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.

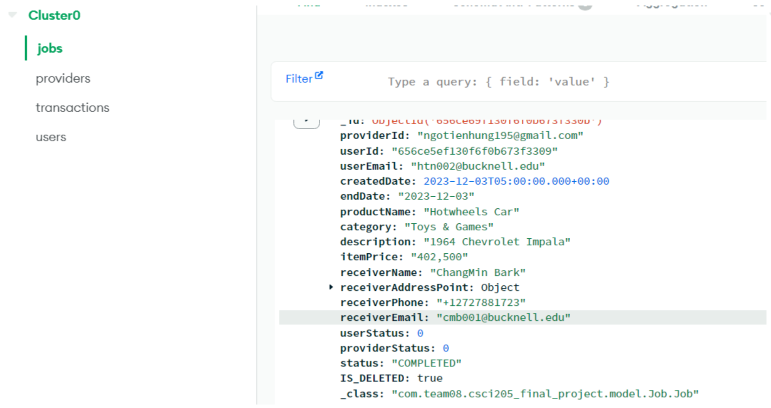
User Stories

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.

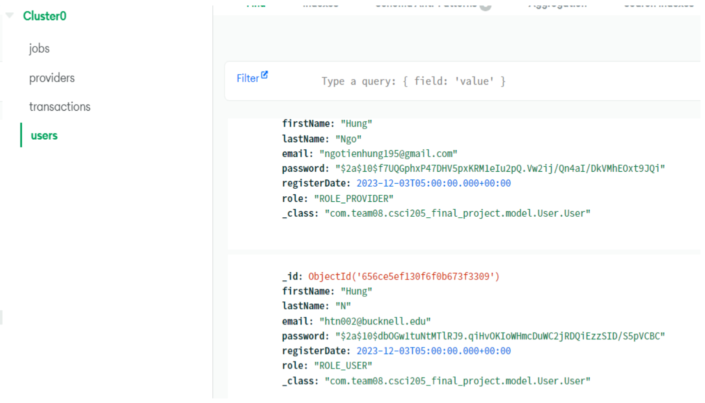
Eddie Hall & Krit Lohia - Similar to Melissa Miller and Camille Dubois' case, we were able to achieve their user stories.

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.



```
providerId: "ngotienhung195@gmail.com"
userId: "656ce5ef130f6f0b673f3309"
userEmail: "htn002@bucknell.edu"
createdDate: 2023-12-03T05:00:00.000+00:00
endDate: "2023-12-03"
productName: "Hotwheels Car"
category: "Toys & Games"
description: "1964 Chevrolet Impala"
itemPrice: "402,500"
receiverName: "ChangMin Bark"
receiverAddressPoint: Object
receiverPhone: "+12727881723"
receiverEmail: "cmb001@bucknell.edu"
userStatus: 0
providerStatus: 0
status: "COMPLETED"
IS_DELETED: true
_class: "com.team08.csci205_final_project.model.Job.Job"
```

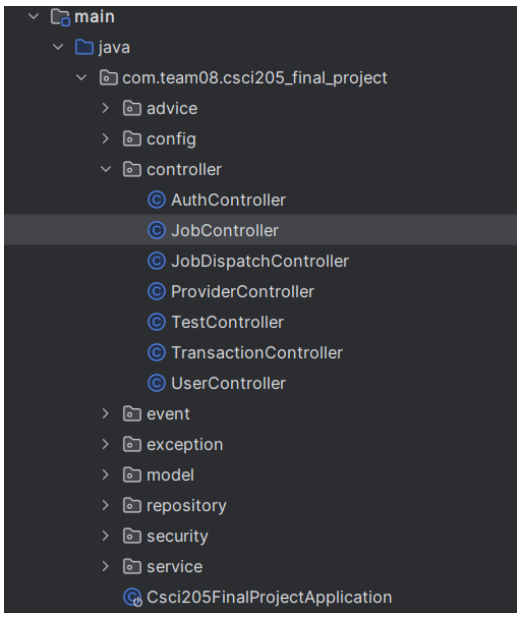


```
firstName: "Hung"
lastName: "Ngo"
email: "ngotienhung195@gmail.com"
password: "$2a$10$f7UQGphxP47DHV5pxKRMleIu2pQ.Vw2ij/Qn4aI/DkVMhEOxt9JQi"
registerDate: 2023-12-03T05:00:00.000+00:00
role: "ROLE_PROVIDER"
_class: "com.team08.csci205_final_project.model.User.User"

_id: ObjectId('656ce5ef130f6f0b673f3309')
firstName: "Hung"
lastName: "N"
email: "htn002@bucknell.edu"
password: "$2a$10$dbOGw1tuNtMTlRJ9.qiHvOKIoWHmcDuWC2jRDQiEzzSID/S5pVCBC"
registerDate: 2023-12-03T05:00:00.000+00:00
role: "ROLE_USER"
_class: "com.team08.csci205_final_project.model.User.User"
```

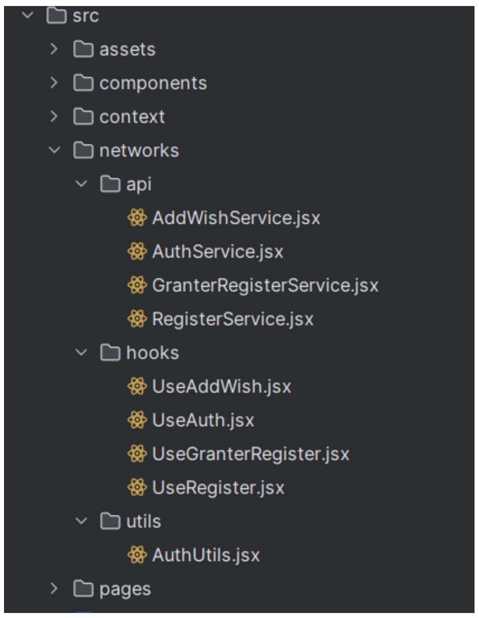
OOD

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.



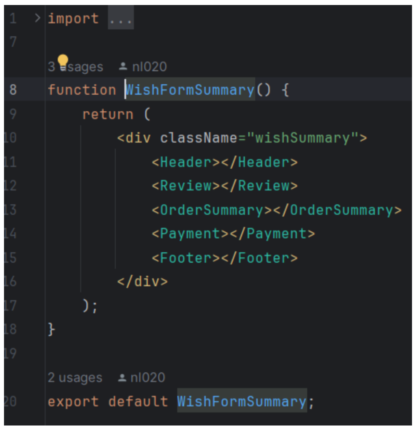
```
main
  java
    com.team08.csci205_final_project
      advice
      config
      controller
        AuthController
        JobController
        JobDispatchController
        ProviderController
        TestController
        TransactionController
        UserController
      event
      exception
      model
      repository
      security
      service
      Csci205FinalProjectApplication
```



```
src
  assets
  components
  context
  networks
    api
      AddWishService.jsx
      AuthService.jsx
      GranterRegisterService.jsx
      RegisterService.jsx
    hooks
      UseAddWish.jsx
      UseAuth.jsx
      UseGranterRegister.jsx
      UseRegister.jsx
    utils
      AuthUtils.jsx
  pages
```

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.



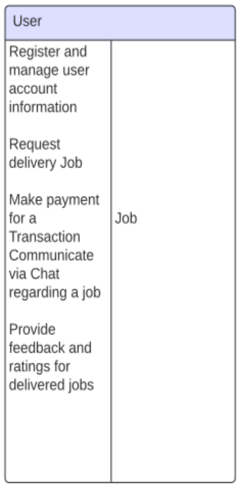
```
> import ...

3 usages  nl020
function WishFormSummary() {
    return (
        <div className="wishSummary">
            <Header></Header>
            <Review></Review>
            <OrderSummary></OrderSummary>
            <Payment></Payment>
            <Footer></Footer>
        </div>
    );
}

2 usages  nl020
export default WishFormSummary;
```

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.

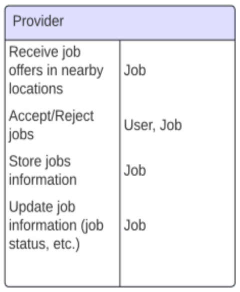
Back-end

1. **CRC Cards:**

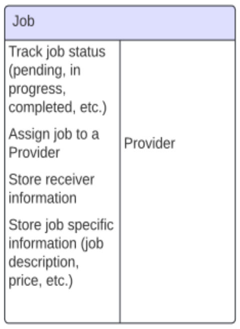
We've designed 3 CRC cards corresponding to 3 models of our system. The User class encapsulates the end-user functionalities of the system, handling personal information and job postings, while interacting with the Provider and Authentication system. The Provider class extends User capabilities to service 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	
Register and manage user account information	
Request delivery Job	
Make payment for a Transaction	Job
Communicate via Chat regarding a job	
Provide feedback and ratings for delivered jobs	



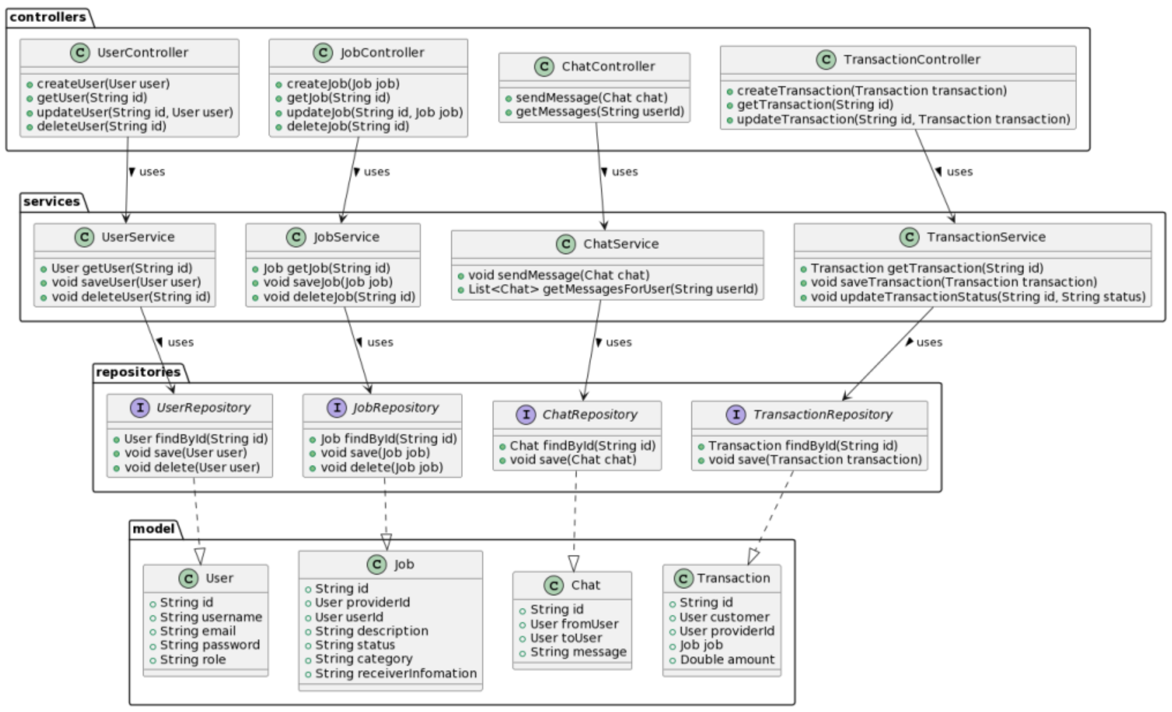
Provider	
Receive job offers in nearby locations	Job
Accept/Reject jobs	User, Job
Store jobs information	Job
Update job information (job status, etc.)	Job



Job	
Track job status (pending, in progress, completed, etc.)	
Assign job to a Provider	Provider
Store receiver information	
Store job specific information (job description, price, etc.)	

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.

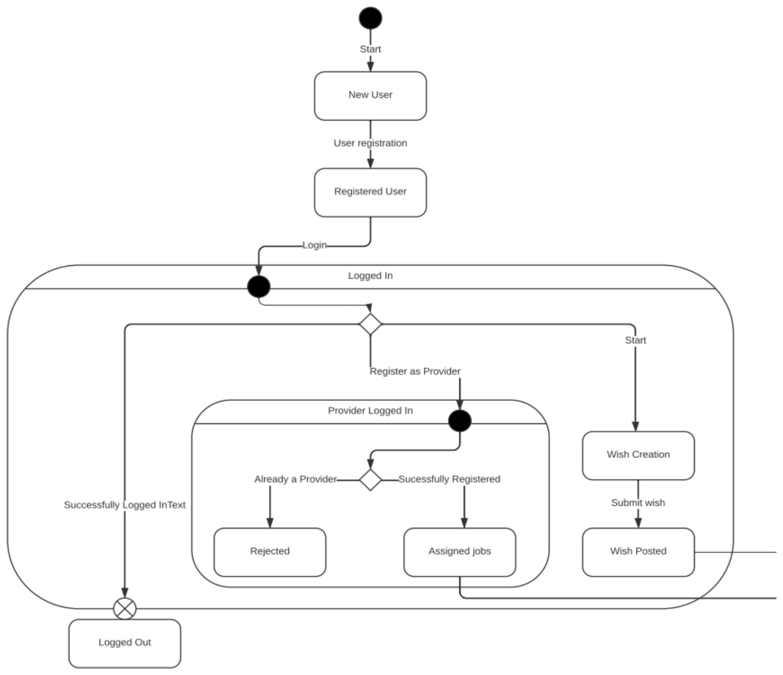
2. UML Class Diagram:



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.

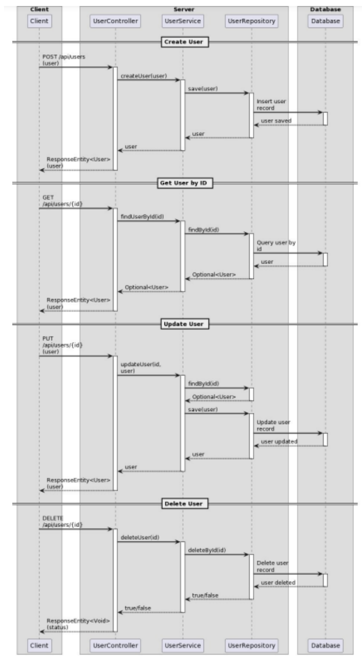
Note: Chats and Transactions are the features that we planned to implement, but we did not have enough time to work on.

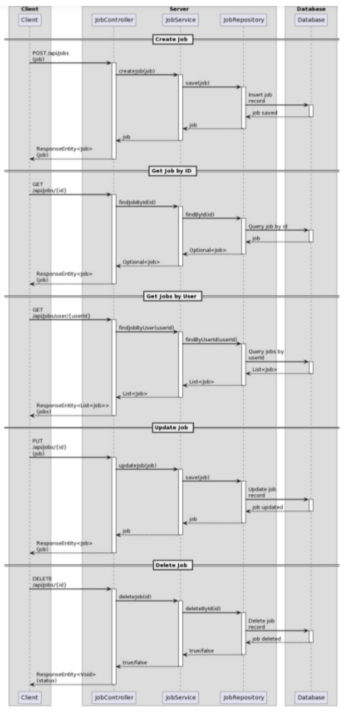
3. UML State Diagram:



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:

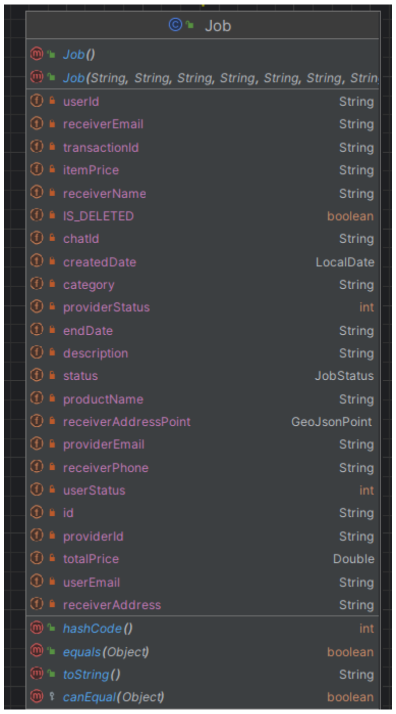




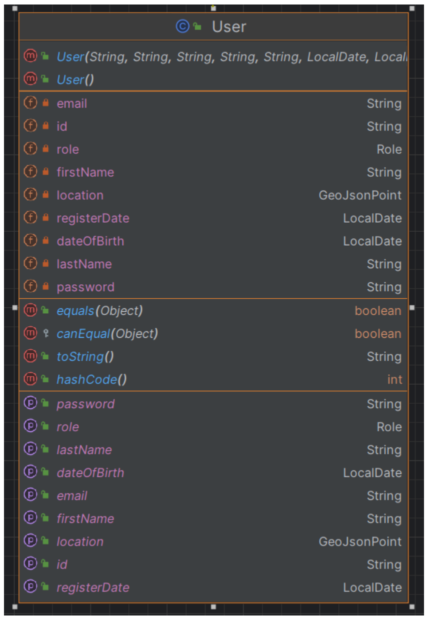
This UML sequence diagram illustrates interactions between user management and job processing within an application. User-related functionalities are handled by the UserController, allowing creation, retrieval by ID, updating, and deletion of user records, while job-related actions follow a parallel 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:**

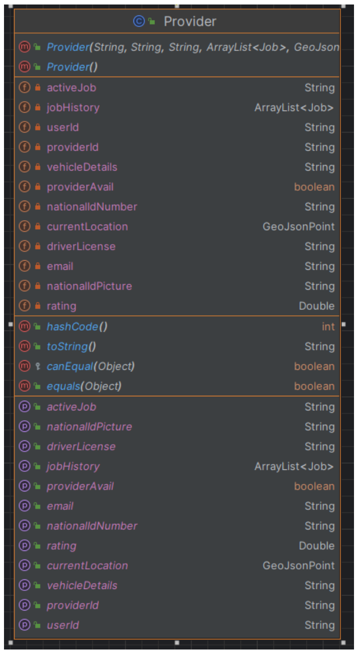
The following diagrams were generated by IntelliJ and show the main classes of the project: the Job, User, and Provider classes. Here, we can see that there are several different fields as well as properties objects of these classes can have. Due to the nature and structure of our web application, IntelliJ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



```
Job
Job()
Job(String, String, String, String, String, String, Strin
userId                      String
receiverEmail               String
transactionId               String
itemPrice                   String
receiverName                String
IS_DELETED                  boolean
chatId                      String
createdDate                 LocalDate
category                    String
providerStatus              int
endDate                     String
description                 String
status                      JobStatus
productName                 String
receiverAddressPoint        GeoJsonPoint
providerEmail               String
receiverPhone               String
userStatus                  int
id                          String
providerId                  String
totalPrice                  Double
userEmail                   String
receiverAddress             String
hashCode()                  int
equals(Object)              boolean
toString()                  String
canEqual(Object)            boolean
```



```
User
User(String, String, String, String, String, LocalDate, Local
User()
email                       String
id                          String
role                        Role
firstName                   String
location                    GeoJsonPoint
registerDate                LocalDate
dateOfBirth                 LocalDate
lastName                    String
password                    String
equals(Object)              boolean
canEqual(Object)            boolean
toString()                  String
hashCode()                  int
password                    String
role                        Role
lastName                    String
dateOfBirth                 LocalDate
email                       String
firstName                   String
location                    GeoJsonPoint
id                          String
registerDate                LocalDate
```



```
Provider
Provider(String, String, String, ArrayList<Job>, GeoJson
Provider()
activeJob                   String
jobHistory                  ArrayList<Job>
userId                      String
providerId                  String
vehicleDetails              String
providerAvail               boolean
nationalIdNumber            String
currentLocation             GeoJsonPoint
driverLicense               String
email                       String
nationalIdPicture           String
rating                      Double
hashCode()                  int
toString()                  String
canEqual(Object)            boolean
equals(Object)              boolean
activeJob                   String
nationalIdPicture           String
driverLicense               String
jobHistory                  ArrayList<Job>
providerAvail               boolean
email                       String
nationalIdNumber            String
rating                      Double
currentLocation             GeoJsonPoint
vehicleDetails              String
providerId                  String
userId                      String
```

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.

How to run:

1. Open up IntelliJ to the Backend Folder.
2. Open up WebStorm/VSCode to the Frontend Folder.
3. Run the application in IntelliJ to host the back-end server.
4. Type “npm i --force” in WebStorm/VSCode’s terminal to install dependencies.
5. Type “npm start” in WebStorm/VSCode’s terminal to start the front-end website.