Introduction

This module's project the goal was to create a web application for booking meeting rooms in the new Sqills building. The main concern was to reduce the interruptions that happen when people enter the highly sought after meeting rooms that are already reserved. The created end product contains a desktop interface and a tablet interface, each with their own approach. For the tablet the goal was to make it accessible, as we want employees to actually use it. A set requirement was that implementation would not be a burden, but be an addition to the workflow. The desktop interface still aims to be as simple and accessible as possible, but here more functions are added to give the entire system some more depth.

Test_report describes how testing was applied to your project (150-300 words)

While developing your system, testing is very important. Most of us noticed this already during the module 2 project. For this project the front end testing was very hard, because there are a lot of javascript methods that use and produce DOM elements. We didn't really find out how to test those methods. We did test the methods on the front-end that returned something. We used Jest for this. On the front-end we make api calls to the back-end to request or edit data. We tested the api calls manually through postman and using the network tab in developer tools. We checked what json the server would accept and checked whether the front end really sent the correct json. When the data arrives at the back-end, it goes through the recourse classes, the data is turned into a Bean, we checked whether Jersey did this correctly manually. After it goes through the bean, the data will access the database in some way, we do this in the DAO classes. For every DAO class, we made a DAOtest which tests every method in the DAO's by mocking data into the database and checking whether right things were returned. We also did a lot of functional testing to test the system as a whole, of course we could not test all the edge cases in this way but we tested the very basic functionality.

Security_analysis describes how you addressed the relevant security concerns (or why you did not address them) (200-400 words)

Although we were not afraid of security vulnerabilities, certain measures were taken. Due to mostly internal usage, honor code, and limited functionality the threat of a security breach is not as big. Nonetheless, we still know about, and have prevented some, vulnerabilities in our system.

Firstly, there is no HTTPS, because we do not have SSL. This means there is a possibility for easy interception, a man in the middle attack. With an attack like this in a POST request, the user can tamper with the request and server does not have any way to ensure the data is not tampered. Nonce?

We stop the Cross-Site Scripting by using input sanitation on fields that allow the user to use strings (email and booking title)
SQL injection is not possible due to the created prepared statements.

We are not really vulnerable to CSRF as all requests that actually change something on the back-end are POST requests that require JSON as body. If this would require a cookie or token, the other tab/webpage will not have access to it, thus there are almost no vulnerabilities to worry

about here. The threat this would cause is not the greatest either, the worst that could happen is that a meeting gets edited or deleted.
Because we use session so it is very unlikely. Verified via api call, so you cannot hijack, due to the cookies being handled by json.

In case of an internal server error, the attacker does not get any extra information of the server as we disabled printing the stack trace to the user.

Authentication is handled by Google, which makes the probability of security failures on our end much smaller. The info of the user that is currently logged in is stored in a session, using the standard cookie system employed by Jersey, which contains long random strings as identifier.

One of the requirements for the project is that you have applied database transactions in a proper manner. What this essentially means is the following.

Our application will not have very many users. This meant that database speed was not a big concern as the provided database is more than fast enough to deal with the low amount of requests. To add some efficiency however, we share connections with methods not called directly from an endpoint. For instance, DAO methods used for validation get passed a connection for the sake of efficiency. We also disabled auto committing so that all endpoints are handled in a single transaction. Lastly we also used connection pooling to make sure that we can not surpass the maximum amount of connections to the database.

Technical aspects we're proud

We are proud of the structure of our project. We feel that all code is there were you expect it to be and is in conformity with conventions. Separation of concerns is also something that was taken to heart and implemented well. Almost all classes and methods do their task, no more, no less.

Great use was also made of stored procedures. Almost all of the access through the database was done through stored procedures.

We also feel our error handling is quite elegant. Is is done in a structured way by throwing exceptions. This allows us to easily relay the correct information to the user in case of an error.

Lastly we also did not forget that Java is an object oriented language. Where possible, we tried to take the concepts and conventions of object oriented programming to heart. Good examples of this are our model classes. These make use of inheritance to prevent code duplication.