DM571
Project

Jonas Bork Jørgensen	jonjo21
Malthe Hedelund Petersen	malpe21
Mikkel Nielsen	mikkn21
Sofus Hesseldahl Laubel	slaub21

8th of December, 2023

Contents

Malthe Hedelund Petersen - malpe21
Sofus Hesseldahl Laubel - slaub21
Jonas Bork Jørgensen - jonjo21
Mikkel Nielsen - mikkn21

1. Introduction

Local Cinema have hired us to make them an online booking system. They are a small community-driven cinema, which wants to make it easier for their members to manage booking shifts. Local Cinema wanted the system to be web-based and to have user-specific actions in it, so that all of their members could remotely manage bookings, regardless of what type of work they do in the cinema. In this report, we will list and discuss what software and design decisions we have made to achieve this.

2. Stakeholder Analysis

To approach the stakeholder analysis, we listed all stakeholders on a whiteboard while going through the interview with the chairman to identify stakeholders (see Appendix A). Once the whole group was satisfied with the selection of stakeholders, we had an open discussion on a whiteboard going through each stakeholder's description, classification, and our assumptions about them. For us to better classify our different stakeholders, we decided to use the salience model so that we could determine the quality of our stakeholders, thereby making it easier to classify them.

The Members are the users of our system, i.e. technical staff, salespeople, cleaners, facility service, PR, and Supers. We do not include the chairman here. All of these members need to be able to somehow say how much they have worked, so they can get free tickets, and that will be through our system. They have legitimacy in our system since they use it to book shifts through our program. They also have urgency, since if the program would not work, they would get affected, and demand that it works. The members are therefore classified by the salience model as dependent. They are all internal stakeholders as they are part of the association and operate the cinema.

The Chairman acts as the representation of the association, for whom we are making the system, and thus has a lot of legitimacy, as he would want the system to succeed and do well. For the same reason, the chairman also has a lot of power, as he would be the one to request any would-be changes or additions to the system. Because the chairman is representing the association during the development of this system, he also has a lot of urgency as he can request changes at any moment since he is responsible for the system, this makes the chairman the core stakeholder in this system. The chairman is also an internal stakeholder, as he is a part of the association, despite not being a member who operates the cinema directly.

The Visitors are the people who come to the cinema to watch movies and otherwise consume the products the cinema produces. These have power over our system because if they decide to stop coming to the cinema, our system will become obsolete, effectively making them able to stop our project. Furthermore, assuming that most volunteers come from being happy visitors, they also gain extra power because if they are not happy, we will most likely get fewer members, which could make the cinema as a whole infeasible and thus shut down, making our system obsolete. Therefore, visitors are classified as dormant in the salience model and are external stakeholders because they are not interacting with our system directly.

The Citizens of the City are non-stakeholders according to the Salience Model because they have no power, legitimacy, or urgency over the system since the system is not visible to the citizens, but the citizens are indirectly affected by the system, thereby making them external stakeholders. An example of this could be that a citizen has a friend who is a member of the cinema and is happy about the effects of the new system, and the citizen is happy for their friend.

It is important to note that the citizens of the city are the people who do not fit another stakeholder category such as members or visitors, i.e., a member might be a citizen of the city, but therefore they are still thought of as members when looking at stakeholders and not a citizen of the city. So, in other words, the citizens of the city are all the people of the city who do not work or use the cinema.

The Government is a stakeholder since our system is affected by laws, for example, GDPR because it needs to handle personal data about the members and the Government needs to oversee that our system complies with the GDPR. The Government is, however, not directly affected by the system itself, so they are further classified as an external stakeholder.

Using the Salience model, the Government has power because they might make requirements like how to handle personal data in the form of updating GDPR. They have urgency because the system needs to accommodate the requirements as fast as possible, thereby making them a dangerous stakeholder by the Salience model.

2.1. Stakeholder Overview

The stakeholders of the project are:

Stakeholder	Salience Model Classification	Internal / External
Members - Technical - Sales People - Cleaners - Facility Service - PR - Supers	Dependent	Internal
Chairman	Core	Internal
Visitors	Dorment	External
City of the Local Cinema	Non-Stakeholder	External
Government	Dangerous	External

Table 1: An overview of the stakeholder analysis.

3. User Analysis

To describe and classify our Users we again had an open discussion using a whiteboard (see Appendix B) to list and go through each user talking about their description and their classification in correlation to our system. We further continued the discussion by talking about the user base, our assumptions, maintenance, and internal/3rd party systems.

End Users	Superusers	System Users
- Technical Staff - Sales People - Cleaners - Facility Service - PR	- Supers - Chairman	- Developers

Table 2: An overview of the user classification.

The users of the system are split into three categories: End Users, Super Users, and System Users. The End Users consist of the regular members of the association, who will be using the system daily to perform various scheduling. The Super Users consist of the Supers who are responsible for ensuring that shifts are taken, and also the chairman as he might wish to act as a Super, or at the least view the overall schedule with the same permissions as a Super. Finally, the System Users will consist of some developers, initially us, as we have made the product and will need to provide technical assistance should the association require it. But it might change in the future if a different company were to take over the development of the system, or if the association finds a volunteer to take over this role.

3.1. User Base

The user base is the End User and Super User together, they are the members and chairman since the system is built for them.

For the End Users the system will provide easy online shift scheduling for them, i.e., they can more easily book shifts, and report sickness, since they can see the current Super in the system. The Super Users will have an easier time keeping track of who has earned a free ticket, and who is eligible to become a Super, furthermore, they can easily change shifts for both members and Supers and get an overview of which shifts have not yet been covered.

3.2. Maintenance

The developers will be maintaining the system daily, and will also be the ones who provide technical support. The developers' role, is the ones who have responsibility for the software at the moment. We assume that the currently responsible software developers will also provide support for the system. Who the responsible developer is, might change over time, as volunteer developers could be found or another company could take over.

3.3. Internal and Third-Party Systems

The electronic ticketing system is an internal system and will need to interact with our system to plan shifts accordingly. Assuming that we do not host the system, we would need a third-party cloud

server to store our database and host the web-based application. We also have some potential third-party features. We would like to be able to export our calendar, so other calendars can import it. This would make it easy for our user base to integrate their shifts into their daily calendar. Another potential feature would be 2-factor authentication when logging in, since it provides a more secure platform, but is not necessary at this stage of production.

4. Backlog

In this section, we have our prioritized backlog containing all found requirements, both functional and non-functional. Some items in our backlog we were unable to make independent, and for those, we have chosen to write which items are dependent on other items, shown in the "Blocked by" column. We choose to use magic number estimation, with story points of 1,2,3,5,8,13,20, i.e., the "Fibonacci" scale where the higher the number the bigger the task. Each score is measured by the complexity and uncertainty of the task, and how long it would probably take to implement. To try to remove bias, we used a common baseline, which was backlog item 10 as seen in Table 3.

When making the backlog, we made some assumptions due to a lack of clarity in the interview with the chairman. The assumptions are as follows:

- For the PR to enter an event into the schedule, they need to speak with a Super and have them enter it into the schedule.
- All volunteers are 16 or above to avoid the stricter GDPR rules for children.
- The types of shows are for example "movie", "movie premiere", "event" and such, meaning the type is not the name of the movie.
- that a computer will be available on-site in the cinema, which can be used by members without a computer at home.

ID	Backlog Item	Blocked by	Score
1	Make the application accessible on the web.		1
2	To provide user-specific features, we will create a login system		13
3	As a member, I want to see which shifts I have taken and their types so that I don't forget them.		13
4	As a member, I want to book shifts of my groups' types and cancel them so that I can do it from home.		3
5	As a Super, I want to remove and add shifts of my group's type from the work schedule so that schedule management is possible.		8
6	As a Super, I want to book which weeks to be Super in for my group so that I can schedule my shifts.		8
7	As a member, I want to be able to see who the current Super is for my groups so that I can contact the correct Super.		1
8	As a Super, I want to manage the free tickets that each member has earned (1 free ticket for every 5 shifts) so that members can be given the correct number of free tickets.		3
9	To automatically create shifts for movies, we will pull data from the online ticketing system.		13
10	As a Super, I want a way to set a default number of members of my group to take shifts, so that less time is spent on scheduling.		5
11	As a Super, I want to modify shifts of my group's type so that I can handle special cases.		8
12	To cancel members' shifts within a week (for example if they are sick), we will allow Supers to cancel shifts for members		2
13	To prevent members from cancelling shifts unexpectedly, we will limit them from doing so within a week's notice.	4	1
14	In order for members to be considered for Super, we will display to the Supers how long each member has worked for the cinema and how many shifts they have worked.		2
15	To view the crew for a shift, we will let the schedule be public for all members.	3	1
16	To have specific information per shift, we will let Supers add notes for each shift.		1
17	As a member, I want to have the shows integrated into the schedule so that I am informed.	3	1
18	As a Super, I want to receive a notification if a shift for my group has not been booked within a week before it happens so that I do not need to manually check the schedule if it has been booked.		3
19	To accommodate tech-illiterate members, we will build an intuitive user interface.		20
20	Have the uptime be 99.5% for the system.		5

Table 3: The items in the product backlog and their ID that we use to reference them. The order of them in the table shows their prioritization, where the top item has the highest priority and the bottom element has the lowest priority. For a given row, the "Blocked by" are the items that need to be completed before work can begin on the row's item, where the "Blocked by" reference items by their IDs.

4.1. Discussion

The backlog items were mainly written as user stories because this format works best for most items since it's concise and includes the motivation for the item being in the backlog, while simultaneously not specifying how it should be implemented, leaving this up to the developers who know best how to implement it. Furthermore, user stories support various levels of specificity. For some items, however, job stories and Feature-Driven Development (FDD) features were used instead. FDD features were beneficial for non-functional requirements such as the uptime requirement (backlog item 19). We chose to use the "Fibonacci sequence" to score each item because the sequence increases rapidly to better reflect the increasing effort needed for the higher-scored items while providing more precision for the lower-scored items. Thereby making it a useful metric to differentiate the estimation of each item.

In the backlog, some backlog items are blocked by others because it pointed out that the blocked items could not be prioritized higher than the items they were blocked by, thus simplifying the ordering. For example, backlog item 13 is blocked by item 4 because it needs to modify the functionality of cancelling shifts with the constraint that they cannot be cancelled by a member within a week of the shift, i.e. to complete item 13, it's necessary to modify the work done in item 4.

Backlog item 11 says that a Super should be able to modify shifts, which seems like it is a pretty important feature, but we chose to give it a low priority. That is because backlog item 5, can add and remove shifts, which will make one able to "modify" shifts, but only by removing it and making a new one. So since modification of a shift can be emulated by backlog item 5, it is actually not that important.

An intuitive interface (ID 18) is critical when volunteers are old since they aren't always good with IT. With that in mind, we chose to add it to the backlog, but we placed it near the bottom. We chose to prioritize a lot of the features higher since we think that they still would give a higher value to the stakeholders. While it is at the bottom, it does not mean that one should make a crappy user interface, and it would be unusable, but rather one should make what makes sense at the time, and then it can be adjusted later. Another point is that it is in the backlog, so everybody knows that we will at some point need to make it intuitive, so they need to take that into consideration when building the product.

Initially, we had a backlog item which stated that the product should be cloud-based, but this item was eventually removed from the backlog as we realized it is a decision that should be made by the developers on how to achieve the 99.5% uptime, as that non-functional requirement was the reason we even had an item requiring the product to be cloud-based to begin with. Despite the item being removed from the backlog, the idea that the product be hosted on a cloud-based platform is still present in the final backlog, as the 99.5% uptime item is still in the backlog, but with a rather low score, as we have made the assumption that the item will not be about how the uptime is achieved, but rather how to convince the stakeholders from the "LocalCinema" organization that a cloud-based service is the best solution and which cloud service provider should be picked, but even this should be trivial as the developers are likely experienced and already know of competent cloud service providers.

To order our backlog items, we ordered them based on how much value to the product owner they provided. Our three main ordering criteria were: *basic functionality*, *quality of life* and our *story points score*, where basic functionality had the most weight on the overall score, then the story points score and lastly quality of life. Because we would rather build a Minimum Viable Product, instead of doing a big bang release, such that feedback is possible.

5. Flows in the system

5.1. Class Diagram

Our class diagram, which can be seen in Appendix I, depicts how our models in our website work together. The class diagram contains all the classes and enumerations that make up our backend code, together with several relation arrows signifying what kind of relations these different classes and enumerations have, and how many of each are related to each other. The relationship between the Super and member class does not have this number of relations between each other, as their relationship is that the Super extends the member. Each class in the diagram contains which attributes and which methods they contain. For the attributes they have a connection to other classes where one of their attributes is an instance of another class in the diagram, with their methods they instead contain input and output types. Note that we do not display private methods in the classes, since they only provide internal logic that is not interesting for the overall functionality of the classes.

For the member class that uses the enumeration "Group" as an attribute, the upper limit of how many they can hold is equal to how many values the specific enumeration can take on, as no member should contain duplicates of any enumeration value from Group, as each of these enumeration values represents a group the member is a part of. The class diagram contains three classes that show up often in our code: Schedule, Shift and Show. Despite how often these three classes show up in our code, we have decided to only show one connection for each of them, to make the diagram readable. In the diagram, there also is no connection between Super and the enumeration Group, despite the Super class containing an attribute with type Group. This is because the member class, which the Super class extends, already has a connection to the Group enumeration, and therefore we decided not to show the connection between Super and Group, which has also made the diagram more readable.

5.2. Sequence Diagrams

We have two sequence diagrams. Each depicts a flow in our systems.

Member Books Shift (Figure 1) depicts the flow of a member trying to book a shift. They start off by being logged in and pressing the "book" button on a shift, then a method called "require_login()" is run to ensure that the member is actually logged in, since any user can access the schedule view by typing in the exact URL for the schedule view. After which the details of the booked shift are updated to reflect that the member has now booked it, after which the database is updated as well. During all of this, there are several steps along the way where the process could fail, despite this, we haven't depicted any errors or failures in the sequence flow, as nothing unusual happens should any of the steps fail. A simple exception is thrown, as is to be expected, and therefore it isn't depicted in the flow.

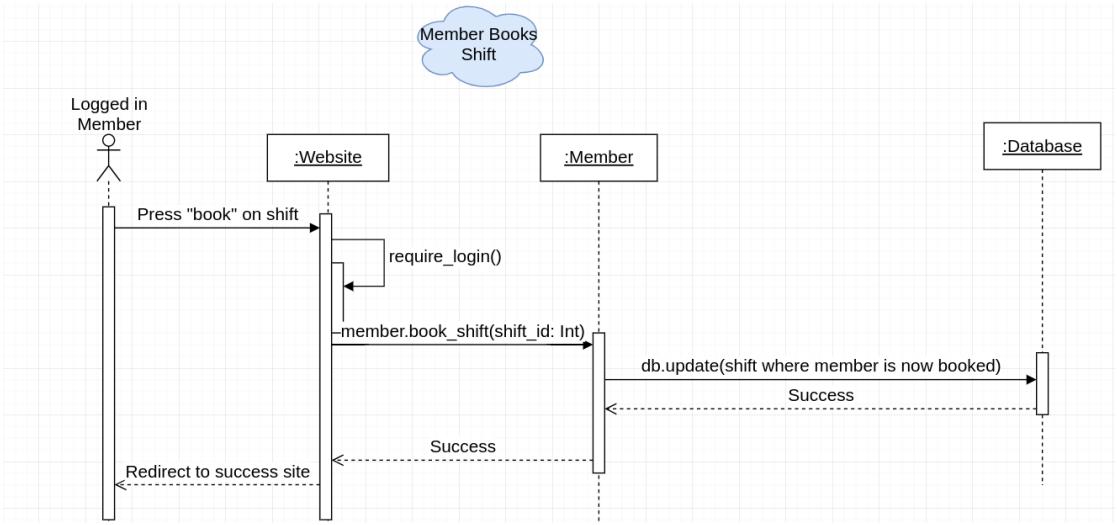


Figure 1: The sequence diagram for a member booking a shift.

Create User (Figure 2) depicts the flow of the system when a Super user wants to create a new member. When the Super pushes the button "create user", it will check if you indeed are a Super. That is because you only need to change the URL, to try to create a new user. That should, of course, not be possible, so that's why we check you are Super.
If a scenario were to fail, nothing extraordinary would happen besides instead of returning success, it would return an error and the user would not be created.

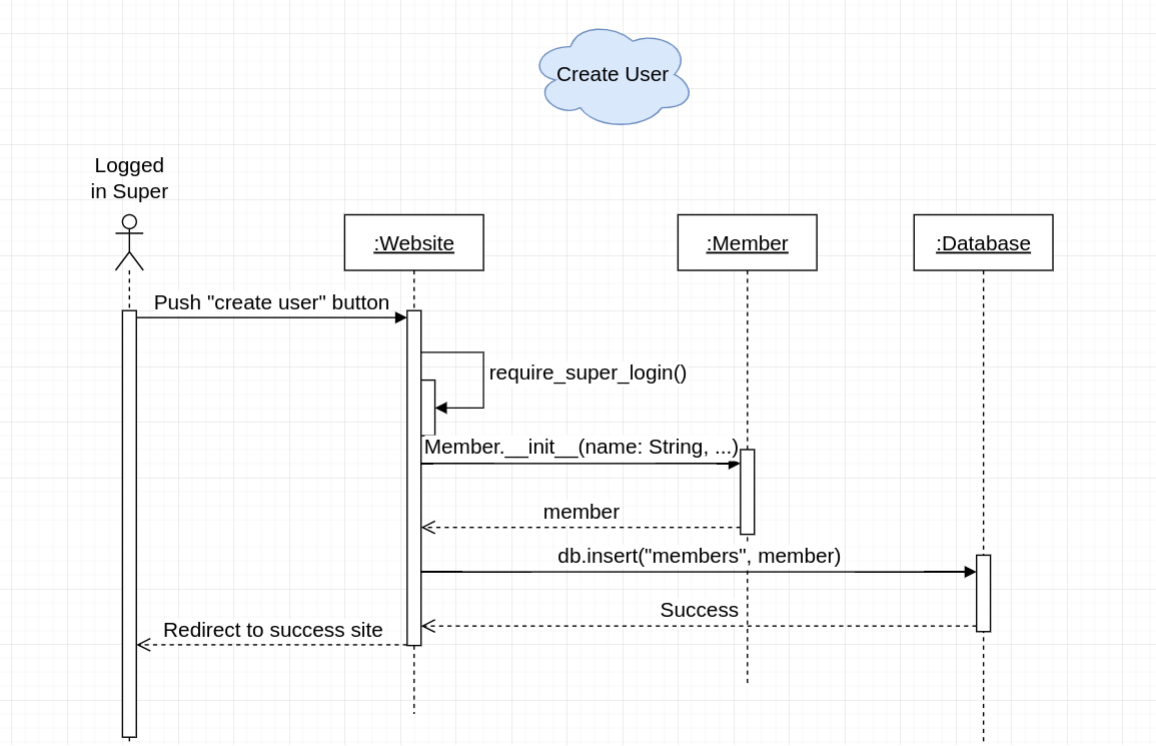


Figure 2: The sequence diagram for a Super creating a user.

6. API Specification

In the first iteration of our API specification, we specified endpoints for the following functionalities; "Fetch future shows and shifts", "Sign up for shift" and "Manage own shifts". These endpoints are both small enough for our first iteration and meaningful enough for the external company to start their work.

The API is documented using OpenAPI, and it is level 2 on the Richardson Maturity scale, meaning that we use the HTTP verbs (GET, POST, etc.) as intended, we decided to use level 2 instead of level 3 to avoid the extra complexity and bandwidth usage that level 3 introduces, since we want to keep the API small and fast. Further, each of our endpoints is secured using an API key.

When possible we decided to return an ID instead of an entire component since the user can then use these IDs to find the relevant information. E.g. when we wish to return members as a part of another component, we only return the IDs of the members, because of this, we decided to include in the first iteration an endpoint `/Members`, to help the user find and filter for relevant information for a given member's ID and increase performance by reducing the amount of data that needs to be sent since the user might not need all the data for a member. We decided to make member-specific endpoints and Supers-specific endpoints, since Supers need extra capabilities, like editing existing shifts. We think of Supers as members, since they might be part of multiple groups and only be Super for some of these, and therefore we have a "member" component in our API, with a property "Super-groups" that tells the user which groups this specific member is Super for, if it is empty then the member is not Super at all. We have decided to only specify non-obvious errors in the specification. For example, we have not written that it will return the response 408 (timeout) because this would apply to every single endpoint, so we assume this is active/available by default. (see Appendix C for the full API specification)

6.1. Fetch future shows and calendar

We have two ways of fetching future shows and the calendar. You can fetch all the shows and the whole calendar from the endpoint `/Schedule` via GET. Here, you also have the option to filter out things you don't want. Each member can also get their personal calendar, where all shows and only shifts from their groups are shown. This is available under the endpoint `/Members/{member-id}/Schedule` via GET. This endpoint also has a filter available.

6.2. Sign up for shift

Using `PUT` on the `/Members/{member-id}/Schedule/Shifts/{shift-id}` endpoint attempts to sign the member with `member-id` up to the shift with `shift-id`. `PUT` is used because it modifies the given shift by assigning the member to that shift. This endpoint will perform some validation of the request as well, for example making sure that the member is a part of the group that the shift is for so that a member cannot book a shift that is not for one of their groups.

6.3. Manage shifts

To provide the ability for members, and Supers, to manage their shifts, additional endpoints have been created to delete, remove oneself from a shift and for Supers to patch a shift, which means to edit anything about the shift they wish. These additional endpoints, in addition to the previously discussed endpoints, allow members and Supers to manage their shifts.

7. Implementation

7.1. Unit test coverage report

To ensure that the implementation was done without errors, we created a code test coverage report that shows how much of the code is tested, to show this, we chose to test the three classes: *Member*, *Super* and *AutomaticScheduler*. This report can be found in Appendix D. We did not have 100% coverage for the member class, since some methods are private, and some of their code is not possible to reach through a unit test at the moment. Only in further development of these functions, should the code be able to be reached, and then it should be unit tested. The *AutomaticScheduler* class also has not reached 100% coverage, as it contains some dummy data that is overwritten in the unit test. When we look away from those cases, we have a 100% coverage.

8. Documenting the architecture

8.1. Layered Architecture

The architecture that we used in the development of the product is layered, specifically it is the MVT architecture: Model, View, Template, which means the architecture consists of 3 layers. This is the architecture that is used by Django, the framework that we used to build our website. The first layer, model, is what would be considered the back end, it is where the logic for the various classes and data structures exist, and it's where communication with the database is handled. The second layer, view, connects the other two layers, by handling the business logic where it manipulates data by sending it to the model layer and makes sure the data is shown to the user by passing it to the template view. The third layer, template, handles all the user interface logic and is responsible for taking requests from the user and passing them to the view layer, which in return can pass the data to a new template, whilst also generating a user interface for the user to interact with.

8.2. Distribution Patterns

We do not use the Distributed Presentation Architecture, as we use Django, which uses the MVT architecture, and does not split it up. While the server handles a lot of the logic for the client, it does not fit in the Remote User Interface pattern, since a lot of logic is still on the client side. We do not use the Distributed Application Kernel. Because Django uses a standard setup to only run on one server, but could be configured for it in the future if we wanted to. The Remote- or Distributed Database pattern is also not used since the database is just implemented as a class for now, so it runs with the other server logic on the same server.

8.3. Agile principles

Here is a list of the relevant principles, with a small description of whether the architecture lived up to them (Showed with "Yes" in the list) or not (Showed with "No").

- 1 - Yes, we prioritised making early and continuous delivery of valuable software
- 3 - We live up to this principle because we have carefully selected the work to be done in the first iteration of the development of our system such that we, in the end, could deliver a working product that provides value to the customer as fast as possible.
- 5 - We were so motivated during development that we made an actual website, instead of a simple mock-up like Figma, and we even extended the website to contain multiple features, just because we could.
- 6 - Yes, throughout the development of the project, most of the team's discussions and decision-making have been done through face-to-face conversations.
- 10 - We often went for simple designs, like building a simple website, but not always, as we, for example, reinvented a database instead of using Django's already working database framework.
- 11 - No, we did not choose our architecture nor requirements, as we were already given the requirements for the project before even starting development. Moreover, the architecture we ended up working with was not one we chose ourselves, it was simply the one that the Django framework happens to use. Other than that, we do not believe the other principles are relevant to our project.
- 12 - No, we agreed on a way to tackle each problem as it arose, and then we just went with that solution until the problem was done, but reflecting on the process when we were done, we realized how we solved each problem was the most optimal, and thus it was not possible to become more effective anyway.

8.4. C4 Model

The first 3 levels of the C4 model have been created to help convey the logic of our application. Pictures showing these three levels can be found in Appendix H The fourth level of the C4 model wasn't created, but it could be created to give a technical view of what the code does. But as the entire point of the C4 model is to make it easier to show the functionality of a system, going in depth with what the code does and how it does it, wouldn't provide much value. Additionally, the fourth level should exclusively be used for essential or complex components, yet none of the components in our system are particularly complex.

9. Extra Requirements

To store the data of our website, we have created a database that contains all the members, shows, and shifts. This database is one that exists in memory and is not actually stored in a non-volatile location anywhere. This has the obvious consequence that the application cannot actually save any data that isn't hardcoded into the code of the application.

10. User interface

To plan the layout of the User Interface (UI), we created sketches and connected them via wireframing to give ourselves an understanding of the flow of the UI without investing time into implementing it, allowing us to plan and experiment with different UIs with very little time commitment. The sketches and wireframing can be seen in Appendix K.

We hooked our prototype up to multiple sequences/flows (we encourage the reader to boot up the website and play around with the different features, see Appendix J), but we will only document our decisions for the following flow:
`Viewing the full schedule (incl. shifts/shows) --> book shift`.
Note that we assumed you are logged in to do the flow described above. Firstly, we agreed that the website needs some "WOW" factor, and therefore we chose the superior colour scheme gruvbox. Furthermore, to have a cohesive colour scheme throughout the website and to keep our code DRY [1], we made the `base.css` and `base.html` files that contain the code that should be shared between all our pages, i.e., the colour scheme, shared page-styling, navigation bar and footer.

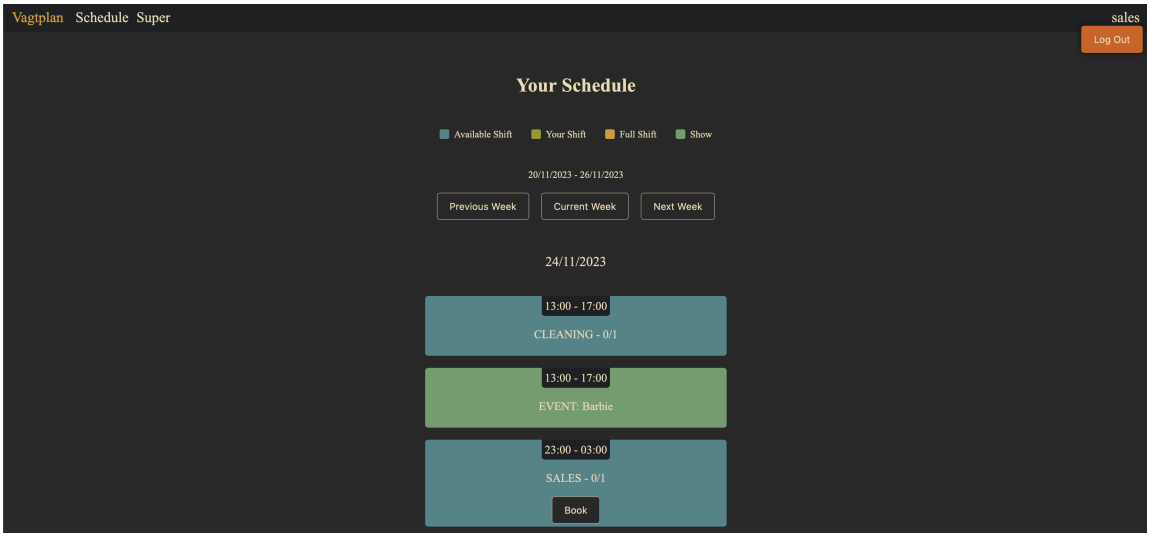


Figure 3: View of the homepage of our website, which is the full schedule, that shows both our navigation bar, buttons, colour scheme and more. The user is logged in as a user whose name is "sales" and is a part of the sales group.

To make information about the shifts and shows in the schedule available at a glance, we colour-coded them, such that:

- If it is a shift that is available to book, it is coloured blue.
- If it is a shift that is fully booked, it is coloured red.
- If it is a shift that the current user has booked, it is coloured green.

[1] Don't Repeat Yourself

- If it is a show, it is coloured aqua.

This should help the user quickly differentiate between the different information, such that they can stay informed about both their own and their colleagues' shifts, while also knowing what type of shift they are working, e.g., working as CLEANING on an EVENT like Barbie premiere. We decided to make the schedule display the current week, and then add buttons to go forward one week to view upcoming shifts or go backwards one week to view previous shifts. This seems like the functionality that you expect from a schedule, and hence it needed to be incorporated into the first iteration of the website. In this first iteration, we decided on a design where you can only view one week at a time, since this is the way most people view their calendars daily. In a later iteration, we can add a "day", "month" and "year" view as well.

To differentiate between the different days of the week, we added a "headline" with the different dates above each set of shifts and shows, to make it clear which day each set of shifts and shows are planned. In a future iteration, we might display Monday, Tuesday, etc instead of the actual date to increase usability, but this was not a priority for the first iteration. Further, each shift displays the start and end time of the shift, along with the group that the shift is for, such as CLEANING, SALES, etc, and how many members are currently booked for the shifts and the capacity e.g., 0/1 booked. If the shift is available to book and the user is in the correct group, then a "Book" button will also display on the shift that, when pressed, allows the user to book the shift for themselves.

If a member has booked a shift that starts in less than a week, they cannot cancel their shift, and thus, this option will not be available to them. If, however, their shift starts later, a "Cancel" button will be displayed to them that they can use to cancel the booking of their shift. Supers can, however, cancel everybody's shifts, so if a member needs a shift cancelled within a week, they still have to contact a Super, which we made easier in this first iteration by providing members with a "Super" tab that currently lists all Supers and their contact information, and then the Super can log in and access an "Admin" page where they can cancel shifts for others. Note that, currently, it is inconvenient for the Supers to cancel multiple shifts for a member at a time because the Super will be returned to their schedule whenever they cancel a single shift, meaning they will have to go through numerous menus to cancel multiple shifts. We deemed this to be okay for the first iteration, since it should only be necessary for a Super to cancel a member's shift when it is within a week, which should happen very rarely. In a later iteration, this will, of course, be fixed. For the first iteration, we decided not to implement that there is a "current Super", instead we simply list all Supers and their contact information along with what groups they are Super for. We decided to prioritise the actual booking and viewing functionalities, since we meant this would be the biggest improvement over the older system.

Further, we have made a navigation bar that takes you to the homepage (Schedule) or to the previously described "Super" page to get contact information about the Supers, and to the right, we have the name of the currently logged-in member (which is "sales" on Figure 3). On the name, a little menu will pop up if you hover over the name that gives you the option to log out. The navigation bar has another option called "Admin" which is only visible for Supers, this page gives Supers the option to access a page where they can list all users, cancel users' shifts, create shifts, and create shows. We thought that it was most intuitive to display the name of the website all the way to the left and the name of the logged-in member all the way to the right, to help the confused user, know which site they are on and that they are actually logged in as themselves. In a future iteration, the pop-up menu that appears on the user's name could also take them to a "Member page" consisting of their information, so they, for example, can change their email if needed. We agreed that personal information like their phone number or email is mostly static, it can change

in the future, but it is unlikely, so this was not something worth considering for the first iteration.

11. Conclusion

We have made a solid prototype, that can with little effort be made available to LocalCinema, and would make their daily life easier regarding bookings. We have also written a specification for an API, that would enable a third-party entity to build on top of our system. We have documented what choices and actions we have made, along with which software engineering principles we have applied successfully to achieve them.

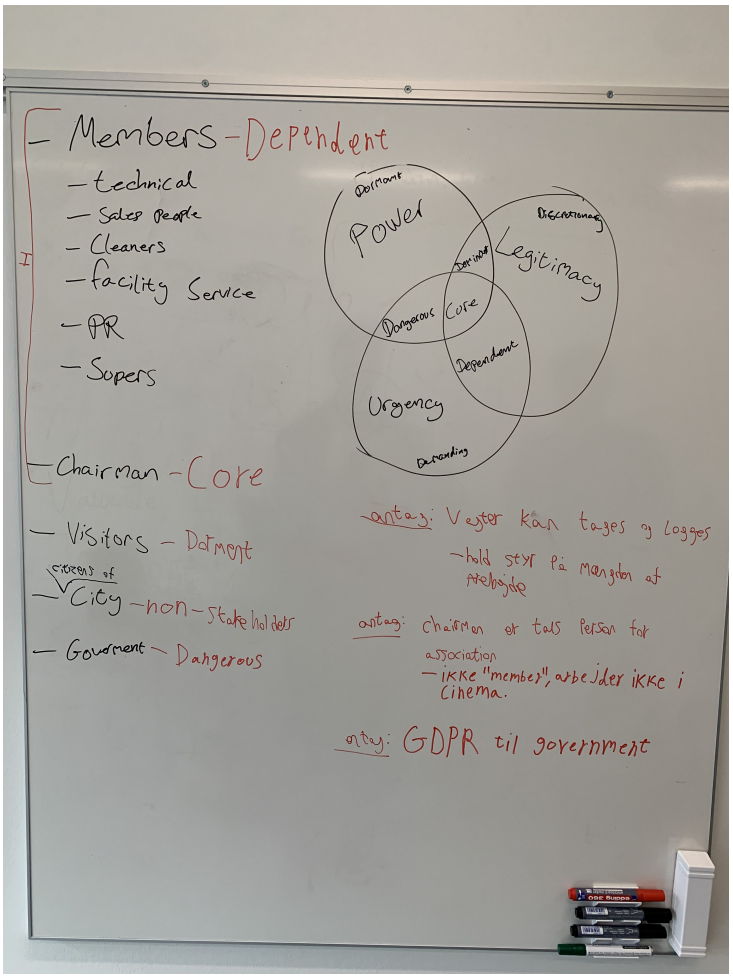
12. Credits

This section will state which member of the group wrote which sections. Note that we will limit ourselves to crediting one member for each section, but this is only an estimate, as most sections have been written in cooperation between all group members.

- 1 Introduction - Sofus
- 2 Stakeholder Analysis - Malthe
- 3 User Analysis - Mikkel
- 3.1 User Base - Jonas
- 3.2 Maintenance - Sofus
- 3.3 Internal and Third-Party Systems - Malthe
- 4 Backlog - Mikkel
- 4.1 Discussion - Jonas
- 5.1 Class Diagram - Sofus
- 5.2 Sequence Diagrams - Malthe
- 6. API Specification - Mikkel
- 6.1 Fetch future shows and calendar - Jonas
- 6.2 Sign up for shift - Sofus
- 6.3 Mange shifts - Malthe
- 7.1 Unit test coverage report - Sofus
- 8.1 Layered Architecture - Jonas
- 8.2 Distribution Patterns - Sofus
- 8.3 Agile Principle - Malthe
- 8.4 C4 model - Mikkel
- 9. Extra Requirements - Jonas
- 10. User Interface - Mikkel
- 11. Conclusion - Malthe
- 12. Credits - Mikkel

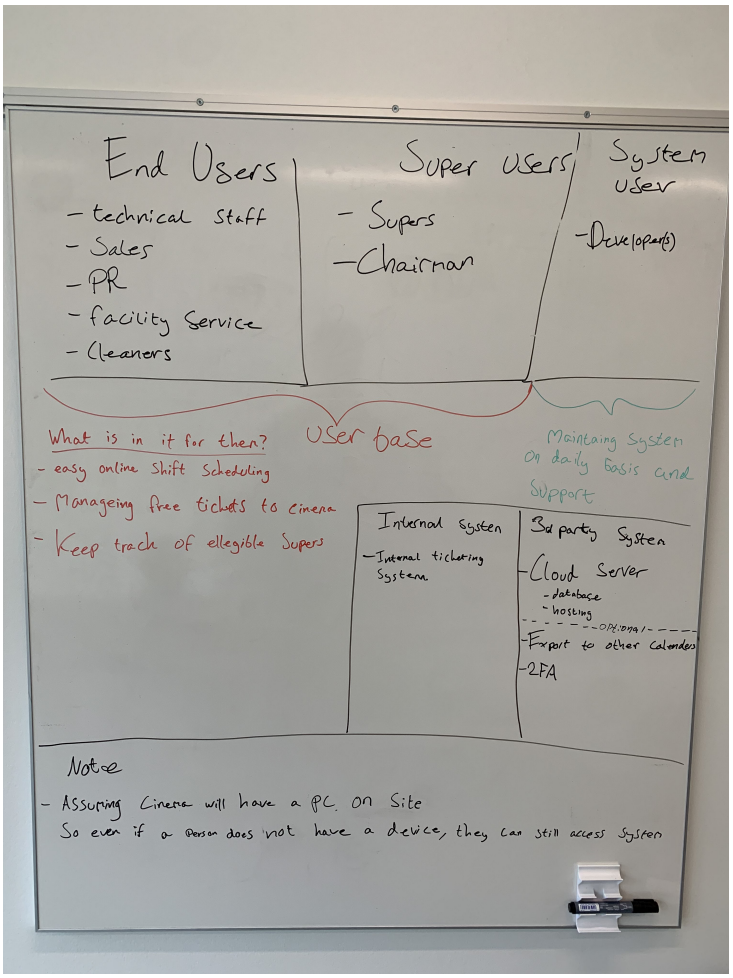
Malthe Hedelund Petersen - malpe21
Sofus Hesseldahl Laubel - slaub21
Jonas Bork Jørgensen - jonjo21
Mikkel Nielsen - mikkn21

A Whiteboard for the Stakeholder Analysis



Malthe Hedelund Petersen - malpe21
Sofus Hesseldahl Laubel - slaub21
Jonas Bork Jørgensen - jonjo21
Mikkel Nielsen - mikkn21

B Whiteboard for the User Analysis



C API Specification

```
openapi: 3.1.0
:
tle: Online Vagtplan
scription: |-
Vagtplan for the local cinema.

rmsOfService: http://swagger.io/terms/
ntact:
email: support@onlinevagtplan.com
name: API Support
cense:
name: Apache 2.0
url: http://www.apache.org/licenses/LICENSE-2.0.html
rsion: 1.0.0
ers:
url: https://onlinevagtplan.com/api/v1
:
name: Member
description: Everything for members
name: Super
description: Everything for Supers
s:
chedule:
get:
  tags:
    - Member
  summary: Retrieves the schedule
  description: Retrieves the schedule
  parameters:
    - name: from
      in: query
      required: true
      description: the first day of the schedule (inclusive)
      schema:
        type: string
        format: date
        example: 2023-02-25
    - name: to
      in: query
      required: true
      description: the last day of the schedule (inclusive)
      schema:
        type: string
        format: date
```

```
        example: 2023-02-25
    - name: filter
      in: query
      required: false
      description: The fields you want to be included
      schema:
        type: array
        items:
          type: string
  responses:
    200:
      description: schedule is retrieved successfully
      content:
        application/json:
          schema:
            $ref: '#/components/schemas/Schedule'
    400:
      description: The date "to" is before the date "from" or the
    ↪ format of the dates is invalid
  security:
    - api_key: []
embers/{member-id}/Schedule/Shifts/{shift-id}:
delete:
  tags:
    - Member
  summary: delete shifts
  description: delete shifts
  parameters:
    - name: member-id
      in: path
      required: true
      schema:
        $ref: '#/components/schemas/Member/properties/id'
    - name: shift-id
      in: path
      required: true
      schema:
        $ref: '#/components/schemas/Shift/properties/id'
  responses:
    200:
      description: successfully canceled booked shift
    400:
      description: you were not able to cancel the shift
  security:
    - api_key: []
put:
```

```
  tags:
    - Member
  summary: book shift
  description: As a member book the shift
  parameters:
    - name: member-id
      in: path
      required: true
      schema:
        $ref: '#/components/schemas/Member/properties/id'
    - name: shift-id
      in: path
      required: true
      schema:
        $ref: '#/components/schemas/Shift/properties/id'
  responses:
    200:
      description: successfully booked the shift
    400:
      description: were not able book the shift
  security:
    - api_key: []
embers/{member-id}/Schedule:
get:
  tags:
    - Member
  summary: Get a specific members schedule
  description: Get the shifts that are available for this specific
   ↪ member and that this member has already booked
  parameters:
    - name: member-id
      in: path
      required: true
      schema:
        $ref: '#/components/schemas/Member/properties/id'
    - name: filter
      in: query
      required: false
      description: The fields you want to be included
      schema:
        type: array
        items:
          type: string
  responses:
    200:
      description: schedule is retrieved successfully
```

```
      content:
        application/json:
          schema:
            $ref: '#/components/schemas/Schedule'
    400:
      description: Bad input parameters
  security:
    - api_key: []
upers/{member-id}/Schedule/Shifts/{shift-id}:
patch:
  tags:
    - Super
  summary: Edit a shift
  description: Edit a shift
  parameters:
    - name: member-id
      in: path
      required: true
      schema:
        $ref: '#/components/schemas/Member/properties/id'
    - name: shift-id
      in: path
      required: true
      schema:
        $ref: '#/components/schemas/Shift/properties/id'
    - name: shift
      description: What to change the shift to. Only include the
   ↪ fields that should be edited, the others will remain the same
      in: query
      required: true
      schema:
        $ref: '#/components/schemas/Shift'
  responses:
    200:
      description: Shift is edited successfully
      content:
        application/json:
          schema:
            $ref: '#/components/schemas/Shift'
    400:
      description: Bad input parameters
    403:
      description: That Super cannot edit that shift
  security:
    - api_key: []
delete:
```

```
  tags:
    - Super
  summary: Delete a shift
  description: Delete a shift
  parameters:
    - name: member-id
      in: path
      required: true
      schema:
        $ref: '#/components/schemas/Member/properties/id'
    - name: shift-id
      in: path
      required: true
      schema:
        $ref: '#/components/schemas/Shift/properties/id'
  responses:
    200:
      description: Shift is deleted successfully
    400:
      description: Bad input parameters
    403:
      description: That Super cannot delete that shift
  security:
    - api_key: []
onents:
hemas:
Schedule:
  type: object
  properties:
    from:
      type: string
      format: date
      example: "2023-02-25"
    to:
      $ref: '#/components/schemas/Schedule/properties/from'
    shifts:
      type: array
      items:
        $ref: '#/components/schemas/Shift/properties/id'
    shows:
      type: array
      items:
        $ref: '#/components/schemas/Show/properties/id'
Shift:
  type: object
  properties:
```

```
    from:
      type: string
      format: date-time
      example: 2023-02-25T10:15Z
    to:
      $ref: '#/components/schemas/Shift/properties/from'
    id:
      type: integer
      format: uint64
    booked-members:
      type: array
      items:
        $ref: '#/components/schemas/Member/properties/id'
    member-type:
      type: string
      description: the type of members that can book this shift
      enum:
        - technical
        - sales
        - cleaners
        - pr
        - facility service
    is-super:
      type: boolean
      description: whether the shift is for Supers
      enum:
        - true
        - false
    notes:
      type: string
      description: Notes for the shift
      default: ""
Member:
  type: object
  properties:
    id:
      $ref: '#/components/schemas/Shift/properties/id'
    name:
      type: string
    groups:
      type: array
      description: the groups that this member is a part of
      items:
        type: string
        enum:
          - technical
```

```
          - sales
          - cleaners
          - pr
          - facility service
    super-groups:
      $ref: '#/components/schemas/Member/properties/groups'
      description: the groups that this member is Super for
    hiring-date:
      type: string
      format: date
      example: 2023-02-25
    shifts-completed:
      type: integer
      description: the amount of shifts that the member has worked
    free-tickets-remaining:
      type: integer
      description: the remaining free tickets that this member has
    ↪ earned
Show:
  type: object
  properties:
    from:
      $ref: '#/components/schemas/Shift/properties/from'
    to:
      $ref: '#/components/schemas/Shift/properties/from'
    id:
      $ref: '#/components/schemas/Shift/properties/id'
    title:
      type: string
      description: the title of the show, for example the name of
    ↪ the movie
      example: Barbie
    type:
      type: string
      enum:
        - event
        - movie
        - premiere

curitySchemes:
api_key:
  type: apiKey
  name: api_key
  in: header
```

D Unit Test Coverage Report

See attached file for unit test coverage report

Malthe Hedelund Petersen - malpe21
Sofus Hesseldahl Laubel - slaub21
Jonas Bork Jørgensen - jonjo21
Mikkel Nielsen - mikkn21

E Member Cyclomatic Complexity

Function Name	Start Line	End Line	Cyclomatic Complexity (Threshold: 10)	Lines of Code (Threshold: 50)	Parameter Count (Threshold: 4)
__init__	12	25	2	14	⚠ 6
__check_valid_phone_number	28	31	2	4	2
__check_unique_email	33	35	2	3	2
__create_id	40	46	2	7	1
_get_element_by_id	48	52	2	5	3
get_free_tickets_remaining_count	54	55	1	2	1
book_shift	58	72	6	15	2
cancel_shift	75	83	3	9	2
get_full_schedule	85	90	2	6	3
get_own_schedule	92	100	4	9	3

Malthe Hedelund Petersen - malpe21
Sofus Hesseldahl Laubel - slaub21
Jonas Bork Jørgensen - jonjo21
Mikkel Nielsen - mikkn21

F Super Cyclomatic Complexity

Function Name	Start Line	End Line	Cyclomatic Complexity (Threshold: 10)	Lines of Code (Threshold: 50)	Parameter Count (Threshold: 4)
__init__	13	17	1	5	⚠ 8
give_free_ticket	19	21	1	3	2
create_shift	23	24	1	2	2
add_super_to_group	26	31	2	6	3
promote_to_super	33	40	2	8	3
cancel_shift	42	48	2	7	3
add_member_to_group	50	54	2	5	3
remove_member_from_group	56	60	2	5	3
create_member	62	63	1	2	2
delete_member	65	66	1	2	2

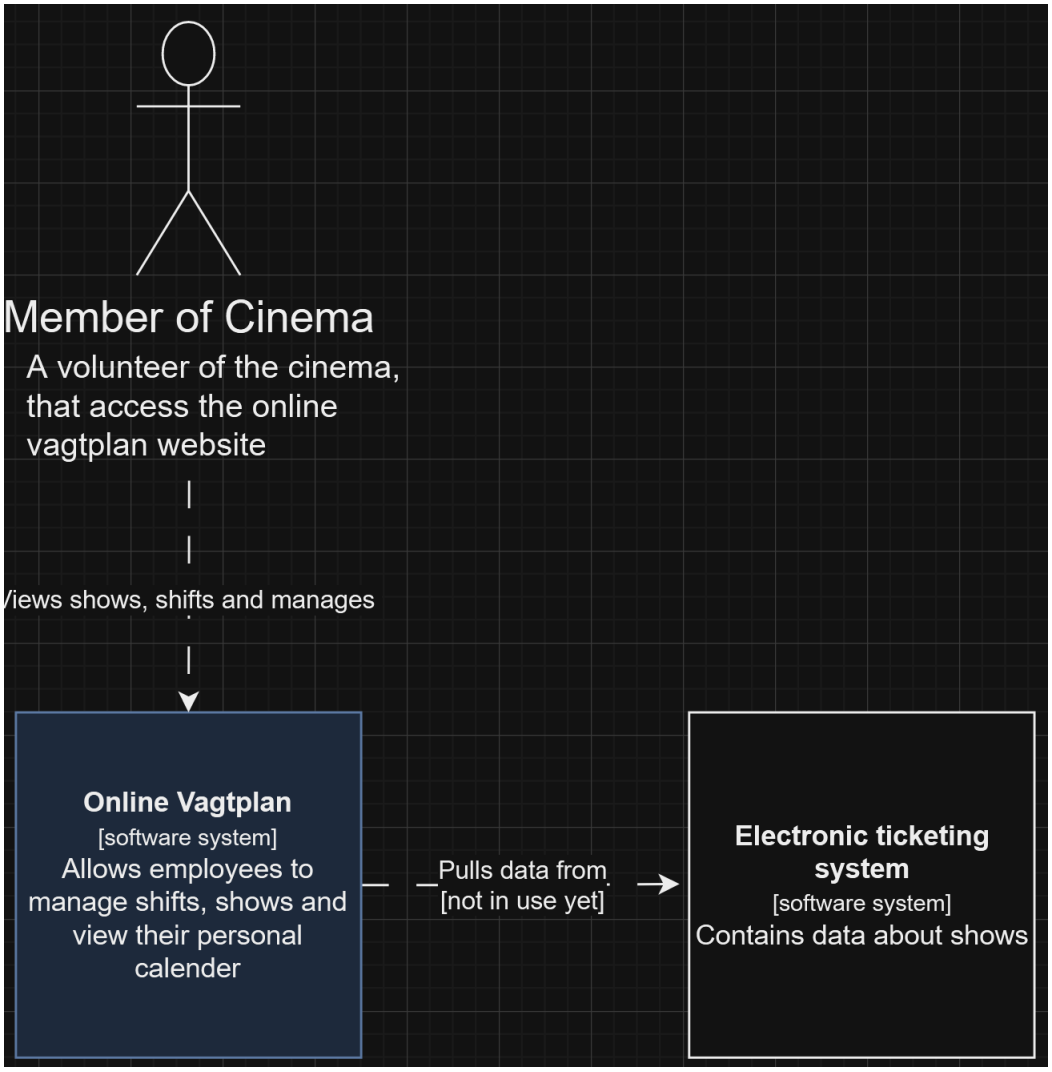
G Automatic Scheduler Cyclomatic Complexity

Function Name	Start Line	End Line	Cyclomatic Complexity (Threshold: 10)	Lines of Code (Threshold: 50)	Parameter Count (Threshold: 4)
__init__	9	13	1	5	3
__get_shows_from_API	15	43	1	29	1
fetch_shows_and_create_shifts	45	50	4	6	1

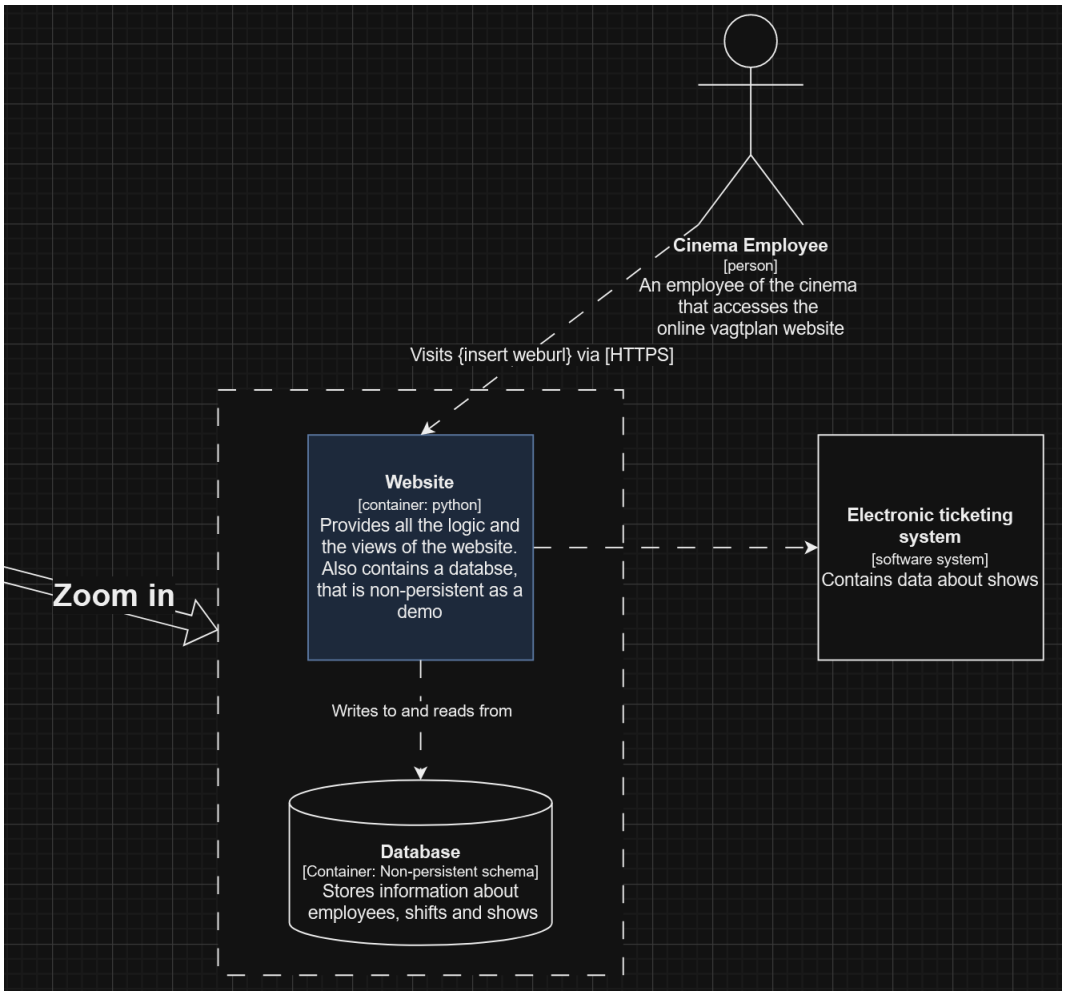
Malthe Hedelund Petersen - malpe21
Sofus Hesseldahl Laubel - slaub21
Jonas Bork Jørgensen - jonjo21
Mikkel Nielsen - mikkn21

H C4 Model

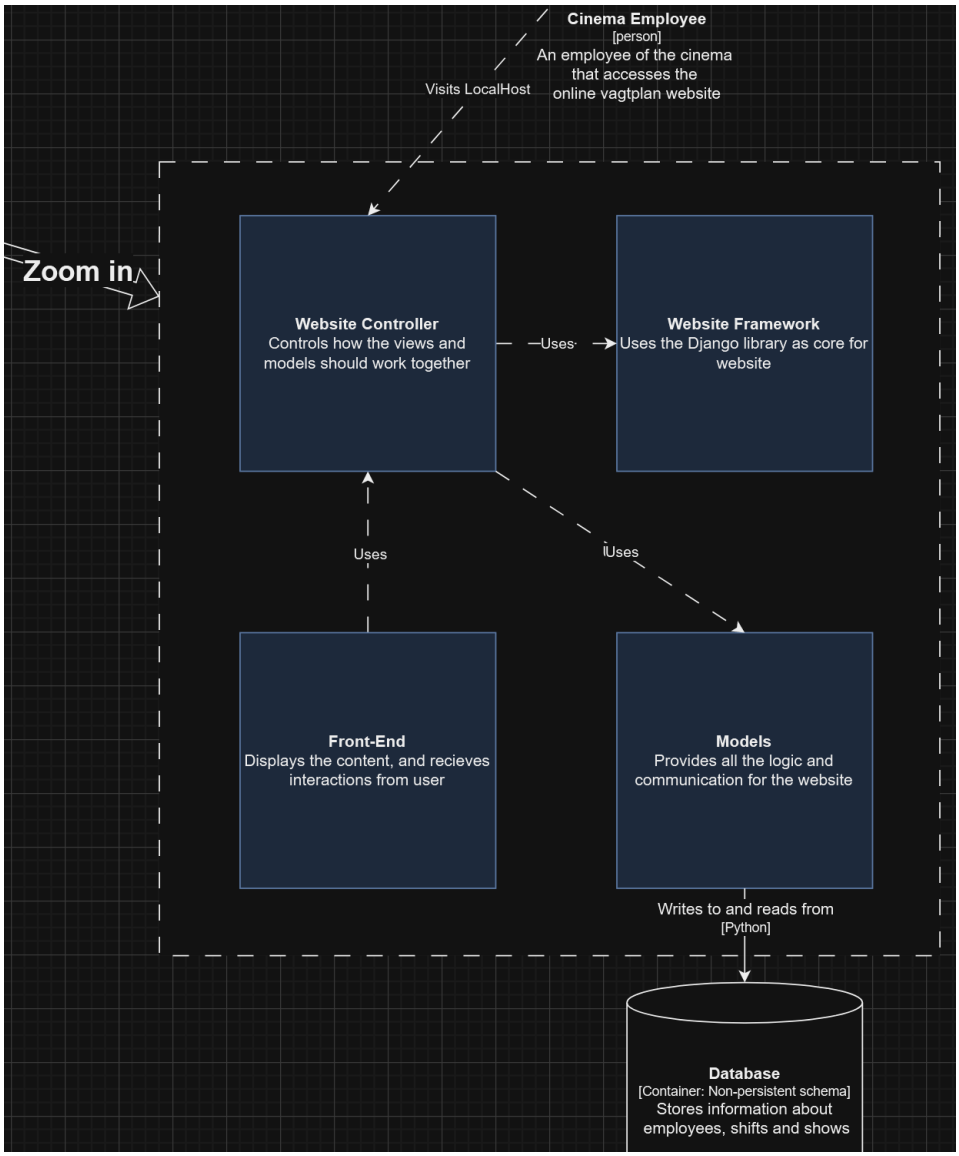
Note the C4 model has been split up into 3 images since it was too big.



The figure below has zoomed in on the "Online Vagtplan" from the previous picture.



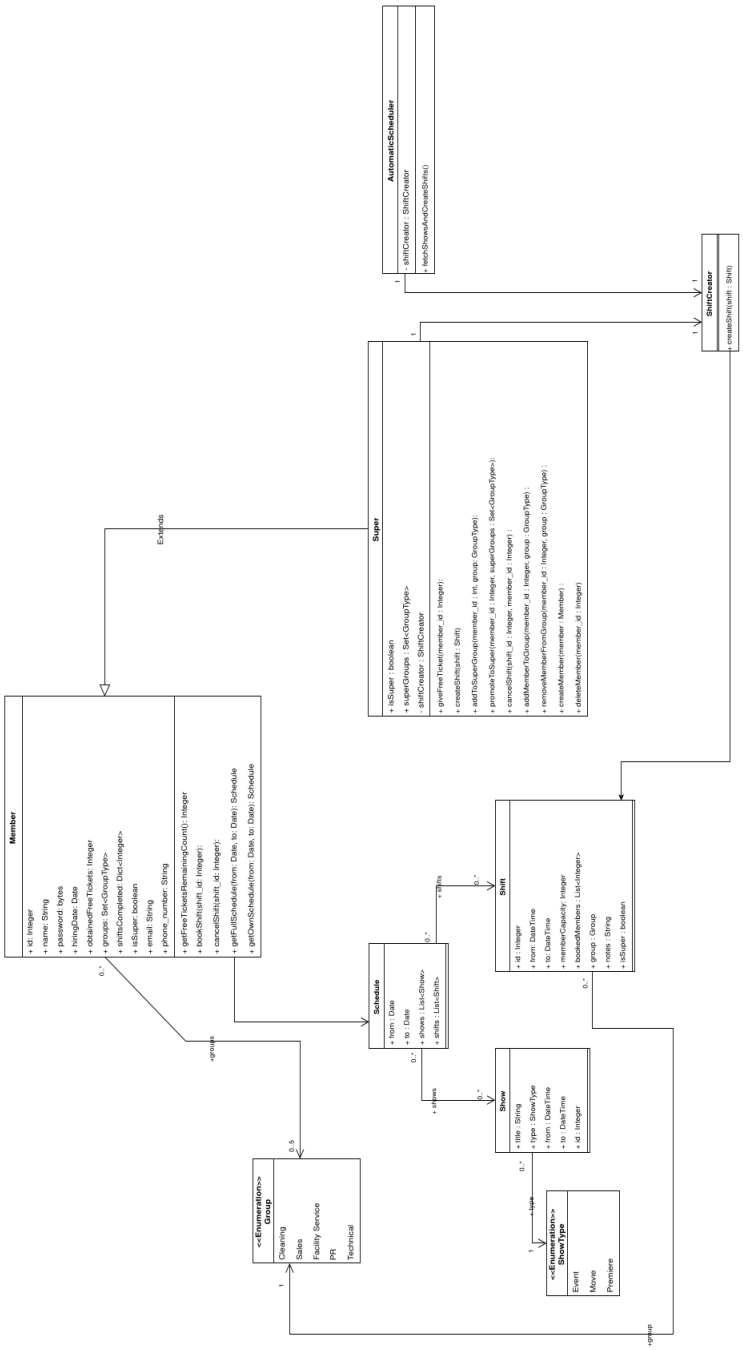
The figure below has zoomed in on the "Website" from the previous figure.



Malthe Hedelund Petersen - malpe21
Sofus Hesseldahl Laubel - slaub21
Jonas Bork Jørgensen - jonjo21
Mikkel Nielsen - mikkn21

I Class Diagram

Malthe Hedelund Petersen - malpe21
Sofus Hesseldahl Laubel - slaub21
Jonas Bork Jørgensen - jonjo21
Mikkel Nielsen - mikkn21



J Website Code

See the attached folder *dm571-project* and the README.MD file for instructions.

Malthe Hedelund Petersen - malpe21
Sofus Hesseldahl Laubel - slaub21
Jonas Bork Jørgensen - jonjo21
Mikkel Nielsen - mikkn21

K Wireframe