



"Ss. Cyril and Methodius" University in Skopje
**FACULTY OF COMPUTER
SCIENCE AND ENGINEERING**

Documentation for the subject

Advanced Web Design

Job Finding Application

Mentors:

Prof. Dr. Boban Joksimoski,

Mila Dodevska

Made by:

Angela Ivanova, 211104,

Andrej Stojov, 213200,

Marija Dimitrieska, 211117,

Filip Gacov, 213174

Skopje, September 2024

Content

1. Introduction.....	6
Key Features	6
2. System Architecture.....	6
2.1 Frontend Architecture (React).....	6
2.2 Backend Architecture (Spring Boot).....	6
2.3 Database (PostgreSQL).....	7
2.4 Data Flow and Communication	7
3. Key Functionalities	7
3.1 Job Finding Page (joboffers).....	7
3.2 User Authentication (Login and Registration Pages).....	7
3.4 Web Scraping Module.....	8
3.5 Chatbot.....	8
4. Backend (Spring Java)	8
4.1 API Endpoints	8
1. Authentication Endpoints	8
2. Job Listings Endpoints	8
3. Company Endpoints.....	8
4.2 Security Configuration.....	9
Overview	9
Security Configuration Code	9
Key Components Explained.....	13
4.3 Web Scraper	13
Dependencies	13
Key Annotations.....	14
Constructor.....	14
Methods.....	14
Configuration	15
4.4 Chat Bot	16
Dependencies	16
Key Annotations.....	16
Constructor.....	16

API Endpoint	16
4.5 Job offer feature	17
1. JobOfferController	17
2. JobOfferService	18
3. JobOfferRepository	18
4. JobOffer Entity	19
4.6 Company feature	19
1. CompanyController	19
2. CompanyService	20
3. CompanyRepository	20
4. Company Entity	20
4.7 Application feature	21
1. ApplicationController	21
2. ApplicationService	22
3. ApplicationServiceImpl	23
4. ApplicationRepository	23
5. Application Entity	23
5. Frontend (React)	24
5.1 Axios Configuration	24
5.2 AppService	24
Job Offers	24
Applications	26
Companies	27
5.3 App Component	29
Imports	29
Component Structure	29
State	29
Lifecycle Method	29
Data Fetching and CRUD Operations	30
Route Definitions	30
Error Handling	31
AppService API Calls	31
5.4 App.js component	31
Imports	31
Component Structure	31

State.....	31
Lifecycle Method	32
Data Fetching and CRUD Operations.....	32
Route Definitions	32
Error Handling	33
AppService API Calls	33
5.5 authService.....	33
Functions:.....	33
API Endpoints:.....	34
LocalStorage Keys:	35
5.6 Header Component	35
5.7 Home Component	35
Key Features:	35
getJobOfferPage() Function:	36
5.8 Details Component.....	36
Key Features:	36
API Integration:	37
5.9 Job Offers Management Components	37
JobOffers (Main Job Offers Component):	37
JobOfferAdd:	38
JobOfferEdit:.....	38
JobOfferTerm:.....	39
5.10 Company Management Components	40
1. Companies Component	40
2. CompanyTerm Component.....	40
3. CompanyAdd Component.....	41
4. CompanyEdit Component.....	41
5.11 ChatBot Component.....	42
Description:	42
Key Features:	42
States:.....	42
Methods:	42
5.12 ApplyForm Component	43
Description:	43
Key Features:	43

States:.....	43
Methods:	44
6. Conclusion	44

1. Introduction

This document provides an overview of a job listing website designed to streamline job searching and recruitment processes. The website allows job seekers to explore job opportunities, and companies can post their vacancies. The platform is designed with a user-friendly interface and provides easy access to job listings through a responsive, card-based layout.

Key Features

- **Job Finding Page:** Displays job offers scraped from various sources.
- **User Authentication:** Allows users to log in, register, and manage their profiles.
- **Admin Dashboard:** Admins can add or manage companies and job listings.
- **Chatbot:** Provides user assistance and guides visitors through the site.
- **Frontend:** Built using React for a dynamic and interactive user experience.
- **Backend:** Developed with Spring Java, ensuring robust data management and API handling.

2. System Architecture

The website is built using a two-tier architecture that consists of a frontend developed with React and a backend implemented with Spring Boot, along with a PostgreSQL database. This architecture separates the user interface from the business logic, providing a clear separation of concerns and enhancing the maintainability, scalability, and performance of the application.

2.1 Frontend Architecture (React)

- **Layered Architecture:** The React frontend follows a layered architecture that promotes modularity and reusability of components. Key layers include:
 - **Presentation Layer:** Responsible for rendering UI components, handling user input, and displaying data fetched from the backend. This includes the main pages (Home, Job Listings, Job Detail, etc.), forms (e.g., job application), and reusable components (e.g., buttons, modals).
 - **API Layer:** Handles communication with the backend through RESTful APIs, managing requests and responses.
 - **Routing:** React Router is used for client-side navigation, enabling users to move between pages without reloading the entire application.

2.2 Backend Architecture (Spring Boot)

- The Spring Boot backend is structured using a layered architecture comprising four main layers:
 - **Domain Layer:** Contains the core business entities and domain logic, representing the main concepts of the application (e.g., JobOffers, User, Application).

- **Repository Layer:** Manages data persistence and retrieval from the PostgreSQL database. This layer interacts directly with the database using JPA (Java Persistence API), ensuring data is stored, updated, and queried efficiently.
- **Service Layer:** Contains the business logic and processes application requests. It acts as an intermediary between the web layer and repository layer, implementing the core functionalities of the application (e.g., adding jobs, applying to jobs).
- **Web Layer:** Exposes RESTful endpoints to handle incoming HTTP requests from the frontend. Controllers in this layer map requests to appropriate service methods, validate inputs, and manage responses.

2.3 Database (PostgreSQL)

- **PostgreSQL:** The backend uses PostgreSQL as the relational database management system to store and manage data. The database schema is designed to support the job-finding platform, including tables for users, job listings, applications, and companies.
- **Data Integrity and Performance:** The database schema is optimized for performance, with indexing strategies to ensure efficient querying. Transactions are handled to maintain data integrity, especially during critical operations like job applications.

2.4 Data Flow and Communication

- The React frontend communicates with the Spring Boot backend through RESTful APIs. The frontend makes HTTP requests to the backend, which processes the requests, interacts with the database if needed, and sends back the appropriate response.
- The backend ensures secure data handling, including input validation, authentication, and authorization to protect user data and enforce access control.

3. Key Functionalities

3.1 Job Finding Page (joboffers)

- Displays a list of available job offers as cards, showing key details such as job title, company name, location, and a brief description.
- **Search Functionality:** Users can filter job offers based on keywords, location, and job type.
- **Job Details:** Clicking on a job card opens a detailed view with more information and an option to apply.

3.2 User Authentication (Login and Registration Pages)

- **Login Page:** Allows users (admins) to log in using their credentials.
- **Registration Page:** Admins can create an account by providing the necessary details.
- **Role-Based Access Control:** Different functionalities are available based on the user's role (e.g., admins can manage listings, companies can post jobs).

3.4 Web Scraping Module

- **Data Extraction:** The module fetches job listings from external websites, parses the data, and stores it in the database.
- **Automation:** Scraping jobs are scheduled to run periodically to ensure the listings are up-to-date.
- **Error Handling:** The system manages inconsistencies and errors in scraped data to maintain a clean dataset.

3.5 Chatbot

- **OllamaChatModel Integration:** The chatbot is built using OllamaChatModel, providing advanced conversational capabilities to assist users.
- **User Assistance:** The chatbot guides users through the website, helps them find job listings, and assists with registration and login processes.
- **Natural Language Processing:** Leverages NLP to understand user queries and deliver contextually relevant responses.
- **Integration:** The chatbot seamlessly interacts with the backend, providing real-time assistance and enhancing the overall user experience.

4. Backend (Spring Java)

4.1 API Endpoints

1. Authentication Endpoints

- **POST /api/auth/login:** Logs a user into the system.
- **POST /api/auth/register:** Registers a new user.

2. Job Listings Endpoints

- **GET /api/jobs:** Retrieves a list of job listings.
- **POST /api/jobs:** Adds a new job listing (admin or company role required).
- **PUT /api/jobs/{id}:** Updates a specific job listing.
- **DELETE /api/jobs/{id}:** Deletes a specific job listing.

3. Company Endpoints

- **GET /api/companies:** Retrieves a list of companies.
- **POST /api/companies:** Adds a new company (admin or company role required).
- **PUT /api/companies/{id}:** Updates company details.
- **DELETE /api/companies/{id}:** Deletes a company.

4.2 Security Configuration

The security configuration for the job listing website uses Spring Security integrated with JWT for authentication and authorization. Below is an overview of the key components and how they contribute to the overall security of the application.

Overview

- **JWT Authentication:** Manages user sessions in a stateless manner, providing tokens that are used to authenticate requests.
- **BCrypt Password Encoder:** Ensures passwords are securely hashed before storing them in the database.
- **CORS Configuration:** Configures Cross-Origin Resource Sharing to allow communication between the frontend and backend.
- **Logout Handler:** Manages user logouts by clearing authentication contexts and handling custom logout logic.

Security Configuration Code

Here is the security configuration implemented in the application:

```
package com.example.backend.config;

import com.example.backend.config.filter.JwtAuthenticationFilter;
import com.example.backend.service.impl.JwtService;
import com.example.backend.service.impl.UserServiceImpl;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.http.HttpStatus;
import org.springframework.security.authentication.AuthenticationManager;
import
org.springframework.security.config.annotation.authentication.configuratio
n.AuthenticationConfiguration;

import
org.springframework.security.config.annotation.web.builders.HttpSecurity;
import
org.springframework.security.config.annotation.web.configuration.EnableWeb
Security;
```

```

import
org.springframework.security.config.annotation.web.configurers.AbstractHttpConfigurer;

import org.springframework.security.config.http.SessionCreationPolicy;
import org.springframework.security.core.context.SecurityContextHolder;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
import org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.security.web.SecurityFilterChain;

import
org.springframework.security.web.authentication.HttpStatusEntryPoint;

import
org.springframework.security.web.authentication.UsernamePasswordAuthenticationFilter;

import org.springframework.web.cors.CorsConfiguration;
import org.springframework.web.cors.CorsConfigurationSource;
import org.springframework.web.cors.UrlBasedCorsConfigurationSource;

import java.util.List;

@Configuration
@EnableWebSecurity
public class WebSecurityConfig {

    private JwtService jwtService;
    private CustomLogoutHandler logoutHandler;
    private UserServiceImpl userServiceImpl;

    @Autowired
    public void setJwtService(JwtService jwtService) {
        this.jwtService = jwtService;
    }
}

```

```

@Autowired
public void setLogoutHandler(CustomLogoutHandler logoutHandler) {
    this.logoutHandler = logoutHandler;
}

@Autowired
public void setUserServiceImpl(UserServiceImpl userServiceImpl) {
    this.userServiceImpl = userServiceImpl;
}

@Bean
public CorsConfigurationSource corsConfigurationSource() {
    CorsConfiguration configuration = new CorsConfiguration();
    configuration.setAllowedOrigins(List.of("http://localhost:3000"));
    configuration.setAllowedMethods(List.of("GET", "POST", "PUT",
"DELETE"));
    configuration.setAllowedHeaders(List.of("*"));
    configuration.setAllowCredentials(true);

    UrlBasedCorsConfigurationSource source = new
UrlBasedCorsConfigurationSource();
    source.registerCorsConfiguration("/**", configuration);
    return source;
}

@Bean
public SecurityFilterChain securityFilterChain(HttpSecurity http)
throws Exception {
    JwtAuthenticationFilter jwtAuthenticationFilter = new
JwtAuthenticationFilter(jwtService, userServiceImpl);

```

```

        return http
            .csrf(AbstractHttpConfigurer::disable)
            .cors(cors ->
cors.configurationSource(corsConfigurationSource()))
            .authorizeHttpRequests(
                req -> req
                    .anyRequest().permitAll()
            )
            .userDetailsService(userServiceImpl)
            .sessionManagement(session -> session

.sessionCreationPolicy(SessionCreationPolicy.STATELESS))
            .addFilterBefore(jwtAuthenticationFilter,
UsernamePasswordAuthenticationFilter.class)
            .exceptionHandling(
                e -> e.accessDeniedHandler(
                    (request, response,
accessDeniedException) -> response.setStatus(403)
                )
            .authenticationEntryPoint(new
HttpStatusEntryPoint(HttpStatus.UNAUTHORIZED)))
            .logout(l -> l
                .logoutUrl("/logout")
                .addLogoutHandler(logoutHandler)
                .logoutSuccessHandler((request, response,
authentication) -> SecurityContextHolder.clearContext())
            ))
            .build();
    }

```

@Bean

```

    public PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }

    @Bean
    public AuthenticationManager
authenticationManager(AuthenticationConfiguration configuration) throws
Exception {
        return configuration.getAuthenticationManager();
    }
}

```

Key Components Explained

1. **CORS Configuration:** The corsConfigurationSource bean configures CORS to allow requests from the frontend (<http://localhost:3000>) and supports all common HTTP methods and headers.
2. **Security Filter Chain:**
 - Disables CSRF protection since JWT handles session security.
 - Configures CORS to work with the provided CORS source.
 - Uses JwtAuthenticationFilter before the UsernamePasswordAuthenticationFilter to validate JWT tokens on incoming requests.
 - Handles session management in a stateless mode using JWT.
 - Provides custom handling for authentication failures and access denials.
3. **Authentication Management:** Integrates JwtService for managing token creation and validation and UserServiceImpl for user-related operations.
4. **Logout Configuration:** Sets up a custom logout handler to manage session termination and security context clearance upon logout.
5. **Password Encoding:** Uses BCryptPasswordEncoder for secure password storage.

4.3 Web Scraper

The Web Scraper class is a scheduled component in a Java Spring Boot application designed to scrape job offers from the Kariera.mk website. It gathers detailed job and company information and stores them in the database using CompanyService and JobOfferService.

Dependencies

- **Spring Framework:** For dependency injection, scheduling, and component management.

- **Jsoup:** For HTML parsing and web scraping.
- **CompanyService:** Service class for handling company-related database operations.
- **JobOfferService:** Service class for handling job offer-related database operations.

Key Annotations

- **@Component:** Marks the class as a Spring component, allowing it to be detected and managed by Spring's component scanning.
- **@Scheduled:** Used to schedule the execution of the `scrapeForOffers` method at specified times using a cron expression.

Constructor

```
public WebScraper(CompanyService companyService, JobOfferService jobOfferService)
```

- **Parameters:**
 - `CompanyService companyService:` Service used to manage company data within the application.
 - `JobOfferService jobOfferService:` Service used to manage job offer data within the application.

Methods

```
@Scheduled(cron = "0 0 12 * * ?")
```

```
public void scrapeForOffers()
```

- *Description:* This method is triggered automatically every day at noon (12:00 PM) based on the cron schedule. It scrapes job offers from the Kariera.mk website, extracts relevant job and company details, and stores them in the application's database if they are not already present.
- *Workflow:*
 1. Connects to the job offers page on Kariera.mk for a specific occupation category using Jsoup.
 2. Retrieves the list of job offers from the page and processes each one:
 - Skips deactivated jobs based on class attributes.
 - Extracts the URL for detailed job information.
 - Fetches the detailed job description and related company details by navigating to the respective pages.
 3. Extracts company details, including:
 - Company name, description, logo URL, address, city, and website.
 - Checks if the company already exists in the database. If not, it creates a new company record.

4. Extracts job offer details, including:
 - Position title, job description, start date, end date, and location.
 - Checks if the job offer already exists in the database under the same company. If not, it creates a new job offer record.
- *Error Handling*: Catches and logs IOException exceptions, which may occur due to network issues or HTML parsing errors.
- *Data Extracted*:
 - **Job Offer**:
 - **Position**: The title of the job.
 - **Description**: A detailed description of the job role.
 - **Starting Date**: The current date when the job is scraped.
 - **Ending Date**: The date when the job offer expires.
 - **Location**: The location of the job.
 - **Company**:
 - **Name**: The name of the company offering the job.
 - **Description**: Additional details about the company.
 - **City**: The city where the company is located.
 - **Address**: The address of the company.
 - **Logo URL**: A link to the company logo image.
 - **Website**: The company's official website link.

Configuration

- **Cron Expression**: 0 0 12 * * ? — This cron expression configures the scrapeForOffers method to run every day at 12:00 PM.

Example

Execution Flow:

1. The scheduled job runs at 12:00 PM daily.
2. It accesses the Kariera.mk search page for jobs within a specified occupation category.
3. Iterates over each job offer found:
 - Extracts and processes the job details.
 - Extracts and processes the company details associated with each job.
 - Checks if the extracted data already exists in the database to avoid duplicates.

4. Saves new job offers and companies to the database if they do not exist.

4.4 Chat Bot

The ChatBotController class provides a REST API endpoint that enables communication with an AI-powered chatbot using the OllamaChatModel. The chatbot is designed to respond to user queries specifically related to IT or job offers, and it will refuse to answer unrelated questions.

Dependencies

- **Spring Framework:** For building RESTful web services and managing HTTP requests and responses.
- **OllamaChatModel:** A chat model interface from Spring AI for processing chatbot interactions.
- **ChatDTO:** A Data Transfer Object (DTO) that encapsulates the chat request data from the client.

Key Annotations

- **@RestController:** Marks the class as a Spring MVC controller where each method returns a domain object instead of a view. It's a shorthand for **@Controller** and **@ResponseBody**.
- **@CrossOrigin(origins = {"http://localhost:3000", "http://localhost:3001"}):** Enables Cross-Origin Resource Sharing (CORS) for specified frontend origins, allowing the frontend applications running on localhost:3000 and localhost:3001 to access the API.
- **@RequestMapping("/api/ai"):** Maps HTTP requests to handler methods of the controller. In this case, it maps requests with the base path /api/ai.

Constructor

```
public ChatBotController(OllamaChatModel chatModel)
```

- **Parameters:**
 - OllamaChatModel chatModel: The AI chat model that handles the interaction logic and processes user queries.

API Endpoint

POST /api/ai/chat

```
@PostMapping("/chat")
```

```
public String chat(@RequestBody ChatDTO chatRequest)
```

- **Description:** This endpoint accepts a user query via a POST request, processes the query using the OllamaChatModel, and returns the chatbot's response. The chatbot is configured to only handle questions related to IT or the job offer application; it will decline to answer other types of questions.
- **Parameters:**
 - **@RequestBody ChatDTO chatRequest:** The request body containing the user's query. The ChatDTO object is expected to have a query property, which contains the user's input message.

- **Response:**

- Returns a String response from the chatbot. If the question is not related to IT or job offers, the chatbot will respond with a message indicating it cannot help with that.

- **Request Example:**

```
{
  "query": "What are the latest IT job offers available?"
}
```

- **Response Example:**

```
{
  "response": "Here are some of the latest IT job offers: ..."
}
```

- **Special Handling:**

- The chatbot response is influenced by the additional prompt: "If the question before this was not related to IT or about the Job offer application you are on do not answer say you can not help with that." This ensures the chatbot stays focused on relevant topics.

4.5 Job offer feature

The Job Offer feature in this application allows users to manage job offers including creating, updating, retrieving, and deleting job offers. This feature consists of a REST API controller, a service layer for business logic, and a repository layer for database interactions.

1. JobOfferController

The JobOfferController is a Spring REST controller that handles HTTP requests related to job offers.

- **Endpoints:**

- GET /api/joboffers: Fetches all job offers. Supports optional filtering by position name.
 - **Query Parameters:**
 - name (optional): Filter job offers by position name.
 - **Returns:** A list of all job offers or filtered job offers based on the provided name.
- GET /api/joboffers/{id}: Fetches a job offer by its ID.
 - **Path Variables:**
 - id: The ID of the job offer.
 - **Returns:** The job offer with the specified ID or a 404 status if not found.
- POST /api/joboffers/add: Creates a new job offer.
 - **Request Body:** A JobOfferDTO object containing job offer details.

- **Returns:** Status 200 if the job offer is created successfully.
- POST /api/joboffers/edit/{id}: Updates an existing job offer.
 - **Path Variables:**
 - id: The ID of the job offer to update.
 - **Request Body:** A JobOfferDTO object containing updated job offer details.
 - **Returns:** Status 200 if the job offer is updated successfully.
- DELETE /api/joboffers/delete/{id}: Deletes a job offer by its ID.
 - **Path Variables:**
 - id: The ID of the job offer to delete.
 - **Returns:** Status 200 if the job offer is deleted successfully, or 400 if deletion fails.

2. JobOfferService

The JobOfferService interface defines the business logic for managing job offers, and its implementation is provided by JobOfferServiceImpl.

- **Methods:**
 - Optional<JobOffer> findById(long id): Finds a job offer by its ID.
 - List<JobOffer> findAll(): Retrieves all job offers.
 - JobOffer create(String position, String details, LocalDate startDate, LocalDate endDate, String location, Long companyId): Creates a new job offer.
 - JobOffer update(long id, String position, String details, LocalDate startDate, LocalDate endDate, String location, Long companyId): Updates an existing job offer.
 - void deleteById(long id): Deletes a job offer by its ID.
 - List<Application> getApplicationsForJobOffer(Long jobId): Retrieves applications associated with a specific job offer.
 - JobOffer findByNameAndCompany(String position, String companyName): Finds a job offer by position and company name.
 - JobOffer saveJobOffer(JobOffer jobOffer): Saves a job offer entity.
 - List<JobOffer> filter(String name): Filters job offers by position name.

3. JobOfferRepository

The JobOfferRepository is a JPA repository interface that handles database interactions for job offers.

- **Methods:**
 - JobOffer findByPositionEqualsAndCompanyEquals(String position, Company company): Finds a job offer by position and company.

- `List<JobOffer> findByPositionContaining(String position)`: Finds job offers where the position contains a specified string.

4. JobOffer Entity

The JobOffer entity represents the job offer model and is mapped to a database table.

- **Fields:**

- `Id (long)`: The primary key of the job offer.
- `position (String)`: The job position title.
- `details (String)`: Detailed description of the job.
- `startDate (LocalDate)`: The date when the job offer starts.
- `endDate (LocalDate)`: The date when the job offer ends.
- `location (String)`: The location of the job.
- `company (Company)`: The company offering the job.
- `applicationList (List<Application>)`: A list of applications associated with the job offer.

- **Constructors:**

- `JobOffer(String position, String details, LocalDate startDate, LocalDate endDate, String location, Company company)`: Constructs a new job offer with specified details and associates it with a company.
- `JobOffer()`: Default constructor.

4.6 Company feature

The Company feature in this application provides functionalities for managing companies, including creating, updating, retrieving, and deleting company records. This feature comprises a REST API controller, a service layer for business logic, and a repository layer for database interactions.

1. CompanyController

The CompanyController is a Spring REST controller that handles HTTP requests related to companies.

- **Endpoints:**

- `GET /api/company/{id}`: Fetches a company by its ID.
 - **Path Variables:**
 - `id`: The ID of the company.
 - **Returns:** The company with the specified ID or a 404 status if not found.
- `GET /api/company`: Fetches all companies.

- **Returns:** A list of all companies.
- POST /api/company/add: Creates a new company.
 - **Request Body:** A CompanyDTO object containing company details.
 - **Returns:** Status 200 if the company is created successfully.
- POST /api/company/edit/{id}: Updates an existing company.
 - **Path Variables:**
 - id: The ID of the company to update.
 - **Request Body:** A CompanyDTO object containing updated company details.
 - **Returns:** Status 200 if the company is updated successfully.
- DELETE /api/company/delete/{id}: Deletes a company by its ID.
 - **Path Variables:**
 - id: The ID of the company to delete.
 - **Returns:** Status 200 if the company is deleted successfully, or 400 if deletion fails.

2. CompanyService

The CompanyService interface defines the business logic for managing companies, with implementation provided by CompanyServiceImpl.

- **Methods:**
 - Optional<Company> findById(long id): Finds a company by its ID.
 - List<Company> findAll(): Retrieves all companies.
 - Company create(String name, String description, String location, String address, String logo, String webSite): Creates a new company with the given details.
 - Company update(long id, String name, String description, String location, String address, String logo, String webSite): Updates an existing company.
 - void deleteById(long id): Deletes a company by its ID.
 - Company findByName(String name): Finds a company by its name.

3. CompanyRepository

The CompanyRepository is a JPA repository interface that handles database interactions for companies.

- **Methods:**
 - Company findByNameEquals(String name): Finds a company by its exact name.

4. Company Entity

The Company entity represents the company model and is mapped to a database table.

- **Fields:**
 - Id (long): The primary key of the company.
 - name (String): The name of the company.
 - description (String): A detailed description of the company.
 - location (String): The general location of the company.
 - address (String): The specific address of the company.
 - logo (String): A URL or path to the company's logo image.
 - webSite (String): The company's website URL.
- **Constructors:**
 - Company(String name, String description, String location, String address, String logo, String webSite): Constructs a new company with specified details.
 - Company(): Default constructor.
- **Annotations:**
 - @Entity: Specifies that the class is an entity and is mapped to a database table.
 - @Data: A Lombok annotation that generates getters, setters, and other utility methods.
 - @AllArgsConstructor: Generates a constructor with all fields as parameters.
 - @Id: Specifies the primary key of the entity.
 - @GeneratedValue(strategy = GenerationType.IDENTITY): Configures auto-increment for the primary key.

4.7 Application feature

The Application feature handles the management of job applications within the system. This includes creating, updating, retrieving, deleting applications, and handling associated functionalities such as uploading and downloading CV files. The feature consists of a REST API controller, a service layer that implements the business logic, and a repository layer for database interactions.

1. ApplicationController

The ApplicationController is a Spring REST controller that manages HTTP requests for job applications.

- **Endpoints:**
 - POST /api/applications/apply: Submits a job application.
 - **Request Parameters:**
 - jobId (long): ID of the job offer the application is for.
 - name (String): Applicant's first name.
 - lastName (String): Applicant's last name.

- email (String): Applicant's email address.
 - phoneNumber (String): Applicant's phone number.
 - cvFile (MultipartFile): Applicant's CV as a file upload (PDF).
- **Returns:** A success message if the application is submitted successfully, or error messages if the job offer is not found or if there's an issue processing the file.
- GET /api/applications/download/{applicationId}: Downloads the CV file of an application.
 - **Path Variables:**
 - applicationId (long): The ID of the application.
 - **Returns:** The CV file as a PDF if found, otherwise a 404 status.
- GET /api/applications: Fetches all job applications.
 - **Returns:** A list of all applications.
- GET /api/applications/{id}: Fetches a specific application by ID.
 - **Path Variables:**
 - id (long): The ID of the application.
 - **Returns:** The application details or a 404 status if not found.
- POST /api/applications/edit/{id}: Updates an existing application.
 - **Path Variables:**
 - id (long): The ID of the application to update.
 - **Request Body:** An ApplicationDTO object containing updated application details.
- POST /api/applications/add: Adds a new application.
 - **Request Body:** An ApplicationDTO object with application details.
- DELETE /api/applications/delete/{id}: Deletes an application by ID.
 - **Path Variables:**
 - id (long): The ID of the application to delete.
 - **Returns:** A success message if deleted successfully, or a 400 status if deletion fails.

2. ApplicationService

The ApplicationService interface defines the business logic for managing job applications, implemented by ApplicationServiceImpl.

- **Methods:**

- `List<Application> findAll()`: Retrieves all job applications.
- `Optional<Application> findById(long id)`: Finds a specific application by its ID.
- `Application update(long id, String name, String lastName, String email, String phoneNumber)`: Updates an existing application.
- `Application create(String name, String lastName, String email, String phoneNumber, long jobOfferId)`: Creates a new job application and associates it with a job offer.
- `void deleteById(long id)`: Deletes an application by its ID.
- `Application saveApplication(Application application)`: Saves an application entity to the database.

3. ApplicationServiceIMPL

The `ApplicationServiceIMPL` class implements the `ApplicationService` interface, providing the actual business logic.

- **Methods:**
 - **findAll**: Fetches all applications from the repository.
 - **findById**: Finds an application by its ID using the repository.
 - **update**: Updates an existing application based on the provided details. If the application ID is invalid, it throws an `InvalidApplicationIdException`.
 - **create**: Creates a new application, saves it to the database, and associates it with a specified job offer. Throws an `InvalidJobOfferIdException` if the job offer ID is invalid.
 - **deleteById**: Deletes an application by its ID.
 - **saveApplication**: Saves an application entity directly.

4. ApplicationRepository

The `ApplicationRepository` is a JPA repository interface that handles database operations for job applications.

- **Methods:**
 - Inherits standard CRUD operations from `JpaRepository` for managing `Application` entities.

5. Application Entity

The `Application` entity represents a job application and is mapped to a database table.

- **Fields:**
 - `id (long)`: The primary key of the application.
 - `name (String)`: Applicant's first name.
 - `lastName (String)`: Applicant's last name.

- email (String): Applicant's email address.
- phoneNumber (String): Applicant's phone number.
- cvFile (byte[]): The CV file stored as a byte array.
- **Constructors:**
 - Application(String name, String lastName, String email, String phoneNumber): Constructs a new application with the provided details.
 - Application(): Default constructor.
- **Annotations:**
 - @Entity: Specifies that the class is an entity mapped to a database table.
 - @Data: A Lombok annotation that generates getters, setters, and other utility methods.
 - @Id: Specifies the primary key of the entity.
 - @GeneratedValue(strategy = GenerationType.IDENTITY): Configures auto-increment for the primary key.
 - @Lob: Specifies that the cvFile field should be treated as a large object (binary data).

5. Frontend (React)

- **React:** The main framework used for building the user interface, offering reusable components and state management for a seamless user experience.
- **React Router:** Used for navigation between different pages such as job listings, login, registration, and company management.
- **Axios:** Handles API requests to communicate with the backend.
- **CSS/Styled Components:** Used for styling components to ensure a modern, responsive design.

5.1 Axios Configuration

In the provided code, Axios is configured with a base URL (`http://localhost:8080/api`) and headers (`'Access-Control-Allow-Origin': '*'`). This configuration is used for all the API calls made by the AppService.

5.2 AppService

AppService is an object that contains methods to handle API requests related to applications, companies, and job offers. It uses Axios to perform HTTP requests (GET, POST, PUT, DELETE) to the backend.

Job Offers

1. **fetchJobOffersFilter(name)**
 - **Description:** Fetches job offers filtered by the provided name.

- **Method:** GET
- **Endpoint:** /joboffers?name=\${name}
- **Returns:** A promise with the filtered job offers.

2. **fetchJobOffers()**

- **Description:** Fetches all job offers.
- **Method:** GET
- **Endpoint:** /joboffers
- **Returns:** A promise with all job offers.

3. **addJobOffer(position, details, startingDate, endingDate, location, company)**

- **Description:** Adds a new job offer.
- **Method:** POST
- **Endpoint:** /joboffers/add
- **Request Body:**

```
{
  "position": "string",
  "details": "string",
  "startingDate": "date",
  "endingDate": "date",
  "location": "string",
  "company": "string"
}
```

4. **updateJobOffer(id, position, details, startingDate, endingDate, location, company)**

- **Description:** Updates an existing job offer by ID.
- **Method:** POST
- **Endpoint:** /joboffers/edit/\${id}
- **Request Body:**

```
{
  "position": "string",
  "details": "string",
```

$$\}$$

5. deleteJobOffer(id)

- **Method:** DELETE

6. `getJobOffer(id)`

- **Method:** GET

Applications

1. fetchApplications()

- **Method:** GET

2. **addApplication(name, lastName, email, phoneNumber)**

- **Method:** POST

$$\{$$

```
"lastName": "string",
```

```
"email": "string",  
"phoneNumber": "string"  
}
```

3. **updateApplication(id, name, lastName, email, phoneNumber)**

- **Description:** Updates an existing application by ID.
- **Method:** PUT
- **Endpoint:** /applications/edit/{id}
- **Request Body:**

```
{  
"name": "string",  
"lastName": "string",  
"email": "string",  
"phoneNumber": "string"  
}
```

4. **deleteApplication(id)**

- **Description:** Deletes an application by ID.
- **Method:** DELETE
- **Endpoint:** /applications/delete/{id}

5. **getApplication(id)**

- **Description:** Fetches a specific application by ID.
- **Method:** GET
- **Endpoint:** /applications/{id}
- **Returns:** A promise with the application data.

Companies

1. **fetchCompanies()**

- **Description:** Fetches all companies.
- **Method:** GET
- **Endpoint:** /company
- **Returns:** A promise with all companies.

2. **addCompany(name, description, location, address, logo, webSite)**

- **Description:** Adds a new company.
- **Method:** POST
- **Endpoint:** /company/add
- **Request Body:**

```
{
  "name": "string",
  "description": "string",
  "location": "string",
  "address": "string",
  "logo": "string",
  "webSite": "string"
}
```

3. **updateCompany(id, name, description, location, address, logo, webSite)**

- **Description:** Updates an existing company by ID.
- **Method:** PUT
- **Endpoint:** /company/edit/{id}
- **Request Body:**

```
{
  "name": "string",
  "description": "string",
  "location": "string",
  "address": "string",
  "logo": "string",
  "webSite": "string"
}
```

4. **deleteCompany(id)**

- **Description:** Deletes a company by ID.
- **Method:** DELETE
- **Endpoint:** /company/delete/{id}

5. **getCompany(id)**

- **Description:** Fetches a specific company by ID.
- **Method:** GET
- **Endpoint:** /company/\${id}
- **Returns:** A promise with the company data.

5.3 App Component

The App React component implements a job application management system with routing to different views for managing applications, companies, and job offers. Below is a detailed explanation of the key parts of the code:

Imports

The component imports various modules and components necessary for the functionality:

- **React and React-Router:** These are essential for defining routes and rendering the UI components.
- **Components:** Includes Applications, Companies, JobOffers, Header, and various forms like ApplicationAdd, CompanyAdd, JobOfferEdit, etc., which are used to manage job-related entities.
- **AppService:** A service object used for making API calls to the backend to perform CRUD operations.

Component Structure

The App component is a React class component that manages the state and handles API calls for different entities (applications, companies, and job offers).

State

The state consists of:

- **applications:** Stores all job applications.
- **joboffers:** Stores all job offers.
- **companies:** Stores all company records.
- **selectedApplication, selectedCompany, selectedJobOffer:** Store currently selected records to edit or view details.

Lifecycle Method

- **componentDidMount():** Invoked immediately after the component is mounted. It calls the `fetchData` method to load all the necessary data by calling `loadApplications()`, `loadCompanies()`, and `loadJobOffers()`.

Data Fetching and CRUD Operations

Each entity (applications, companies, job offers) has corresponding methods for fetching, adding, updating, and deleting:

1. Applications:

- loadApplications: Fetches applications using AppService.fetchApplications() and updates the state.
- addApplication, updateApplication, deleteApplication: These handle adding, updating, and deleting an application respectively by calling the corresponding API method in AppService.

2. Companies:

- loadCompanies: Fetches companies using AppService.fetchCompanies() and updates the state.
- addCompany, updateCompany, deleteCompany: Handle CRUD operations for companies by calling the appropriate methods in AppService.

3. Job Offers:

- loadJobOffers: Fetches job offers from the backend using AppService.fetchJobOffers().
- addJobOffer, updateJobOffer, deleteJobOffer: Manage job offer creation, updating, and deletion.

4. Filtered Job Offers:

- loadFilteredJobOffers: Fetches job offers based on a filter by calling AppService.fetchJobOffersFilter(name).

Route Definitions

The Router from react-router-dom is used to define routes for different pages, with each route linked to a specific component:

- /home: Displays a list of job offers with filtering options.
- /job_details/:id: Displays details for a specific job offer.
- /applications, /applications/add, /applications/edit/:id: Manage applications.
- /companies, /companies/add, /companies/edit/:id: Manage companies.
- /joboffers, /joboffers/add, /joboffers/edit/:id: Manage job offers.
- /chat: Renders the chatbot component.
- /login, /register: Handles user authentication.

A default route is set to redirect users to the job offers page using <Navigate to="/joboffers" />.

Error Handling

Each data-fetching or API-call method includes `.catch()` to handle potential errors, logging them to the console using `console.error()`.

AppService API Calls

The API interactions are managed via `AppService`. These are Axios-based functions that make HTTP requests to the backend:

- **fetchJobOffersFilter(name)**: Fetch job offers filtered by name.
- **fetchApplications, fetchCompanies, fetchJobOffers**: Retrieve applications, companies, and job offers.
- **addApplication, updateApplication, deleteApplication**: Manage CRUD operations for applications.
- **addCompany, updateCompany, deleteCompany**: CRUD operations for companies.
- **addJobOffer, updateJobOffer, deleteJobOffer**: CRUD operations for job offers.

5.4 App.js component

The App React component implements a job application management system with routing to different views for managing applications, companies, and job offers. Below is a detailed explanation of the key parts of the code:

Imports

The component imports various modules and components necessary for the functionality:

- **React and React-Router**: These are essential for defining routes and rendering the UI components.
- **Components**: Includes Applications, Companies, JobOffers, Header, and various forms like ApplicationAdd, CompanyAdd, JobOfferEdit, etc., which are used to manage job-related entities.
- **AppService**: A service object used for making API calls to the backend to perform CRUD operations.

Component Structure

The App component is a React class component that manages the state and handles API calls for different entities (applications, companies, and job offers).

State

The state consists of:

- **applications**: Stores all job applications.
- **joboffers**: Stores all job offers.
- **companies**: Stores all company records.

- `selectedApplication`, `selectedCompany`, `selectedJobOffer`: Store currently selected records to edit or view details.

Lifecycle Method

- `componentDidMount()`: Invoked immediately after the component is mounted. It calls the `fetchData` method to load all the necessary data by calling `loadApplications()`, `loadCompanies()`, and `loadJobOffers()`.

Data Fetching and CRUD Operations

Each entity (applications, companies, job offers) has corresponding methods for fetching, adding, updating, and deleting:

5. Applications:

- `loadApplications`: Fetches applications using `AppService.fetchApplications()` and updates the state.
- `addApplication`, `updateApplication`, `deleteApplication`: These handle adding, updating, and deleting an application respectively by calling the corresponding API method in `AppService`.

6. Companies:

- `loadCompanies`: Fetches companies using `AppService.fetchCompanies()` and updates the state.
- `addCompany`, `updateCompany`, `deleteCompany`: Handle CRUD operations for companies by calling the appropriate methods in `AppService`.

7. Job Offers:

- `loadJobOffers`: Fetches job offers from the backend using `AppService.fetchJobOffers()`.
- `addJobOffer`, `updateJobOffer`, `deleteJobOffer`: Manage job offer creation, updating, and deletion.

8. Filtered Job Offers:

- `loadFilteredJobOffers`: Fetches job offers based on a filter by calling `AppService.fetchJobOffersFilter(name)`.

Route Definitions

The Router from `react-router-dom` is used to define routes for different pages, with each route linked to a specific component:

- `/home`: Displays a list of job offers with filtering options.
- `/job_details/:id`: Displays details for a specific job offer.
- `/applications`, `/applications/add`, `/applications/edit/:id`: Manage applications.
- `/companies`, `/companies/add`, `/companies/edit/:id`: Manage companies.

- `/joboffers, /joboffers/add, /joboffers/edit/:id`: Manage job offers.
- `/chat`: Renders the chatbot component.
- `/login, /register`: Handles user authentication.

A default route is set to redirect users to the job offers page using `<Navigate to="/joboffers" />`.

Error Handling

Each data-fetching or API-call method includes `.catch()` to handle potential errors, logging them to the console using `console.error()`.

AppService API Calls

The API interactions are managed via `AppService`. These are Axios-based functions that make HTTP requests to the backend:

- **`fetchJobOffersFilter(name)`**: Fetch job offers filtered by name.
- **`fetchApplications, fetchCompanies, fetchJobOffers`**: Retrieve applications, companies, and job offers.
- **`addApplication, updateApplication, deleteApplication`**: Manage CRUD operations for applications.
- **`addCompany, updateCompany, deleteCompany`**: CRUD operations for companies.
- **`addJobOffer, updateJobOffer, deleteJobOffer`**: CRUD operations for job offers.

5.5 authService

This module handles user authentication and registration in a React application by interacting with an API using `axios` and managing user data through `localStorage`. It provides functions for registering, logging in, logging out, and retrieving the current authenticated user.

Functions:

1. *`register(name, surname, username, password, role)`*

Registers a new user by sending a POST request to the API.

- **Parameters:**
 - `name`: The user's first name.
 - `surname`: The user's last name.
 - `username`: The desired username for the account.
 - `password`: The desired password for the account.
 - `role`: The user's role (e.g., admin, user).
- **Behavior:**

- Sends a POST request to the `/register` endpoint with the user's information.
- If the registration is successful and an `access_token` is returned in the response, it stores the user data in `localStorage`.
- Returns the response data.

2. *login(username, password)*

Authenticates a user by sending a POST request to the API.

- **Parameters:**
 - `username`: The user's username.
 - `password`: The user's password.
- **Behavior:**
 - Sends a POST request to the `/login` endpoint with the user's credentials.
 - If the login is successful and an `access_token` is returned, it stores the user data in `localStorage`.
 - Redirects the user to the `/joboffers` page.
 - Reloads the page after 50 milliseconds to update the view based on authentication.
 - Returns the response data.

3. *logout()*

Logs the user out by removing the user data from `localStorage` and reloading the page.

- **Behavior:**
 - Removes the user item from `localStorage`.
 - Reloads the page to reset the user's session.

4. *getCurrentUser()*

Retrieves the currently logged-in user from `localStorage`.

- **Returns:**
 - The parsed user object if the user is logged in.
 - `false` if no user is currently logged in.

API Endpoints:

- **`/register`**: The endpoint for registering new users.
- **`/login`**: The endpoint for user login.

LocalStorage Keys:

- **user:** The key used to store the user object, which includes the access token and other user information, in localStorage.

5.6 Header Component

The Header component is a navigation bar that displays links based on whether the user is logged in or not. It fetches the current user from localStorage using the authService.

- **State:**
 - currentUser: A boolean value indicating whether a user is logged in.
- **Methods:**
 - useEffect: Retrieves the current user when the component mounts.
 - logout: Logs the user out and reloads the page.
- **Behavior:**
 - Displays different navigation links depending on whether the user is logged in. If logged in, the "Logout" link is shown, otherwise "Register" and "Login" links are shown.
- **Props:** None.

5.7 Home Component

The Home component is the main page of the job listing site, responsible for displaying recent job offers and providing a search bar for filtering jobs based on a keyword.

Key Features:

State Management:

- page: Tracks the current page for pagination (not fully implemented in this version).
- size: Specifies the number of job offers per page.
- name: Captures the keyword from the search form.

Event Handling:

- **handleChange(e):** Updates the name field in the component's state when the user types into the search bar.
- **onFormSubmit(e):** Handles form submission when the user clicks "Find job." It prevents the default form behavior and triggers the onFilter prop function with the keyword name as a parameter. This function is expected to handle filtering of the job offers.

Rendering Job Offers:

- The component calls `getJobOfferPage()` to map through the list of job offers (`this.props.joboffers`) and display each job offer as a card. Each card contains:
 - The company logo.
 - The job position.
 - The company's name and location.
 - A link to job details, passing the offer ID to the route.

Preloader:

- While loading the page, a preloader is displayed with the company logo.

Footer:

- Contains links and contact information, including team members and university information.

getJobOfferPage() Function:

This function maps over the `joboffers` array passed via props and generates JSX for each job offer, including a company logo, job title, company name, location, and dates.

5.8 Details Component

The Details component displays detailed information about a specific job offer selected by the user.

Key Features:

React Router:

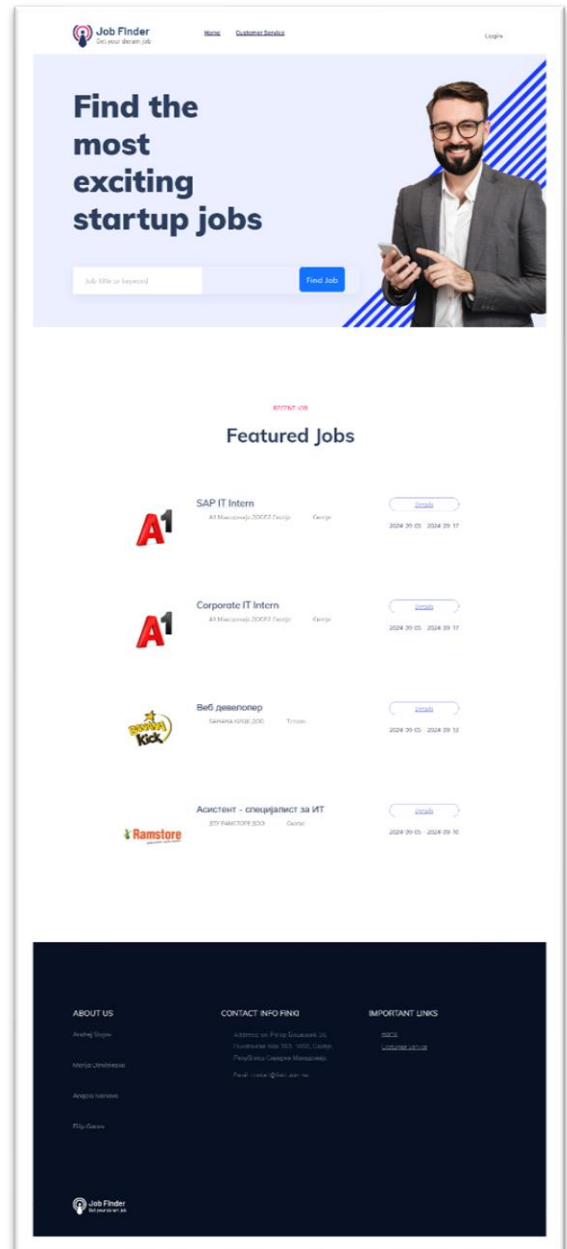
- The component uses `useParams` from `react-router-dom` to extract the job offer id from the URL.

State Management:

- `isLoading`: Boolean flag that tracks whether the data is still loading.
- `jobOffer`: Stores the detailed job offer data fetched from the API.

Fetching Data:

- **`useEffect()`**: The component fetches job offer details when it mounts or when the id in the URL changes. The `AppService.getJobOffer(id)` function is used to retrieve the job offer data by its ID.



- If the data is successfully fetched, it updates the jobOffer state and turns off the loading flag (isLoading).

Conditional Rendering:

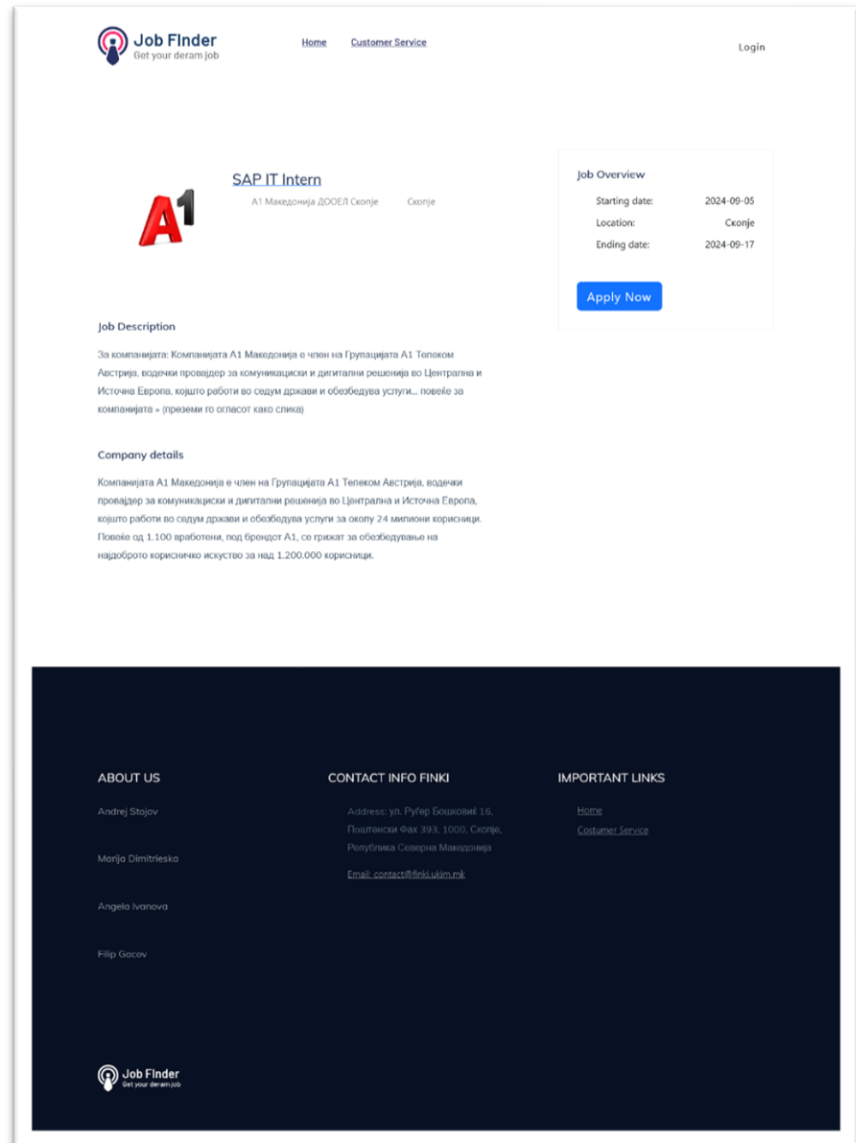
- Displays a preloader while the data is being fetched (isLoading is true).
- Once the data is loaded, the job offer details are displayed, including:
 - Company logo.
 - Job position, company name, and location.
 - Job description and company details.

Job Overview:

- Provides a summary of the job offer, including the start and end dates and a button linking to the application form.

Footer:

- Similar to the Home component, the footer contains contact information and links to other parts of the site.



API Integration:

The component integrates with an external API (`AppService.getJobOffer(id)`) to retrieve job offer data. The API call is made inside `useEffect()` to ensure it triggers on component mount or when the id changes.

5.9 Job Offers Management Components

This group of components handles the entire functionality related to job offers, including viewing, editing, adding, and deleting job offers. The components involved are:

- **JobOffers (Main Job Offers Component):** Displays a paginated table of job offers. Users can navigate through pages, add new job offers, or interact with existing ones.

- **State:**
 - page: The current page number.
 - size: Number of job offers per page.
- **Methods:**
 - handlePageClick: Changes the page when the user interacts with pagination.
 - getJobOfferPage: Returns a subset of job offers for the current page.
- **Props:**
 - joboffers: Array of job offers.
 - onDelete: Function to delete a job offer.
 - onEdit: Function to edit a job offer.
- **Behavior:**
 - Renders the job offers in a table, each row represented by JobOfferTerm. Includes pagination and an "Add new Job-Offer" button.
- **JobOfferAdd:** A form component for adding a new job offer.
 - **State:**
 - formData: Tracks the input values for the new job offer.
 - **Methods:**
 - handleChange: Updates form input values.
 - onFormSubmit: Submits the form and calls the parent onAddJobOffer function.
 - **Props:**
 - onAddJobOffer: Callback function to handle adding the job offer.
 - companies: Array of available companies for selection.
 - **Behavior:**
 - Form with fields for entering job offer details like position, details, dates, location, and company. Redirects to the job offers list after submission.
- **JobOfferEdit:** A form component for editing an existing job offer.
 - **State:**
 - formData: Tracks input values for the job offer being edited.
 - **Methods:**
 - handleChange: Updates form values.

- onFormSubmit: Submits the edited job offer and calls the parent onEditJobOffer function.
- **Props:**
 - joboffers: The current job offer being edited.
 - onEditJobOffer: Callback function to handle editing the job offer.
 - companies: Array of companies for selection.
- **Behavior:**
 - Prefilled form to edit job offer details like position, details, dates, location, and company. Redirects to the job offers list after submission.

Position	Details	Starting Date	Ending Date	Location	Company	Applications
SAP IT Intern	За компанијата: Компанијата A1 Македонија е член на Групацијата A1 Телеком Аустрија, водечки провајдер за комуникациски и дигитални решенија во Централна и Источна Европа, којшто работи во осум држави и обезбедува услуги... повеќе за компанијата » (презими го огласот како слика)	2024-09-05	2024-09-17	Скопје	A1 Македонија ДООЕЛ Скопје	Delete Edit Apply
Corporate IT Intern	За компанијата: Компанијата A1 Македонија е член на Групацијата A1 Телеком Аустрија, водечки провајдер за комуникациски и дигитални решенија во Централна и Источна Европа, којшто работи во осум држави и обезбедува услуги... повеќе за компанијата » (презими го огласот како слика)	2024-09-05	2024-09-17	Скопје	A1 Македонија ДООЕЛ Скопје	Delete Edit Apply
Веб дeвeлoпeр	За компанијата: Банања Кикс ДОО е нов јазец на спортски објектувалници со уплатни места во неколку градови во државата, а на осмро во сите градови и помали места, повеќе за компанијата » БЕБ ДЕВЕЛОПЕР Опис на работни задачи: Работа на проекти, веб програми, дизајн на веб страна По потреба, давање техничка поддршка на нашите вработени Бакор и одржување на база на податоци Одржување на техничките уреди во нашите објектувалници и канцеларии (Работка и имплементација на код, дилетистрира и поправка на грешки Други обврски поврзани со општо на работното место Баране Точност, прецизност, одговорност, Претходно искуство на иста или слична работна позиција ќе се смета за предност Нашата понуда: Компетитивна плата Работно време: 08:30 – 18:30 1 слободен ден Одлични услови за работа и тесно работна атмосфера Моментот за живети и професионален развој (одговор одговор согласно ЗРК) Доколку сакате да бидете дел од тимот на Банања Кикс , Ве молиме Вашето CV(со фотографија) да го испратите на е-пошта: bjanajki@bjanajki.mk или преку пошта "Аплицирајте" Во насокот на мимолет (судет) наведете ја работната позиција на која аплицирате. Само селекционерите кандидатите ќе бидат повикани на интервју.	2024-09-05	2024-09-13	Тетово	БАНАНА КИКС ДОО	Delete Edit Apply
Асистент - специјалист за ИТ	За компанијата: Рамсторе Македонија е ланец од 27 мартети, каде онлајн мартет "Рамсторе до дома" и еден трговски центар Рамсторе кои ги искористи можностите на пазарот на трга да се дојде како работодавец кој нуди... повеќе за компанијата » (презими го огласот како PDF)	2024-09-05	2024-09-10	Скопје	ДТУ РАМСТОРЕ ДОО	Delete Edit Apply

- **JobOfferTerm:** Represents a single job offer within a table row. It displays details like position, dates, location, and company.
 - **Props:**
 - term: The job offer object.
 - onDelete: Function to delete the job offer.
 - onEdit: Function to edit the job offer.

- **Behavior:**
 - Displays the job offer details and provides buttons for editing, deleting, or applying to the job.

5.10 Company Management Components

This group of components collectively manages the companies in an application. It covers listing, adding, editing, and deleting companies.

1. Companies Component

The Companies component handles the display of the list of companies, paginated for easier navigation. It provides users with the ability to add new companies and paginate through the list.

- **State:**
 - page: The current page number being viewed.
 - size: The number of companies displayed per page.
- **Props:**
 - companies: An array of company objects to be displayed.
 - onDelete: Function to delete a company.
 - onEdit: Function to edit a company.
- **Methods:**
 - handlePageClick(data): Updates the page number when pagination is clicked.
 - getCompaniesPage(offset, nextPageOffset): Filters the companies list based on the current page and size.
- **Rendering:**
 - Displays a table with company details (name, description, location, address, logo, website).
 - Pagination is managed using the ReactPaginate component.

2. CompanyTerm Component

The CompanyTerm component represents a single row in the company table. It displays the details of a single company and includes options to edit or delete the company.

Props:

- term: The individual company object containing its data (name, description, location, address, logo, website).
- onDelete: Function to trigger the deletion of the company.
- onEdit: Function to trigger the edit of the company.

Rendering:

- Displays the company's name, description, location, address, logo, and website.
- Provides "Edit" and "Delete" buttons for each company entry.

3. CompanyAdd Component

The CompanyAdd component provides a form for adding a new company to the list. Upon submission, it calls the onAddCompany function passed from the parent component to create a new company.

State:

- formData: An object storing the form inputs (name, description, location, address, logo, website).

Props:

- onAddCompany: Function to handle adding a new company, passed from the parent.

Methods:

- handleChange(e): Updates the formData state with the input field changes.
- onFormSubmit(e): Handles form submission and calls the onAddCompany function with the form data.

4. CompanyEdit Component

The CompanyEdit component allows editing an existing company. It pre-populates the form with the company's current data and updates the record upon form submission.

State:




- formData: An object storing the updated company information (name, description, location, address, logo, website).

Props:

- company: The current company object being edited (containing its current name, description, etc.).
- onEditApplication: Function to handle updating the company.

Methods:

- handleChange(e): Updates the formData state with input field changes.
- onFormSubmit(e): Handles form submission and updates the company by calling onEditApplication.

Job Offers						
Applications Companies Job Offers ChatBot						
Logout						
Job Finder						
Get your dream job						
Home Customer Service						
Login						
Name	Description	Location	Address	Logo	Website	
A1 Македонија ДООЕЛ Скопје	Компанијата А1 Македонија е член на Групацијата А1 Телеком Австрија, водечки провајдер за комуникациски и дигитални решенија во Централна и Источна Европа, којшто работи во седум држави и обезбедува услуги за околу 24 милиони корисници. Повеќе од 1.100 вработени, под брендот А1, се грижат за обезбедување на најдоброто корисничко искуство за над 1.200.000 корисници.	Скопје	Почтааа Пресвета Богородица бр. 1		http://www.a1.mk	Delete Edit
БАНАНА КИЦК ДОО	Банана Кикс ДОО е нов ланец на спортски објектувалници со уплатни места во неколку градови во државата, а на скоро во сите градови и помали места.	Тетово	Благодја Токаа 222		http:// www.bananakick.mk	Delete Edit
ДТУ РАМСТОРЕ ДОО	Рамстор Македонија е ланец од 27 маркети, еден онлајн маркет „Рамстор до дома“ и еден трговски центар Рамстор мол кој ги искористи можностите на пазарот на труд да се докаже како работодавец кој нуди успешна кариера за своите вработени. Освен чувствителноста за потребите на потрошувачите, Рамстор има извонредна сензитивност за потребите на своите вработени, препознавајќи ги нивните потенцијали и нудејќи можности за развој, благодарение на тој непосреден пристап, веќе 18 години создава успешни кариерни приказни каде што има за цел да не е само компанија, туку и многу повеќе од тоа - семејство на своите вработени. Рамстор е семејството неограничено расте и се развија, постојано се збогатува со нови маркети, нови вработени, но и новитети. Доколку сакате да бидете дел од динамична компанија, која ги цени своите вработени и е свесна дека успехот е можен само со вистинскиот тим, аплицирајте на огласите. ВИЕ ТРИЖЕТЕ СЕ ЗА РАБОТАТА, НИЕ КЕ СЕ ПОТРУДИМЕ ЗА ВАШАТА КАРИЕРА ПРИДРУЖЕТЕ НИ	Скопје	улица Кирил и Методиј бр.13		http:// ramstore.com.mk/	Delete Edit
Add New Company						

[back](#) [next](#)

5.11 ChatBot Component

This component provides a simple chatbot interface for customer service.

Description:

- Allows users to interact with an AI chatbot by sending and receiving messages.
- Manages conversation state using useState hooks.
- Handles asynchronous communication with a backend API using axios.
- Displays a loading state when a message is being sent to prevent multiple simultaneous submissions.

Key Features:

- **Messages:** Displays user and bot messages in a chat window.
- **Input Field:** Lets users type messages and send them either by clicking a button or pressing "Enter".
- **Loading State:** Disables input and changes the button text while awaiting the response.

States:

- messages: An array of messages in the conversation.
- input: The current message input from the user.
- loading: Boolean to track if the app is waiting for a response from the backend.

Methods:

- sendMessage(): Sends a message to the backend and updates the UI with the bot's response.

- `handleKeyPress(e)`: Sends the message if the Enter key is pressed.


[Home](#)
[Customer Service](#)
[Login](#)

AI Customer Service

5.12 ApplyForm Component

This component provides a form for users to apply for a job.

Description:

- Allows users to fill out a form with personal information and upload a CV to apply for a job.
- Submits the form data, including file upload, to a backend API using axios.
- Uses `useParams` to fetch the job offer ID from the URL parameters and sends it with the form data.

Key Features:

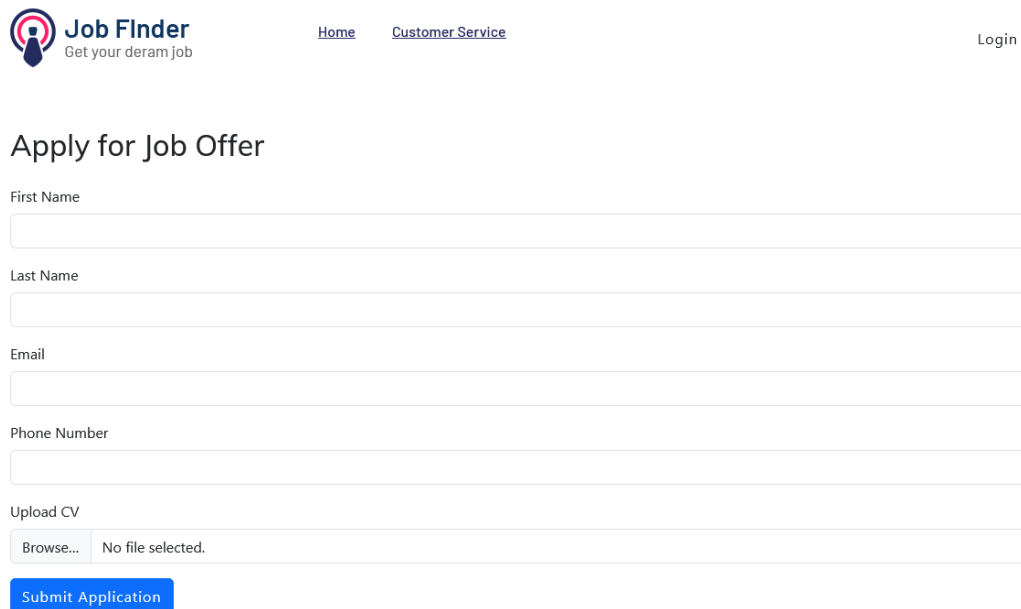
- **Form Submission:** Submits user details (name, email, phone number) and a CV file to the backend.
- **File Upload:** Allows users to attach a CV file.
- **Error Handling:** Displays success or error messages based on the result of the submission.

States:

- `formData`: Object that holds the user input values (name, lastName, email, phone number, and CV).
- `message`: Success message displayed after successful submission.
- `error`: Error message if the form submission fails.

Methods:

- `handleChange(e)`: Updates the form data as the user types.
- `handleFileChange(e)`: Handles the CV file upload.
- `handleSubmit(e)`: Submits the form data, including the file, to the backend.



The image shows a web application interface for 'Job Finder'. At the top left is the logo with the text 'Job Finder' and 'Get your deram job' below it. To the right of the logo are two links: 'Home' and 'Customer Service'. Further right is a 'Login' link. Below the header is a section titled 'Apply for Job Offer'. This section contains several input fields: 'First Name', 'Last Name', 'Email', and 'Phone Number', each with a corresponding text input box. Below these is an 'Upload CV' section with a 'Browse...' button and the text 'No file selected.'. At the bottom of the form is a blue 'Submit Application' button.

6. Conclusion

The job listing website provides a comprehensive platform for IT job seekers and employers. By combining web scraping, dynamic job listings, and a secure authentication system, it offers an efficient and user-friendly experience. The modular architecture ensures that the site can be easily maintained and scaled as new features are introduced.