Ferchau Technical Assignment Report

Martijn Clemenkowff

1 User instructions

Navigate to the **Customers** (**Vehicles**) tabs to manage your customers (vehicles). Click the plus icon, or press **Command+C/Command+V** to create a new customer/vehicle respectively. Pressing the ellipsis presents the options to edit or delete a customer/vehicle. In the **Vehicles** tab you can see which vehicles are available or rented today and see the rented mileage which is automatically maintained.

To create a new ride, press the plus icon in the **Ride Calendar** tab, or press **Command+R**, or click a customer/vehicle in the respective tabs to create a ride for that customer/vehicle. A car or calendar icon denotes whether a vehicle is rented today or has a planned ride. To complete a ride and move it to **Ride History**, click **Edit/Complete** in the ellipsis menu and then click **Complete Ride** after you have filled in the odometer fields; the vehicle mileage will automatically update.

2 Design choices

The application stack is as follows:

- MySQL/SQLite database
- Django REST API
- Swift graphical user interface

The application is designed around an SQL-based database, exposed through a Django REST API. Having had no prior experience with Django or SQL before this assignment, Django drew me as a framework that allows quick iteration, has wide community support, and as being appropriate for the scale of the fictional car rental company while still allowing for potential growth of the business. FastAPI would have been an alternative option, but may have been overkill for the request volume of a car rental company. The choice for whichever database (supported by Django) the user wants to use with the system also plays a role in this; SQLite is lightweight and portable, but MySQL would be more scalable and is more flexible in how it can be hosted. At least for SQLite vs. MySQL, the API is agnostic to whichever is used. For this specific application, I believe MySQL would be the best choice.

All business logic is handled in the Django API, leaving only the GUI implemented in Swift and ensuring consistent behavior and validation across any possible different (future) UI implementation. My choice for Swift was frankly mainly because I prefer to work on Mac and I was most inspired to go for a native MacOS app, having spent a lot of time in the language recently. In this case I sacrificed cross-platform capabilities for native behaviour. SwiftUI is the logical choice here over UIKit in regards to future proofing and ease of development.

The Swift user interface follows a model–view–viewmodel pattern, with **CustomerViewModel**, **VehicleViewModel** and **RideViewModel** handling the endpoints from the respective tables and the corresponding **ListViews** rendering the GUI by calling functions on the **ViewModels**. Data entry is delegated to the respective **FormViews**. Enforcing separation of concerns, I believe this pattern allows for the cleanest iteration.

In designing the GUI, I initially wanted to go for a calendar/timeline view to render the rides, but in the interest of time decide to implement simple list views for all tables instead. However, when implemented thoroughly, I do believe such a representation would be a strong addition to the application. Further simple improvements that could be made to the user experience are sorting and filtering functionality and better input validation.