Design Document: AutoDirect

Ruofan Chen, Samm Du, Nada Eldin, Shalev Lifshitz

December 12, 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 pre-approved for an auto loan. Based on buyers' financial information, AutoDirect recommends car options that buyers can afford with a pre-approved 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 pre-approved 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 pre-approved 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 pre-approved.

Brief Instructions

Upon accessing our website, users can search for available cars by ascending or descending price. However, to determine and include loan offers for each car, users must click "Agree & continue", which signs them into our system (auto-generated user ID). Then, once signed in, users can enter their down payment and monthly budget (already auto-filled) and search for cars with their desired sorting parameters. Specifically, the two buttons at the top left of the blue bar allow users to choose which attribute they would like to sort by and whether they want results to appear in ascending or descending order.

After claiming some loan offers, users can click on "See all offer details", where they can see the details of each loan offer and also update the principal for each offer (i.e., with add-ons acquired at the dealership). After updating their principal and clicking "Update", the loan information for users will update if a loan offer is approved by the Senso /rate Api for this new principal. Otherwise, if not approved, users will receive a notification saying that a loan offer has not been approved for this new principal amount.

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. Another class `SensoAPI`, is used to get the loan offer. Additionally, we have separate classes for each database table, a class just for 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 to avoid duplicating code. These classes follow the Liskov Substitution Principle as the methods in Table can be called in the children and still behave the same (i.e., resultSetToList).

5. Flaws in our code

Early on, we realized 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 receive 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 makes 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 of is using issues to share useful resources. We have two pinned "resources" issues, for the frontend and backend repositories, respectively. For the frontend, we stored useful online tutorials for HTML, CSS and Javascript. And for the backend, we have resources for spring framework, SQL basics, PostgreSQL, JDBC, and Java.

4. Repository Organization

We use Github features to organize repositories. We have two repositories: "Frontend" and "Backend". This helps us organize our project in a meaningful manner. Our frontend is organized into 3 folders: "assets" for static assets like images and icons, "js" for JavaScript, and "css" for stylesheets. Our Backend is organized 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 to host our frontend. It automatically serves the latest changes on the specified branch (currently the main branch) to our domain, without us needing to manually configure a web server. It also allows us to view changes and new updates quickly, as they get merged.

6. Github Actions

We are using Github Actions for automated testing. Whenever a PR is made, Github automatically checks that our tests pass before we review. This is very useful, as we don't

need to pull the branch to check that our tests run.

7. Github Release

We use the GitHub Release feature specifically on the backend repository, in order to mark a state in the main branch that has major functionality working, without any significant breaking behavior, and is ready to be deployed to the API server. In each release we include a `.war` file created at that point in time, which is the binary file that is ready to be deployed to a Java Servlet server (we use Apache Tomcat).

Code Style and Documentation

On the backend, 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. 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.

On the frontend, the code is formatted with Prettier — an automatic code formatter that supports common frontend file types, including HTML, CSS, and JavaScript. There are ample comments inside JavaScript files as well, whereas in HTML and CSS comments are used sparingly to mainly indicate separation of different sections.

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 a 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. We used Mustache.js as a templating engine, because it helps simplify our code, and is very lightweight itself.

Testing

We have a lane that 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 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 week leading up to Phase 1 submission, we 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 took this semester. The results of this refactoring step can be seen in the Backend repo PR `#41` "Reorganized codebase to match with our updated CRC card design which ...".

We have since continuously evaluated our code organization, intermittently refactored our code, and made necessary changes that make our code more flexible and maintainable.

Code Organization

Our backend 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 the `services` classes to use `database`/`upstream` classes.

On the frontend, we have two main HTML files, `index.html` is the initial discover page, which is the page that allows the user to configure their options and discover cars with loan offers; `details.html` is the details page, which displays the list of claimed offers, and allows the user to see the details for each offer.

Each HTML file also includes a set of templates at the end, in `<script type="text/html">` tags. These are HTML templates are used by Mustache.js, which dynamically renders either repeated code, or code that is used to display different content, such as each of the car/loan listings on the discovery page, or the loan details on the details page. We did not separate these templates out to separate files, in order to improve speed and performance, and cut down on TCP requests.

Our frontend folders are organized based on file type, as follows:

- `assets`: static assets including images and icons
- `css`: CSS stylesheets
  - `minireset.css`: open-source CSS stylesheet that resets some default broswer styling to make custom styling easier
  - `main.css`: common styles across all pages on the frontend
  - `index.css`: styles exclusive to `index.html`
  - `details.css`: styles exclusive to `details.html`
- `js`: JavaScript files
  - `mustache-*.js`: Mustache.js — open-source templating engine
  - `api.js`: provides the `api` object with methods that allow for interaction with the API backend
  - `main.js`: scripts that are used across the entire frontend
  - `discover.js`: scripts that are used only on the discover page (`index.html`), including the `onPageLoad()` function that gets run upon every page load of `index.html`

- `details.js`: scripts that are used only on the details page (`details.html`), including the `onPageLoad()` function that gets run upon every page load of `details.html`

We received feedback during phase 1 about repeated code on the footer of our pages. Since our frontend is reasonably simple with only two pages, we decided to keep the footer code directly in each page, rather than loading them dynamically, in order to improve performance and minimize layout shift. We also discussed this with Antoine and he said it was okay to leave as is. In a more substantial website, we can introduce a build process that injects the footer code into each page during build time, so we can keep the footer separate for development, while maintaining good performance after deployment.

Functionality

The program loads and stores state. The URL of our website encodes certain information that identifies a user session, and the backend saves the state of the user session inside the database. Using the same URL on any device at a later time would recall the same session that the user was on before. This 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 row 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 pre-approved loans based on the user's credit score and Senso rate API.

User Credit Scores

Currently we expose the user ID in the URL parameters for demonstration purposes. We decided to allocate the last three characters of the user ID to represent the user's credit score, which makes it easier to debug and show the effect of having different credit scores. Simply changing the last three digits of the user ID in the browser address and reloading

the page would create a new user with a different set of offers. At this time, we generate the credit score on the frontend (in a range between 300 and 900), encode it in the user ID, and the backend simply slices the last three digits and returns it as the user's credit score for use in the rest of the application.

Of course, in a real production environment, the user's credit score would be retrieved from the bank or the credit bureau and would not be sent to the frontend, the user ID would be generated on the server side, and sessions would be saved as cookies or JWTs. But since we chose not to implement a full-fledged authentication system, we use the URL to keep track of user sessions for simplicity.

Loan Offers

We use the Senso `/rate` API as a loan pre-approval system. Specifically, we query the Senso `/rate` API with a user's financial information (in addition to information about the target car) and use the query result to determine whether a loan for this car has been pre-approved or not. If the loan has been pre-approved, we consider this a viable loan offer for the user.

Accessibility Report

Principles of Universal Design

- Equitable Use: We have image labels (alt text) for all images on the frontend that allows for use of the TTS feature in the user's operating system or browser. In the future, we can incorporate a way for the user to customize our website so that they would be able to interact with our program in a way that is easiest for them. This can be done by including an option to change the appearance of the website (dark mode, high contrast mode, etc.). We can also include a way for the user to interact with our website using a microphone. This can be incorporated into our search feature to allow the user to verbally search for the car specifications they want.
- Flexibility in Use: Our website runs on most contemporary operating systems and web browsers. Our visual elements are located in accessible parts of the screen that allow users to easily access them (on a large or small screen), and employ responsive design principles. We also automatically save the user's session on the backend, which allows users to switch devices as they wish without losing progress.
- Simple and Intuitive Use: We only collect two pieces of information from the user, and perform as many operations automatically as possible. There are only two pages on the frontend, and navigation is very straightforward. We use icons and transition animation to accompany text so that it is easy to understand what each control is used for

- Perceptible Information: We use pictorial methods to show essential information. We provide a contrast between essential information and it's surroundings. We include a small description of what our website is and what it is used for.
- Tolerance for Error: We have ample area around our interactive features which allows the user some leeway for clicking or tapping options, even on a touchscreen. We considered adding a pop-up screen on our website that says, "Are you sure you want to claim this offer?". However, we felt that this added too much friction in our application as, currently, users can easily unclaim an offer.
- Low Physical Effort: We have strategically aligned our interactive features in accessible parts of the screen to allow mobile users to choose options with minimal effort. We also automatically save user's budget and down payment upon every search to allow users to return to where they left off at a later time, without having to re-input their information or car choices (all the users needs to start back is a copy of their URL).
- Size and Space for Approach and Use: We made the effort to have high contrast colors so that users can see the information clearly, although Senso's green branding color does not have a good contrast against a white background; this is something that can be improved in the future. All text has been configured to appropriate sizes so that users can clearly see it. Our website allows users to zoom in with their browsers to see information that they might not be able to see clearly otherwise. We also position objects on the screen so that they are directly in the user's line of sight.

Target Audience

We could like to market our program to banks. The bank would buy our program then incorporate it into their own website (or suite of web services), similar to how the program canvas works. The bank would allow our website to access the user's financial information and based on that information would provide appropriate loan offers for the customer. This would help the integration of banks in the auto trade industry and help banks interact directly with their customers (the buyer).

Who cannot use our program

Our program specifically caters to banks, any other categories will not be able to use it. Anyone looking to buy cars can use our program through the bank; however, it only caters to that specific type of situation. If the user was in the market for any other product, they would not be able to use our program.

Progress Report

Ruofan Chen

Since phase 1, Ruofan has attended group meetings and participated in presentations.

Samm Du

Since phase 1, Samm has completed the entire frontend functionality. A notable PR is `#14` Details page complete except the ability to update loan principal, where most major features of the details page have been completed. Since then, more functionality and modifications have been made to the details page. He participated in presentations, design document drafting, major design decisions, troubleshooting, and code review. He also performed all server deployments to DigitalOcean.

Nada Eldin

Since phase 1, Nada has contributed to the backend repository by adding tests for most classes and methods that did not have sufficient coverage. One of Nada's notable PRs is `#82` Tests/table users and table cars. This PR included most tests to TableOffers, the biggest Table class. Another notable PR that she assisted in is `#59` Added tests for TableOffers and updated TableOffers, also merged with feature/SvcSearch branch while working on this. Nada learned how to use BeeKeeper Studio, which is a SQL client that allows interaction with SQL databases and facilitates design of database schema. She also helped prepare presentations and was a deciding factor on how the process of the presentation would go. She wrote up a significant amount of the design document which included SOLID Principles, Design Patterns, some of the github features and testing. She also incorporated all the feedback given after phase 1 and made sure that the design document was up to date. Nada also wrote up most of the accessibility report.

Shalev Lifshitz

Since phase 1, Shalev has built almost all backend functionality and many of the associated tests. Shalev also helped prepare and present presentations, and contributed to the design of this design document. The biggest feature Shalev added since phase 1 was the search functionality. One of Shalev's notable PRs is `#57` Added search functionality. This PR added most of the key search functionality in our program (however, search has changed and evolved since this PR). Another notable PR is `#95` better tables where Shalev used inheritance to greatly improve the `database` code and reduce code duplication by a significant factor. This change proved extremely useful in future PRs since new methods in the `database` classes could now be simple one-liners.

Epilogue

We have finished implementing our project; however, we had to remove some features that we wanted to implement due to time constraints. The main feature we had to remove was the full-text search. Instead, we only support sorting by different loan offer features (interest rate, car price, etc.) and we can choose whether to sort in ascending or descending order.