

Large System Development - report

Group 4 - backend:

- Alexander Nielsen
- David Carl
- Hazem Saeid
- Tjalfe Møller

Group 4 - frontend:

- Murched Kayed
- Mikkel Lindstrøm
- Murad Sarker
- Mathias Bartels Jensenius
- Rasmus Balder Nordbjærg

Teacher: Anders Kalhauge

Github repository:

<https://github.com/DavidCarl/LargeSystemsDevelopment>

Handin date: 18/12-19

Table of content

Introduction	3
System requirements	3
Use case model	4
Use case roles	4
Travel agent	4
3rd party system	5
Use case descriptions	5
Get flight offer	5
Make a booking	5
See a booking	6
Cancel a booking	7
Sub-system sequence diagrams	7
Contract	10
Logical data model	10
Setting up the contract	11
Artifactory	12
Development process	13
Backend	15
Software architecture	15
Software design	15
Software implementation	16
Evaluation	17
Frontend	17
Introduction to the frontend	17
JSP & Servlets	17
Using the contract	17

Coding standards	18
Front-end Process	18
Branching strategy	19
Encapsulating the contract	21
View models	21
Backend connector	22
Endpoint factory	22
Front-end Tests	23
Code Coverage	24
Logging	25
Evaluation for frontend	25
CI/CD	26
What we did	26
What we want in the future	27
Maintenance and SLA	29
Self evaluation	31
Communication	31
Contract rework	31
Missing exceptions	31
Interfaces for DTOs	32
Conclusion	33

Introduction

We were supposed to make one of 3 applications, in our case we choose to make an Airplane booking system. A part of this task was to attempt to create an older service with older technologies. This was to better prepare us for what we could experience in the real world, where everything isn't the newest and flashiest technologies. This set some limitations on what we could use and challenged us to learn older and more outdated ways to do things.

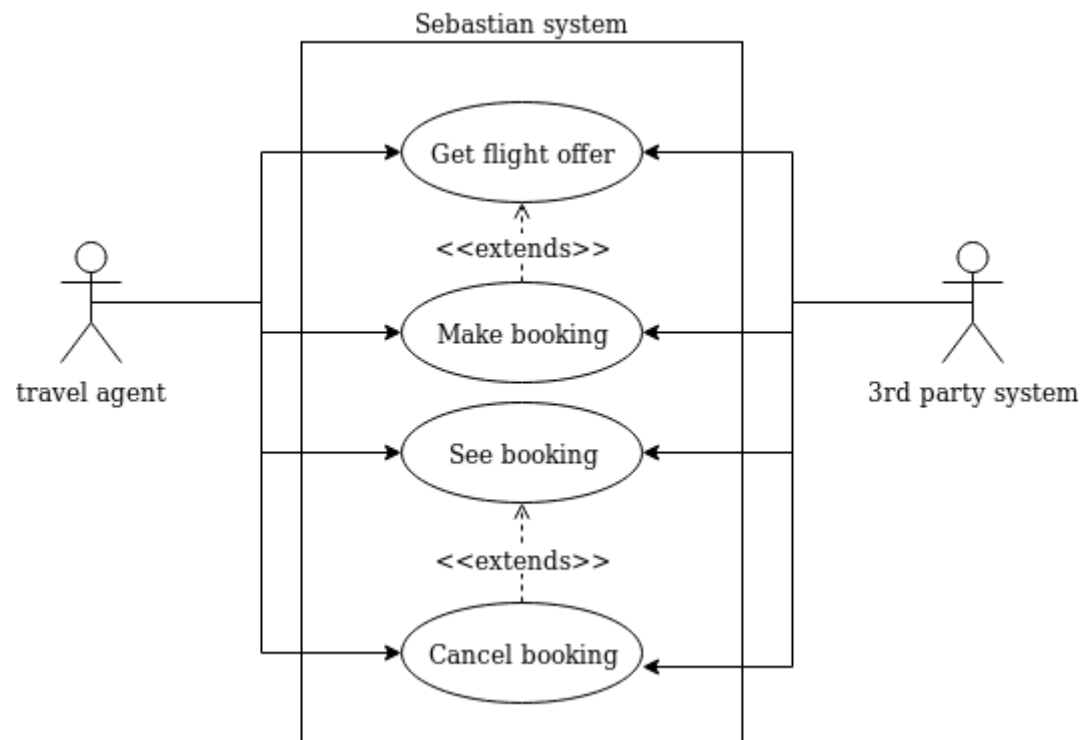
We were a group of 9 people where we got dealt up into two groups back-end and front-end, which both of the groups should agree on one contract to follow.

System requirements

As a user you should be able to go in and search for a trip by airplane. Their search had to fulfill some of our requirements which is give us outbound date, outbound airport as a constant requirement and inbound date and inbound airport if the customer decided to take a round trip.

Another requirement on our system is that the user that books the flights are not the customers themselves, its travel agencies. So there had to be a login system where the travel agent should identify themselves with a username, password and agency number.

Use case model



We identified 4 use cases from project description. These were based on the following verbs “Show (a time schedule...)”, “Make (a booking...)”, “See (a booking ...)”, “Cancel (a booking)”. Although some names might have changed from the original verb due to clarification the use cases remains the same.

Before making a booking you need to get the flight offers. Therefore “Make booking” extends “Get flight offer”. When you want to cancel a booking you first need to find it in the system by following the “See booking” use case steps.

Use case roles

Travel agent

Name: Travel agent

Description: Employee working in a travel agency interacting with the system using the web application.

Responsibilities:

- Make sure that the correct information are entered in the system

3rd party system

Name: 3rd party system

Description: 3rd party systems using a web API to perform actions in the system.

Responsibilities:

Use case descriptions

Get flight offer

Name: Get flight offer

Description: Display an overview of the flight offers, that matches the search parameters.

Actor: Travel agent

Precondition: No preconditions.

Main success scenario:

1. The actor goes to the time schedule overview.
2. The actor enters departure airport, destination, departure date, destination date and whether it is a one-way or return ticket.
3. The system finds the schedules between the 2 airports.
4. The user choose one of the time schedules.
5. The system shows information about the schedule.

Alternative scenarios:

Make a booking

Name: Make a booking

Description: The actor fills in necessary information and creates a booking.

Actor: Travel agent

Precondition: No preconditions.

Postcondition: The booking is stored in the DB

Main success scenario:

1. "Show a time schedule"
2. The actor sends passenger information.
3. The system confirms the booking.

Alternative scenarios:

1. Not enough seats in flight
 - a. The actor sends passenger information.
 - b. The system responds that the selected flight doesn't have the capacity for the number of passengers entered.

See a booking

Name: See a booking

Description: The actor finds a specific booking in the system by PNR

Actor: Travel agent

Precondition: No preconditions.

Main success scenario:

1. The actor navigates to bookings page.
2. The actor enters the booking PNR in the search field.
3. The system displays the information about the booking.

Alternative scenarios:

1. Booking does not exist

- a. The actor navigates to bookings page.
- b. The actor enters the booking PNR in the search field.
- c. The system responds that the no booking with the PNR exists

Cancel a booking

Name: Cancel a booking

Description: The actor cancels a specific booking

Actor: Travel agent

Precondition: No preconditions.

Main success scenario:

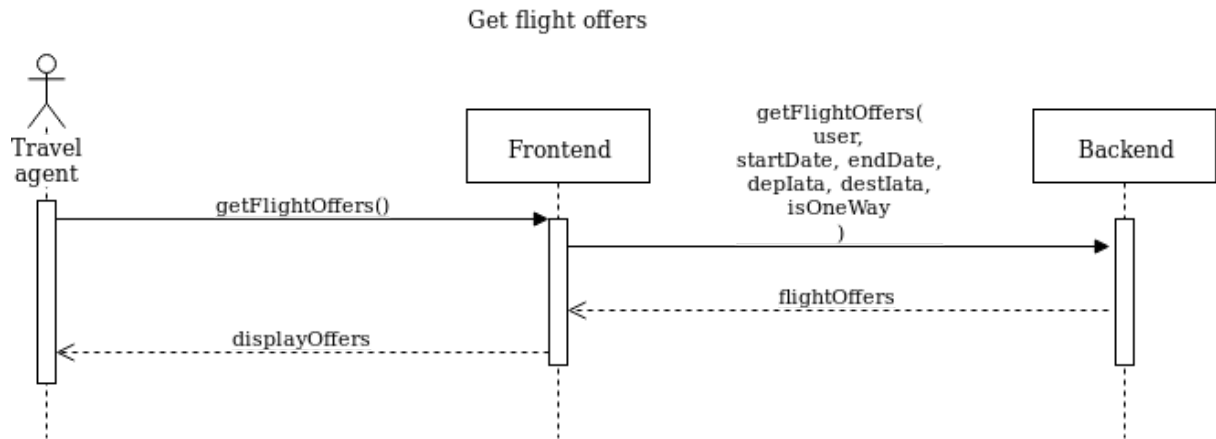
1. "See a booking"
2. The actor clicks on "cancel booking"
3. The system cancels the booking
4. The system confirms to the user that the booking was cancelled

Alternative scenarios:

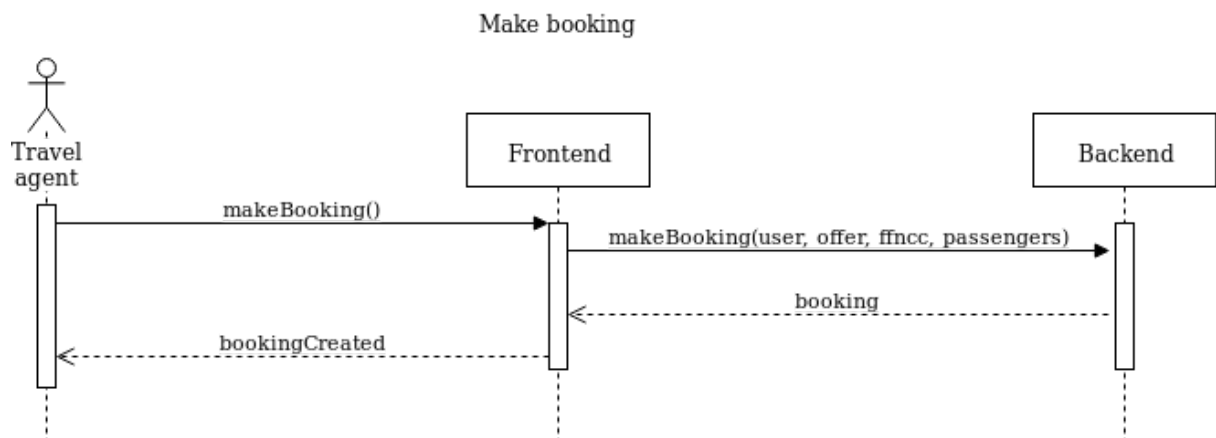
1. Too late to cancel
 - a. "See a booking"
 - b. The system does not show a cancel button since the first flight in the booking will happen within 24 hours.

Sub-system sequence diagrams

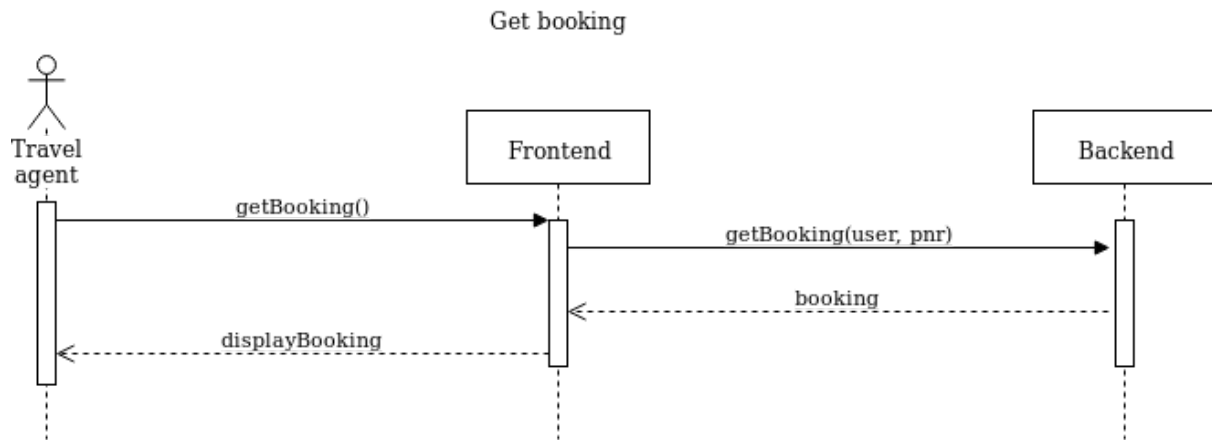
In order to map out the communication between the frontend and the backend we created subsystem sequence diagrams for the use cases.



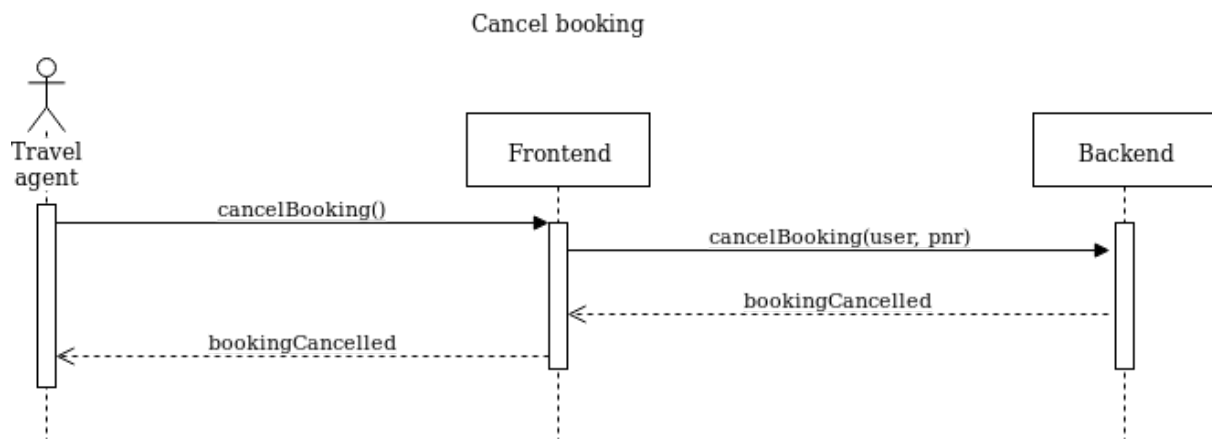
In the “Get flight offer” the user enters the start date, end date, departure IATA, destination IATA and whether it is a one way ticket or not. The frontend also attaches the travel agent that is logged in in order for the backend to perform audit logging. The backend should then return a number of flight offers that meets these search criteria.



When making a booking the frontend sends the logged in travel agent, the flight offer, the FFN/credit card number and information about the passengers for the booking. If successful the backend returns the created booking.



In “Get booking” the frontend sends the logged in travel agent alongside the PNR for the booking. If the booking is found the backend returns the booking.



In order to cancel a booking the frontend needs to provide the PNR of the booking to cancel and the travel agent performing the operation. The backend returns a boolean value which is true if the cancel was successful and otherwise false.

In the contract we ended up adding more methods than what is shown in the SSSDs. However these methods were either too small or too technical to find its place in our use cases.

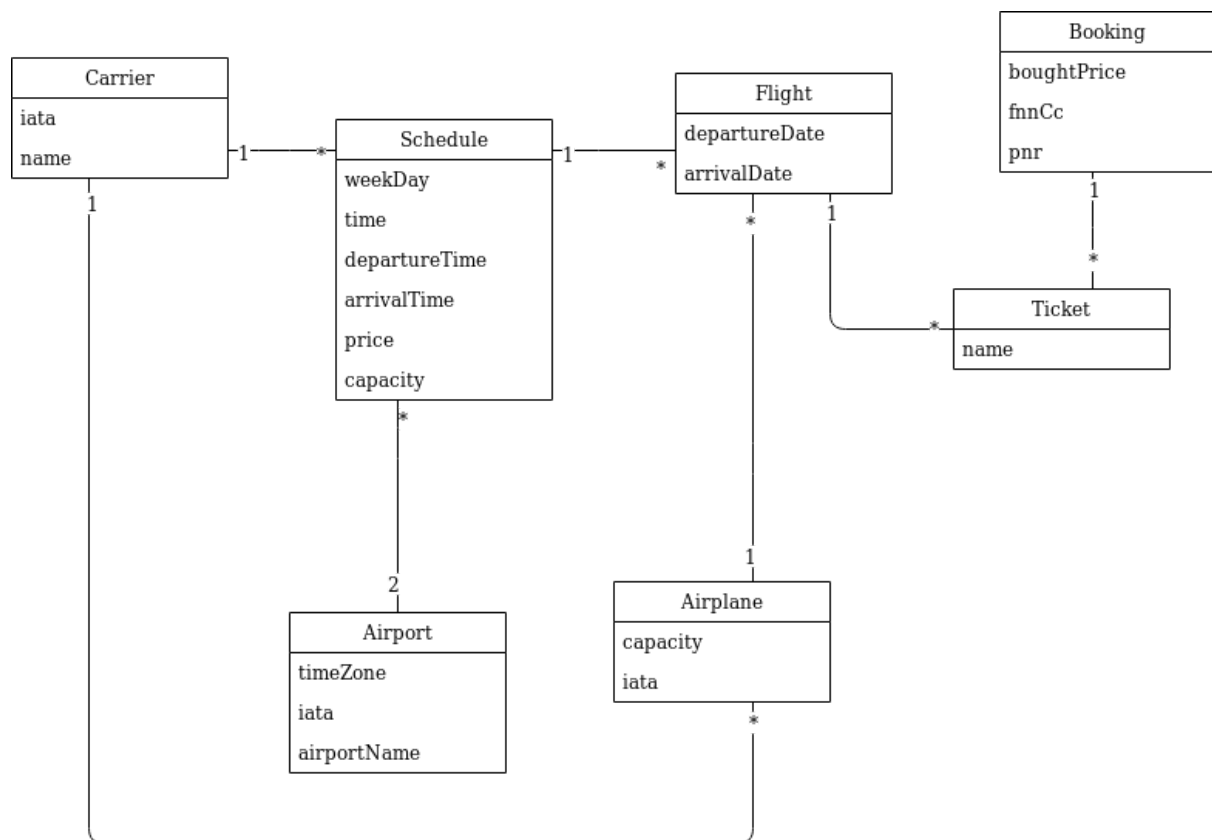
Contract

We were supposed to use Contract-Driven development. This was to enable multiple independent teams to work on multiple things that had to work together when done in the end.

Logical data model

Its a smart thing since it allows our Frontend and Backend to work simultaneously with minimal contact, however it required us to start out with sitting down together and figuring out what endpoints we needed, how our classes should look, and how to communicate with each other. This was a hard part of it since, we pretty much had to nail it the first time, since every time we wanted a change to our contract we had to sit down with both teams and work out what we wanted to change and was it okay to do this, do we want something new into our contract etc.

Our plan was to get an idea of how the data was structured in the system. This helped us understand what was needed to pass around in the system in order to implement the use cases. Having an overview of the conceptual data structures also helped getting an idea of what data was related to the backend subsystem and what was related to the frontend system.



Our Logical Data Model is quite simple: Carriers have many schedules, a schedule only belongs to a single carrier. A schedule always occur on the same weekday at a specific time. The schedule is connected to 2 airports; departure airport and arrival airport. There can be multiple flights from the same schedule but at different dates; one flight one week another flight next week.

When a booking is created the tickets are reserved for the passengers. One booking can book tickets for multiple passengers. All the tickets are related to the booking and the respective passenger. The ticket refers to the flight.

Setting up the contract

One of the big concerns we had was that we were supposed to make contact between our frontend and our backend through something else than REST Endpoints since that was what we were used to do. Suddenly we were given a framework called Bean which we needed to implement for communication between frontend and backend. We understood what was smart about it, since we could (did)

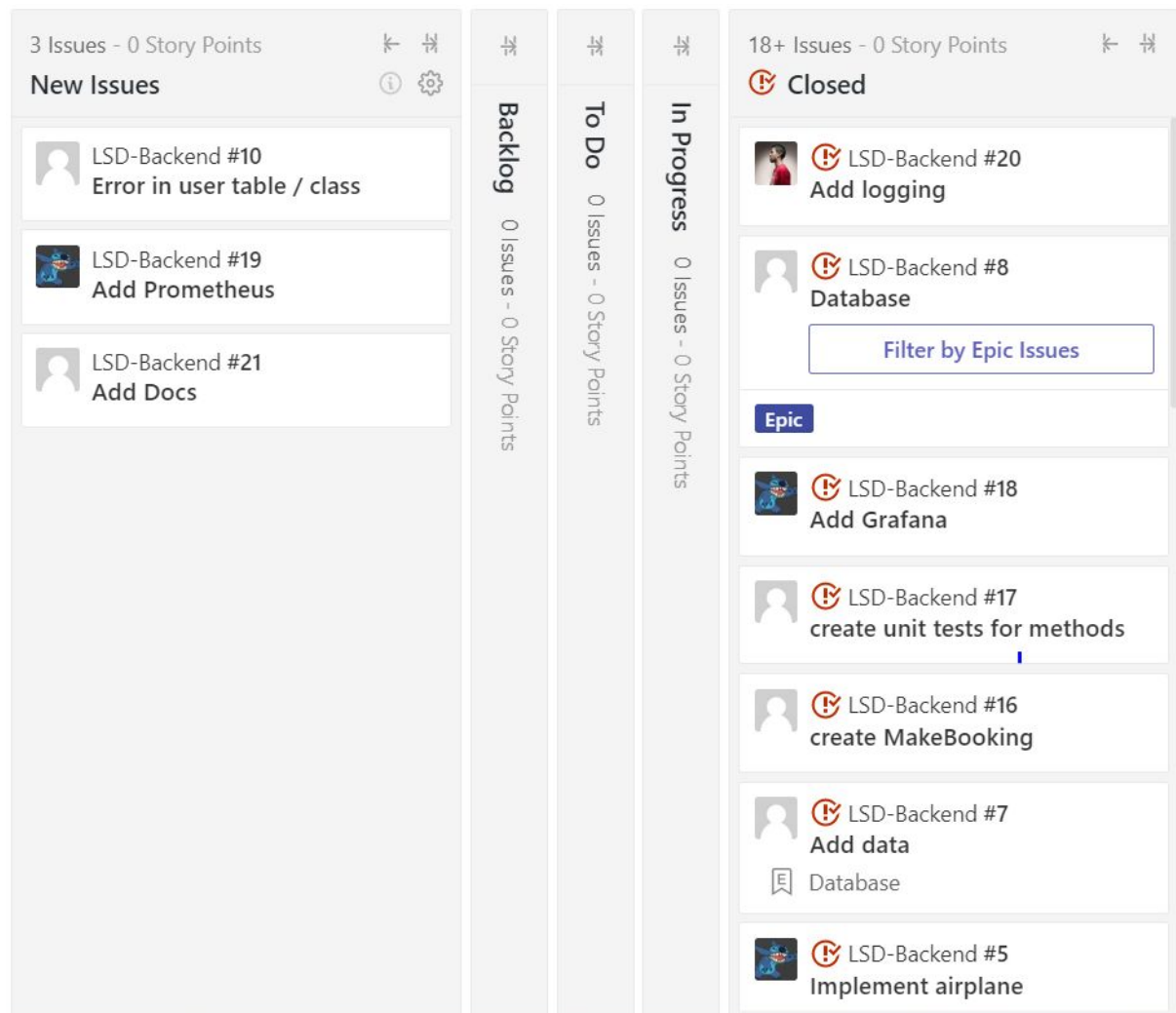
make a contract we had to follow, so it was really hard for us to screw up the communication link, and if we both did it correctly it wouldn't take more than 5 minutes to implement the real backend from some mock state. However after a couple of long days of debugging we managed to get it up and running correctly, this was both from problem of our selection of application server and configuration in our code. We started out with using Tomcat in our Docker images and this was fault number 1, since Tomcat didn't support Bean. After reading up on it we figured out we needed to use Wildfly, however this didn't work out of the box either, we had to change some settings for it to work.

Artifactory

At first we found a way to resolve our issue with sharing our contract through a GitHub repository. It worked well for a couple of weeks but at some point we were no longer able to pull the latest JAR file from github. We tried to debug this issue however we found it troublesome and decided to switch to a JFrog artifactory community edition that had the features we needed for our project. We decided to self host in a Docker container on the same server, this was done to get away from github and hopefully removing these problems we had.

This ended up solving our problem for hosting our contract, however it took quite some time to set it up.

Development process



In our development process we used several tools while developing our system. The tools we used for our development flow are the following, a VCS tool for better collaboration between each other, a issue tracker / kanban board, several CI tool, a communication tool.

For our VCS tool we used github since its based on git, and we all had a good knowledge about how to use git and github. This made it a fairly painless setup process for us to setup and use together.

As our issue tracker / kanban board we used [zenhub](#), which is based on github issues. One of the features that hooked us on this issue tracker compared to so

many other trackers was the possibility to access it straight on the GitHub website by adding their extension to your browser. This made it easy for us to access our issue tracker / kanban board.

As one of our 2 CI (Continuous Integration) tools we used [CircleCi](#). We have it configured to run all tests once we commit to the branch we work on, and run test once we make a pull request. This way we know if the code works (depends on developers making the necessary tests) before we decide to rebase our code base unto our master branch. As for our second CI tool we decided to use [Docker hub](#) for automatically building our Docker image we would then use for deployment. We decided to use docker to simplify our deployment process and safety for our runtime, more on this later.

As our communication tool we decided to use Discord, this decision was based on it both having a feature rich text chat, voice chat, and the possibility to screenshare also if we had to help each other. As for the text chat it had the possibilities to upload images directly and had markdown support to make it easier to share code snippets and more.

So how the workflow then went using our tools would be the following. I go to see what current issues we have available in our kanban board (Zenhub), I then assign myself to let others know that I am working on this issue. Then when I am done with this task I push the branch up to our VSC tool and make a merge request. An independent third party of the group will look through the pull request and either accept or deny it depending on CircleCI result and the code quality. If it's being accepted we perform a rebase into the master branch and Docker Hub would automatically perform a build of the newest master branch. If we want to make a semver release on Docker Hub we should do a Github Release with the version number, Docker Hub will then build the image with that tag.

We also used Artifactory to serve a contract project for frontend and backend, so we always had the up to date version and easy access to it. This was to ensure our

projects would be easy to connect once we came to it and it would work as flawless as possible.

Backend

Software architecture

As we had to split into 2 teams, a frontend and a backend team we sat down together at the start of the project, discussed how the project should look when done. We decided how our Data Transfer Objects classes were supposed to look, so we had the same classes and what things the frontend and the backend should be able to communicate. All of this information we put down into a contract project which we then hosted on a Artifactory so we both could add it as a dependency and always be up to date on changes to the contract.

For communication we between our frontend and backend we decided to use Bean on recommendation from our teachers since we were supposed to emulate working on an old system.

[Picture of the code architecture here]

Software design

We started off by gathering the requirements from the Sebastian GDS whitepaper. Firstly, we spent time discussing the architecture of the database that helped us with an understanding on how to build the system and implement the respective business logic layer.

[Picture of initial DB schema design]

As mentioned in the Contract chapter our communication between the frontend and backend, needed to be through Remote Procedure Calls using a contract interface and Data Transfer Objects, so the frontend knew what method was accessible on the backend and how the objects transferred was structured.

Composition over Inheritance Pattern*

“Here you should sketch your thoughts on the software design *before* you started implementing the system. This includes describing the technical concerns you had about the system before you started development, together with all the technical components you came up with to fix these concerns and meet the requirements.”

Software implementation

In correspondence with the frontend team, we chose to use EJB (Enterprise Java Beans) as our means of communication between the two systems, due to the many features it brings and the ease of use.

In order to increase our development velocity we choose to use an ORM framework to communicate to our database, as this made it easier and faster to query complex objects across multiple tables. For this we choose to use JPA as our ORM framework connected to a MySQL database.

Due to the fact we had chosen to make all our DTO's classes instead of interfaces, meant we had to make new Objects to use as Entities as we needed to add annotations for the variables in order to use JPA.

In the beginning we tried to make Entity object that would be more optimized for the database, as some of our DTO's that could be separated out of a single object and shared between multiple objects. Our plan was to use a Model Mapper to convert our entities to the DTO objects, but after hours of trial and error we had to scrap that plan and edit our Entity classes to have a more one-to-one layout compared to our DTO's.

Evaluation

Already from the start we were 1 man down compared to the frontend team, since we were 4 developers and the frontend team where 5. Once we were supposed to start developing our system we suddenly got no response from Alexander in our group so we ended up just being 3 persons.

Frontend

Introduction to the frontend

JSP & Servlets

We developed the frontend with the help of JSP and servlets as we had to mimic an old legacy system this was the ideal way to do it. We had previous experience in using JSP and servlets from our AP degree in computer science, so we decided to use this as we already knew how it worked. We use JSP to create and render our HTML pages and run our java code directly on the given page with JSTL. The Servlets will have the responsibility to save the data in the session, so that we can access it across all pages. It also does all the redirecting between the different pages using the built in request dispatcher in servlets, and makes sure to get the needed parameters everytime we switch page using the request.getParameter method.

```
RequestDispatcher requestDispatcher = request
    .getRequestDispatcher("home.jsp");
requestDispatcher.forward(request, response);
```

Using the contract

We have integrated the contract as a maven dependency in our projects POM.xml file and this allows us to access the remote bean interface, which serves as our contract. The Bean interface has every DTO and exposed functionality, which is used for both frontend and backend implementation. In order to make sure you can

pull and push the updated contract you have to put our settings.xml file in your users .m2 folder. You also have to have it in your repository root folder.

In the beginning of our project we didn't have a connection to the backend and the implementation of the backend logic had not been made, so we had to make our own implementation of the bean interface in order to test that our code would behave as we wanted it to. Once the backend was developed we switched over to use the bean interface as it was originally intended.

Coding standards

One of our coding standards we set up for the frontend team in the beginning was that we wanted our naming of our files, variables and methods to reflect what the purpose was of that given method, variable or file. By doing this it would become easier to look back at the code and understand what class serves what purpose for a brand new developer or a developer that has not yet worked on the given feature.

We agreed upon using the servlet request dispatcher to redirect between our different JSP pages, as it is a built in feature in servlets and we could reuse the request dispatcher multiple times and only alter the string input so it would redirect to a specific page if the conditions were met for a given if-statement. We also used the session in order to store necessary data that we had to use later on in our application as this is the easiest way in servlets and we had previous experience doing so.

Front-end Process

As a front-end team we used [scrum](#) as agile process. The first thing we started with is creating some usestories and then cut them into small tasks that we can deal out to our team members, we used clickup.com for that.

The scrum master of the team was Rasmus Balder Nordbjærg, because we all agreed that he is the right one to be that. He was also our contact person between the front-end and the backend group, so if we needed some extra data from the

backend or there is something went wrong with backend, we told Rasmus about it so he can tell them what we need and why.

Example: We had improvements to add to the contract that we needed to do, so before doing it Rasmus took a meeting with the contact person from the back-end to tell him that, so we don't create problems to the backend if we do the change. After Rasmus get the acceptance from the back-end that we can do the chang, we start doing that.



We worked in sprint to complete the user stories. Her is a screenshot of some of our completed user stories from the backlog. You can see that there are three different user stories which was dealt out to at least two different members.

The reason we were minimum two on each user story, is that we could use the agile [pair programming](#) technique, which was very useful for us, so if one of the team could not come over a problem the other members can join to solve it on one workstation.

As you can see the first user story was three members to complete it, because after our estimations and velocities, we agreed that we need so many people to complete such a big user story that contains many tasks.

We made some stands up meetings every morning where the whole team met up either online or physical, to give a status about how it is going with completing the user story.

Branching strategy

We have a repository on Github to serve our development code. This means that we are using git to upload and pull from the repository.

In the start of the project we tested out the Bean implementation and a lot of other stuff so this was our testing phase where all commits went into the master as seen here:



We then tried to implement a branching strategy which made every servlet a branch with all its features. We decided on this branching strategy because we were working in teams of 2 or 3, so instead of having every feature in a branch we decided to only use it for a whole and complete Servlet. We did this to also reduce the amount of merge conflicts that will occur because we then were creating new files with a lot of code and only minor changes to files that the other branches were also adding changes to. You can see the picture below to see how we did a branch for the Login-Servlet.



This was the idea behind our branching strategy, but our usage of the strategy wasn't performed very well, as some members forgot to use their branch or even create a branch for their code and features. As seen in the picture above we created a branch Login-Servlet and developed on the branch, in the meantime some of the other guys made commits on the master, so when we realised this we merged the master into the Login-Servlet branch to sort out all the errors and merge conflicts so this wouldn't happen in the master where we only want code that works smoothly. We then talked about the importance to at least not work on the master, so we

created a development branch where we would merge all the different branches into when they were done and then fix the conflicts there and test it before committing to the master.

Encapsulating the contract

Ideally the contract should not be changed after it was created. However after starting implementing the functionality for the frontend and backend we realized that a few small changes need to be made. This made us realize that we should not count on the data transfer objects from the contract as being usable data models in the frontend. Our solution to this was to try and encapsulate the contract and its DTOs in a layer of abstraction so we would not need to change the frontend related functionality in case of changes happening to the contract.

View models

We decided to create view models as the frontend data structure. The idea here was to model the data in the most structural convenient way for us to display data in the JSP pages and to retrieve data from the web application. Each page had its own view model that contained data to be displayed on that page.

Some of our pages however would need rather complex data structures and so a flat view model object would not suffice. An example of such a page was the flight deals page. The page displays an overview of a varying number of flight offers but if the user were to click on any of them a small window pops up revealing more details about the offer. We therefore needed some general view models like flight and airport that could be nested in the view model for the flight deals page.

We created a class for converting DTO objects to view models and the other way around. The idea was that if changes were made to the contract then the only thing that we needed to change in the frontend was the conversion between DTOs and view models. This would work for smaller changes like adding or removing data fields from the DTOs. However if more structurally drastic changes were to be made this might force the view view models to change as well. But if that big changes were

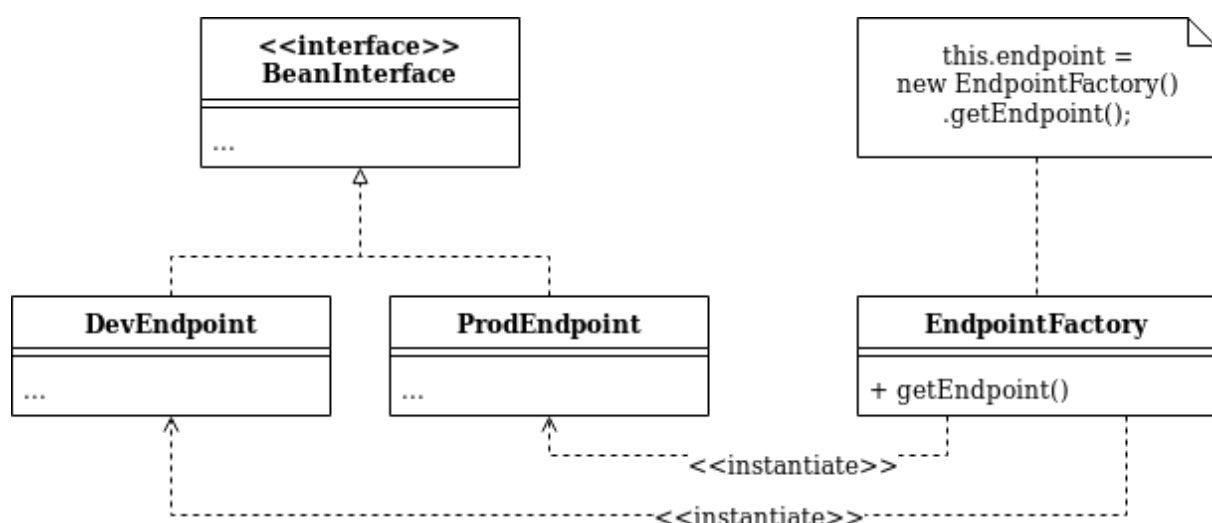
to be made then there is a big chance that the frontend would need to be reworked as well.

Backend connector

As a part of the encapsulation of the contract related code we also went and created an object for handling the RPC connection to backend application. In order to increase testability we decided to create an interface called BackendConnectable and code against the interface instead of the BackendConnector class. The responsibility of the BackendConnector is to call an endpoint implementing the contract interface and then call the necessary objects in order to convert DTOs into view models. The view model would then be returned back to the caller. However BackendConnector was not responsible for creating or establishing an endpoint to the backend.

Endpoint factory

In order to distinguish between production and development environment we made use of the factory design pattern. The idea here is to remove the instantiation of the endpoint from the BackendConnector into a separate class with that only responsibility. We created an EndpointFactory class with the responsibility of creating either a production or development endpoint.



The the EndPointFactory reads from an environment variable “LSD_FE_PROD” whether it should return a DevEndpoint or a ProdEndpoint. If the environment variable is set to 1 then a production endpoint is instantiated, otherwise the development endpoint is instantiated.

Front-end Tests

It is very important to make sure that every functionality of software work together. We have followed various types of the test throughout the whole development cycle. Testing has to be performed both the implementation and integration phase. Testing has been done to check all possible input and output according to the business document. User satisfaction is our ultimate goal so we have check and recheck different scenarios and make sure the stability of system and robustness of code.

In our Airline Reservation system we have test each single module. The main purpose of unit testing is to isolate each part of the program and check the correctness of code. The main focus is the unit test is the implementation that means does the code work according to the developer intended. The conditional work correctly and any error case correctly detected. There are many benefits for unit testing for example faster development because we don't spend much time for debugging. It will also reduce lot of future costs because in software development that the cost is significantly more money fix bug found later in its release than earlier in the development.

Integration Testing sometimes called Integration and testing is the testing process when two or more individual modules integrate together. When we have integrated our backend and frontend the we have check every functionality is working together or not. Integration testing is follows unit testing where each module is tested as a separate unit. The main goal of integration testing is to make sure the project developed individually would be combined together as a whole system work according to the expectations.

We also made some user acceptance test, which we used two different technologies:

1. Cucumber: We used it to support our behavior-driven development (BDD) process. We created some scenarios that we can use to run automated acceptance tests .
2. Selenium: We used it to create automated tests that is cried out on the web browser from the scenarios we had.

Example of one of our scenarios is Login:

```
Feature: Login

Scenario Outline: Login
  Given the user <user>
  Given the password <password>
  Given the agencyNumber <agencyNumber>
  When logs in
  Then the session should be <output>

Examples:
|user|password|agencyNumber|output|
|admin|admin|62|Hello admin UserId: (4)|
```

As you can see this is a cucumber scenario for the login, which we starts with identifying a user(admin), password(admin) and agencyNumber(62). After doing that it will expect an action which is logging in. Then it will expect a “Hello admin userId: (4)” as a session.

This scenario will be executed using selenium which it will be automated.

Code Coverage

Code coverage is a measure used to describe how well you have tested your source code. Ideally you want a percentage as high as possible, because it suggests that there is a lower chance for bugs or program failures to occur compared to a low

percentage. In our project we have not used any tool to calculate or get a code coverage report such as JaCoCo, however it could be a good idea in order to see how big a percentage of our system actually has been tested. Even with a code coverage report saying 100% is tested we can still not be sure that there will be no bugs within our system.

We could have used JaCoCo as it is a maven plugin for java code coverage that generates a report for us to see where we are lacking tests. It is possible to set up a rule in JaCoCo for how big a code coverage percentage is has to have in order to complete a build or else it will fail because of a rule violation, which is smart because it can ensure that you keep testing your code and makes sure it lives up to a certain quality.

Logging

Logging was not implemented in the frontend part of our system but we have made some thoughts on how would like to have implemented that. First of all the java library that we would have used was [Logback](#). Logback is supposed to replace log4j and provides the features that we thought would be necessary in our system.

Logback supports various log levels which we would want to use in order to distinguish between different types of logs. The log levels supported by logback are TRACE, DEBUG, INFO, WARN, ERROR.

With Logback you can specify different appenders. An [appender](#) is responsible for actually writing the log. This could be to a file, the console or to an external server. In our system would have liked to use the [appender made for prometheus](#) running on a separate server. This way we do not store the logs on the same server as the application is running on.

Evaluation for frontend

The process went very well with us, we could complete most of the user stories. The most effective thing that gave us a boost, is creating the dummy data (development environment) instead of waiting on the back-end to get the needed data.

We had a hard time, managing to make a connection from our front-end to the back-end from the contract using the EJB, since it is an old technology and we never used it before. But the good with that, as a team we learned a way of how a legacy system communicates.

The most changing part was, that after changing into the development environment we had some time problem getting the unexpected data format from the back-end, which Rasmus needed to contact the back-end to know why, so we can manage to know where the problem is. Which we could improve by making a better agreement with the back-end from the contract and inform them prictly which data we need from the start.

In overall, as a front-end group we learn how to build a legacy front-end. The best thing that we learned is how to work independently from the back-end so we don't spend time waiting to get data from them.

CI/CD

What we did

We set up CircleCi to simply run the test we had, and used that to let us know if the current code on a given branch failed or did succeed. This helps our development team with not having to keep pulling pull requests or branches to see how their testing went. This is also another step to avoid the *"It works on my machine"* problem, since we are spinning up a docker image on an unrelated instance hosted by another company where they check it.

✓ SUCCESS	booking-logic #60 fix	workflow build	15 days ago 03:00 3108a00
✓ SUCCESS	booking-logic #59 SLA	workflow build	15 days ago 03:58 9c07ee4
✓ SUCCESS	booking-logic #58 Fixed IATA vs Name	workflow build	21 days ago 02:42 6654ede
✗ FAILED	booking-logic #57 Why tf does this not work	workflow build	22 days ago 02:30 5088f8e
✗ FAILED	booking-logic #56 currentThingy	workflow build	22 days ago 02:16 40e0711

Here is an image displaying how it looked in our CI part of our chain. This is CircleCi and its displaying first 2 runs (in our cases push to a branch) wherein it failed and after that 3 runs where in succeeded.

We also setup Docker Hub to build anything that reached master branch since its supposed to be our 'only working code goes here' branch, these images it build as latest since there is no semver on these builds. We also set it up to build our github releases where here we wanted to build on our release version so we could build an image with that tag.

latest		docker pull davidcarl/lsd-backend:latest
Last updated 24 days ago by davidcarl		
DIGEST	OS/ARCH	COMPRESSED SIZE
97b64736622d	linux/amd64	371.96 MB

0.2.2		docker pull davidcarl/lsd-backend:0.2.2
Last updated 15 days ago by davidcarl		
DIGEST	OS/ARCH	COMPRESSED SIZE
472b328028c8	linux/amd64	380.66 MB

This is 2 examples of build types, 1 where it just build whatever was on master and another where we had made a github release with a semver release so it got tagged as 0.2.2 as shown.

What we want in the future

An ideal setup would also implement our CD part, as it is right now we only have like half of the CD. We continue to make builds with pushes on github however we do not deploy them automatically. Optimally we would have a stage environment where we

would be able to push our new image onto automatically, this would be a huge part of our CF part of our chain.




Another step to take would be to integrate a service like [Harness](#) this would give us the possibility to make a new release strategy, which could look like the following. Once there is a new Docker image, we deploy it onto a stage environment where there is run some automated test, this can be frontend test or API test if our project had those. We then gain the possibility to approve manually that it is working and can now click the update production button. It is also possible to deploy without the manual approval if you trust your test that much and don't feel like checking it out yourself.

With Harness and our stage environment implemented we would have implemented a complete CI/CD chain. However some of this is also out of scope of what is taught in this course, and is more of a dream scenario for us.

Maintenance and SLA

For checking our uptime of our application we have setup Grafana to run on the server monitoring the following things, Server performance, Docker containers.

Monitoring our Docker Containers enables us to be able to monitor how much storage, RAM, CPU and network traffic they use. We are also able to see when the containers are running and when they are not. This gives us the ability to monitor their uptime and downtime. Ideally we would also implement other ways to do this, one way is setup uptimerobot, its a service that pings your service with a 5-min interval and send you an email if it no longer receives a response from the service that it pings. It is already implemented on some other services on the server, so it has a half cover since if the server dies completely we will also be alerted of this. However we will not be alerted if just the backend or frontend application crash.

Event	Monitor	Date-Time	Reason	Duration
 Up	Dcarl	2019-12-06 22:26:42	OK (200)	240 hrs, 58 mins
 Down	Dcarl	2019-12-06 21:55:47	Connection Timeout	0 hrs, 30 mins
 Up	Dcarl	2019-07-04 12:30:05	OK	3729 hrs, 25 mins

This is how uptime robot looked for the server this project is hosted upon, with a total 99.93% uptime the last 30 days, and 100% uptime both the last 7 days and the last 24 hours.

As for maintenance it would be ideal for us to run this in a kubernetes setup or Docker swarm so we could make a rolling release and this way not affect people updating our service. Even though its something we have not implemented, we still want to write about how it could be done as if we used Docker swarm, and what it would mean for our end users. When we decide to release a new version we would perform a rolling release on our swarm, we simply replace the image on a node one after one, to make sure this process goes as smoothly as possible it WILL only release on 1 image at a time, per default settings. If one of our nodes fail to update to our new version, the rolling release will pull the handbrake and stop the release by itself. This ensures you have a chance to see what went wrong in the deployment and make sure you still have a service that works once you went on a total rollout.

If we choose to run this in a kubernetes Cluster instead of a Docker swarm we will have even more update strategies than just a rolling release. This gives us even more control over how we want to perform our update, for some features it could be interesting to test them out first, this is where we would deploy a/b testing strategy.

Self evaluation

We think it is important to always be critical of ourselves and evaluate things we did well and what we could have done better.

Communication

We tried our best to act as two different developer teams. This included us having a spokesperson on each team, which meant if the Backend team needed some information from the Frontend team, it would have had to go through David (Backend) to Rasmus (Frontend), and then to whoever could answer the question.

We understand this is how it could be in a real scenario however since we all went to the same class we found it a bit frustrating to do it this way around.

Other than that we had our communications tools we used, this included both a Discord server for talking and messaging (for the teams internal use) and facebook chats for easier reachability with people on the go.

Contract rework

Throughout the project we realised that our contract could need a lot of rework. However due to the system being highly coupled we hesitated at making any changes to the contract. However in the final state of the project we have gathered some things that we would like to change in the future development of the project.

Missing exceptions

As the backend team we figured out too late in the process we were missing exceptions to throw to the frontend. We ended up sending nulls back to the frontend when they requested information that was not available. This resulted in bad error handling since the frontend never knew what really was the problem with their request, or what went wrong in the backend.

Interfaces for DTOs

As the backend team we had some trouble when having to store the different DTO's in our database, as we were using JPA we needed to create entities where we could control the different annotations of each variable, in this case we were forced to create a separate entity class for JPA that could transfer its data to a DTO when it needed to get sent to the frontend.

If we had used interfaces as DTO's we would have had more control over the Objects and would not have had to convert it before transferring between systems. This would not have had an impact on the frontend system as the different accessible methods would be defined in the contract layer.

Conclusion

It was an interesting assignment since we needed to create a legacy system which we have not done before, since we are the new generation with the new technologies. It is still good to know how to work with legacy systems, because there are still companies running their infrastructure on older systems, that are hard / expensive to replace every 5-10 years, a good example of this is the banking world.

One of our bigger struggles in this project was to get our Backend and Frontend to communicate remotely with each other using EJB, as described earlier in our report.

We had also learned how big teams communicate with each other, and how to be interdependent as a frontend and backend group by agreeing on a contract, so both teams have the expectations from both sides.

Overall we found some of the project hard to work with since there was a lot of old technologies we had never worked with, and since it was older technologies it wasn't always as easy to find solutions to these problems on the internet. We had to play around with a bunch of different attempts to resolve some of our problems but managed in the end.