Exercise Report Oliver Brunner

For implementing the task at hand my decision-making was mostly influenced by two factors: Speed of development and possibility for further development. The reason behind is that on the one hand I wanted to use the given timeframe to come up with a prototype as complete as possible while not sacrificing options to turn said prototype into a “real” product to be used by a customer.

I implemented the exercise using a quite standard approach: Using a cross-platform client that calls a REST service with a database in the background. One other idea I had was to implement everything as a single C++ application with a database, this however restricts the use of and the ability of the system to scale further by locking the database to a single system. In the following I will describe my technological decisions for each of the parts of the system.

Client

For the client I decided to go for a cross-platform app using the Flutter framework. Other candidates were C++/Qt, React Native or a web-based approach. All are well-established options throughout the industry and used for similar projects. I went for the Flutter option since it offered me the best overall outcome regarding my experience with using the framework and the resulting expected speed of development. The client is implemented in a relatively simple way. I made sure the UI classes are decoupled already from the underlying data-fetching mechanism. This can be easily extended into a larger system given more development time in the future. For the UI itself I decided to give the users’ focus on the open car rentals and not. Management of cars and customers moved to individual pages to be used when necessary since I considered them tasks that are less frequently done than managing rentals. This way the UI is much simpler and does not overload the user with options.

Backend and Database

For the REST-service I went for the combination of Python and Flask. The reason behind this decision was the rapid development speed it offers. Other candidates considered were a C++-Rest Service (higher implementation and setup complexity and runtime speed not really needed) and C#/ASP.net (higher complexity with lower expected development speed). I decided to not use the typical approach of using Flask with SQLAlchemy for the database but went for SQLite instead. The use of a relational database in comparison with NoSql-systems was preferred due to the structured nature of the data. This way I could set up and work with the database quickly, while ensuring it can easily be changed into a larger system like Oracle or MS-SQL. For the same reason I decided to keep the service itself as lightweight as possible; to be more easily able to exchange it in the future.

Next steps

Given there was more time to continue the project the next steps would be to improve the robustness and security of the application as well as finishing the required set of features. I went on fast when implementing, assuming a best case, since I wanted to get done as much as possible given the time. I therefore decided to reduce the effort for error handling for cases like no or invalid responses from service or database. This must be done before the system is ever to be used in a productive scenario. Same holds for security flaws. The most prominent that comes to mind is SQL-injection, against which the application is currently not shielded against. Further required features are encrypted communication and authentication mechanisms. Finally, before the system can go productive, it should be thoroughly unit-tested in all relevant components.