Design Document: AutoDirect

December 10, 2021

Today, Senso.ai announces AutoDirect, a disruptive new web application that enables banks to directly extend loans to consumers looking to buy a car.

AutoDirect is an all-in-one solution that allows buyers to quickly find their perfect car and get instantly preapproved for an auto loan. Based on buyers' financial information, AutoDirect recommends car options that buyers can afford with a preapproved loan, all before they set foot into a dealership, which allows buyers to shop and compare cars with financing in mind. AutoDirect also allows buyers to incorporate any changes made at the dealership directly into the web application.

Customer Problem

Banks face three major problems in auto lending:

- Buyers do not typically apply for auto loans directly through banks, as the borrowing experience is decoupled from the purchasing process.
- Dealerships often persuade the buyer to incorporate add-ons and even change the car they were planning to finance, which invalidates any previous preapproved loan made with the bank.
- Unnecessary costs are incurred to incentivize dealerships for loan referrals.

Solution

The new AutoDirect product from Senso.ai enables buyers to easily view an assortment of cars with preapproved financing, customized for their individual financial profiles. Buyers are empowered to choose among affordable cars, and to compare cars with financing in mind. AutoDirect allows banks to provide these services to their clients, creating a win-win situation between buyers and their banks.

AutoDirect facilitates a direct lending process for banks looking to lend to prospective car buyers. It removes the need for paying rebates to dealerships. The preapproval process is automated, which saves time and hassle in the lending process. Buyers can also adjust their loan parameters (i.e., add-ons or discounts) if changes are made at the dealership, which allows banks to get ahead of any changes in the field.

What do buyers need to do?

It's easy! Just go to https://autodirect.tech, fill in some financial information, and get immediate access to hundreds of cars for which they are already preapproved. Choose a model and be on your way to your new car!

SOLID Principles

1. Single Responsibility Principle

Every class in our project has a single responsibility, which means they manage a single functionality in our program. For example, we have a class `SvcUserLogin`, that is responsible for user login. In this class, we can get user information, or create a new user account. And another class `SensoAPI`, is used to get the loan offer. Additionally, we have separate classes for each database, searching, etc. Each of the `TableOffers`, `TableCars`, `TableUsers` classes are responsible for accessing only one table in our database, this is another example of the Single Responsibility Principle.

2. Dependency Inversion Principle

We introduced an abstraction layer (using dependency injection) between high-level and low-level classes in order to avoid having dependency between each type of class. As a result, we adhere to CLEAN architecture and also decrease the reliance of our classes on one another. We mainly use this principle by creating interfaces to allow Application Business Rules to call methods from our Frameworks and Drivers classes.

3. Interface Segregation Principle

We are following the Interface Segregation Principle because a gateway class implementing one of our interfaces only needs to talk to one table in the database, it does not need to interact with other classes.

4.Liskov Substitution Principle

Inheritance is not something heavily implemented in our code,however; we do use it in the databases between Table and TableCars, TableOffers, TableUsers in the methods `resultSetToList`. This method does not change from the parent class to the children class which means that it follows Liskov Substitution Principle.

5. Flaws in our code

Early on, we realised that our code design did not follow CLEAN architecture because Frameworks and Drivers were interacting with Entities and Use Cases. We modified this by removing the dependency aspect and by using Dependency Injection between Use Case classes and Frameworks and Drivers classes (described below).

Clean Architecture

In our project, we use all four layers of the clean architecture. Our entire Frontend repository is part of our Frameworks and Drivers layer and the following is the organization of our Backend repository packages w.r.t. clean architecture:

- Enterprise Business Rules: `entities`
- Application Business Rules: `services`, `utils`

- Interface Adapters: `endpoints`
- Frameworks and Drivers: `database`, `upstream`

The `entities` package contains classes such as `EntCar` and `EntUser`. These entities contain representation data and are used by classes in the `services` package to perform the program's application business logic. Furthermore, classes in the `services` package need to make use of classes in the `database` package (which contains classes that talk). As we are following clean architecture, we use interfaces and the dependency injection to manage the dependency of the `services` classes on the `database` classes.

The `ApiEndpoints` class is the only class in the Interface Adapters layer of our architecture. Note that `ApiEndpoints` is both a presenter and a controller as it both receives and sends data to and from the frontend. This behaviour comes from the fact that we use Spring as our backend web framework, and this is how Spring manages communications with the frontend. `ApiEndpoints` class employs the facade design pattern and delegates all of its behaviour to classes in the `services` package (which also follows the clean architecture).

Our main class; however, violates CLEAN architecture. Ideally, the main class would not be part of any layer, which is why it is allowed to pull different objects together. In our project, we use Spring which facilitates the use of endpoints, these act as controllers and presenters in the main class. This allows our Controller/Presenter to interact with Frameworks and Drivers. This is a violation, albeit an involuntary one, of CLEAN architecture.

Design Patterns

1. Dependency Injection

We are intensively using the Dependency Injection Design pattern. Our program relies heavily on interfaces, most of which are created in order to adhere to CLEAN Architecture. We use Dependency Injection to allow classes from a higher level to interact with classes in a lower level. We use the dependency injection to maintain the use of the dependency inversion principle, because they are heavily reliant on one another.

2. Private Class Data

Another design pattern that we are using is the Private Class Data design pattern. The goal of this design pattern is to reduce exposure of attributes by making the variables private. We use this pattern in order to prevent the manipulation of certain parts of our program and keep things compartmentalized. This design pattern was not introduced in the scope of our course, however it is relevant to our coding structure.

3. Facade

We use the Facade design pattern in the main entry class of our backend, `ApiEndpoints`. `ApiEndpoints` delegates all operations to classes in the `services` package (which is a hallmark of the Facade design pattern). The only responsibility of the `ApiEndpoints` class is to send and recieve data to and from the frontend, This is done by the Spring web-framework (and so there is no code related to this inside the methods, only delegation).

Use of Github Features

1. Github Projects and KANBAN Board

One of the Github features that we have been using is Github Projects. Specifically, the integrated KANBAN board. We use github issues as cards in the board and have found it very useful to assign issues to specific team members! The following are the lanes of our KANBAN board and how we use them:

- TODO: Issues to be started.
- In Progress: Issues currently being worked on.
- Reviewing/Testing: Issues that are under reviewing or currently being tested.
- Complete: Issues that have been completed.

2. Branches, Pull Requests, and Reviews

We are also using pull requests. Based on task assignments from the KANBAN Board, every member works on their parts. We've been creating new branches, to fix a bug or add new features. When an issue is completed, one member make a pull request, which will be reviewed and, usually, approved to merge into main by another member. Often, members will have specific comments on a pull request that their creator must address before it can be merged into the main code.

3. Using Issues to Store Useful Resources

Another feature we've been taking advantage is using issues to share useful resources. We have two pinned "resources" issues, for the frontend and backend repositories, respectively. For frontend, we stored useful online tutorials for HTML, CSS and Javascript. And for backend, we have resources for spring framework, SQl basics, PostgreSQL, JDBC, and Java.

4. Repository Organization

We use Github features to organise repositories. We have two repositories: "Frontend" and "Backend". This helps us organise our project in a meaningful manner. Our frontend is organised into 3 folders: "assets" for static assets like images and icons, "js" for JavaScript, and "css" for stylesheets. Our Backend is organised into a package for tests and for main. The main package contains the packages `database`, `entities`, `interfaces`, `services`, `upstream`, and `utils` as well as the class `ApiEndpoints.java`. This helps us differentiate classes according to their different functionalities. It also makes our code easy to read for those who need to check or use our code.

5. Github Pages

We are using Github pages for our frontend. Github pages is a deployment tool that supports the creating of websites. It functions as a host for the website and has allowed us to view changes and new updates.

Code Style and Documentation

There are no substantial errors and warnings in the IDE. There are false-positive typo alerts, and SQL syntax warnings that require configuring the IDE to connect to the database server.

Javadoc is used considerably throughout the project. There are still a few places that could use more documentation, and we intend to add more documentation going forward. We also intend to add class-level documentation as part of our next steps.

We believe that a Java programmer can open any code file in our project and understand the code. We made every attempt to ensure the code is clear on its intentions and documented thoroughly. However, we are aware that there is room for improvement in our documentation.

Tech Choices

Backend

Java seemed like an appropriate programming language for us to use. We all have a strong desire to learn the language and it is useful for object-oriented projects.

We chose Spring.io as our backend web framework because it provides rapid API development functionality with a simple annotation-based syntax, which is superior to other frameworks or to writing the API features from scratch. We also chose DigitalOcean as our cloud provider because they grant each person a $100 credit for 12 months, and it provides a simple VPS solution for our database and API hosting needs.

Next, we use a relational SQL database because the structure of our data is naturally tabular, which is suited for a SQL database. We chose PostgreSQL specifically because it is a popular and mature open-source SQL database solution, and it also has good full-text search capability, should we decide to implement that feature.

Frontend

We chose to implement our frontend as a web interface because of the ease of development and cross-platform compatibility.

We are using HTML, CSS, and plain JavaScript at this time because our frontend is a fairly simple two-page static website, which means that it does not warrant the use of a large framework, as this will introduce additional complexity.

Testing

We have a lane which is specific to testing in our Github KANBAN board ("Reviewing/Testing"). We implemented multiple unit tests. Our testing was mainly focused on the services, upstream and database methods. This is to test the functionality of each of our methods and to make sure that they are working properly. Most methods were tested in a multitude of ways in order to test them thoroughly.
We use unit tests to verify the functionality of our code and to find errors and bugs early on. We have separate tests for database, services and upstream. For instance, when testing TableOffers, we implement test functions to check whether we could successfully set users, add offers, mark offers, remove offers, etc.

We implemented multiple unit tests. Our testing was mainly focused on the services, upstream and database methods. This is to test the functionality of each of our methods and to make sure that they are working properly. Most methods were tested in a multitude of ways in order to test them thoroughly.
We are using a "peer review" method, this means that before accepting a PR, reviewers pull the branch and test the added functionality on their machine. This enables our code to be reviewed by more than one person and decreases chances for errors.

Refactoring

In the past week, we have spent much time discussing how we can refactor our code in order to adhere to the CLEAN architecture. This multi-session discussion was the biggest refactoring step that our group has partaken in thus far. The results of this refactoring step can be seen in the Backend repo PR "Reorganized codebase to match with our updated CRC card design which ...". We have also intermittently refactored our code throughout the semester.

Code Organization

Our code is organized into packages and classes based on major functionalities. Further, classes are grouped semantically, which makes the code easy to navigate. At the root level, there are the `main` and `test` folders, customarily representing functional code and testing code. Within the `tech.autodirect.api` folder under `main`, we have an entry point class `ApiEndpoints`, which is an annotated Spring REST application class, that serves our API endpoints via Spring. Our packages are organized as follows:

- The `upstream` folder contains classes that access upstream APIs.
- The `services` package contains classes that perform the major functionalities exposed by our API endpoints.
- The `utils` package contains project-level utilities.
- The `database` package contains classes that enable interaction with the database.
- The `entities` package contains project entities, which are mainly data classes that represent different data.
- The `interfaces` package contains interface definitions that enable dependency injection for upper level code to interact with lower-level code.

Functionality

The program loads and stores state. This means that when the URL of our website is copied, it saves the state the website is in. That increases user accessibility because it allows the user to transfer between devices and not lose the progress made so far in the website. We use a PostgreSQL database server to store and load information that needs to persist across time. The details of these examples of such behaviours are as follows:

- We have a `cars` table and a `users` table that are initialized once during deployment, by the `InitDatabase` class. They store our car collection and user information respectively.
- We have an `offers` table for each active user, created dynamically by the `TableOffers` class, that contains loan offers based on the user's financial information. Each offers rows has a `claimed` field that supports a "cart" functionality, which allows users to claim offers they are interested in.
- Each row of the `users` table also refers to an `offers` table corresponding to that user.
- The program accesses data via classes under the `database` package, using interfaces for dependency injection to adhere to CLEAN architecture.
- The program aims to provide possible preapproved loans based on the user's credit score and Senso rate API.

Summary

We have finished implementing our project; however, we had to remove some features that we wanted to implement due to time constraints. We do not filter using car specifications, but we do filter using monthly budget and down payment.