man-in-the-middle attack and so on, but they are all modelled in the same way as the SQL injection example above.

2.5.2 **Risk Assessment** (Karrtiigehyen and Nicolas Popal)

In this subsubsection, a risk assessment table can be created based on the threat modelling from the previous subsubsection. Before making the risk assessment table itself, the meaning of risk and the way it is calculated must be declared.



Risk is calculated by following the equation below:

$$Risk = Incident\ Likelihood + Incident\ Consequence$$

Incident likelihood is how likely a potential threat is to happen, described by the following equation (the plus sign is more of a metaphorical plus):

$$Incident\ Likelihood = Threat\ frequency + Preventive\ measures$$

The threat frequency is how often the threat can occur. The preventive measures are what measure can be taken to prevent the threat. The threat frequency cannot be controlled, but the preventive measures can be. This means that the incident likelihood can only be lowered by controlling the preventive measures.

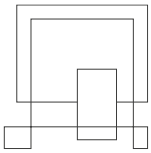
In incident consequence, it is assumed that threat has occurred, and is described by the following equation (the plus sign is more of a metaphorical plus):

$$Incident\ Consequence = Threat\ effect + Corrective\ measures$$

Threat effect is the magnitude of effect that the threat had, and the corrective measures are the measures taken to nullify the threat effect. The way to lower the incident consequence is by controlling the corrective measures, since the threat effect cannot be controlled.

Together, the incident likelihood and incident consequence make up risk. The lower the incident likelihood and incident consequence, the lower the risk will be.

Now that the risk is defined, the information gathered while modelling the threats can be compiled into a risk assessment table below. The *Threat* column represents which of the STRIDE categories are involved. The *Impact* column says what the attack is. *Security Policies Compromised* is regarding which of the security objectives are violated. *Vulnerability* is regarding of which vulnerabilities the attack will be possible. *Asset and Consequences* is about which assets will be attacked and what are the consequences of the attack. *Risk* is discussed above, but it has a rating from Low to High. Finally, the



Counter Measure column represents what counter measures can be taken to circumvent the threats.

Threat	Impact	Security Policies compromised	Vulnerability	Asset and consequences	Risk (Low, Moderate, and High)	Counter measure
Spoofing Indentity	Website spoofing	Authentication	Having an easily forgotten domain name and simplistic UI design that can be copied easily	Website. The emails and passwords of customers and operators can be stolen and used to manipulate data in the airline reservation system while masquerading as the stolen customer or operator	Moderate	Two-factor authentication with the user's email
	Man-in-the-middle attack	Authentication, Confidentiality	Exchanging public keys instead of using certificate authority	Data. The threat agent can listen to data such as account information (email and password), ticket information, and so on	Moderate	Use certificate authority and encryption
	Replay attack	Integrity	Unencrypted	Data. The threat agent can intercept data while imposing as the genuine user	High	TLS (add sequence numbers and MACs to packages, and use nounces)
Tampering	Password cracking	Integrity	No password limits and no captcha during login.	User accounts. Accounts can be stolen	High	Add captcha for login page and advise a stronger password during registration
	URL query strings	Integrity	Sensitive information is displayed in the URL such as ids.	User information can be displayed in the URL and accessed by typing in the URL	Moderate	Encode the sensitive information in the URL
Denial of Service	DDoS attack	Availability	Firewall is not properly configured to prevent DDoS attacks	Website. The website will be unavailable	High	Configure the firewall to prevent DDoS attacks and monitor the firewall
Tampering, Information Disclosure, Elevation of Priveleges	SQL injection	Integrity Confidentiality, Authorization	Preventions for SQL injections were applied	Database. The database can be manipulated	High	Keep the query separate from the database and use prepared statements

Table 3: Risk assessment table

2.6 **Domain Model** (Karrtiigehyen and Nicolas Popal)

The domain model is used as a visual guide for the problem domain. The development of the domain model was influenced by the user stories from the Requirements subsection.

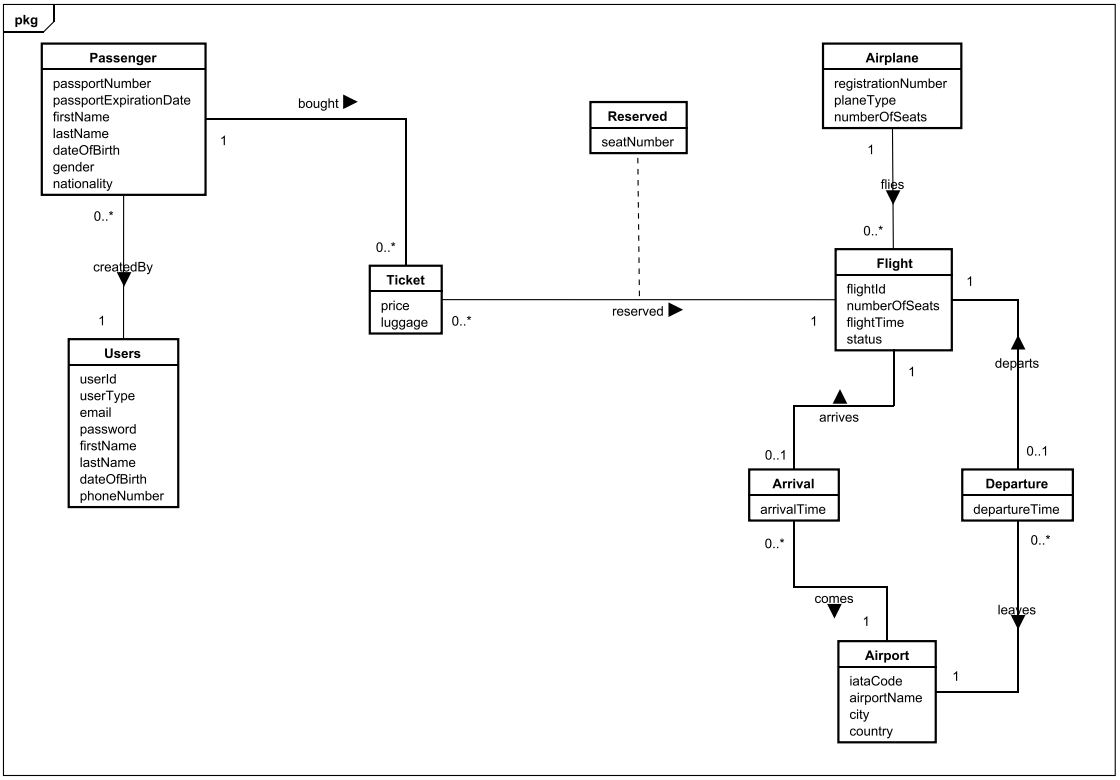
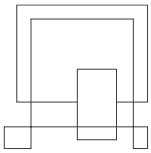


Diagram 6: Domain model

For more explanation on the domain model, please refer to subsubsection 3.3.1 since the conceptual model of the database is explained there, and the conceptual model of the database is heavily influenced by the domain model.



3 **Design** (Everybody)

In this section, the software will be designed more in-depth. The topics that will be discussed are the architecture of the system, design patterns, technologies used, and UI design choices. A section discussing security mechanisms is also included. The outcome of this section will contain the necessary knowledge and diagrams that will be used for the implementation of the system.

3.1 **Architecture** (Karrtiigehyen and Nicolas Popal)

The architecture chosen for the airline reservation system is the three-tier architecture. The three-tier architecture is a client-server architecture pattern where the presentation, application and data tiers are separated. This has many benefits. For example, there is a separation of concerns. This reduces coupling and dependencies, while improving cohesion and increasing reusability of code. Another advantage to using the three-tier architecture is that any tier can be replaced or modified independently.

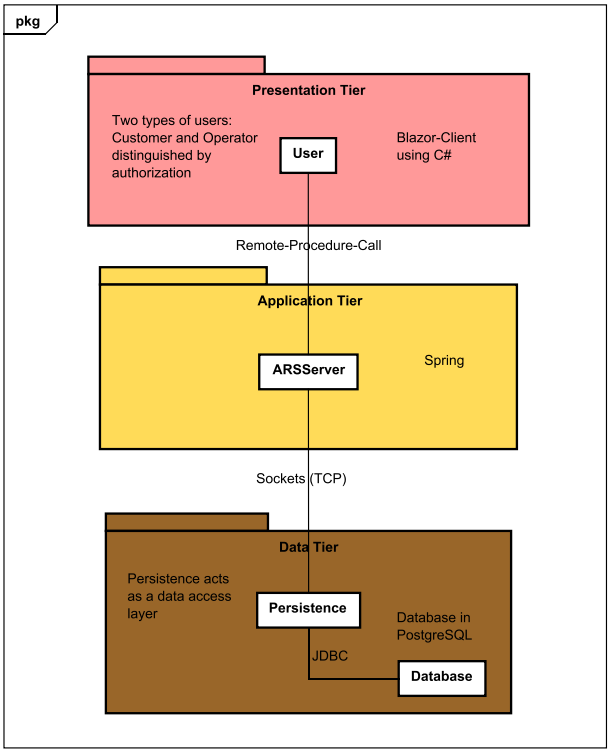
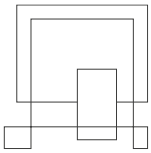


Diagram 7: System architecture diagram



3.1.1 **Tiers** (Karrtiigehyen and Nicolas Popal)

In the presentation tier, the topmost level of the application, displays information regarding services such as booking flights, viewing available flights, and creating flights. The result will be displayed on a web page in the form of a GUI. This is achieved by using *Blazor*, namely *Blazor WebAssembly*. The reason for choosing to work with *Blazor* is that it offers more flexibility and options compared to other GUIs. The entire presentation tier is programmed in *C#.* As shown in the use case diagram in the Analysis section, the two actors, *Customer* and *Operator.* These actors are distinguished by authorization in the presentation tier. For more on how this is achieved, look in the Implementation section.

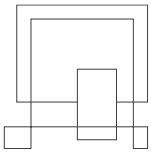
The application tier holds the business rules of the system. This tier is also the middleware of the system. This is a component-based middleware instead of an object-based middleware. This is so that the dependencies will be made explicit and will provide a more complete contract for system construction. The middleware is created using the *Spring Boot* framework. *Spring Boot* is used since it gives a flexible way to database transactions and *XML* configurations. It also manages *REST* endpoints. The middleware is programmed in *Java*.

The data tier houses the persistence layer and the database. The persistence layer consists of the Data Access Objects (DAO), and data can be stored and retrieved from the database. The persistence layer is programmed in *Java*, while the *PostgreSQL* is used for the database. This tier depends on no other tiers, while the application tier is dependent on the data tier, and the presentation tier is dependent on the application tier. For more on the *DAO*, look in the subsubsection Persistence.

3.1.2 **Communications Between the Tiers** (Karrtiigehyen and Nicolas Popal)

The communication used between the presentation tier and application tier is webservices using Remote Procedure Call.

Sockets are used for the communication between the middleware (application tier) and the persistence layer in the data tier. The socket protocol used for this is *TCP* since *TCP*



ensures that there will be a lossless and reliable data transmission, which is needed for this system. *UDP* would be incompatible since reliability of the data transmission is of a higher importance than performance.

The persistence layer in the data tier can perform *CRUD* operations on the database via *JDBC*.

3.2 **Design of The Airline System** (Jan Vasilcenko and Karrtiigehyen)

In this subsection, the package diagram will be shown first in order to get a bird's eye view of the system. Then, the package diagram will be discussed further into details with the help of some class diagrams, highlighting the design patterns. At the end, some sequence diagrams will be shown and discussed to illustrate the flow of the system.

The SOLID principles were used as a guideline while designing the classes so that dependencies are reduced, meaning that the code should be loosely coupled, highly cohesive, context independent, and easy to divide into separate standalone components. This also makes testing the system much easier.

3.2.1 **Package Diagram** (Jan Vasilcenko and Karrtiigehyen)

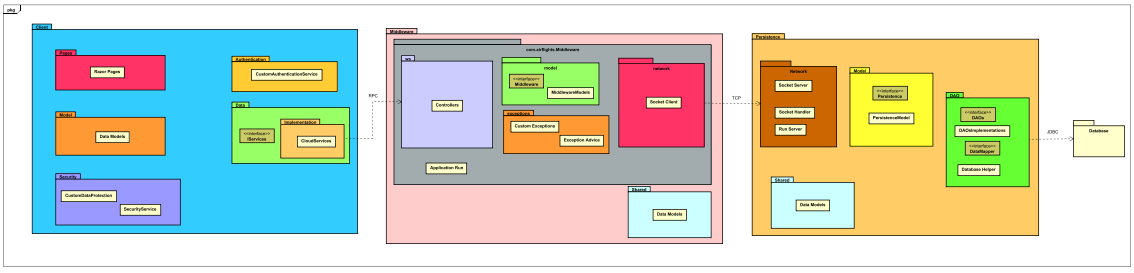
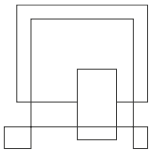


Diagram 8: Package diagram of the system

Each of the three packages represents a tier, with the corresponding communications between them. The leftmost package (blue) called *Client* is the presentation tier, the middle package (pink) called *Middleware* is the application tier, the orange package, *Persistence*, and the database package represent the data tier.



The communications between *Client* and *Middleware* are handled by web services, RPC. *TCP* is used between *Middleware* and *Persistence,* while *JDBC* is used between *Persistence* and the database.

The package diagram can be found in Appendix D.

3.2.2 **Client** (Jan Vasilcenko)

The full class diagram can be located in Appendix E.

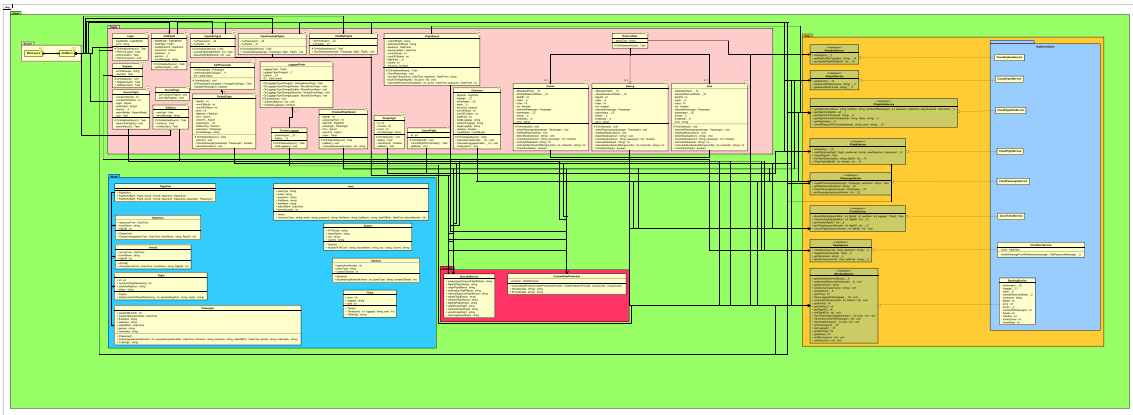
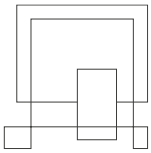


Diagram 9: shows the Client package

The *Client* is implemented in *Blazor*, programmed in *C#*. The *GUI* is implemented in the form of Razor pages, which are divided into two parts: *HTML* and *C#* code. The values from the *HTML* are bound to the code, and various operations are performed on them by services.

Services are classes that perform operations on data inputted. These services can be found in the *Security* and *Data* packages. These services are divided into two, local services and cloud services. Local services perform actions on local scale, for example saving and getting data, and encryption and decryption. On the other hand, cloud services contact the *Middleware*.

The dark pink package, *Security,* contains two local services. The first is used to encrypt and decrypt the URL (because of security reasons, and for more on the matter refer to



subsections 2.5 and 3.4). The second one is used for storing and getting those URLs. Both of these services make use of the **singleton pattern** so the same instance can be called in various pages.

3.2.3 **Middleware** (Jan Vasilcenko)

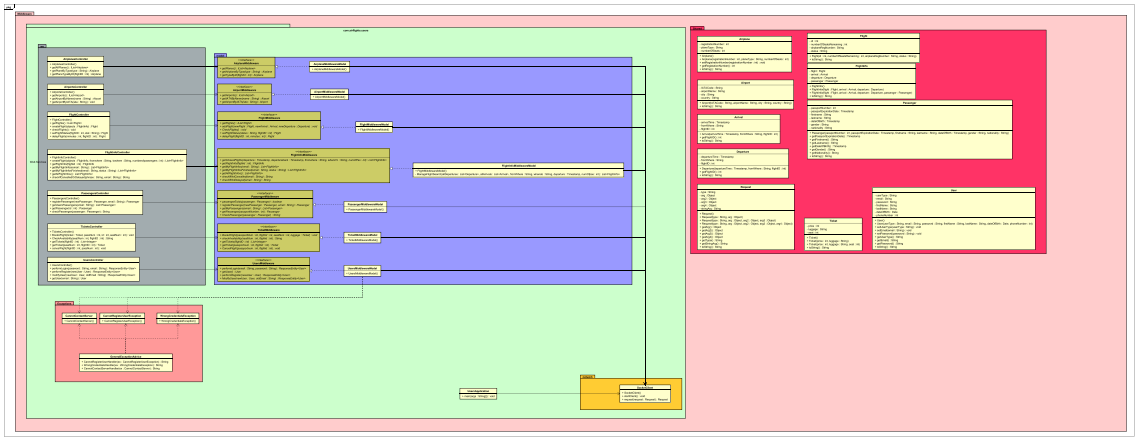


Diagram 10: shows the Middleware package

In the *Middleware*, the business logic is located. It is programmed in *Java* using the *Spring* framework. The middleware is a server-side web service, which is exposing itself to the *Client* for use.

In the package *ws*, there are many controllers which contain HTTP methods which are called by the client to be consumed. Controllers are implemented mainly by Spring annotations, specifying methods, requests and parameters. Each controller has its own instance of the middleware model, which can be found in the *Model* package.

The *Model* package contains all of the middleware models and their interfaces. They are divided so that they comply with the SOLID principles. The business logic of the program can be found here. Custom exceptions are used here. Finally, each one of them have an instance of *SocketClient* to communicate with the *Persistence.*

The *Exceptions* package (light pink) contains all of the custom-made exceptions, and general exception advice to return throw those exceptions to the *Client.*

In the *Network* package, the *SocketClient* can be found. *SocketClient* sends a *Request* object to *Persistence* and awaits a response in form of another *Request* object.

The *Request* object is a DTO, with a *String* type which is used to implement the custom protocol for sockets. DTO can also store other objects, that can be send or received in the form of object arguments.

3.2.4 **Persistence** (Jan Vasilcenko and Karrtiigehyen)

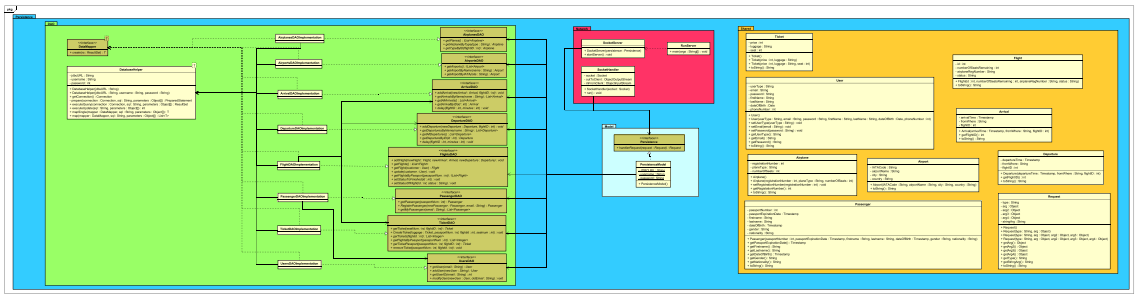


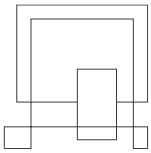
Diagram 11: shows the Persistence package

The *Persistence* package is a data access layer, with an access to the database. It is programmed in *Java*. It also has a *Network* package, which houses *SocketServer*. It is responsible for accepting connections from *SocketClient*. There is also a *SocketHandler*, which receives requests and passes it to the persistence model.

PersistenceModel has instances of the DAOs. It handles the requests by reading its types and calls the proper DAO, and sends a response in the form of *Request* object.

A connection to the database must be made in order to read or manipulate data from the database. This is done by using an API called Java Database Connectivity (JDBC), which will be explained further in the Technologies Used subsection. A class called *DatabaseHelper* is used here so that the DAOs can get access to the database.

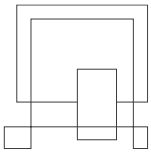
Data Access Object pattern is a design pattern that is used in separating low level data accessing API or operations from high level business services (Tutorialspoint, 2020). DAO interfaces are used to define the methods to be performe. DAO concrete classes



are classes that implement the DAO interfaces, where it can get data from the database. The DAOs are mainly designed using CRUD (Create, Read, Update, Delete) as a guideline, but there are some classes where it is not used. For example, some classes will only need to read data from the database. The DAOs are also designed using the SOLID principles, which is why there are many DAO interfaces and concrete classes, to comply with the Single Responsibility Principle.

An **adapter pattern** is used so that the *PersistenceModel* can call methods from the DAOs. In this case, the *DatabaseHelper* is the Adaptee, the DAO concrete classes are the Adapter, the DAO interfaces are the Target, and *PersistenceModel* is the Client.

There is a problem with the *Persistence* however. Firstly, the DAOs violate the DRY (Don't Repeat Yourself) rule. There are several lines of code that will repeat in the DAOs. The command pattern could have been used to avoid this problem. The command pattern turns a request into a stand-alone object that contains all information about the request (Refactoring Guru, 2020). This same problem can also be found in the *Client* package, where pages *Boeing, Comac* and *Irkut* contain a lot of the same duplicate code.



3.2.5 **Sequence Diagram** (Nicolas Popal)

Sequence diagrams are based on the methods in the actual code and are drawn, so that it can be seen what methods and classes are used sequentially in the system. The examples shown here are for creating flights and viewing available flights.

All of the sequence diagrams can be found in Appendix F.

3.2.5.1 Creating Flights (Nicolas Popal)

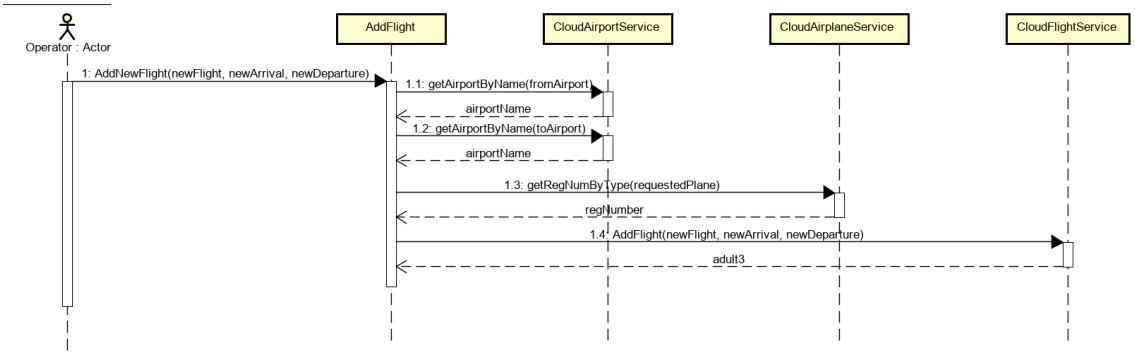


Diagram 12: Sequence diagram of creating flights in Client

Flight is added, when the confirmation button by *Operator* is clicked. System needs arguments *flight*, *arrival* and *departure* to add the actual flight. Argument *arrival* and *departure* is read by the system in *CloudAirportServices*, which contains information about all the saved airports in the system. Argument flight will be looked up from the system based on the argument *regNumber*, which is registration number of the airplane already saved in the system. Finally, flight is sent to the middleware through *json*.

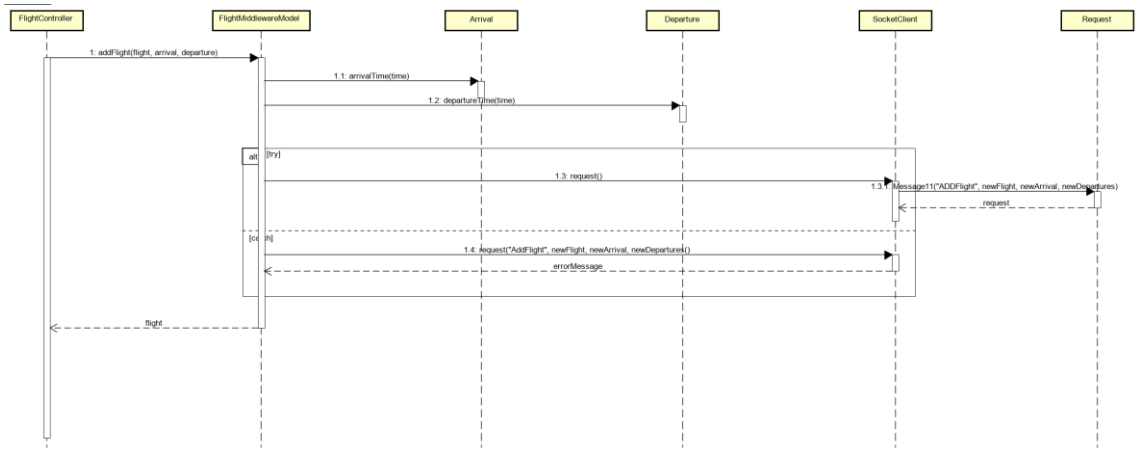


Diagram 13: Sequence diagram of creating flights in Middleware

In middleware new flight is received in *FlightController*. *FlightControllers* handles this request in *FlightMiddlewareModel* in *addFlight* method. In this method *arrivalTime* and *departureTime* is set and system continues to exception. In this exception system continues to *SocketClient*, which is sending request to class *Request* with parameter *ADDFlight* and based on this parameter system distinguish, that new flight is added. Next 3 arguments *newFlight*, *newArrival* and *newDepartures* are data received by middleware from client with all the flight information.

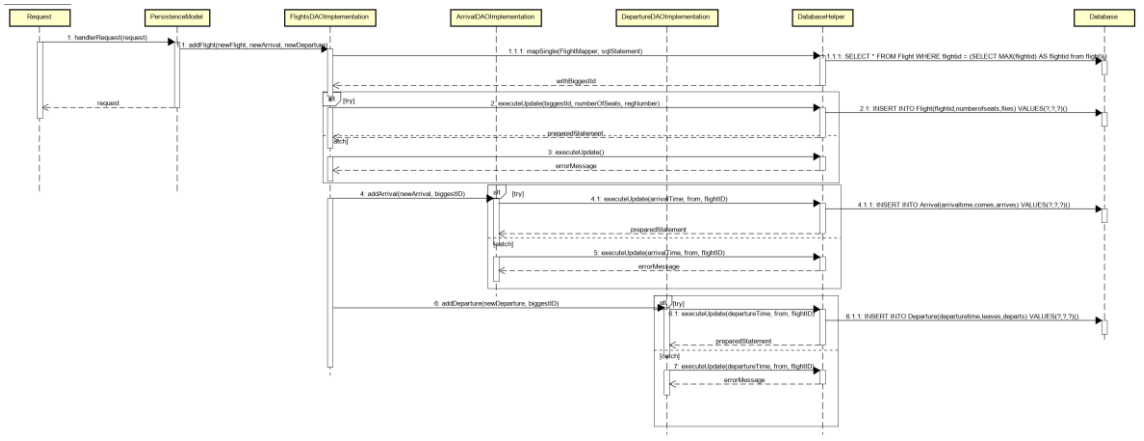
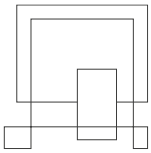


Diagram 14L Sequence diagram of creating flights in Persistence

In middleware request to the Request class was sent and as this class is shared with persistence, persistence can receive it. *Request* is then calling method *handlerRequest*



in *PersistanceModel* and based on the first argument sent from middleware “ADDFlight” method *addFlight* is called in class *FlightsDAOImplementation*. In this class persistence looks up the biggest ID saved in flights, so the new flight can be saved with the next biggest ID. Finally, flight is added to the database in class *DatabaseHelper* through method *executeUpdate*. Then class *FlightsDAOImplementation* is calling methods *addArrival* in class *ArrivalDAOImplementation* and method *addDeparture* in class *DepartureDAOImplementation* and saving to database departure and arrival of this flight.

3.2.5.2 Viewing Available Flights (Nicolas Popal)

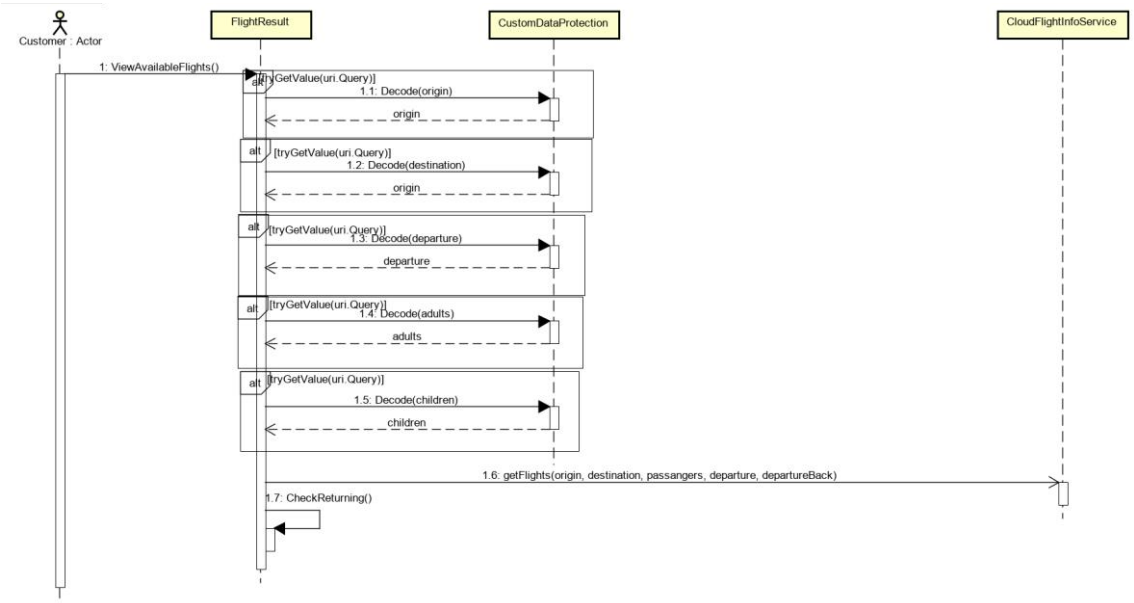


Diagram 15L Sequence diagram of viewing available flights in Client

View available flights is displayed when customer search for a flight. Firstly, *FlightResults* try all the arguments used in previous page during search and check them in *CustomDataProtection*. If all arguments are okay, client looks for all flights available in class *CloudFlightInfoService*, which are getting information in middleware. This will be explained bellow. Also, class *FlightResult* checks for return date in method *CheckReturning*.

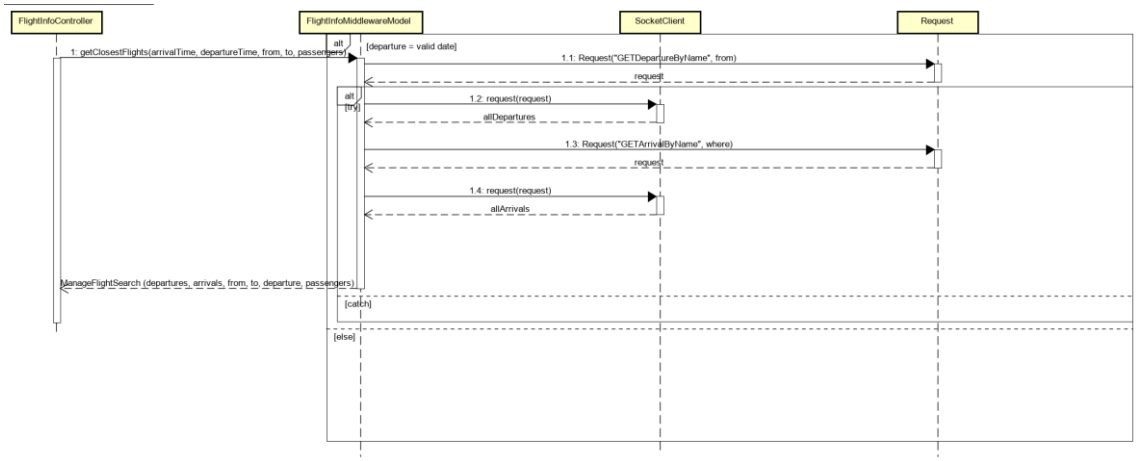


Diagram 16: Sequence diagram of viewing available flights in Middleware

Client sends a message to middleware, where class *FlightController* receives it. Based on this request, *FlightController* uses method *getClosestFlights* from *FlightInfoMiddlewareModel*. In this class date of the departure is checked and request method to the *Request* class to get departures can be sent. This sends a request to the persistence where all the departures are saved. Then system continues and saves all the available departures. Same happens for arrivals.

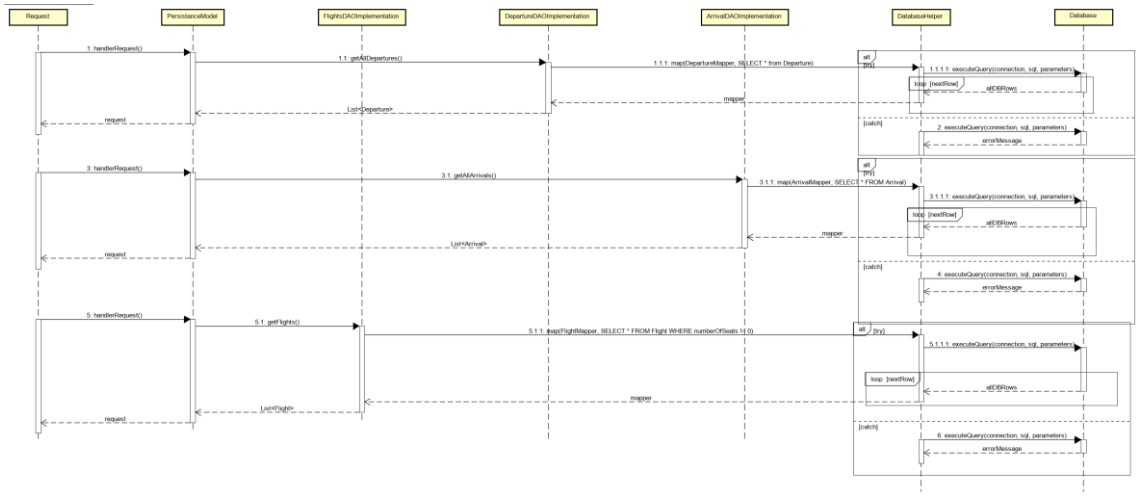
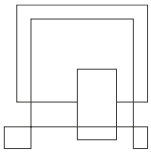


Diagram 17: Sequence diagram of viewing available flights in Persistence



Request methods from middleware are received in the persistence. Then all these requests are handled in *PersistanceModel* and each request is handled by its own DAO class, flight request is handled in *FlightsDAOImplementation*, departure request is handled in *DepartureDAOImplementation* and arrival request is handled in *ArrivalDAOImplementation*. In these classes' method map from class *DatabaseHelper* is called and SQL statements from arguments are executed. Lists are then returned back and sent as a request back to the middleware.

3.3 **Database** (Karrtiigehyen and Nicolas Popal)

3.3.1 **Conceptual Model** (Karrtiigehyen and Nicolas Popal)

The conceptual model is heavily influenced by the domain model, the only change being *Users* is a superclass to *Operator* and *Customer* subclasses.

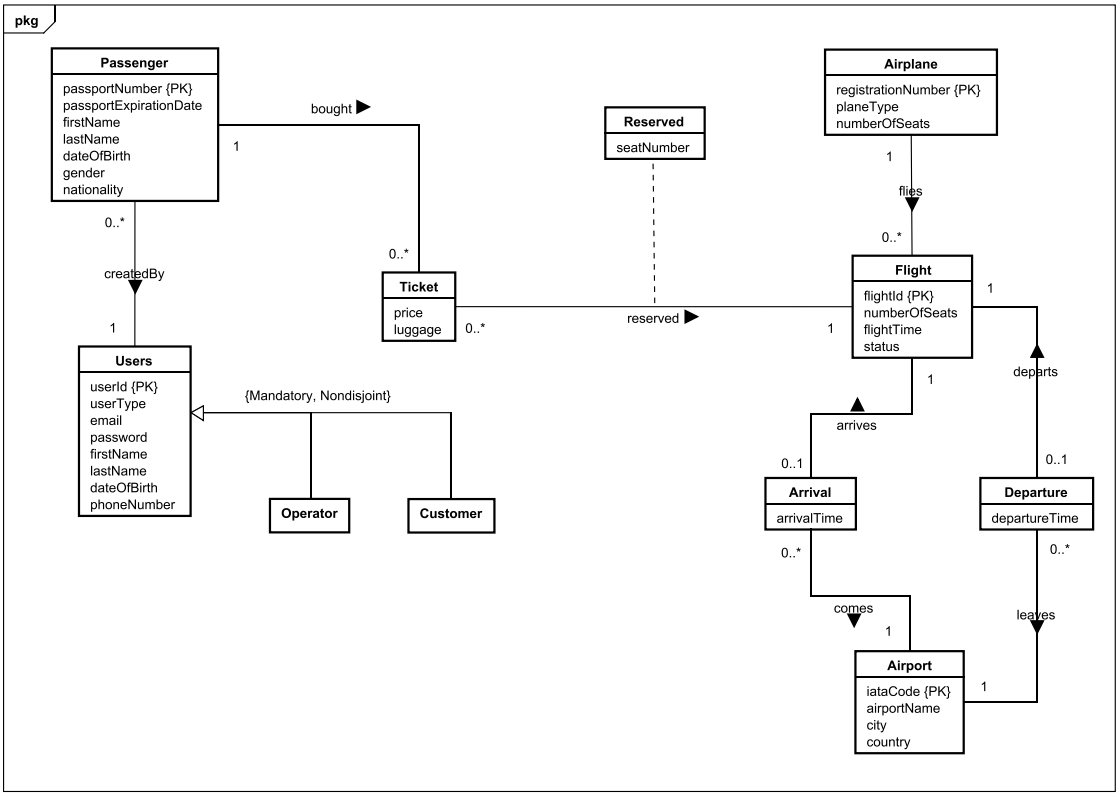
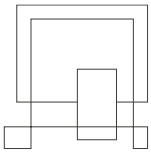


Diagram 18: EE/R diagram of the database

Entity/Relationship modelling models a business situation by describing the relevant entities and their relationships. For example, *Passenger* and *Ticket* are entities, and *boughtBy* describes the relationship between the entities.

This conceptual model is in third normal form. Normalization is a technique for producing a set of relations with desirable properties, given the data requirements of an enterprise (Connolly and Begg, 2015). It is in third normal form since it satisfies the first and second



normal form, and no non-primary key attribute are transitively dependent on the primary key.

To better explain the conceptual model, it will be separated into three sections - Section 1, Section 2 and Section 3.

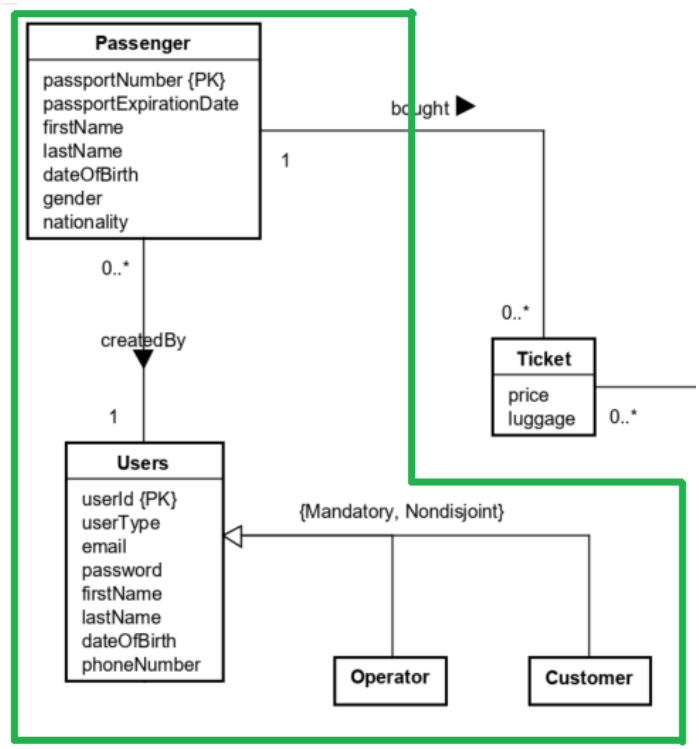


Diagram 19: Section 1 of the EE/R diagram

In Section 1, the strong entities present are the *Users* and *Passenger* entities. The *Users* entity holds the account information of users that use the system. The extended entity/relationship modelling is used between *Users*, *Customer* and *Operator*. *Users* acts as the superclass to the subclasses *Customer* and *Operator.* The participation constraint is mandatory and nondisjoint, which means it has a single relation with one or more discriminators to distinguish the type of each tuple (Connolly and Begg, 2015).

The other strong entity in this section is the *Passenger* entity. The *Passengers* entity has the information of passengers, with the *passportNumber* as the primary key The *Passenger* entity is created for the handling of creating multiple tickets by one customer.

For example, if one customer decides to buy tickets for 3 people for one flight, each of the 3 people will be regarded as individual passengers, with their own *passportNumber,* names and so on. The relationship, *createdBy*, describes the one-to-many relationship, since multiple passengers can be created by a user, but a passenger can only be created by one user.

The Section 2 primarily deals with the handling of passengers who buy tickets.

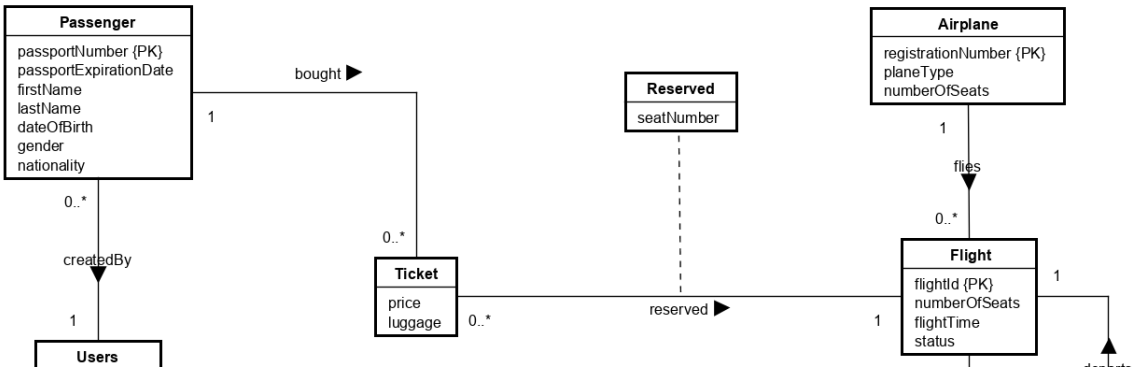
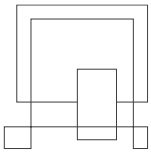


Diagram 20: Section 2 of the EE/R diagram

The new strong entities in this are *Airplane* and *Flight.* The *Airplane* entity describes the airplane, *Flight* describes the flight information. The status in *Flight* refers to whether the flight is on time, delayed or cancelled. The relationship between *Airplane* and *Flight*, flies, is a one-to-many relationship because a flight can only have one airplane flying and an airplane can fly zero to many flights.

The *Ticket* is a weak entity. *Ticket* is a weak entity because passengers' *passportNumber* can be stored alongside *flightId* from the *Flight* entity. The *passportNumber* and *flightId* will act as the primary keys, which then prevents passengers with the same *passportNumber* to buy more than one ticket for the same flight. *Ticket* and *Passenger* are related by the relationship *boughtBy*, which is a one-to-many relationship. It is a one-to-many relationship since a passenger can buy multiple tickets, but a ticket must belong to only one passenger.

How the reservation of seats is handled can be seen in this section as well. Each airplane has a fixed number of seats. The attribute, *numberOfSeats,* in the *Flight* entity shows



how many seats are left that are unreserved in the airplane. The relationship, *reserved,* describes the reservation of seats. The attribute, *seatNumber,* indicates the seat number belonging to a particular flight reserved by a passenger. As seats are reserved, the attribute *numberOfSeats* in *Flight* should decrease.

In section 3 are 3 strong entities and 2 weak entities. Strong entities present are *Airplane*, *Flight* and *Airport* and weak entities are *Departure* and *Arrival*.

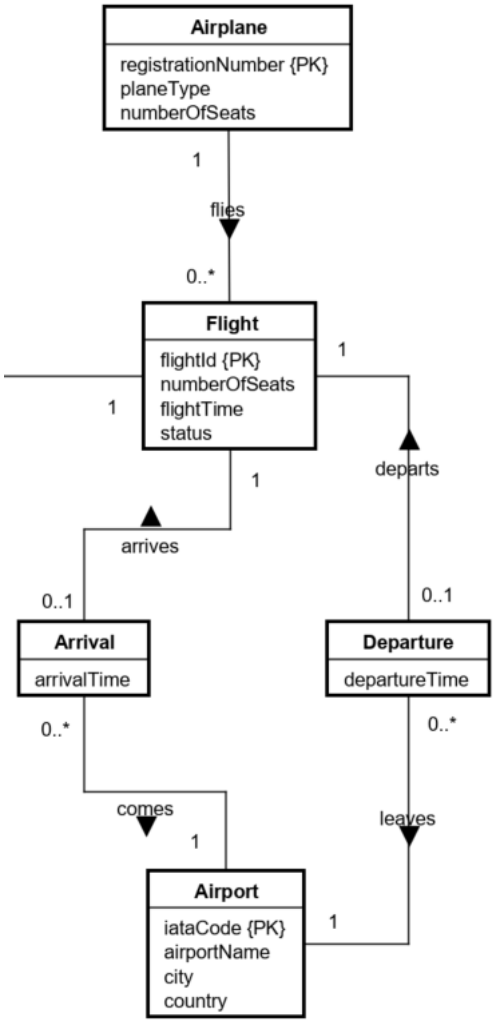
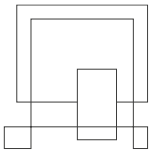


Diagram 21: Section 3 of the EE/R diagram



The *Flight* entity holds the information of flights that use the system, while *Airplane* holds information about airplanes. The relationship between these strong entities is called *flies*, and it is a one-to-many relationship. The reason, why it is a one-to-many relationship, is because *Flight* can store only one airplane (as the one flight will be done with one airplane) and *Airplane* can store multiple flights (as the airplane can be used for many flights). One-to-many relationship is also used between *Airport-Arrival* and *Airport-Departure*.

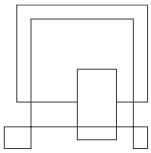
Arrival and *Departure* are weak entities because there is no need for primary keys. There is a one-to-one relationship between strong entity *Flight* and weak entities *Arrival* and *Departure*. These one-to-one relationships are called *arrives* and *departs*. This relationship is used, because each flight can have only one departure and one arrival, and that Arrival/Departure can belong to only one *Flight*.

The conceptual model of the database can also be found in Appendix G.

3.3.2 **Logical Model** (Karrtiigehyen and Nicolas Popal)

The logical model will be used as the basis for the creation of the physical database. Deriving relations for logical data model was done by following the following steps (Connolly and Begg, 2015):

1. strong entity types
2. weak entity types
3. one-to-many (1:*) binary relationship types
4. one-to-one (1:1) binary relationship types
5. one-to-one (1:1) recursive relationship types
6. superclass/subclass relationship types
7. many-to-many (*:*) binary relationship types
8. complex relationship types
9. multi-valued attributes



An example for each of the steps will be showcased, but since there were no one-to-one (1:1) recursive relationship types, many-to-many (*:*) binary relationship types, complex relationship types, and multi-valued attributes in the conceptual model, steps 5, 7, 8, and 9 are skipped.

The full logical model can be found in Appendix H.

Only one example for each step will be shown.

Step 1: Strong entity types.

```
Users(userId, userType, email, password, firstName, lastName, dateOfBirth, phoneNumber)
PRIMARY KEY(userId)
```

An entity *Users* has attributes *userId, userType, email, password, firstName, lastName, dateOfBirth, phoneNumber,* with the *userId* being the primary key.

Step 2: Weak entity types AND **Step 3:** One-to-many (1:*) binary relationship types

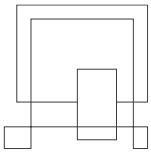
An interesting weak entity is the *Ticket.*

```
Ticket(bought, reserved, price, luggage, seatNumber)
PRIMARY KEY(passportNumber)
PRIMARY KEY(reserved)
FOREIGN KEY(reserved) REFERENCES Flight(flightId)
FOREIGN KEY(bought) REFERENCES Passenger(passportNumber)
```

The *Ticket* entity only had the price. But since its relationships with the *Passenger* and *Flight* entities are one-to-many, the *Passenger* and *Flight* entities are designated as the parent entities (which is Step 3). This makes the *Ticket* the child entity. This means that the *fligthId* and *passportNumber* will be foreign keys in the *Ticket* entity. The relationship attribute, *seatNumber,* is also added as an attribute.

Step 4: One-to-one (1:1) binary relationship types

The relationship between *Flight* and *Arrival* is a one-to-one type.



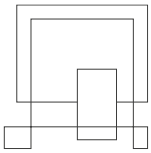
```
Arrival(arrivalTime, comes, arrives)
FOREIGN KEY(comes) REFERENCES Airport(iataCode)
FOREIGN KEY(arrives) REFERENCES Flight(flightId)
```

The *arrivalTime* will still be an attribute, but the *flightId* will be added as a foreign key. The other foreign showed in the code above comes from the one-to-many relationship with *Airport.*

Step 6: Superclass/subclass relationship types

```
Users(userId, userType, email, password, firstName, lastName, dateOfBirth, phoneNumber)
PRIMARY KEY(userId)
```

Since the constraint is mandatory and nondisjoint, just one table, *Users,* will be needed.



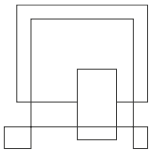
3.4 **Security Mechanisms** (Karrtiigehyen and Nicolas Popal)

In the Analysis section, potential security threats were discussed, and a risk assessment was created. In this subsection, the security mechanisms that can be implemented to prevent the analysed potential threats will be discussed here.

For this system, a symmetric key exchange and public key cryptography can be used. The reason for choosing both symmetric and asymmetric encryption is because of efficiency and the ability to get both confidentiality and authentication security objectives. If only the symmetric encryption is used, the solution to key exchange would be to meet, which is impractical. Asymmetric encryption is more inefficient compared to symmetric encryption.

The way symmetric and asymmetric encryption can be combined is by using symmetric key exchange and public key cryptography. The public key system to be used is Diffie-Hellman key exchange. Alternatively, RSA could have also been used as well, but Diffie-Hellman is chosen because its better efficiency. Diffie-Hellman key exchange is secure because of the discrete log problem. Diffie-Hellman does have one problem though. It is vulnerable to a man-in-the-middle attack. This issue can be solved with the use of Certificate Authority. The Certificate Authority needs only to be sent from the airline reservation system to the users and the users can authenticate themselves with email and password. The reason why the users do not need to send Certificate Authority to the airline reservation system is because it would be inefficient and impractical.

The user authentication, like stated above, will include an email, which is the ID of the user, and a password. To address the password cracking threat stated in the risk assessment, some measures can be taken to nullify the threat. Firstly, instead of focusing on password length, more emphasize should be placed upon using different keystrokes for the password. Secondly, instead of transmitting the password in plaintext, it should be instead be a hash value. Users should be aware how their password is stored, and the password should only be stored as a hash value in the database.



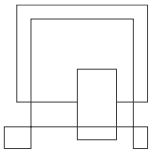
Speaking of hash functions, the hash functions used should be resistant to preimage, second preimage and collision attacks. An example of a hash algorithm that can be used is SHA-3 since no collisions have been found yet in the algorithm.

So far, using symmetric key exchange and public key cryptography using Diffie-Hellman and Certificate Authority, will guarantee confidentiality, and message authenticity and integrity. But nonrepudiation would also be a nice security objective to have. This cannot be achieved using hash functions or MACs, but it can be achieved using Digital Signatures. Digital Signatures grants two factor authentications, which was the counter measure needed for the website spoofing threat. Plus, using Digital Signature alongside Certificate Authority will make sure that nonrepudiation is achieved. Elgamal can be used can be used as the digital signature scheme.

But so far, encryption itself has not been discussed yet. This is because encryption of anything should be the last step. After all the above-mentioned systems have happened, it should be encrypted. Advanced Encryption Standard (AES) would be a good choice for encryption since it is one of the most widely used encryption systems.

All of the cryptographic systems discussed above can be summarized by using SSL/TLS. Since the Certificate Authority will only need to be sent one way, a one-way TLS would be best suited. The TLS protocol implements all of the discussed subjects above, making it a prime candidate to implement. It uses symmetric and asymmetric encryption, with Diffie-Hellman key exchange. Certificate Authority can also be used with TLS. In the Record Protocol in TLS, sequence numbers and MACs will be added to the raw data before being encrypted, which will prevent replay attacks (replaying packages during ongoing sessions). And nounces are also used during the TLS Handshake, which also prevent replay attacks (connection replay). TLS also ensures that denial of service attacks cannot be performed efficiently.

Almost all of the threats stated in the risk assessment table is addressed in the discussion above. The only threats that are not addressed are URL query string attack and SQL injection. The URL query string attack can be prevented by encoding the URL and SQL

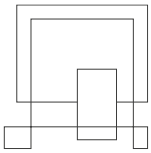


injections can be avoided by using prepared statements and doing the query in the persistence layer in the data tier.

The table below shows which security mechanisms can be used to address which threats.

Threat	**Security Mechanism**
Website spoofing	Digital Signatures
Man-in-the-middle attack	Certificate Authority
Replay attack	TLS
Password cracking	Use stronger passwords, transmit and store the hash value of the password
URL query strings	Encode the URL
DDoS attack	TLS
SQL injection	Prepared statements

Table 4: Security mechanisms for the potential threats



3.5 **Technologies Used** (Karrtiigehyen and Nicolas Popal)

The programming languages used are *Java* and *C#.* This mixture of programming languages leads to the system being heterogeneous. *C#* is used in *Blazor*, alongside *HTML* and *CSS* for construction of the web application. Alternatively, JavaFX could have been used for constructing a GUI, but since building a heterogeneous system using *Java* and *C#* was a requirement, *Blazor* was chosen for the presentation tier. *Java* is used in the middleware (application tier) and in the persistence layer (data tier). In the middleware, *Java* is using the *Spring Boot* framework.

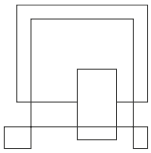
Remote Procedure Call protocol is used for the communication between the presentation and application tiers.

Sockets are used to establish a connection between the middleware and the persistence layer. The protocol used was TCP, since it is lossless and reliable, meaning if a segment is dropped, TCP will resend it. While TCP might not be as fast as UDP, a reliable data transfer is more important than better performance in this system. Alternatively, RMI could have been used instead of sockets, but sockets are chosen since they are part of the requirement.

As the requirement was that the system needs to use a database, *PostgreSQL* is used. *PostgreSQL* is an open-source object-relational database system (PostgreSQL, 2020). All the relevant data is stored and can be fetched in the database.

The persistence layer in the data tier and database must be connected as well. For this connection *PostgreSQL JDBC Driver API* is used. It is an open-source *JDBC* driver written in pure java (PostgreSQL JDBC, 2020). It is necessary to use this to establish a connection between database and persistence layer, so that the DAOs will function.

In the testing part, JUnit 5 is used. JUnit is a testing tool used for testing class methods in order to check the functionality of the code. It is usually used in parts of the code that involve logic to verify code integrity. Spring's MockMVC is also used since a web layer is also tested (Spring, 2020).



The last used technology is the Maven repository. Maven is an automation tool used in Java to help with managing file structure, tools, and plugins.

3.6 **UI Design Choices** (Patrik Horny)

This semester project the UI of the whole system could be made much more appealing thanks to using HTML/CSS. UI of the system is designed to be intuitive for customer, as well as for operator. No fancy transitions or animations had been implemented, but instead the focus was put more on the functionality and consistency of the UI.

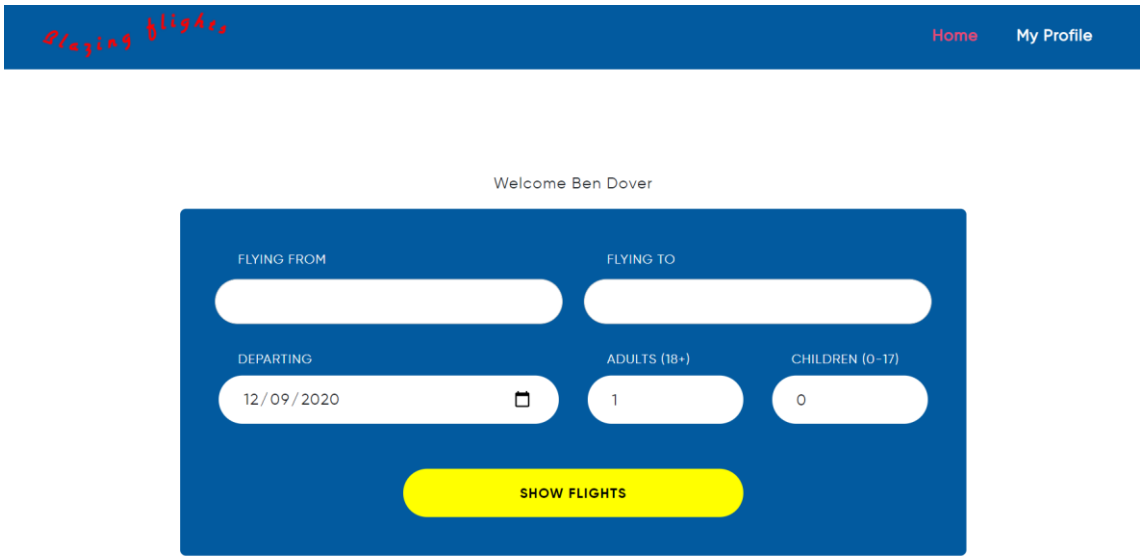


Figure 1: Searching flights

After logging in to the system as a user, user can search for desired flight with his preferences for passengers and date.

In the list of flights available flights will appear based on previous search of the customer, if the criteria are met. Customer can see the details of the flight, such as the length of the flight and its price.

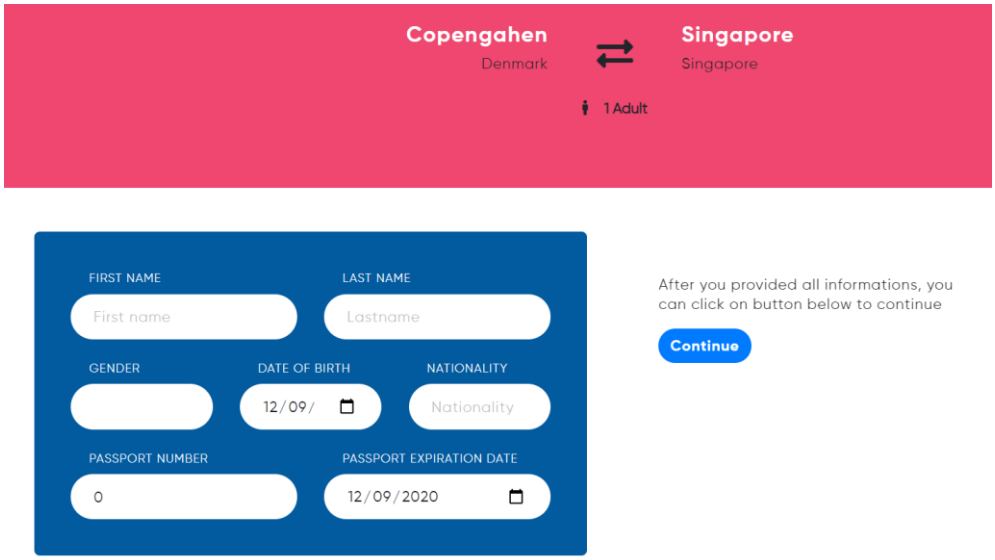


Figure 2: Inserting personal information

When the customer is ready with his deciding, he books a flight and is prompted to fill out their personal information, as well as other passengers' information, if the customer chose to travel with somebody. On top of the page, customer has an overview of how many passengers are going with and if the destination is the right one that the customer has chosen.

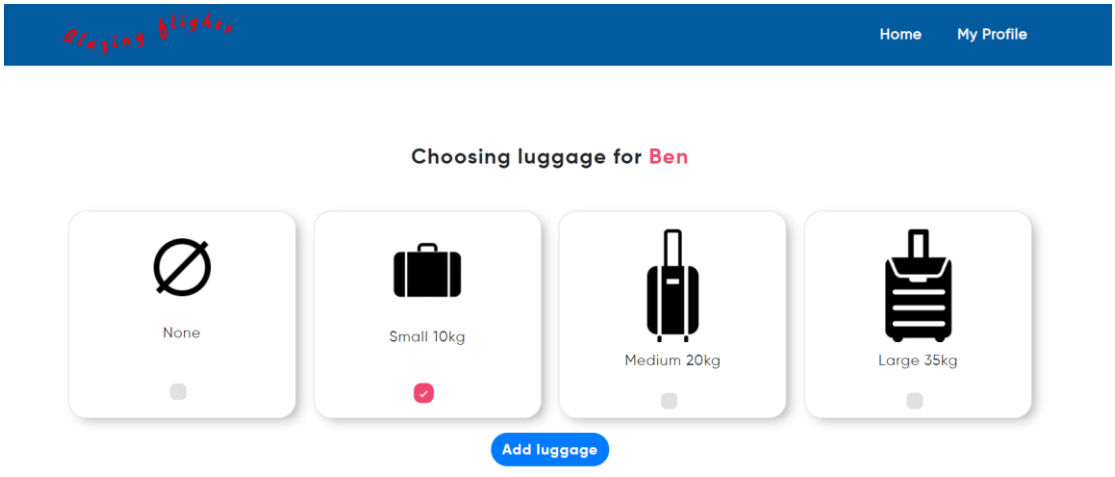


Figure 3: Choosing luggage option

Then the customer can choose an additional luggage based on their preferences or to choose no luggage at all.

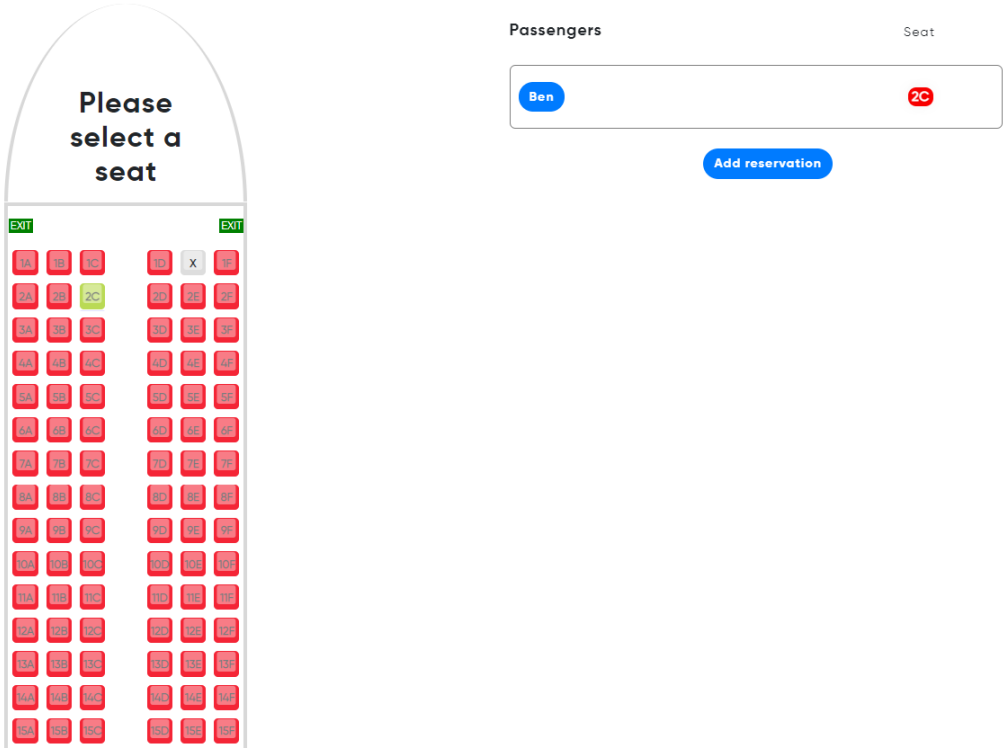


Figure 4: Reserving seat

In reserving a seat section customer needs to choose a seat where they want to sit. Customer can only choose a seat where with red colour. Reserved seats are filled out in grey colour with an X-mark as an indication that the seat was reserved by somebody else. Customer has no option to choose already reserved seats by somebody else since they would not be able to click on it.

On the right side, customer can see the chosen seat, as well as their name if more passengers are coming with.

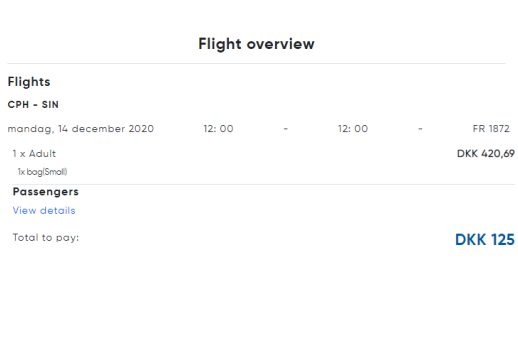


Figure 5: Payment

After seat selection, overview of the flight is showed to customer with the whole information of the flight and the price. When everything is checked, customer is prompted to pay for the flight with his credit card.

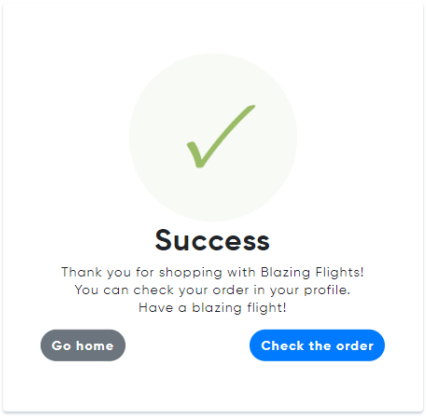
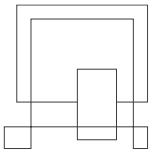


Figure 6: Booking is successful

If there are no error in the reservation, success page will appear, and customer can check his flight or to continue in booking another flight.



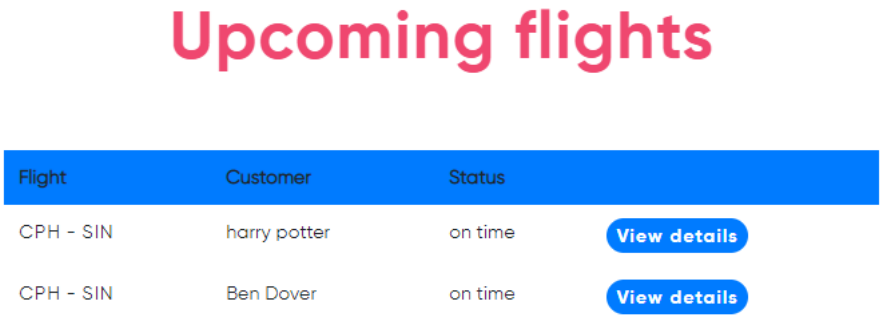


Figure 7: Overview of booked flights

Overview page consists of upcoming flights by customer. Customer can clearly see whether the flight is on time, delayed or if by any chance was cancelled. If customer changes their mind, he can easily cancel the flight. Details of the flight are available as well with all necessary information.

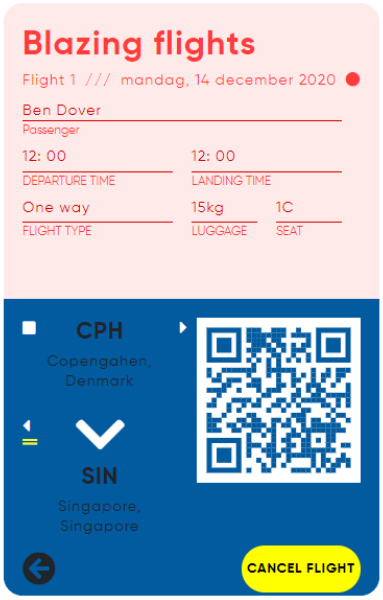


Figure 8: Ticket

Finished flight page is just a simple page with history of passenger flight.

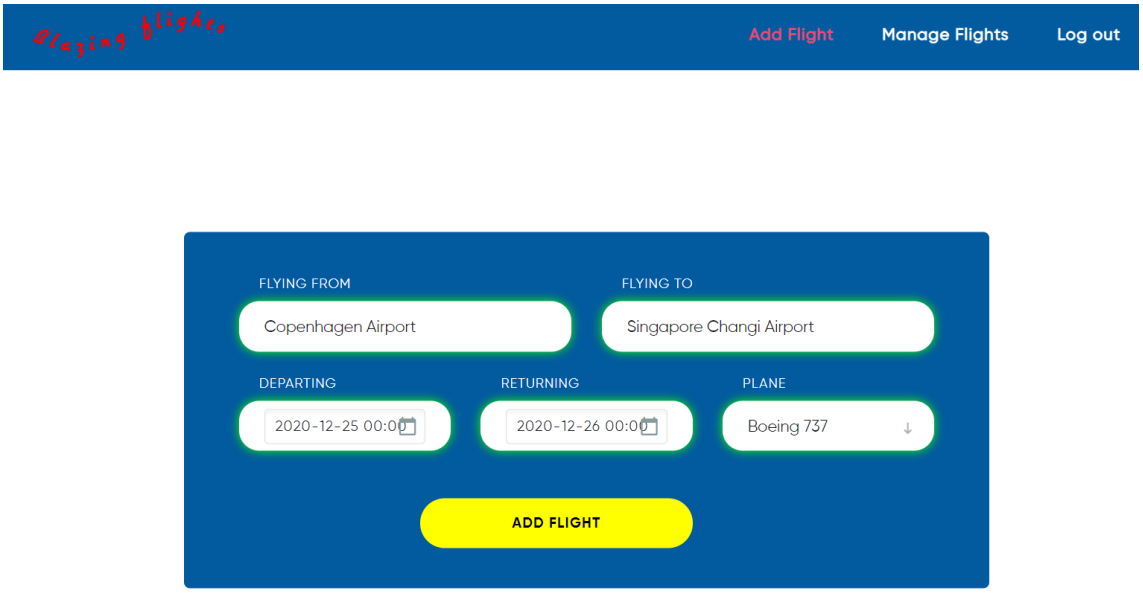


Figure 9L Creating flights

The operator has an option to create a new flight with the destination option, departure time and date as well as choosing which plane is going to partake on the flight. Time inputs are included in date for the sake of saving space and less objects to focus on the screen.

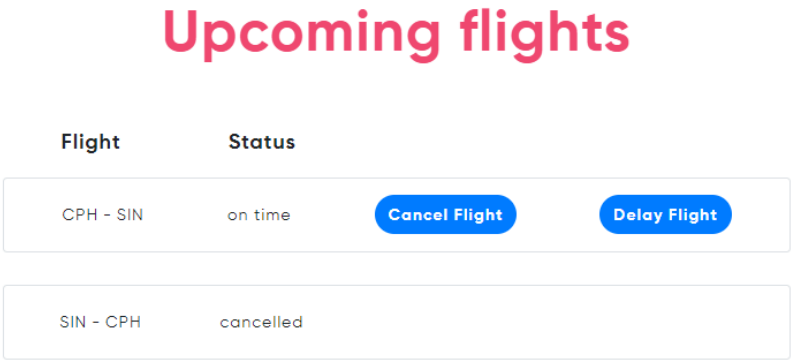
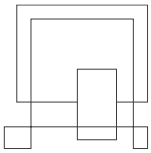
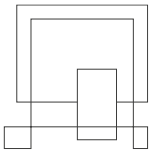


Figure 10: Overview of flights for Operator

Overview page has the same layout as the layout of the customer. There operator has an option to cancel to flight from which he is asked second time by the system if the



decision is final to prevent accidents. Status and details of the flight are present as well in the table.



4 **Implementation** (Jan Vasilcenko)

In this section, only interesting and significant parts of the code will be shown and explained. The code for the database is not shown in this section since it is simply the creation of tables according to the logical model shown in the subsubsection 3.3.2 and populating them with data.

The interesting code snippets will be shown by explaining how the flow of logic is done through the three-tier architecture using the View Available Flights and Create Flight use cases, with particular focus on the communications between the tiers. This way, all of the important parts of the three-tier architecture will be touched upon.

The source code for the database and the airline reservation system can be found in Appendix I.

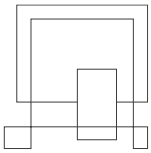
4.1 **Presentation Tier** (Jan Vasilcenko)

To communicate with user, Blazor implements UI which is in form of razor pages, which holds HTML content and C# code.

```
<div class="row">
    <div class="col-md-6">
        <div class="form-group">
            <span class="form-label">Departing</span>
            <InputDate class="form-control" type="date" placeholder="Departing" @bind-Value="searchModel.departing" />
            <ValidationMessage For="@(() => searchModel.departing)" />
        </div>
    </div>
```

```
@code {
    private IList<Airport> airports;
    private Airport origin;
    private Airport destination;
    private Departure wishedDeparture = new Departure();
    private int numberOfAdults;
    private int numberOfChildren;
    private SearchModel searchModel = new SearchModel();
    private User user = new User();
```

Code 1: shows the code and HTML part of a razor page and binding in SearchFlight



For getting input for user, Blazor form is used with validation. Values from those forms are bound onto C# objects and then used to perform some action, usually communicating with rest of the system.

4.2 **Web Services** (Jan Vasilcenko)

To establish the web service communication, there must be a client and server, and both of these need to be connected to the internet, so that they can communicate over it. Server, running on some port, is exposing the services to the client, so that the client can invoke and call the service. The platform for our web service is HTTP protocol, meaning that mainly methods like GET, DELETE, PUT and POST are used. For security, HTTPS or SSL was introduced, so that data are encrypted and decrypted, so that only sender and receiver can see it.

4.2.1 **Server-side** (Jan Vasilcenko)

```
@RestController @RequestMapping("/flightinfo") public class FlightInfoContoller
```

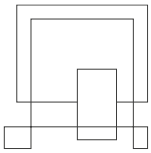
Code 2: shows the annotation of a controller in Spring

Classes on the server-side, which is in the Middleware (implemented in Java), are responsible for handling clients' requests are called controllers, and are mapped, so that when there are more controllers, you can easily distinguish between them and call the right one.

```
private FlightInfoMiddleware flightInfoMiddleware;

public FlightInfoContoller()
{
  this.flightInfoMiddleware = new FlightInfoMiddlewareModel();
}
```

Code 3: shows that the controller implements an instance of the middleware

Each controller holds and implements in constructor instance of middleware interface, so that they can call methods and perform logic.



```
@PostMapping List<FlightInfo> searchForClosestFlights(@RequestBody FlightInfo objects,
    @RequestParam String fromwhere, @RequestParam String towhere,
    @RequestParam int numberofpassengers)
{
  return flightInfoMiddleware.getClosestFlights(objects.arrival.arrivalTime,
      objects.departure.departureTime, fromwhere, towhere,
      numberofpassengers);
}
```

Code 4: shows the web services method in Spring

Example of controller method will be *searchForClosestFlights()* method. With beginning annotation, it is set as a Post method, which returns a list of Flights. In argument section it requests body from http message, and three parameters from URI (Uniform resource identifier). Finally, controller calls middleware interface, which holds all the logic and returns list of desired flights.

4.2.2 **Client-side** (Jan Vasilcenko)

Client is implemented in C#. It calls HTTP methods to be performed in Server and awaits answer.

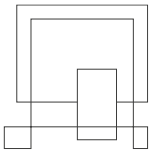
```
flightInfos = await flightinfoservice.getFlights(originOfFlight, destinationOfFlight, (numbOfAdults + numOfChildren), departure, departureBack);
```

Code 5: shows the service call for showing flights

Somewhere in the code, method for searching flights is called and the result from server is awaited.

```
public async Task<List<FlightInfo>> getFlights(string fromWhere, string toWhere, int numberOfPassengers
    , DateTime departure, DateTime departureback)
{
    HttpClient client = new HttpClient();
    StringContent content = new StringContent(JsonSerializer.Serialize
        (new FlightInfo(new Flight(0, null, null), new Arrival(departure, null, 0), new Departure(departureback, null, 0)))
        , Encoding.UTF8, "application/json");
    HttpResponseMessage message = await client.PostAsync("https://localhost:8443/flightinfo" + "?fromwhere=" + fromWhere
        + "&towhere=" + toWhere + "&numberofpassengers=" + numberOfPassengers, content);
    string response = await message.Content.ReadAsStringAsync();
    List<FlightInfo> result = JsonSerializer.Deserialize<List<FlightInfo>>(response);
    return result;
}
```

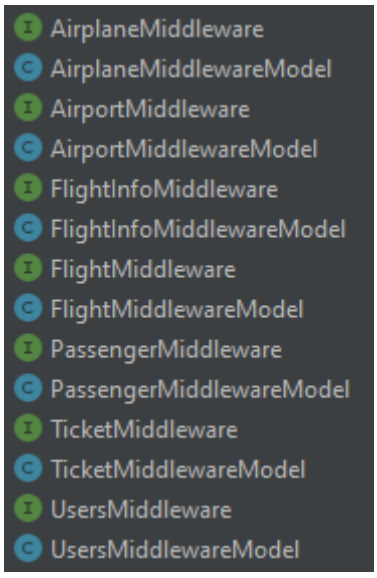
Code 6: shows the implementation of service for calling flights



Then, the requested service is called and method is invoked. *HttpClient* is created, content of http message serialized into *JSON* and send as a content in invoked POST method. It is supplied with full URI, which contains port and controller mapping plus arguments. Then a response is awaited, deserialized from *JSON* and returned to the client.

4.3 **Middleware Model** (Jan Vasilcenko)

Models (Middleware) are interfaces, which hold all the logic for the program. They are implemented by models and divided into fields, by objects they are working with.

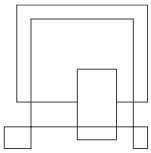


Code 7: shows all of the middleware models

For example, *FlightInfoMiddleware* holds these methods.

```
List<FlightInfo> getClosestFlights(Timestamp departure,Timestamp dearptureback,String fromwhere,String whereto,int numberofpeople);
FlightInfo getFlightInfo(int flightid);
List<FlightInfo> getMyFlightInfos(String email);
List<FlightInfo> getMyFlightInfosFinished(String email,String status);
List<FlightInfo> getAllFlightInfos();
String checkIfNotCancelled(String email);
String checkIfNotDelayed(String email);
```

Code 8: shows the middleware interface

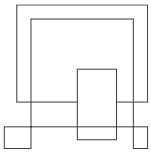


They are implemented in *FlightInfoMiddlewareModel*, so that method *getClosestFlights* look like this.

```
@Override public List<FlightInfo> getClosestFlights(Timestamp departure,
    Timestamp dearptureback, String fromwhere, String whereto,
    int numberofpeople)
{
  Date date = Date.valueOf("0001-01-02");
  if (dearptureback.before(date))
  {
    Request request = new Request( type: "GETDepartureByName", fromwhere);
    try
    {
      List<Departure> allDepartures = (List<Departure>) client
          .request(request).getArg();
      request = new Request( type: "GETArrivalByName", whereto);
      List<Arrival> allArrivals = (List<Arrival>) client.request(request)
          .getArg();
      return ManageFlightSearch(allDepartures, allArrivals, fromwhere,
          whereto, departure, numberofpeople);
    }
    catch (IOException | ClassNotFoundException e)
    {
      e.printStackTrace();
    }
  }
  else
  {

  }
  return null;
}
```

Code 9: shows the implementation of viewFlights method in middleware model

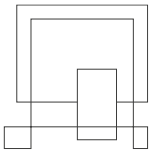
In this method, it is checked that the flight is one-way and then the Departures and Arrival are obtained from the persistence layer. Finally, there is returned a result from *ManageFlightSearch()* method.



```
private List<FlightInfo> ManageFlightSearch(List<Departure> allDepartures,
    List<Arrival> allArrivals, String fromwhere, String whereto,
    Timestamp departure, int numberOfPassengers)
{
  List<FlightInfo> flightInfos = new ArrayList<>();
  List<Arrival> wishedArrivals = new ArrayList<>();
  List<Departure> wishedDepartures = new ArrayList<>();
  for (int i = 0; i < allDepartures.size(); i++)
  {
    if (allArrivals.get(i).fromWhere.equals(whereto) && allDepartures
        .get(i).fromWhere.equals(fromwhere))
    {

      wishedArrivals.add(allArrivals.get(i));
      wishedDepartures.add(allDepartures.get(i));
    }
  }
  for (int i = 0; i < wishedArrivals.size(); i++)
  {
    if (wishedDepartures.get(i).flightID == wishedArrivals.get(i).flightID)
    {
      Request request = new Request( type: "GETFlightByID",
          wishedArrivals.get(i).flightID);
      try
      {
        flightInfos.add(
            new FlightInfo((Flight) client.request(request).getArg(),
                wishedArrivals.get(i), wishedDepartures.get(i)));
      }
      catch (IOException | ClassNotFoundException e)
      {
        e.printStackTrace();
      }
    }
  }
```

Code 10: shows the ManageFlightSearch() method part 1

ManageFlightSearch() method has two parts in first part only those flights that match departure location and arrival location are chosen.



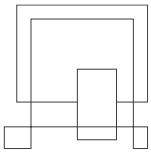
```
Date wishedDepartureDateSQL = new Date(departure.getTime());
LocalDate wishedDepartureLocalDate = wishedDepartureDateSQL.toLocalDate();
List<FlightInfo> flightInfosWithRightDate = new ArrayList<>();
for (int i = 0; i < flightInfos.size(); i++)
{
  Date dateOfDepartureSQL = new Date(
      flightInfos.get(i).departure.departureTime.getTime());
  LocalDate localDateOfDeparture = dateOfDepartureSQL.toLocalDate();
  if (wishedDepartureLocalDate.minusDays(1).equals(localDateOfDeparture)
      || wishedDepartureLocalDate.minusDays(2).equals(localDateOfDeparture)
      || wishedDepartureLocalDate.equals(localDateOfDeparture)
      || wishedDepartureLocalDate.plusDays(1).equals(localDateOfDeparture)
      || wishedDepartureLocalDate.plusDays(2).equals(localDateOfDeparture))
  {
    flightInfosWithRightDate.add(flightInfos.get(i));
  }
}
List<FlightInfo> finalFlightInfos = new ArrayList<>();
for (int i = 0; i < flightInfosWithRightDate.size(); i++)
{
  if (flightInfosWithRightDate.get(i).flight.numberOfSeatsRemaining
      - numberOfPassengers >= 0)
  {
    finalFlightInfos.add(flightInfosWithRightDate.get(i));
  }
}
return finalFlightInfos;
```

Code 11: shows the part 2 of ManageFlightSearch() method

Then dates are checked, with tolerance of three days around clients chosen date. The result is then returned and displayed on client side.

4.4 **Socket Communication** (Jan Vasilcenko)

This communication occurs in our system between middleware (client) and persistence (server) layer. They communicate over *TCP*, to ensure that the data is safely transferred



between those two and that both sides understand each other. To communicate *DTO* is used, in this case it is *Request* class.

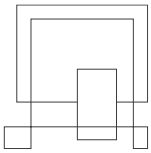
```
public class Request implements Serializable
{
  private String type;
  private Object arg;
  private Object arg2;
  private Object arg3;
  private Object arg4;
  private String stringArg;

  public Request(String type, Object arg)
  {
    this.type = type;
    this.arg = arg;
  }

  public Request(String type, Object arg,String stringArg)
  {
    this.type = type;
    this.arg = arg;
    this.stringArg = stringArg;
  }
```

Code 12: shows the Request class

This class has String type which determines, what action is requested and many arguments, in which objects can be inputted. It of course implements serializable so that it could be converted to bytes and send over network.

4.4.1 **Client-side** (Jan Vasilcenko)

In middleware requests are created, with wished type and objects.



```
@Override public void addFlight(Flight newFlight, Arrival newArrival,
    Departure newDepartures)
{
  newArrival.arrivalTime.setTime(newArrival.arrivalTime.getTime()-3600000);
  newDepartures.departureTime.setTime(newDepartures.departureTime.getTime()-3600000);
  try
  {
    client.request(
        new Request( type: "ADDFlight", newFlight, newArrival, newDepartures));
  }
  catch (Exception e)
  {
    e.printStackTrace();
  }
}
```

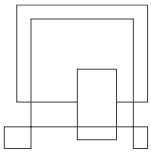
Code 13: shows the addFlight() implementation in middleware model

For example, in method *addFlight()* request with type “ADDFlight” is created and then is supplied with three objects, *newFlight, newArrival* and *newDeparture*. Then client is called, and request method is invoked with request argument.

```
public Request request(Request request)
    throws IOException, ClassNotFoundException
{
  Socket socket = new Socket( host: "localhost",  port: 2910);
  ObjectOutputStream outToServer = new ObjectOutputStream(socket.getOutputStream());
  ObjectInputStream inFromServer = new ObjectInputStream(socket.getInputStream());
  outToServer.writeObject(request);
  return (Request) inFromServer.readObject();
}
```

Code 14: initializing client socket and writing the request out to the server socket

The client socket is then sending the request to persistence and then is waiting for an answer in shape of another request.

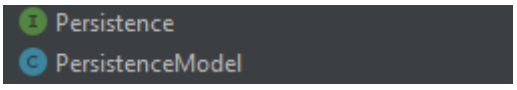


4.4.2 **Server-side** (Jan Vasilcenko)

```
@Override public void run()
{
  try
  {
    Request request = (Request) inFromClient.readObject();
    Request result = persistence.handlerRequest(request);
    outToClient.writeObject(result);
  }
  catch (IOException e)
  {

  }
  catch (ClassNotFoundException e)
  {
    e.printStackTrace();
  }
}
```

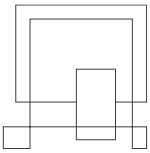
Code 15: run method for receiving request and calling persistence to handle it

On server side (persistence), the socket is waiting and accepting client sockets. Then if the request arrives persistence is called with method *handlerRequest()*.



Code 16: shows the Persistence interface and PersistenceModel class

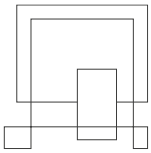
Persistence is an interface with one method, which is implemented in Persistence Model.



```
public class PersistenceModel implements Persistence
{
  public static final String JDBCURL = "jdbc:postgresql://localhost:5432/Airline?currentSchema=airline";
  public static final String username = "postgres";
  public static final String password = "123456789";

  private FlightsDAO flightsDAO;
  private UsersDAO usersDAO;
  private AirplanesDAO airplanesDAO;
  private AirportsDAO airportsDAO;
  private DepartureDAO departureDAO;
  private ArrivalDAO arrivalDAO;
  private PassengerDAO passengerDAO;
  private TicketDAO ticketDAO;

  public PersistenceModel()
  {
    this.airportsDAO = new AirportsDAOImplementation(JDBCURL, username,
        password);
    this.flightsDAO = new FlightsDAOImplementation(JDBCURL, username, password);
    this.usersDAO = new UsersDAOImplementation(JDBCURL, username, password);
    this.airplanesDAO = new AirplanesDAOImplementation(JDBCURL, username,
        password);
    this.departureDAO = new DepartureDAOImplementation(JDBCURL, username,
        password);
    this.arrivalDAO = new ArrivalDAOImplementation(JDBCURL, username, password);
    this.passengerDAO = new PassengerDAOImplementation(JDBCURL, username,
        password);
    this.ticketDAO = new TicketDAOImplementation(JDBCURL, username, password);
  }
```

Code 17: shows the PersistenceModel class

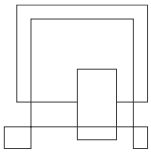
In *PersistenceModel, JDBCURL* for database is set, same for username and password. Then it has instances of all the Data Access Objects, initializing them with all the information.



```
@Override public Request handlerRequest(Request request)
{
  Request request1 = new Request( type: null,  arg: null);
  switch (request.getType())
  {
    case "GETUser":
      request1 = new Request( type: null,
          usersDAO.getUser(request.getArg().toString()));
      break;
    case "REGISTERUser":
      request1 = new Request( type: null, usersDAO.addUser((User) request.getArg()));
      break;
    case "ADDFlight":
      flightsDAO
          .addFlight((Flight) request.getArg(), (Arrival) request.getArg2(),
              (Departure) request.getArg3());
      break;
    case "GETFlights":
      request1 = new Request( type: null, flightsDAO.getFlights());
      break;
    case "GETPlanes":
      request1 = new Request( type: null, airplanesDAO.getPlanes());
      break;
    case "GETAirports":
      request1 = new Request( type: null, airportsDAO.getAirports());
      break;
    case "GETAirplaneByType":
      request1 = new Request( type: null,
          airplanesDAO.getAirplaneByType(request.getArg().toString()));
      break;
    case "GETIATACodeByName":
      request1 = new Request( type: null,
          airportsDAO.getAirportByName(request.getArg().toString()));
      break;
```

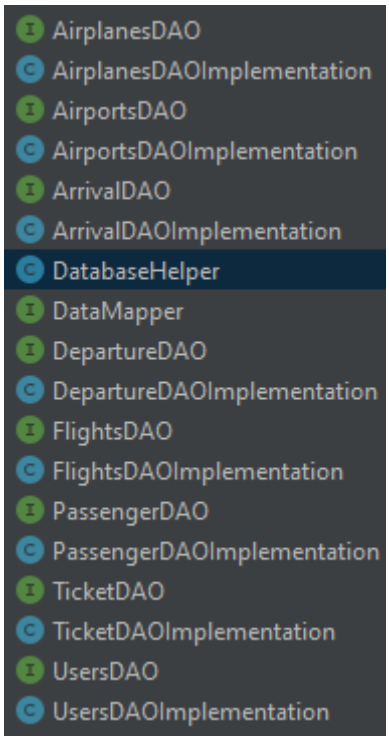
Code 18: shows the handlerRequest() method in the PersistenceModel

The final part of custom-made protocol is the method *handlerRequest()*, which reads all request types and does things invoked by client. Calling proper DAO and performing action. In some cases, returning something in the form of *Request*.



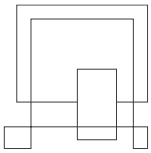
4.5 **Database Access** (Jan Vasilcenko)

JDBC is used for access to the database. There are two important objects. First is *DatabaseHelper*, which prepares statements and holds general logic to map object from database to Java objects and *DataMapper*, which is used to map these objects. For access to database Data Access Objects are used, divided by each table of database.



Code 19: shows the DAO interfaces and classes

Each DAO holds methods, to be called by *PersistenceModel*.



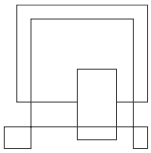
```
public class FlightsDAOImplementation implements FlightsDAO
{
  private DatabaseHelper<Flight> helper;
  private ArrivalDAO arrivalDAO;
  private DepartureDAO departureDAO;
  private TicketDAO ticketDAO;

  public FlightsDAOImplementation(String jdbcURL, String username,
      String password)
  {
    helper = new DatabaseHelper<>(jdbcURL, username, password);
    arrivalDAO = new ArrivalDAOImplementation(jdbcURL, username, password);
    departureDAO = new DepartureDAOImplementation(jdbcURL, username, password);
    ticketDAO = new TicketDAOImplementation(jdbcURL, username, password);
  }
```

Code 20: shows a DAO implementation with DatabaseHelper

For example, *FlightsDAOImplementation* has *DatabaseHelper* instance and some other DAO instances, which are initialized in constructor and is implementing its interface *FlightsDAO*.

```
public static class FlightMapper implements DataMapper<Flight>
{

  @Override public Flight create(ResultSet rs) throws SQLException
  {
    int id = rs.getInt( columnLabel: "flightid");
    int numberOfSeats = rs.getInt( columnLabel: "numberOfSeats");
    int regnum = rs.getInt( columnLabel: "flies");
    String stat = rs.getString( columnLabel: "status");

    return new Flight(id, numberOfSeats, Integer.toString(regnum), stat);
  }
}
```

Code 21: shows create() method in FlightMapper

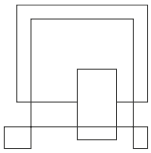


In each DAO there is also a class which is implementing *DataMapper* and code to map that object properly.

```
@Override public void addFlight(Flight newFlight, Arrival newArrival,
    Departure newDeparture)
{
  int biggestID = 0;

  Flight withBiggestId = helper.mapSingle(new FlightMapper(),
       sql: "SELECT * FROM Flight WHERE flightid = (SELECT MAX(flightid) AS flightid from flight)");
  if (withBiggestId != null)
  {
    biggestID = withBiggestId.id;
  }
  helper.executeUpdate(
       sql: "INSERT INTO Flight(flightid,numberofseats,flies) VALUES(?,?,?)",
      (biggestID + 1), newFlight.numberOfSeatsRemaining,
      Integer.parseInt(newFlight.airplaneRegNumber));
  arrivalDAO.addArrival(newArrival,  flightID: biggestID + 1);
  departureDAO.addDeparture(newDeparture,  FlightID: biggestID + 1);
}
```

Code 22: shows addFlight() method

Then, the methods are implemented with *DatabaseHelper* being called, SQL command written and parameters supplied. In this example biggest id already in database is calculated and new flight with the biggest id plus one is inserted into *Flight* table and also departure and arrival are inserted into their respective tables.



5 **Test** (Everybody)

In this section, the tests that were conducted on the system and their results will be presented. The results obtained from this section will be discussed in the next section, Results and Discussion.

The tests can be divided into two sections: white-box tests and black-box tests. White-box tests are conducted to test the internal structures and processes of the system, while the black-box tests are to test the functionalities of the system.

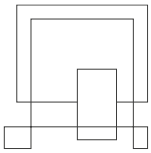
The white-box test chosen was the integration testing. Ideally, the system should have been tested using unit testing and integration testing. Only integration testing was chosen since the airline reservation system is a very user-centric application, meaning the application rarely does anything when it is not in an interaction with a user. Because of this, it would be better to divert the focus more onto black-box tests such as usability testing and test case testing. The usability testing and test case testing are included since these tests are conducted with the end user in mind (the usability testing being tested directly by end users).

5.1 **White-Box Testing** (Jan Vasilcenko and Karrtiigehyen)

5.1.1 **Integration Testing** (Jan Vasilcenko and Karrtiigehyen)

The type of integration testing done is the Big-Bang integration testing, where the testing only took place after the development of all the components (QATestLab, 2018). Alternatively, incremental integration testing could have been done, where the components are tested when they are available, rather than waiting for the development of all of them (Tutorialspoint, 2020). This could have been done using stub and driver classes.

The best idea for the integration testing is would be to do it in the presentation tier, since the presentation tier is dependent on the application tier, and the application tier is in turn dependent on the data tier. This means that if the integration tests succeed in the presentation tier, the methods involved in the other tiers are also working. But the



integration test was ultimately conducted on the application tier. The reasoning behind this is that the application tier houses the business logic of the system, and in a way the data tier will also be tested since the application tier is dependent on the data tier. Another crude justification for the absence of integration testing in the presentation tier is that this tier is only regarding the displaying of the services of the system with no business logic. For more on this, check subsubsection 5.6.4 in the Process Report.

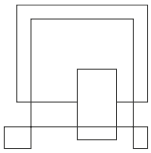
All of the integration tests passed with no failures. The integration tests results can be found in Appendix J, while the code for the integration tests is in Appendix I along with the source code.

5.2 **Black-Box Testing** (Karrtiigehyen, Nicolas Popal and Patrik Horny)

5.2.1 **Test Cases** (Karrtiigehyen)

Test cases are made to check compliances with specific requirements. The type of test case done is the formal test cases, where the information such as preconditions, test steps, input data, and more are included. The test cases will have defined set of inputs, which will give the expected output. The test cases made are based off of the functional requirements, with each requirement having one positive test and one negative test.

An example of a test case is shown below, and all of the test cases can be located in Appendix K.



Test Case ID	Test Scenario	Pre-Condition	Test Steps	Test Data	Expected Results	Actual Results	Pass / Fail
7	Check the creation of flights by the Operator with valid data	Operator is logged in and the airports and airplanes are stored in the database	1. Choose the departure airport 2. Choose the arrival airport 3. Set the departure date and time 4. Set the arrival date and time 5. Choose the airplane	departure airport = Copenhagen Airport arrival airport = Singapor Changi Airport departure date = 10/12/2020 departure time = 13:00 arrival date = 11/12/2020 arrival time = 09:00 Airplane = Boeing 737	Flight is added and should be visisble to operator and customer	As Expected	Pass
8	Check the creation of flights by the Operator with invalid data	Operator is logged in and the airports and airplanes are stored in the database	1. Choose the departure airport 2. Choose the arrival airport 3. Set the departure date and time 4. Set the arrival date and time 5. Choose the airplane	the invalid data here is that the dates are in the past departure airport = Copenhagen Airport arrival airport = Singapor Changi Airport departure date = 02/12/2020 departure time = 13:00 arrival date = 03/12/2020 arrival time = 09:00 Airplane = Boeing 737	An error should be shown and the flight should not be added	As Expected	Pass

Table 5: Test cases for checking the creation of flights

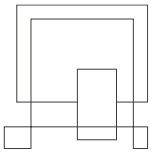
The test case above is regarding the creation of flights. There are two test cases, where one creation of a flight is inputted with valid data while the other is inputted with invalid data.

From these test cases, it can be deduced which functional requirements were fulfilled and which failed. An acceptance test based on the test cases was made to show this.

5.2.2 **Acceptance Testing** (Patrik Horny)

Acceptance testing was done to check whether the system meets the functional requirements.

The acceptance tests are shown in the table below. The user stories that match the numbers can be seen in the Functional Requirements subsubsection 2.1.1.



Functional requirements number	Result (Successful or Failed)
1	Successful
2	Successful
3	Successful
4	Successful
5	Successful
6	Successful
7	Successful
8	Successful
9	Successful
10	Successful
11	Successful
12	Successful
13	Successful
14	Successful
15	Successful
16	Successful
17	Successful
18	Successful
19	Successful
20	Successful
21	Successful

Table 6: Results of the Acceptance test

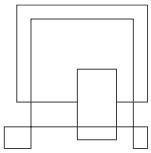
As seen from the table above, all of the functional requirements were met. This outcome is discussed in the Results and Discussion section.

5.2.3 **Usability Testing** (Nicolas Popal)

Usability testing is concerned with the intuitiveness of the system, testing on users who had no prior experience with the system (Nielsen, 1994). The number of users to conduct the test with was chosen to be five, since it is said that the best result come from testing no more than 5 users (Nielsen, 2000).

The usability testing is meant to be iterative, meaning that it should be done multiple times throughout the implementation of the system for constant feedback. In this case, it was not done like that. The feedback acquired from the users can be considered as possible improvements that can be made in the Project Future section, and some of them will be discussed in the Results and Discussion section.

There were a series of scenarios made for the users to follow. Then the users are asked to complete the proposed scenario and give any thought and criticisms upon it.



1. Registering yourself

Inputting information about the customers to create an account to be saved in the system.

Figure 11: A scenario of the usability test

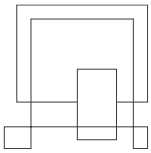
The above figure is one of the examples of the scenario presented to the user. Users were testing system firstly as a customer and then as an operator. Both types of these users had their own scenarios. These scenarios can be found in Appendix L.

The users were not provided with a user manual prior to the testing. The user manual can be found in Appendix M.

Part of the result of one of the testers is shown below. The full usability test can be found in Appendix L.

Action	Result	Comments
Register	Success	• Tester likes that she is informed about inputs through changing colors (red and green). • Phone number has increase and decrease arrow, which is confusing • Birthdate could be defaulted set on different date then year 0 • Tester thinks that not enough information was given during registration (missing passport number, nationality etc.)
Login	Success	• Tester likes that she is informed about inputs through changing colors (red and green). • Forgot password option is missing
Edit user info	Success	• Same as registration
Search flight	Success	• Confused about shortcut of airports • Sorting flights by longest does not make a sense • Missing option one-way flight and direct
Book flight	Success	• Passport number does not work, while trying to add bigger number than 2200000000 • Passport usually has letters • Passport number has decreasing and increasing arrows, which does not make a sense • When expiration date of passport is before actual flight date, customer is not notified • There could be a middle name as well

Table 7: Result of one of the testers for the usability test

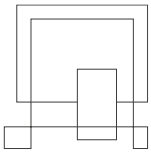


One of the biggest criticisms from all the test subjects was during the cancellation of the flight. When testers cancelled their flight, no confirmation about cancelling was displayed and flight was cancelled. Another big criticism was about searching flight. Most of the test subjects wanted to look up for flights with return flight, but this functionality is not available. Another big criticism came during paying for the ticket. Most of the test subjects were confused about the price and they did not know how the price was counted. Another criticism came during choosing luggage, where no price is displayed. The test subjects would like to be more informed about prices for each luggage as about the overall price for the ticket. The test subjects also noticed an oddity in the insertion their passport number, where any number bigger than approximately 2200000000 could not be inputted.

There were also two other notable criticisms made by the test subjects. One was that filling out the credit card information at the end of booking each flight ticket was cumbersome. The other is that the notification that the customers receive when their booked flight is delayed or cancelled on takes place when the login operation is performed.

Other than these criticisms, the test subjects found the airline reservation system to be intuitive and easy to navigate.

These problems and criticisms will be further discussed in the Results and Discussion, and Project Future sections.



6 **Results and Discussion** (Jan Vasilcenko and Karrtiigehyen)

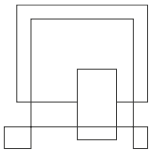
The acceptance test proved that all of the functional requirements were met. The usability test also informs that the test subjects found the system to be intuitive and easy to navigate. All of the non-functional requirements are also fulfilled.

While the airline reservation system is functioning and operating as intended, there are some flaws highlighted by the testings.

The integration test showed that the flow of the data through the program is working fine, but as mentioned in the integration testing section, the integration test was made in the application tier. Ideally, the integration test should have taken place in the presentation tier. Also, unit testing was ignored this time because of the reasons stated in the Test section. Another flaw with the integration test was that it was conducted at the end of the system's development, instead of testing it incrementally using stubs and drivers.

As mentioned in the Test section, the airline reservation system is a very user-centric system. Because of this, much emphasis was put upon the usability test, and many useful criticisms were garnered from the test subjects. One of the biggest criticisms by the test subjects was the lack of confirmation when cancelling booked flights from the customer's point of view. This can be easily fixed by adding a confirmation box when a customer wants to cancel their flight. Another big criticism from the test subjects was that there was no option for choosing round-trip flights, only one-way flights. This is because it was included in the delimitations that round-trip flights will not be considered. The test subjects also noticed that when filling out their passport numbers while booking a flight, the passport number has to be smaller than 2200000000. This is because it the passport number was implemented as an *int* in *Java*, which meant that the highest possible number with this is 2147483647 (Runestone, 2020). An easy fix would be to change the passport number's type to *long.*

Other big criticisms from the usability test were from the payment for the flight tickets at the end of booking flights. The main problem of which was that the test subjects were confused with how the price was deduced. This problem arises because the price is deduced at random since it was included in the delimitations that nothing will be included



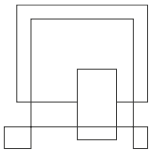
money or transaction-wise. Another problem with the payment was that the users must fill out their credit card information each time they book a flight. This was noted to be cumbersome. A way around this is to add a functionality where the users can store their credit card information. This can be done adding an entity *Credit Card* in the database, connected to the *User* entity with a one-to-many relationship.

Finally, some test subjects noticed that when a flight is delayed or cancelled, the notification received was only activated on performing login. This is due to the implementation of the notification in the *SearchFlight* page, where the customer is notified each time a page is refreshed or changed when the customer is logged in. This could become an annoyance to users, so it was changed so that the notification is only activated once when the customer logs in. Alternatively, an indirect communication technique such as a publish/subscribe system could have been used.

There were other minor criticisms from the usability test, but the criticisms discussed above were the recurring criticisms from the test subjects. Other than these, the test subjects found the system intuitive and easy to use.

A gripe from a design perspective can be found within the implementation of the sockets. Currently, custom requests have to be made with a type in the client-side (middleware) in order to make a new functionality. On the server-side (persistence layer) a new case using switch have to be added in order to receive the request. This could be cumbersome for future expansions of the program. So, another approach for the sockets would have been better.

The possible security mechanisms that can be placed for this system was discussed in the subsection 3.4, but of these only three security mechanisms were implemented. One being the security mechanism to prevent SQL injections by using prepared statements. The second one being the encryption of the URL, and another is implementing TLS. It can also be argued that by implementing TLS, all the other security mechanisms were also achieved.



7 **Conclusion** (Jan Vasilcenko and Karrtiigehyen)

Overall, the project was a resounding success on fulfilling the purpose of this project, which is to help customers book flight tickets and operators manage the reservation system with ease.

Some limitations were made, such as excluding round-trip flights, to define the scope of the project.

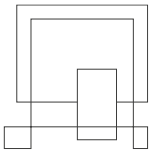
The project was a distributed, heterogeneous system with a three-tier architecture, and is programmed in *C#* and *Java.* The tiers (presentation, application, and data) communicate with each other using webservices and sockets. A persistence layer in the data tier communicates to a database using *JDBC.*

Design patterns such as singleton, DAO, and adapter patterns are used, to solve common problems. They are also reusable and overall general solutions to some problems.

SOLID principles were used as guidelines for designing the system, so that dependencies are reduced, it is highly cohesive, context independent, and easy to divide into separate standalone components and modify. The DRY rule was also used to avoid code repetition, but some code repetition is still present. An alternative solution for this is to command design pattern.

The database is made to store the data, and normalized into third normal form so that there are no partial and transitive dependencies on the primary keys.

The acceptance test showed that all of the functional requirements were met. The usability test showed that the test subjects found the airline reservation system to be easily navigable and intuitive from both customer and operator’s perspectives, which was the goal of the entire project. Since this is addressed and solved, it could be said that this project is a success.



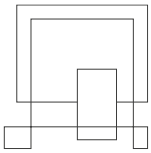
8 **Project Future** (Jan Vasilcenko and Karrtiigehyen)

There are multiple improvements that can be added in the future. Some of these improvements are derived from the integration test and usability test. Others are to expand and add some features to improve the functionality of the system.

The integration test should have been in the presentation tier as opposed to the application tier (end-to-end test). Unit tests will also be a good addition to validate that each unit of the program performs as designed.

While the round-trip flights feature was included in the delimitation, it can be implemented in the future since it was noted by many test subjects to be missing. The same can be said with the transactions with money or some type of currency. This feature can already be seen crudely implemented to an extent with the credit card information. Speaking of the credit card information, a feature storing the credit card information can also be added. The implementation of the notification of the delay and cancellation of flights could be changed to a publish/subscribe system.

As discussed in the Results and Discussion section, the implementation of the sockets could be expanded upon, possibly creating a custom made RMI instead. The webservices, instead of just using RPC, RESTful webservices could also be added.



9 Sources of Information

1. Altexsoft, 2019. *History of Flight Booking: CRSs, GDS Distribution, Travel Agencies, and Online Reservations*. [online] Available at: <https://www.altexsoft.com/blog/travel/history-of-flight-booking-crss-gds-distribution-travel-agencies-and-online-reservations/> [Accessed 08 December 2020]

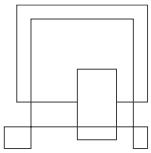
2. ICAO, 2020. *Economic Impacts of COVID-19 on Civil Aviation.* [online] Available at: <https://www.icao.int/sustainability/Pages/Economic-Impacts-of-COVID-19.aspx> [Accessed 08 December 2020]

3. Blessing, A., Abisoye, O., Umar, A., 2017. *Challenges of Airline Reservation System and Possible Solutions (A Case Study of Overland Airways)*. [pdf] I.J. Information Technology and Computer Science. Available at: <https://www.researchgate.net/profile/Abisoye_Blessing/publication/312923998_Challen
ges_of_Airline_Reservation_System_and_Possible_Solutions_A_Case_Study_of_Overland_Airways/links/5c680b454585156b57014486/Challenges-of-Airline-ReservationSystem-and-Possible-Solutions-A-Case-Study-of-Overland-Airways.pdf> [Accessed 30 September 2020].

4. Aviation Job Search, 2012. *Aviation Planner: Job Description*. [online] Available at: <https://blog.aviationjobsearch.com/aviation-planning-job-description/> [Accessed 30 September 2020].

5. Tutorialspoint, 2020. *Data Access Object Pattern*. [online] Available at: <https://www.tutorialspoint.com/design_pattern/data_access_object_pattern.htm
>[Accessed 25 November 2020].

6. Refactoring Guru, 2020. *Command*. [online] Available at: <https://refactoring.guru/design-patterns/command> [Accessed 25 November 2020]



7. Connolly, T. and Begg, C., 2015. Database Systems A Practical Approach to Design, Implementation, and Management. 6th edition. Essex: Pearson Education Limited.

8. PostgreSQL, 2020. *PostgreSQL: The world's most advanced open source database*. [online] Available at: <https://www.postgresql.org/> [Accessed 02 December 2020].

9. PostgreSQL JDBC, 2020. *PostgreSQL JDBC About.* [online] Available at: <https://jdbc.postgresql.org/about/about.html> [Accessed 02 December 2020].

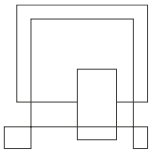
10. Spring, 2020. *Testing the Web Layer*. [online] Available at: <https://spring.io/guides/gs/testing-web/> [Accessed 02 December 2020]

11. QATestLab, 2018. *Big Bang Testing.* [online] Available at: <https://qatestlab.com/resources/knowledge-center/big-bang-testing/> [Accessed 08 December 2020]

12. Tutorialspoint, 2020. *Incremental Testing*. [online] Available at: <https://www.tutorialspoint.com/software_testing_dictionary/incremental_testing.htm> [Accessed 08 December 2020]

13. Nielsen, J., 1994. *Usability Engineering*. Unknown: Academic Press Inc.

14. Nielsen, J., 2003. *Why You Only Need to Test with 5 Users*. [online] Available at: <https://www.nngroup.com/articles/why-you-only-need-to-test-with-5-users/> [Accessed 31 May 2020].



10 Appendices

A – Project Description

B – Use Case Descriptions

C – System Sequence Diagrams

D – Package Diagram

E – Class Diagram

F – Sequence Diagrams

G – Conceptual Model of the Database

H – Logical Model of the Database

I – Source Code

J – Integration Test Results

K – Test Cases

L – Usability Test

M – User Manual

N – Installation Guide

R – Link to Github

S – Link to Video Demonstration