

Destinex - Design Manual

Professor: Joshua Stough

Product Owner: Nolan Lwin

Scrum Master: Chang Min Bark

Developers: Hung Ngo, Hung Pham

Introduction

Destinex is our solution to the problem of overseas gifting. It is a web-based platform where users can request other people living in foreign countries to deliver a gift or complete a service. The website has two interfaces: the front-end and the back-end. These two components are stored separately in their own directories and communicate using methods. The front-end is developed using React.js, HTML, and CSS. The back-end is developed using Java SpringBoot, with the unit tests using the Spring Test framework. It contains several different pages, including a signup/login page, a home page, and pages related to making the order. We used RESTful APIs to communicate with the MongoDB server, where we store information about the users and jobs. We also used the Nominatim Map API to convert addresses to geographic coordinates as it was free as opposed to Google Maps' API.

Users are automatically assigned to be wishers or the default user role. If they want to become a granter/provider role, they would have to register to become providers on a separate page.

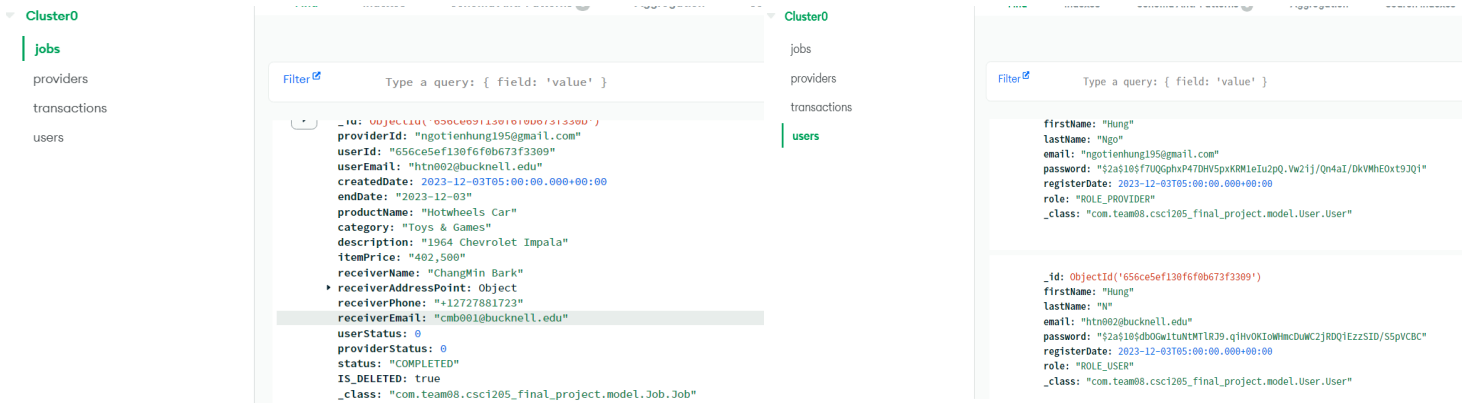
User Stories

Melissa Miller & Camille Dubois - We were able to complete this pair of user stories. Melissa wanted to send flowers to her friend in Paris for their birthday. Camille, on the other hand, is a resident of Paris who wants to spend some of her free time earning some pocket money. Destinex is able to achieve this by allowing Melissa to make a request to send flowers to a certain address in Paris, which pops up as a job on Camille's app. Here, Camille is able to accept the job and complete it, whereafter she is paid for her work.

Eddie Hall & Krit Lohia - Similar to Melissa Miller and Camille Dubois' case, we were able to achieve their user stories.

Jackson Zhang & Terry Gray - Similar to the first two user stories, Destinex is able to accomplish their user stories as their stories are similar in structure to one another.

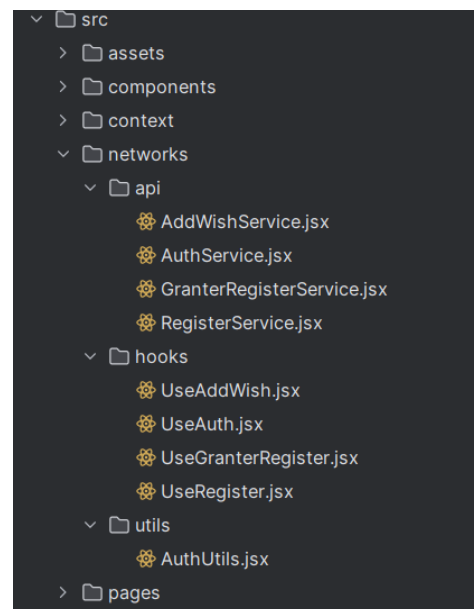
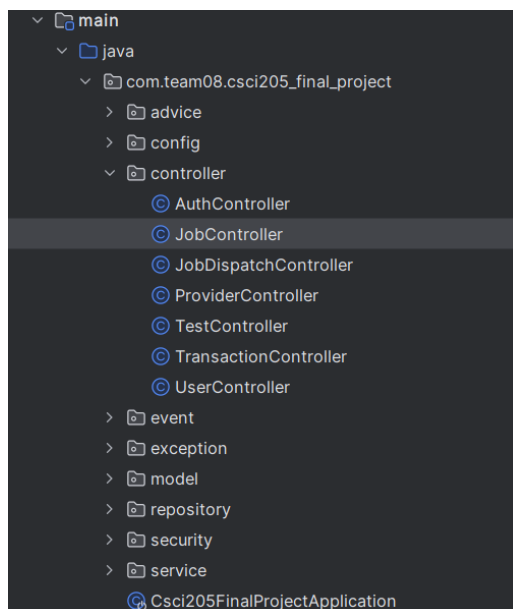
Hung Min Lwin - Hung Min Lwin is an administrator who wishes to look at past records for customer support reasons. Destinex is not directly able to accomplish this as we have not created a page specifically for the admin role. However, the administrator can log into the MongoDB database website to access the data on the server. Here, the admin can access the different collections of data on the server. An example of this can be seen in the images below. If we are to completely achieve this user story, we would need to implement an admin portal and page to access this data.



OOD

MVC

In our implementation of Destinex, we adapted the MVC design pattern. In simple terms, our model is everything in the back-end that excludes the controller package. This includes packages/directories, such as the “model”, “service”, “security”, and “repository” packages. The view is represented by the front-end, where our interface is coded in React components (HTML/CSS). The controller is located in the back-end’s “controller” package as well as the front-end “networks” module. It handles the network requests from the web interface (front-end) and the response from the server.



Front-end

- 1. Pages and Components:** In our front-end design, we implemented some OOD principles, such as splitting up a page into individual components. This helped us manage our components and design easier as modifying one component did not change the other components. Furthermore, it helped with code readability and reusability as we just needed to reference these components in the HTML portion of the pages.
- 2. Fallback UI:** Another OOD principle we incorporated into our front-end design was a fallback UI where users will be redirected to an error page when they try to access an unknown page. Users will also be redirected to the signup/login page if they are not authorized (logged in) to access certain pages like creating a wish/job or trying to look at the list of jobs in their area.
- 3. Prop Types:** Properties were passed to static UI components to represent different states so that they will change according to that state. An example of this is the different header type that is displayed depending on whether the user is logged in or not. If the user is logged in, the logged-in state is passed to the header component, so that it shows the username and avatar instead of the sign-up and sign-in button.

```
1 > import ...
7
8 function WishFormSummary() {
9   return (
10    <div className="wishSummary">
11      <Header></Header>
12      <Review></Review>
13      <OrderSummary></OrderSummary>
14      <Payment></Payment>
15      <Footer></Footer>
16    </div>
17  );
18 }
19
20 export default WishFormSummary;
```

Back-end

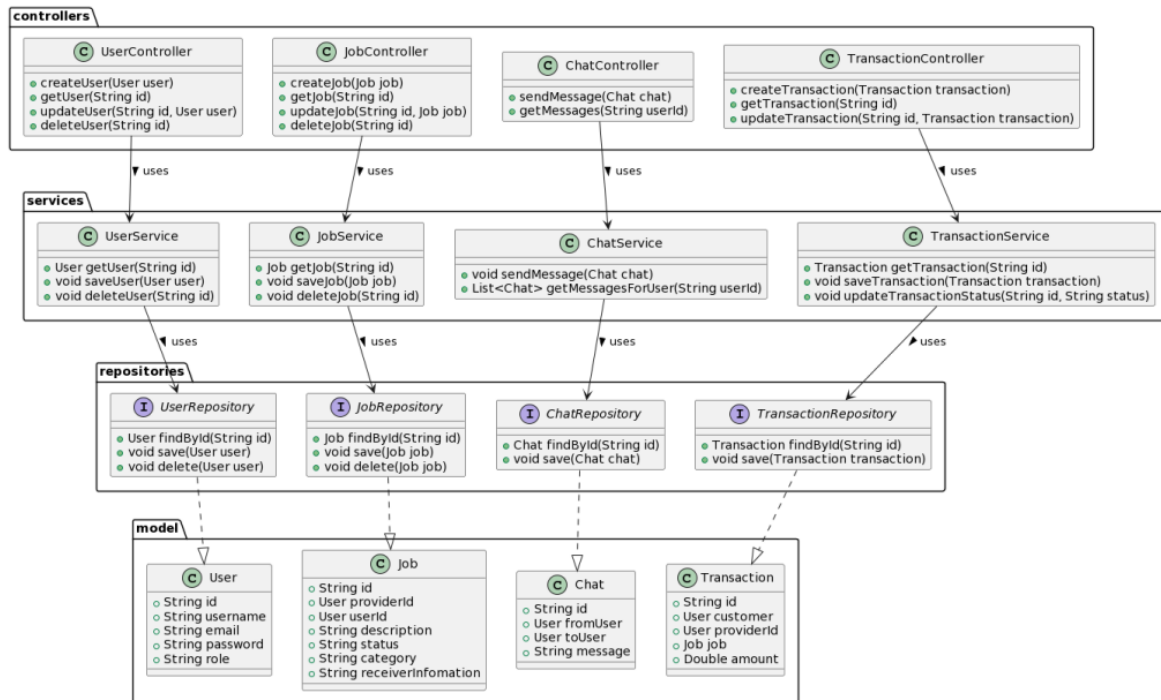
1. CRC Cards:

We've designed 3 CRC cards corresponding to 3 models of our system. The User class encapsulates the end-user functionalities of the system, handling personal information and job postings, while interacting with the Provider and Authentication system. The Provider class extends User capabilities to service offerings, responding to job dispatches and updating statuses, collaborating with Users and the Job Dispatch Service. Lastly, the Job class focuses on the lifecycle

User	Provider	Job
Register and manage user account information	Receive job offers in nearby locations	Track job status (pending, in progress, completed, etc.)
Request delivery Job	Accept/Reject jobs	Assign job to a Provider
Make payment for a Transaction	Store jobs information	Store receiver information
Communicate via Chat regarding a job	Update job information (job status, etc.)	Store job specific information (job description, price, etc.)
Provide feedback and ratings for delivered jobs		

of job postings, from creation to completion, and serves as a bridge between Users and Providers, facilitating the assignment and tracking of jobs within the system.

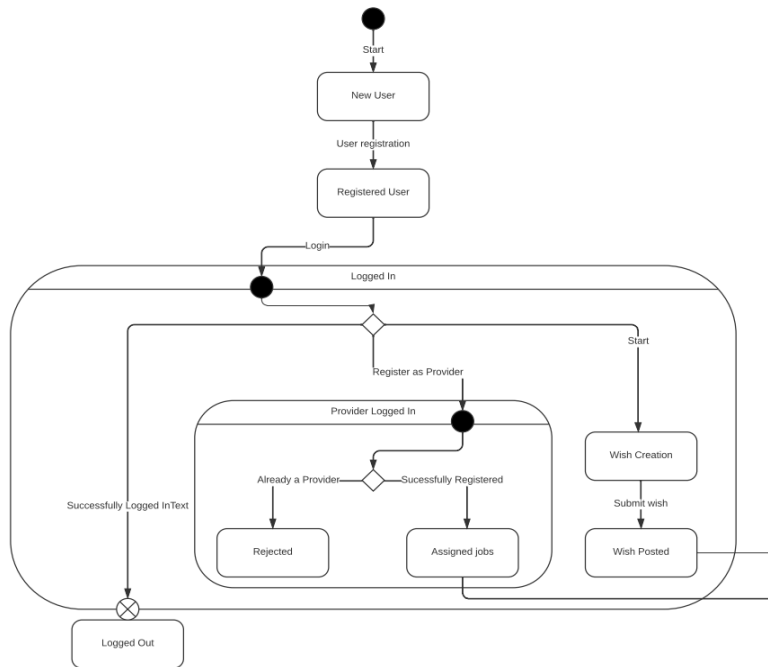
2. UML Class Diagram:



We've designed an UML class diagram according to the Model-View-Controller (MVC) pattern. Controllers serve as the system's interface, handling user interactions, jobs, chats, and transactions. The corresponding services encapsulate the core business logic, interfacing with specialized repositories that manage data persistence. At the foundation are the models—User, Job, Chat, and Transaction—each representing the system's key entities with their specific attributes. This architecture ensures a separation of concerns, facilitating maintainability and scalability of the application.

Note: Chats and Transactions are the features that we planned to implement, but we did not have enough time to work on.

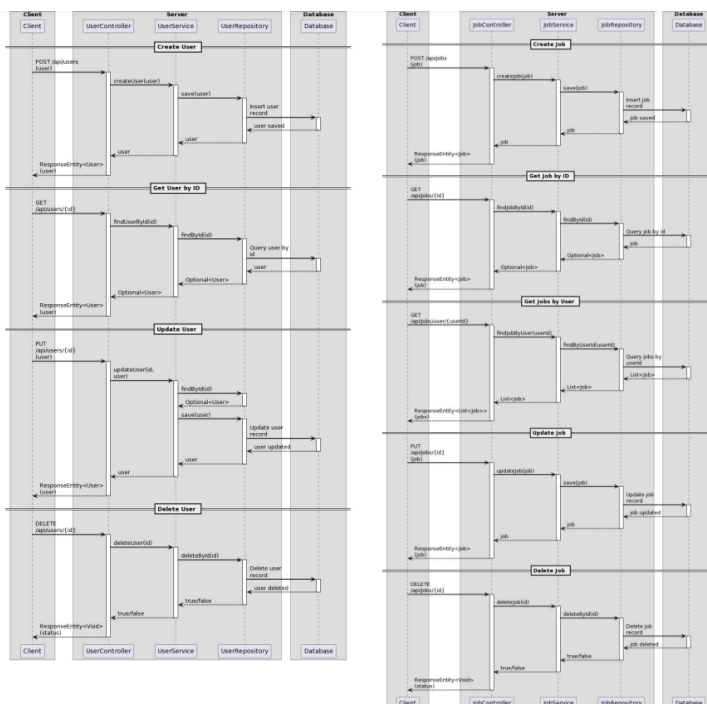
3. UML State Diagram:



The UML State Diagram shows the pipeline of our application. New users can register and log in, with the system handling both successful and failed login attempts. Once logged in, users can elevate their status to a provider unless already designated as such, in which case they are rejected. Registered users can also post jobs, which then enter a pending state until dispatched to providers. The system accommodates concurrent job dispatching, handling multiple jobs simultaneously through

multithreading, which allows providers to accept or reject jobs, and ultimately complete them.

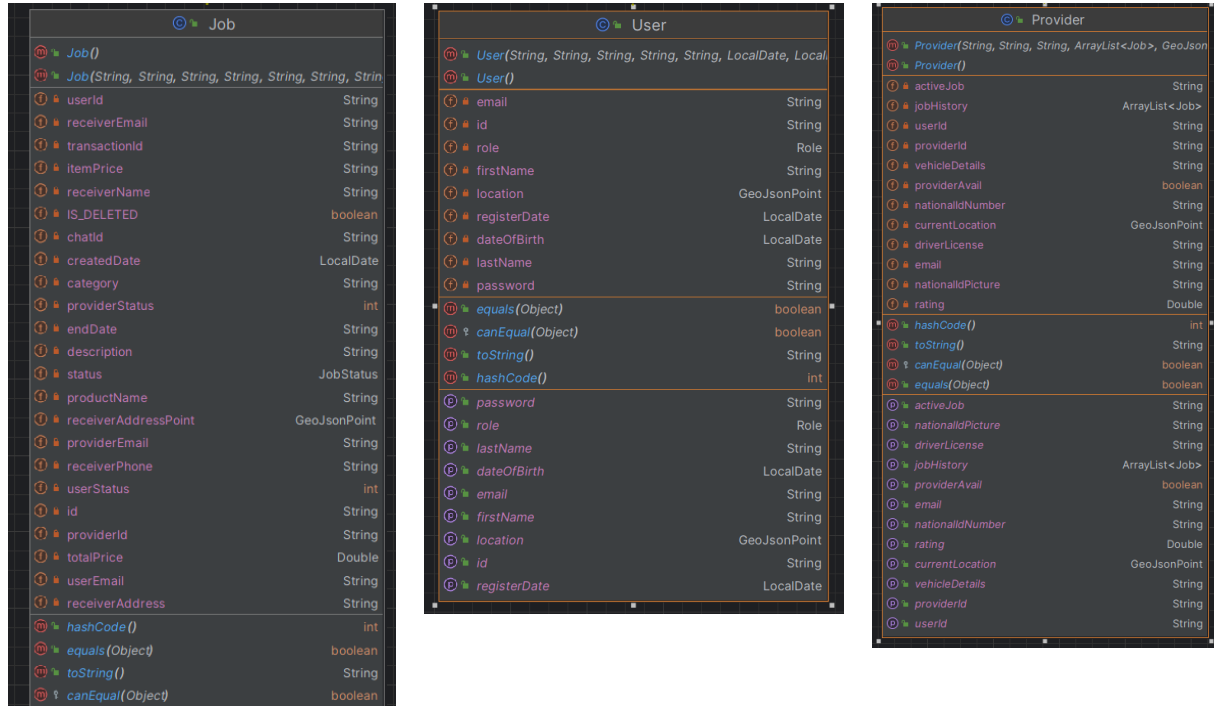
4. UML Sequence Diagrams:



This UML sequence diagram illustrates interactions between user management and job processing within an application. User-related functionalities are handled by the UserController, allowing creation, retrieval by ID, updating, and deletion of user records, while job-related actions follow a parallel structure with the JobController. Each controller communicates with its service layer, which in turn interacts with the repository layer to perform database operations. This diagram effectively maps out the flow of data and control across the system for key operations.

5. IntelliJ UML:

The following diagrams were generated by IntelliJ and show the main classes of the project: the Job, User, and Provider classes. Here, we can see that there are several different fields as well as properties objects of these classes can have. Due to the nature and structure of our web application, IntelliJ is not able to produce a complete diagram showing the interactions between the various components of our MVC model.



Miscellaneous Information

Note: When running “./gradlew run”, it may stop at 75% executing. This is normal when running long-lived tasks like web servers; it typically stops at 75% executing even though the server is running completely fine.