Pawsup

Systems Design Document

CSCC01

Instructor: Ilir Dema

Group Members: Gary Xie, Jonathan Bai, SuTong Kong, Ivan Shao, Meng Zhao, Jesse Zhang, Tyler Reichert

Table of Contents

CRC Cards

Frontend

Class: App.js
Responsibilities: Controls the entire frontend and routing
Collaborators: HomePage.js SignInPage.js ServicePage.js ProductPage.js MediaPage.js

Class: SigninPage.js
Responsibilities: Sign-in page for users
Collaborators: HeaderMenu.js

Class: SignupPage.js
Responsibilities: Sign-up page for users
Collaborators: HeaderMenu.js

Class: FeaturedProducts.js
Responsibilities: Overview of the products offered
Collaborators: None

Class: FeaturedServices.js
Responsibilities: Overview of the services offered
Collaborators: None

Class: HomePage.js
Responsibilities: HomePage for users
Collaborators: HeaderMenu.js Footer.js HomePageSearch.js FeaturedServices.js FeaturedProducts.js

Class: HomePageSearchService.js
Responsibilities: Search Component for services based on Location, time, and pets.
Collaborators: None

Backend

Class Name: db	
Parent Class: None **Subclasses**: None	
Responsibilities: ● Data access object which allows the user to connect to and query the PostgreSQL database.	**Collaborators**: ● None

Class Name: Model	
Parent Class: None **Subclasses**: UserModel, ServiceModel	
Responsibilities: ● Abstract class with responsibility to manage the logic and data of the application. Contains basic methods applicable for all model extensions.	**Collaborators**: ● db

Class Name: UserModel	
Parent Class: Model **Subclasses**: None	
Responsibilities: ● Handles backend database logic related to Users.	**Collaborators**: ● db

Class Name: ServiceModel	
Parent Class: Model **Subclasses**: None	
Responsibilities: ● Handles backend database logic related to Services.	**Collaborators**: ● db

Class Name: AuthController	
Parent Class: None **Subclasses**: None	
Responsibilities: ● Handles the logic for HTTP requests to the /api/auth route. ● User login authentication. ● User session.	**Collaborators**: ● None

Class Name: ServiceController	
Parent Class: Model **Subclasses**: None	
Responsibilities: ● Handles the logic for HTTP requests to the /api/service route.	**Collaborators**: ● None

Class Name: UserController	
Parent Class: Model **Subclasses**: None	
Responsibilities: ● Handles the logic for HTTP requests to the /api/user route.	**Collaborators**: ● None

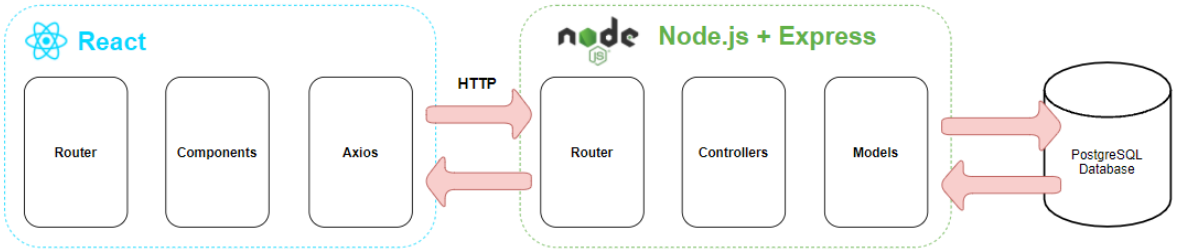
Class Name: Router	
Parent Class: None **Subclasses**: None	
Responsibilities: ● Handles the routes for our Express server and connects them to the correct controller.	**Collaborators**: ● AuthController ● ServiceController ● UserController

System Interaction Description

Our project must be run on one of the following operating systems: Windows 10, MacOS, or Linux. **Node.js** (Version 14.7+) as well as **npm** must be installed in order to compile the front-end code and run the application server. For the database, **PostgreSQL** (Version 13+) must be installed and running. To allow users to connect to the application, the server must be hosted on a machine with adequate storage and bandwidth, alongside a domain name (website URL) registered in a DNS for users to access via a browser.

System Architecture

We are using React and Bootstrap for our front-end, and it connects to the back-end server using Axios to send HTTP requests. In the backend, the router handles Requests and routes them to the correct Controller endpoint, which will use Models to communicate with the database (PostgreSQL) using a connection pool. Here is a diagram of our system architecture:



System Decomposition

The user will access the React frontend client for our application by connecting to the correct URL via the Internet. The client will connect with our Node.js + Express server by using Axios to send HTTP requests to our REST API, such as for creating an account, logging in/out, searching for services or products, or making a purchase. When the backend server receives a request, it will process it and query the PostgreSQL database to update the stored data and send back relevant information. We are using a three-tier architecture to prevent users from directly accessing the database, for security purposes.

To handle errors and edge cases (such as empty user input for a required field), the frontend will prevent the user from sending an invalid request and show a descriptive error message to the user explaining what caused the error. In case a request with invalid data is sent to the server, the backend controller will validate that the data is correct before processing it to the database, and will reject the request if it is invalid.