Institut spécialisé dans les métiers de l'offshoring Tétouan

NTT DATA

INTERNSHIP REPORT

2023-2024

JSON CONVERSION AND PERSISTENCE

Made by:

Ouail KAHI

Marouan AKECHTAH

Mohamed Rida El BADRI

Anas Ben Omar

Academic supervisor:
Mr. Oussama Rahmouni

Professional supervisor:
Mr. Hatim El Khalkhali

ACKNOWLEDGEMENT

To my parents, whose unwavering support and endless sacrifices have been the cornerstone of my success; to my family, whose love and encouragement have been my strength; to my mentors, whose guidance has been instrumental; and to my friends, whose support and camaraderie have been invaluable. Lastly, I thank myself for the perseverance and dedication that brought me here. To all of you, my heartfelt gratitude.

-Ouail KAHI

I dedicate this work to my devoted parents, whose constant support and encouragement have been my anchor; to my family, whose love and backing have been my foundation; to my mentors, whose guidance and insights have been invaluable; and to my friends, whose loyalty and support have been a source of strength. Lastly, I extend my gratitude to myself for the determination and perseverance that led to this achievement. Thank you all.

-Anas BEN OMAR

To my wonderful parents, thank you for your endless love and unwavering support. To my family, your encouragement has been my driving force. To my friends, your support and friendship have been invaluable. To my mentors, your guidance and wisdom have been instrumental. Lastly, I thank myself for the hard work, dedication, and resilience that have brought me to this point. My deepest appreciation to all.

- Mohamed Rida El BADRI

To my incredible family, whose steadfast support has been my foundation; to my parents, whose encouragement has driven me forward; to my mentors, whose wisdom has guided me; and to my friends, whose companionship has been a source of joy and strength. This work stands as a testament to the collective love, support, and inspiration that have shaped my journey. Thank you all.

-Marouan AKECHTAH

ABSTRACT

This report details the process and implementation of a project aimed at automating the extraction, transformation, and storage (ETL) of data from PDF resumes into a PostgreSQL database. Our team's mission was to utilize a Java library, created by another team, to extract data from PDF resume files and convert it into JSON format. Since the library was not available during our development phase, we simulated its functionality using a JSON file representing the expected output of the library.

Using the Jackson library, we then converted the JSON data into Java objects, which were subsequently persisted in the PostgreSQL database through Spring Data JPA. This approach ensured efficient data handling and storage, leveraging the robust features of Spring Boot for application configuration and management, and the powerful repository abstraction provided by Spring Data JPA for seamless database interactions.

The report outlines the technical stack, implementation details, and workflow, including JSON parsing and database entity setup. It also discusses the challenges encountered, such as data mapping issues and performance optimization, and presents solutions to these problems. Future enhancements to the project are suggested, including data validation mechanisms and bulk processing capabilities.

This project demonstrates a successful integration of various technologies to automate a crucial data processing task, thereby improving the overall efficiency and accessibility of resume data for the organization.

Methodologies:

To ensure a well-structured and efficient design, we utilized UML (Unified Modeling Language) diagrams during the planning phase. Specifically, we designed the structure of our entities using UML class diagrams, which helped us visualize the relationships and attributes of each entity in our system. Additionally, we created a sequence diagram to model the flow of interactions between different components of the system during the process of converting and storing data. These diagrams provided a clear blueprint for implementation and facilitated better communication among team members.

Our system was developed using an n-tier architecture, which includes:

- Business Logic Layer: Contains the core functionality and business rules of the application.
- Data Access Layer: Manages interactions with the PostgreSQL database using Spring Data JPA.

This architectural approach ensured a modular and maintainable codebase, enabling efficient development and future scalability.

INDEX

Contents

ACKRONYMES

- **DAO**: **D**ata **A**ccess **O**bject
- **DTO**: **D**ata **T**ransfer **O**bject
- **ER: E**ntity **R**elationship
- **ETL**: **E**xtraction, **T**ransformation and **L**oading
- **JAR**: **J**ava **AR**chive
- **JDK**: **J**ava **D**evelopment **K**it
- **JPA**: **J**akarta **P**ersistence **A**PI / **J**ava **P**ersistence **A**PI
- **JRE**: **J**ava **R**untime **E**nvironment
- **JSON**: **J**ava **S**cript **O**bject **N**otation
- **JVM**: Java Virtual Machine
- **PDF**: **P**ortable **D**ocument **F**ormat
- **SQL**: **S**tructured **Q**uery **L**anguage
- **UML**: **U**nified **M**odeling **L**anguage

OVERVIEW OF INTERNSHIP ACTIVITIES

Activity	Duration	Start Date	End Date
Specifications	1 week	01/05/2024	**08/05/2024**
Design	1 week	08/05/2024	**15/05/2024**
Programming	**2 weeks**	**15/05/2024**	**29/05/2024**

LIST OF FIGURES

I. INTRODUCTION

In today's fast-paced digital world, organizations are increasingly relying on automated systems to handle routine tasks efficiently. One such task is the processing and storage of resumes received in PDF format. Manually extracting data from these resumes is not only time-consuming but also prone to errors. Recognizing this challenge, our project aims to automate the extraction, transformation, and loading (ETL) of resume data into a PostgreSQL database, thereby streamlining the recruitment process and enhancing data accessibility.

The project is divided into three main phases, following the ETL framework. The first phase, handled by another team, involves developing a Java library that extracts data from PDF files and converts it into JSON format. This library is designed to capture relevant information from resumes, such as personal details, educational qualifications, and work experience, and represent it in a structured JSON format. Unfortunately, due to the unavailability of the library during our development phase, we simulated its functionality using a pre-defined JSON file to represent the expected output.

In the transformation phase, our team converts the JSON data into Java objects using the Jackson library. This involves data mapping, validation, and preparation for storage. Finally, in the loading phase, we persist these Java objects into a PostgreSQL database using Spring Boot and Spring Data JPA. This robust system automates the end-to-end process of resume data extraction, transformation, and storage, reducing manual effort and minimizing errors.

A key aspect of our development process is the adoption of a **code-first approach**. In this approach, we start by defining our database schema directly in the code using entity classes. This allows us to manage and evolve the database schema through code, providing a more agile and maintainable development process. The ORM tool, Spring Data JPA in our case, translates these entity classes into the corresponding database schema, ensuring that the database structure always aligns with our application's data model.

This report delves into the technical stack and methodologies employed in the project, including the design of entity structures using UML class diagrams and the interaction flow modeled by sequence diagrams. We also discuss the challenges encountered during implementation, such as data mapping issues and performance optimization, and provide solutions to these problems. Furthermore, we suggest future enhancements to the project, including data validation mechanisms and bulk processing capabilities.

By successfully integrating various technologies, this project not only automates a crucial data processing task but also improves the overall efficiency and accessibility of resume data for the organization. The automation of this ETL process is expected to have a significant impact on the recruitment workflow, allowing for faster and more accurate data handling, and ultimately supporting better hiring decisions.

Objectives

- Convert PDF resumes to JSON using an existing Java library.
- Parse JSON data to Java objects using the Jackson library.

- Persist parsed objects into a PostgreSQL database using Spring Boot and Spring Data JPA.

II. Presentation of the Host Company – NTTDATA

At its core, NTT DATA is driven by a relentless commitment to facilitating digital transformation for its clients. Offering a comprehensive suite of consulting, strategic advisory, and IT services, the company serves as a trusted partner to organizations across diverse sectors, empowering them to navigate the complexities of the digital age. Whether it's streamlining business processes, enhancing customer experiences, or fostering innovation, NTT DATA's solutions are meticulously crafted to address the unique challenges and aspirations of its global clientele.

Central to NTT DATA's ethos is its unwavering focus on client-centricity. By deeply understanding the distinct needs and objectives of each client, the company ensures the delivery of tailored solutions that drive tangible business value. Through enduring partnerships founded on trust, reliability, and unparalleled service, NTT DATA has earned a reputation as a strategic advisor and catalyst for digital transformation.

As the digital landscape continues to evolve at a rapid pace, NTT DATA remains at the vanguard of technological innovation. With a steadfast focus on talent development and a relentless pursuit of excellence, the company is poised to shape the future of the digital world. Through the development of cutting-edge technologies and transformative solutions, NTT DATA is committed to driving economic growth, fostering societal progress, and unlocking new possibilities for clients across the globe.

III. Technology Stack

The technology stack forms the backbone of any software project, providing the essential tools and frameworks necessary for its development and execution. In the context of our automated resume processing project, the carefully chosen technology stack plays a pivotal role in ensuring efficiency, reliability, and scalability throughout the application lifecycle.

By understanding the intricacies of our technology stack, including its strengths, capabilities, and integration points, stakeholders gain valuable insights into the underlying architecture and mechanisms driving our project forward. From Java and Spring Boot to Hibernate and PostgreSQL, each technology serves a distinct purpose, collectively forming a cohesive ecosystem that empowers us to deliver a robust, feature-rich solution.

1. Java

Java is one of the most popular programming languages used in developing environments today. It is primarily employed for back-end development projects, game development, and desktop and mobile computing.

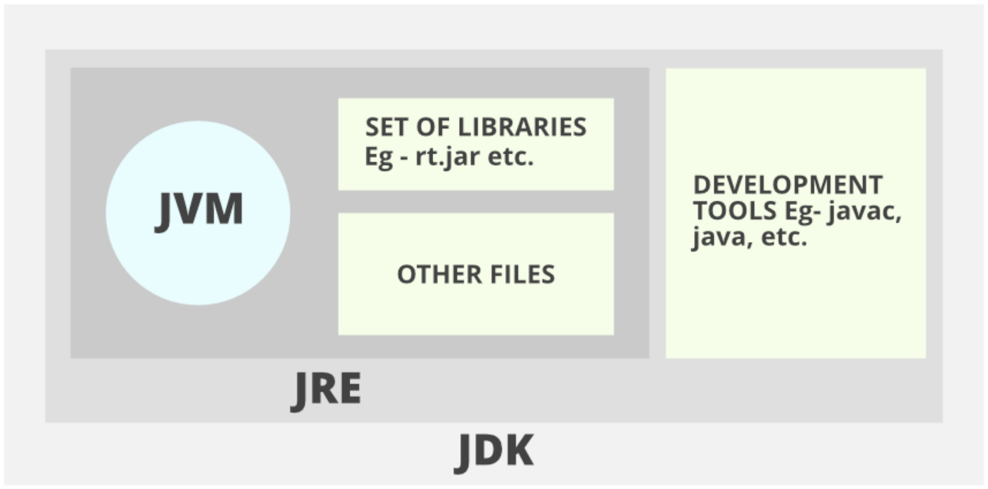

JVM
SET OF LIBRARIES Eg - rt.jar etc.
OTHER FILES
DEVELOPMENT TOOLS Eg- javac, java, etc.
JRE
JDK

Figure 1 Illustration of java's JDK, JRE and JVM

Java Virtual Machine (JVM)

The Java Virtual Machine (JVM) is the core of the Java programming language. It loads, verifies, and runs Java bytecode, making Java applications platform-independent by converting bytecode to machine-specific code.

Java Runtime Environment (JRE)

JRE is a set of software tools used for running Java applications. It provides the libraries, Java Virtual Machine (JVM), and other components to run applications written in Java. JRE is necessary for executing Java programs. It uses heap space for dynamic memory allocation and includes deployment technologies (like Java Web Start and Java plug-in), development toolkits (like Java 2D, AWT, Swing), and integration libraries (like JDBC, JNDI).

Java Development Kit (JDK)

JDK is a software development kit that is a superset of JRE. It includes the tools required to compile, debug, and run Java applications.

JDK is essential for Java development. It includes JRE, a compiler, a debugger, an archiver, and other tools necessary for developing Java applications.

2. Spring Framework

The Spring Framework is a comprehensive framework for building enterprise Java applications. It provides a wide range of functionalities, including dependency injection, aspect-oriented programming, transaction management, and more, making it a highly flexible and powerful tool for developers.

a. Dependency Injection (DI)

Dependency Injection is a core feature of Spring that allows the creation and management of dependencies between application components in a loosely coupled manner.

Instead of creating dependencies manually, Spring handles the creation and injection of dependencies, which promotes easier development, unit testing and maintainability.

b. Aspect-Oriented Programming (AOP)

Aspect-Oriented Programming (AOP) is a key feature of the Spring Framework that enhances modularity by allowing the separation of concerns. In our project, AOP helps to manage functionalities that are not central to the business logic but are necessary for the overall application, such as logging or security.

AOP allows additional behavior to be added to existing code without modifying it. This is achieved by defining aspects that encapsulate these additional behaviors. Although we did not implement any specific cross-cutting concerns in our project, understanding AOP principles ensures that our application design remains flexible and maintainable for future enhancements.

c. Spring Boot

Spring Boot is an extension of the Spring Framework that simplifies the development of Spring applications by providing out-of-the-box configurations, embedded servers, and a convention-over-configuration approach.

In our project, Spring Boot significantly streamlined the development process:

- **Quick Setup and Configuration**

  Auto-Configuration: Spring Boot's auto-configuration handled most of the initial setup, reducing the time and effort required to configure the application manually.

- **Simplified Database Interactions**

  Spring Data JPA: Integrated seamlessly with Spring Boot, allowing us to perform database operations with minimal boilerplate code.

  HikariCP: Enhances the performance of database interactions by providing efficient connection pooling out-of-the-box.

- **Enhanced Development Experience**

  Dependency Management: The use of Spring Boot starters simplified the inclusion of necessary dependencies, ensuring compatibility and reducing configuration errors.

Production-Ready Features: Features like health checks and metrics facilitated easier monitoring and management of the application, ensuring reliability and performance in production environments.

d. Spring Data JPA

In our project, we adopt a **code-first approach**. This approach involves defining our database schema directly in the code using entity classes. These entity classes are annotated to map them to corresponding tables in the database. The ORM (Object-Relational Mapping) tool, in this case, Spring Data JPA, translates these entity classes into the actual database schema. This ensures that the database structure is consistently aligned with our application's data model and allows us to manage and evolve the schema through code, which is more agile and maintainable.

- **Creating Entities:**

  To map the JSON data to the database, we define a set of entities that represent the structure of the resumes. Each entity corresponds to a table in the PostgreSQL database.

- **Using Repositories:**

  To handle CRUD operations on these entities, we create repositories that extend the **JpaRepository** interface. This interface provides methods for standard database operations without requiring us to write any boilerplate code.

By using Spring Data JPA and HikariCP, we achieve a robust, scalable, and efficient data access layer that simplifies the complexity of interacting with the PostgreSQL database, ensuring seamless persistence of resume data.

3. Jackson Library

The Jackson library is a powerful and flexible tool used for converting between Java objects and JSON. In our project, we utilized Jackson to parse JSON data representing resumes and convert this data into Java objects, which are then persisted in our PostgreSQL database.

Using Jackson involves a few key steps:

- **Reading JSON Data:** The JSON file containing the resume data is read into a string format.
- **Parsing JSON to Java Objects:** The **ObjectMapper** class from the Jackson library is used to convert the JSON string into Java objects, specifically instances of our **ResumeDTO** class.
- **Saving Java Objects:** These Java objects are then processed and saved into the database, ensuring that all the relevant data from the resumes is accurately stored.

4. Lombok

In our application, we use Lombok to reduce boilerplate code, making our codebase cleaner and more maintainable. Lombok is a Java library that automatically generates common methods such as getters, setters, constructors, toString, equals, and hashCode during the compilation phase. This helps us focus on the business logic rather than writing repetitive code.

Key Lombok Annotations:

- **`@Data:`** The `@Data` annotation is a convenient shorthand that bundles the features of `@ToString`, `@EqualsAndHashCode`, `@Getter`, `@Setter`, and `@RequiredArgsConstructor` in a single annotation. When applied to a class, it automatically generates all the necessary getters, setters, and other utility methods.

- **`@AllArgsConstructor`**: The `@AllArgsConstructor` annotation generates a constructor with one parameter for each field in the class. This is useful when you need to create objects with all properties initialized.
- **`@NoArgsConstructor`**: The `@NoArgsConstructor` annotation generates a no-argument constructor. This is especially useful for dependency injection which often require a default constructor to create instances of a class through reflection. Additionally, it is beneficial for JPA entities, as JPA requires a no-argument constructor to instantiate entity objects.
- **`@Builder`**: The `@Builder` annotation produces complex builder APIs for your classes. It allows you to instantiate your objects in a more readable and maintainable way, especially when dealing with objects with many properties.
- **`@Getter`** and **`@Setter`:** These annotations are used to generate getter and setter methods for individual fields or for all fields in a class.

5. PostgreSQL

PostgreSQL is an advanced, open-source relational database management system (RDBMS) known for its robustness, scalability, and compliance with SQL standards. In our project, PostgreSQL serves as the primary data storage solution, efficiently handling the persistence of resume data extracted from JSON files.

In our project, PostgreSQL is used to store and manage the resume data processed from JSON files. Here’s how PostgreSQL fits into our technology stack:

1. **Data Modeling**: We designed the database schema to represent resumes and their associated information accurately. Entities such as Resume, Experience, Education, and Skills are mapped to corresponding tables in the PostgreSQL database.
2. **Persistence with Spring Data JPA**: Using Spring Data JPA, we interact with the PostgreSQL database seamlessly. Spring Data JPA abstracts the low-level database interactions, allowing us to focus on the business logic.
3. **Transaction Management**: PostgreSQL’s transaction management ensures that all database operations are executed reliably and consistently. In our project, we utilize Spring’s transaction management capabilities to handle database transactions.
4. **Query Optimization**: With PostgreSQL’s advanced indexing and query optimization features, we ensure efficient data retrieval and manipulation, which is crucial for handling large volumes of resume data.
5. **Scalability**: As our application grows, PostgreSQL provides the scalability required to handle increased data loads and user queries without compromising performance.

IV. System Design and Architecture

Our project is designed to automate the extraction and storage of data from PDF resumes into a PostgreSQL database. The system architecture leverages several technologies and follows a layered architecture to separate concerns and improve maintainability. The key components of the system include:

- **Java Library for PDF to JSON Conversion**: Created by another team, this library is responsible for converting PDF resumes into JSON format.
- **JSON Parsing and Object Mapping**: Using the Jackson library, we parse JSON data and map it to Java objects representing the resume's structure.
- **Service Layer**: This layer handles the business logic, including the validation and transformation of data.
- **Data Access Layer**: Utilizing Spring Data JPA, this layer manages database interactions, including saving and retrieving data from the PostgreSQL database.
- **Database**: PostgreSQL is used to store the structured resume data.

1. UML Class Diagram

The UML Class Diagram below illustrates the main entities in our system and their relationships.

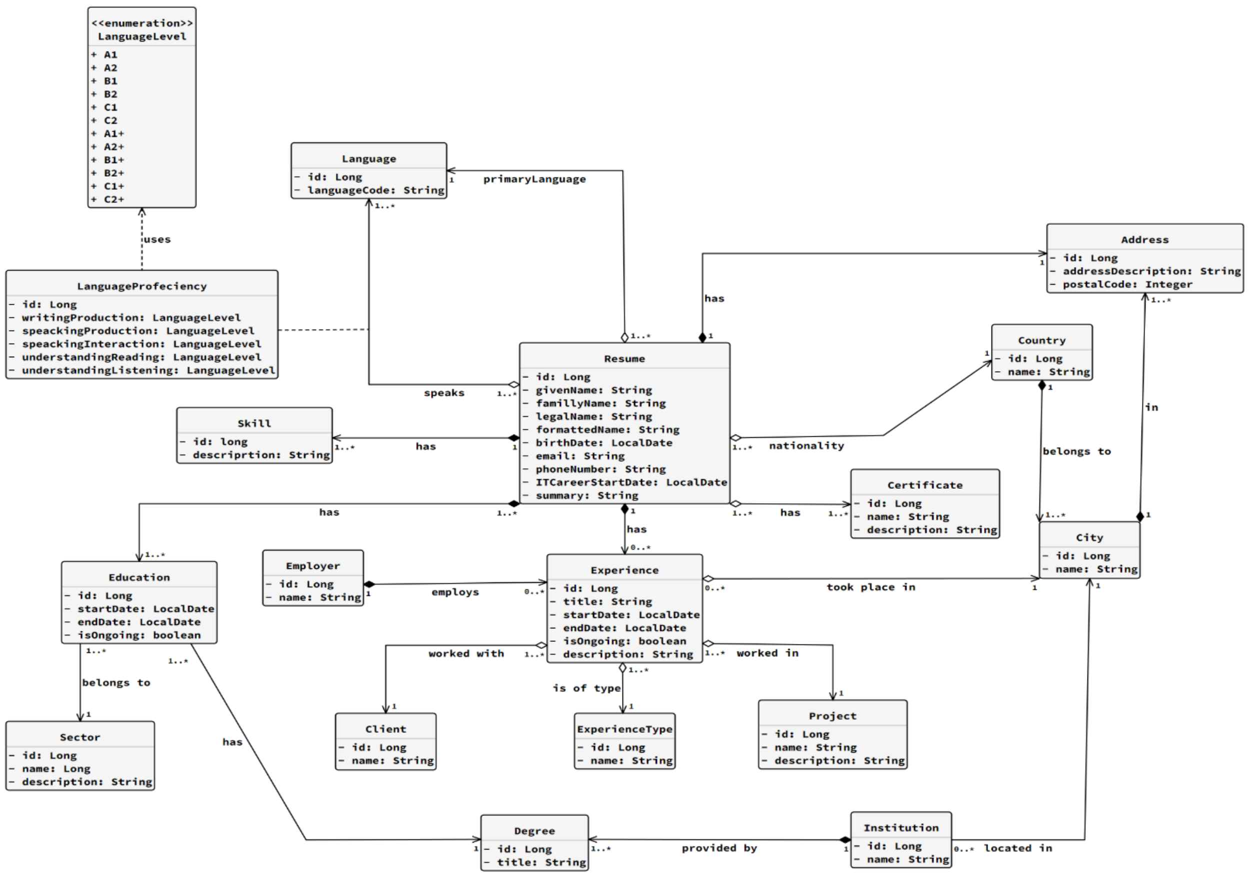


Figure 2 UML class diagram of the resume's structure

The provided class diagram serves as a blueprint for creating entities in our project, which will be used to model the data and interactions within the application. Here's how each part of the diagram translates into entity creation.

Each box in the class diagram represents an entity that will be mapped to a database table. We will use JPA annotations to define these entities.

a. Types of Associations

Aggregation:

- Represented by a hollow diamond at the aggregate (whole) side.
- Indicates a "whole-part" relationship where the part can exist independently of the whole.

Composition:

- Represented by a filled diamond at the composite (whole) side.
- Indicates a "whole-part" relationship where the part cannot exist independently of the whole.

Dependency:

- Represented by a dashed arrow pointing from the dependent to the independent entity.
- Indicates a temporary relationship where one entity relies on another for some functionality.

Example: **LanguageProficiency** has a dependency on **LanguageLevel**, where LanguageProficiency has attributes of type **LanguageLevel**.

Association:

- Represented by a solid line connecting entities.

- Indicates a general relationship between entities, without implying a whole-part hierarchy.

b. Cardinalities

Cardinalities define the number of instances of one entity that can be associated with one instance of another entity. They are specified at both ends of an association line.

One-to-One (1: 1):

- A single instance of one entity is associated with a single instance of another entity.
- Example: between Address and Resume Address where Resume has only one Address.

One-to-Many (1: *):

- A single instance of one entity is associated with multiple instances of another entity.
- Example: Resume has many Skills, indicating a one-to-many relationship.

Many-to-One (*: 1):

- Multiple instances of one entity are associated with a single instance of another entity.
- Example: Many Experiences are associated with one Employer.

Many-to-Many (*: *):

- Multiple instances of one entity are associated with multiple instances of another entity.
- Example: Resume and Language through LanguageProficiency, where each Resume can have multiple Languages, and each Language can be spoken by multiple Resumes.

2. Sequence Diagram:

The UML sequence diagram below represents a simplified illustration of the project :

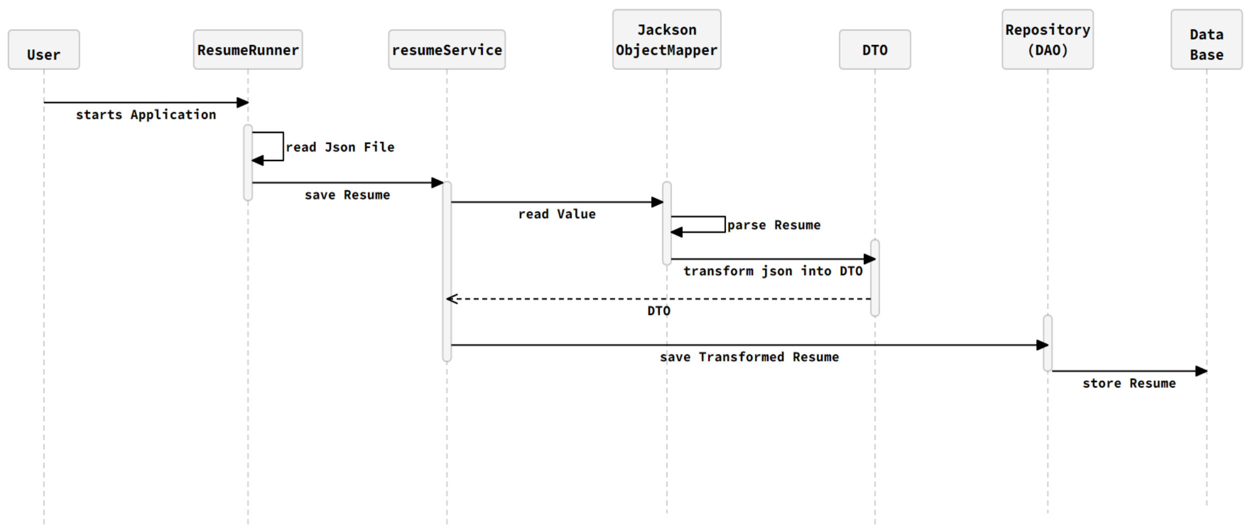


Figure 3 simplified UML sequence diagram of the application

The sequence diagram illustrates the process of converting a JSON file into a resume object and then persisting that object into the database. Here's a step-by-step breakdown of the interactions:

1. **User Interaction**:
    - The process begins when the user starts the application. This triggers the ResumeRunner component.
2. **ResumeRunner**:
    - The ResumeRunner component reads the JSON file containing the resume data.
3. **resumeService**:
    - After reading the JSON file, ResumeRunner calls the saveResume method on the resumeService to initiate the processing and saving of the resume data.
4. **Jackson ObjectMapper**:
    - The resumeService utilizes the Jackson ObjectMapper to convert the JSON string into a ResumeDTO object. The readValue method of ObjectMapper is called to parse the JSON.
5. **DTO (Data Transfer Object)**:
    - The JSON string is transformed into a ResumeDTO object, which serves as an intermediate data structure used for transferring data between layers of the application.
6. **Repository (DAO)**:
    - Once the ResumeDTO is created, it is passed back to the resumeService, which then converts the DTO into an entity object suitable for persistence.

- The resumeService calls the repository (DAO) to save the transformed resume entity to the database.

7. **Database**:

    - The repository performs the necessary operations to store the resume entity in the database, completing the process.

V. Implementation Details

1. Code Structure

The project is organized into packages to ensure separation of concerns. Below is an overview of the code structure within the **ntt.cv.europass** package:

a. dto Package

The dto (Data Transfer Object) package contains classes that are used to transfer data between different layers of the application.

in our application Data Transfer Objects (DTOs) serve as recipient objects for data extracted from the JSON output of the library developed by the other team to read from PDF resume files.

In our simulation, the JSON file contains objects representing different sections of the resume. Using Jackson's ObjectMapper, we can extract the data from the JSON and populate instances of our DTO classes with the data from these JSON objects. Example:

```
"language": {
                "languageCode": "en"
            },
```

The corresponding DTO class, **LanguageDTO**, is designed to match the JSON structure:

```java
@Data
public class LanguageDTO {
    private Long id;
    private String languageCode;
}
```

`@Data`: Generates getters, setters, toString(), equals(), and hashCode() methods.

b. entity Package

The entity package includes the JPA entities that map to the database tables. Each class in this package is annotated with JPA annotations to define the mapping between the class and the corresponding database table.

Each entity will be a class annotated with `@Entity`. The fields in each class in the previous UML class diagram will be mapped to class attributes, with JPA annotations specifying the details.

Example: Language.java: Represents the Language entity.

```java
@AllArgsConstructor
@NoArgsConstructor
@Entity
@Table(name = "Language")
@Data
@Builder
public class Language {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(name = "language_code")
    private String languageCode;

    @OneToMany(mappedBy = "language", fetch = FetchType.EAGER)
    @Builder.Default
    private List<LanguageProficiency> languageProficiencies = new
ArrayList<>();

}
```

- **`@AllArgsConstructor`**: Generates a constructor with one parameter for each field in the class.

- **`@NoArgsConstructor`**: Generates a no-argument constructor.
- **`@Entity`**: Marks the class as a JPA entity, making it a table in the database.
- **`@Table(name = "Language")`**: Specifies the name of the database table to be "Language".
- **`@Data`**: Generates getters, setters, toString(), equals(), and hashCode() methods.
- **`@Builder`**: Implements the builder pattern for the class, allowing for easy and readable object creation.

c. mapper Package

The mapper package contains classes that convert between entities and DTOs. These mappers help in transforming the data from the entity format to the DTO format and vice versa.

Example: LanguageMapper.java: Contains methods to map **LanguageDTO** to Language.

```
public class LanguageMapper {
    private LanguageMapper() {
    }

    public static Language mapToEntity(LanguageDTO dto) {
        if (dto == null) {
            return null;
        }

        return Language.builder()
                .id(dto.getId())
                .languageCode(dto.getLanguageCode())
                .build();
    }
}
```

d. repository Package

The repository package houses the Spring Data JPA repositories. These interfaces extend JpaRepository and provide CRUD operations for the entities. Custom query methods can also be defined in these interfaces.

Example:

LanguageRepository.java: Repository interface for Language entity.

```java
@Repository
public interface LanguageRepository extends JpaRepository<Language, Long> {

    Language findByLanguageCode(String code);
}
```

- **`@Repository`** Annotation: This annotation indicates that the interface is a Spring Data Repository, which allows Spring to discover it and create a bean for it.
- `` `extends JpaRepository<Language, Long>` ``: By extending JpaRepository, the **LanguageRepository** interface inherits various methods for performing CRUD operations. generic parameters Language and Long specify the domain type the repository manages (i.e., Language entity) and the type of the entity's primary key (i.e., Long).
- Custom Query Method: This method follows Spring Data JPA's naming convention. By naming the method **findByLanguageCode**, Spring Data JPA understands that this method should find a **Language** entity where the **languageCode** field matches the given code, and **Long** indicate the type of entity and its primary key type, respectively.

e. runner Package

The runner package contains the class **ResumeRunner** that will be executed when the application starts.

The **ResumeRunner** class is a component in the Spring Boot application that implements the **CommandLineRunner** interface. This class is designed to execute specific code at application startup, primarily focused on reading a JSON file and processing its contents.

ResumeRunner will Read the JSON file and call `resumeService.saveResume()` to save the resume data.

```java
@Component
@RequiredArgsConstructor
public class ResumeRunner implements CommandLineRunner {

    private final ResumeService resumeService;

    @Override
    public void run(String... args) throws Exception {
        // Object pointing to the europass_cv.json file
        ClassPathResource resource = new
ClassPathResource("europass_cv.json");

        // Convert URI to a Path object
        Path path = Paths.get(resource.getURI());

        // Read the JSON file into a String
        String json = new String(Files.readAllBytes(path));

        // Save the resume along with its associated data
        resumeService.saveResume(json);
    }
}
```

`@Component`:

- Marks the ResumeRunner class as a Spring-managed component.
- Ensures that Spring will detect this class during component scanning and create an instance of it in the application context.

`@RequiredArgsConstructor`:

- Generates a constructor for the ResumeRunner class that includes a parameter for the **ResumeService** dependency.
- Ensures that **resumeService** is properly injected when ResumeRunner is instantiated by Spring.
- Promotes constructor injection, which is generally preferred in Spring applications for its benefits in testing and immutability.

f. service Package

The service package includes the service classes that contain the business logic of the application. These classes use the repositories to interact with the database and perform operations such as saving, updating, or retrieving data.

ResumeService

The **ResumeService** is annotated with **`@Service`**. In Spring, **`@Service`** is used to indicate that this class is a service, meaning it holds the business logic of the application. It's typically used to encapsulate transactional or business logic operations.

1) **Parsing JSON to DTO**: The parseResume method is responsible for converting a JSON string into a ResumeDTO object. It uses Jackson's ObjectMapper to deserialize the JSON string into a Java object:

```
    private ResumeDTO parseResume(String json) throws 
JsonProcessingException {

        ObjectMapper objectMapper = new ObjectMapper().registerModule(new 
JavaTimeModule());

        return objectMapper.readValue(json, ResumeDTO.class);
    }
```

2) **Saving a Resume**: The saveResume method is the entry point for saving a resume. It takes a JSON string as input, parses it into a ResumeDTO, and then maps it to a Resume entity using ResumeMapper. It checks if a resume with the same email already exists. If not, it proceeds to save the new resume.

```
    @Transactional
    public void saveResume(String json) throws JsonProcessingException {
        ResumeDTO resumeDTO = parseResume(json);
        Resume resume = ResumeMapper.mapToEntity(resumeDTO);

        Optional<Resume> existingResumeOptional = 
Optional.ofNullable(resumeRepository.findByEmail(resume.getEmail()));
        if (existingResumeOptional.isPresent()) {
            System.out.println("Email already registered.");
        } else {
            saveNewResume(resume);
        }
    }
```

- **`@Transactional`** annotation is used to mark a method or a class as transactional, meaning that any database operations performed within the marked method or class will be executed within a transaction. If the transaction is successful, the changes will be committed to the database.

3) **Saving a New Resume**: The saveNewResume method is called internally by saveResume to save a new resume entity along with its associated data. It saves address, primary language, certificates, educations, and countries associated with the resume.

```
    private void saveNewResume(Resume resume) {
        // Save the Resume entity along with its associated data
        Address address = addressService.save(resume.getAddress());
        resume.setAddress(address);

        Language language =
languageService.save(resume.getPrimaryLanguage());
        resume.setPrimaryLanguage(language);

        List<Certificate> certificates =
certificateService.saveMultiple(resume.getCertificates());
        resume.setCertificates(certificates);

        List<Education> educations =
educationService.saveMultiple(resume.getEducations());
        resume.setEducations(educations);

        List<Country> nationalities =
countryService.saveMultiple(resume.getCountries());
        resume.setCountries(nationalities);

        // Save the Resume entity
        Resume savedResume = resumeRepository.save(resume);

        // Save and associate other entities
        saveAndAssociateEntities(savedResume);
    }
```

4) **Associating Entities**: After saving the main resume entity, saveAndAssociateEntities is called to save and associate experiences, skills, and language proficiencies with the resume. It iterates through the lists of these entities, associates them with the resume, and then saves them.

```
    private void saveAndAssociateEntities(Resume resume) {
        // Save and associate experiences
        List<Experience> experiences = resume.getExperiences();
        experiences.forEach(experience -> experience.setResume(resume));
        experienceService.saveMultiple(experiences);

        // Save and associate skills
        List<Skill> skills = resume.getSkills();
        skills.forEach(skill -> skill.setResume(resume));
        skillService.saveMultiple(skills);

        // Save and associate language proficiencies
        List<LanguageProficiency> languageProficiencies =
resume.getLanguageProficiencies();
        languageProficiencies.forEach(languageProficiency ->
languageProficiency.setResume(resume));
        languageProficiencyService.saveMultiple(languageProficiencies);

 }
```

MainService:

The **MainService** class is designed to provide a generic foundation for CRUD operations, which other service classes in the application can extend to reduce redundancy and ensure consistency. This abstract class encapsulates common data operations, allowing derived classes to focus on their specific business logic.

```java
@Service
@RequiredArgsConstructor
public abstract class MainService<T> {

    protected final JpaRepository<T, Long> repository;

    @Transactional
    public T save(T entity) {
        if (entity == null) {
            return null;
        }
        return repository.save(entity);
    }

    @Transactional
    public List<T> saveMultiple(List<T> entities) {
        return repository.saveAll(entities);
    }

    @Transactional
    public void delete(T entity) {
        repository.delete(entity);
    }

    @Transactional
    public void deleteMany(List<T> entity) {
        repository.deleteAll(entity);
    }
}
```

- **Generic Operations**: The class is generic, parameterized by type T. This allows it to handle operations on any type of entity.

- **Transactional Methods**: All methods in the **MainService** class are annotated with `@Transactional`, meaning that each method will execute within a transactional context. This ensures that if any operation within a method fails, the entire operation will be rolled back, maintaining data consistency.
- **Save Operation**: The save method takes an entity of type T as input and saves it using the injected repository's save method. It checks if the entity is null before saving it.
- **Save Multiple Operation**: The saveMultiple method takes a list of entities of type T and saves them all at once using the repository's saveAll method.
- **Delete Operation**: The delete method takes an entity of type T and deletes it from the database using the repository's delete method.
- **Delete Multiple Operation**: The deleteMany method takes a list of entities of type T and deletes them all at once using the repository's deleteAll method.
- **Transactional Management:** The `@Transactional` annotation on methods in the MainService class ensures that all database operations are performed within a transactional context. This means that any failure in these operations will result in a rollback, preserving data integrity.

To illustrate how the MainService class is utilized, consider the LanguageService class, which extends MainService<Language>.

```java
@Service
public class LanguageService extends MainService<Language> {

    private final LanguageRepository languageRepository;

    public LanguageService(LanguageRepository repository, LanguageRepository
languageRepository) {
        super(repository);
        this.languageRepository = languageRepository;
    }

    @Override
    public Language save(Language entity) {
        if (entity == null) {
            return null;
        }
        Language languageExist =
languageRepository.findByLanguageCode(entity.getLanguageCode());

        if (languageExist != null) {
            return languageExist;
        } else {
            return super.save(entity);
        }
    }
}
```

- **Inheritance of CRUD Operations**: By extending MainService<Language>, the LanguageService class inherits common CRUD operations such as saving, updating, and deleting language entities. This eliminates the need to redefine these operations in each service class, promoting code reuse and consistency.

- **Custom Business Logic**: While the MainService class provides generic implementations of CRUD methods, the **LanguageService** and other service classes can override these methods to add custom business logic. In this example, the save method is overridden to include a check for existing language entities based on their language code before saving a new entity.
  If a language entity with the same language code already exists, it returns the existing entity instead of creating a new one. Otherwise, it calls the save method of the MainService class to save the new entity.

g. EuropassApplication.java

This is the main class of the Spring Boot application. It contains the main method which is the entry point of the application. The class is annotated with `@SpringBootApplication` to enable Spring Boot's auto-configuration and component scanning.

```
@SpringBootApplication
public class EuropassApplication {

    public static void main(String[] args) {
        SpringApplication.run(EuropassApplication.class, args);
    }

}
```

2. Database generation

In our application, we adopted the code-first approach to generate the database schema directly from our JPA entity classes. This approach ensures that the database structure is always in sync with our application's data model, allowing for more agile development and easier maintenance.

a. Code-First Approach with JPA Entities

We began by defining our domain models as JPA entity classes. Each class represents a table in the database, and the fields in the class correspond to columns in the table. We used various JPA annotations to map the class fields to database columns and to define relationships between entities.

- **Entity Definitions:** We previously defined our domain models as JPA entity classes in the entity package. Each class represents a table in the database, and the fields in the class correspond to columns in the table. We used various JPA annotations to map the class fields to database columns and to define relationships between entities.
- **Configuration of Spring Data JPA**: We configured Spring Data JPA in our application.properties file to automatically generate the database schema based on these entities. The key configuration property is `spring.jpa.hibernate.ddl-auto`, which controls how Hibernate handles schema generation.

```
spring.datasource.url=jdbc:postgresql://localhost:5432/database
spring.datasource.username=username
spring.datasource.password=password
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format_sql=true
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.PostgreSQLDialect
```

b. Generated Database

the following ER diagram represents the structure of the database tables generated using

spring data JPA and hibernate:

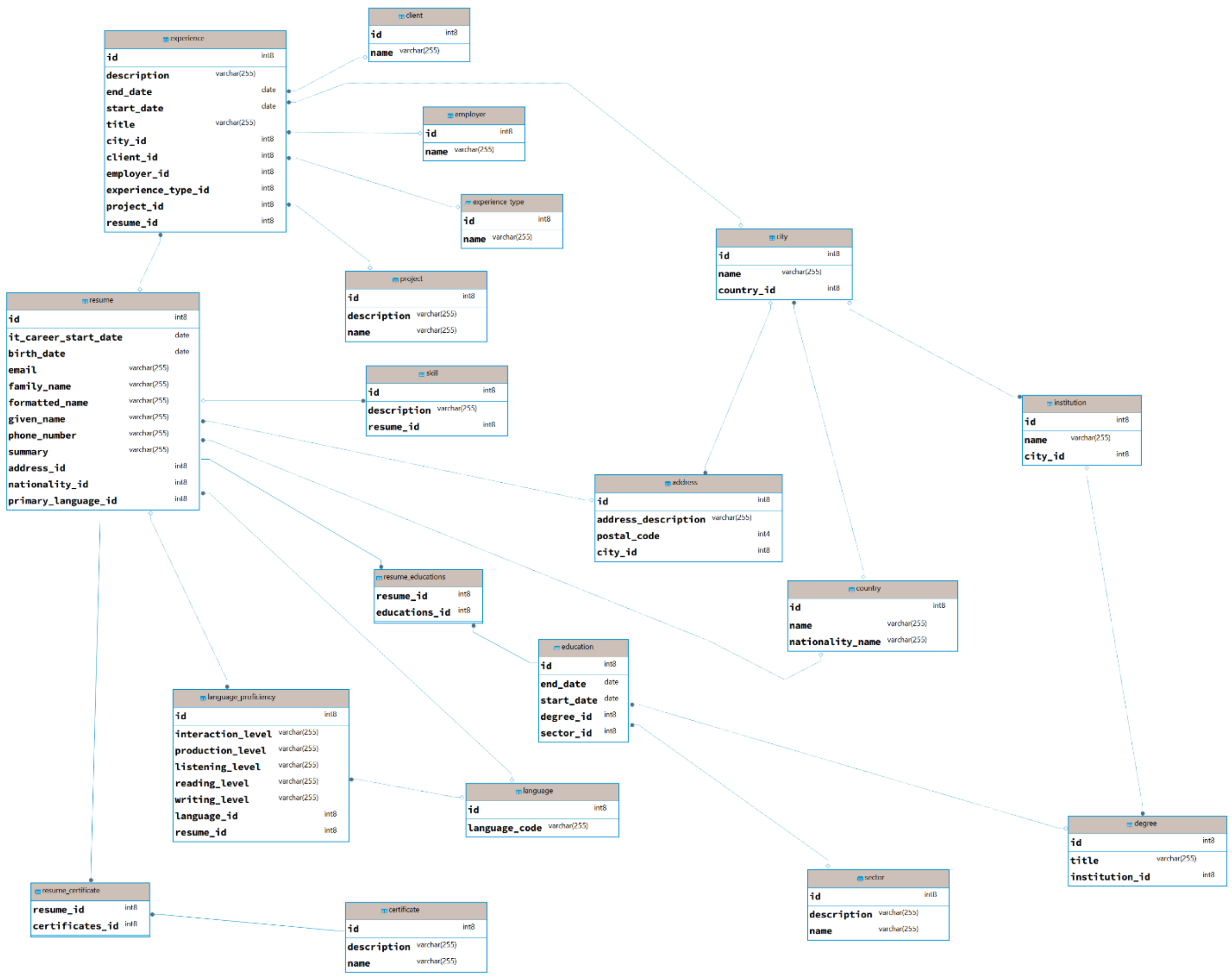


Figure 4 ER diagram of the generated database

DDL explanation:

- Certificate Table: The certificate table stores information about certifications.

```
CREATE TABLE certificate (
      id int8 GENERATED BY DEFAULT AS IDENTITY NOT NULL,
      description varchar(255) NULL,
      "name" varchar(255) NULL,
      CONSTRAINT certificate_pkey PRIMARY KEY (id)
);
```

- Client Table:

The client table stores client information.

```
CREATE TABLE client (
	id int8 GENERATED BY DEFAULT AS IDENTITY NOT NULL,
	"name" varchar(255) NULL,
	CONSTRAINT client_pkey PRIMARY KEY (id)
);
```

- Country Table:

The country table stores country information including nationality.

```
CREATE TABLE country (
	id int8 GENERATED BY DEFAULT AS IDENTITY NOT NULL,
	"name" varchar(255) NULL,
	nationality_name varchar(255) NULL,
	CONSTRAINT country_pkey PRIMARY KEY (id)
);
```

- Employer Table:

The employer table stores employer information.

```
CREATE TABLE employer (
	id int8 GENERATED BY DEFAULT AS IDENTITY NOT NULL,
	"name" varchar(255) NULL,
	CONSTRAINT employer_pkey PRIMARY KEY (id)
);
```

- Experience_type Table:

The experience_type table stores different types of experiences.

```
CREATE TABLE experience_type (
	id int8 GENERATED BY DEFAULT AS IDENTITY NOT NULL,
	"name" varchar(255) NULL,
	CONSTRAINT experience_type_pkey PRIMARY KEY (id)
);
```

- Language Table

The language table stores language information.

```
CREATE TABLE "language" (
	id int8 GENERATED BY DEFAULT AS IDENTITY NOT NULL,
	language_code varchar(255) NULL,
	CONSTRAINT language_pkey PRIMARY KEY (id)
);
```

- Project Table

The project table stores information about projects.

```
CREATE TABLE project (
	id int8 GENERATED BY DEFAULT AS IDENTITY NOT NULL,
	description varchar(255) NULL,
	"name" varchar(255) NULL,
	CONSTRAINT project_pkey PRIMARY KEY (id)
);
```

- Sector Table

The sector table stores sector information.

```
CREATE TABLE sector (
	id int8 GENERATED BY DEFAULT AS IDENTITY NOT NULL,
	description varchar(255) NULL,
	"name" varchar(255) NULL,
	CONSTRAINT sector_pkey PRIMARY KEY (id)
);
```

- City Table

The city table stores information about cities and references the country table.

```
CREATE TABLE city (
	id int8 GENERATED BY DEFAULT AS IDENTITY NOT NULL,
	"name" varchar(255) NOT NULL,
	country_id int8 NULL,
	CONSTRAINT city_pkey PRIMARY KEY (id),
	CONSTRAINT fk_country FOREIGN KEY (country_id) REFERENCES
country(id)
);
```

- Institution Table

The institution table stores information about educational institutions and references the city table.

```
CREATE TABLE institution (
	id int8 GENERATED BY DEFAULT AS IDENTITY NOT NULL,
	"name" varchar(255) NULL,
	city_id int8 NULL,
	CONSTRAINT institution_pkey PRIMARY KEY (id),
	CONSTRAINT fk_city FOREIGN KEY (city_id) REFERENCES city(id)
);
```

- Address Table

The address table stores address details and references the city table.

```
CREATE TABLE address (
	id int8 GENERATED BY DEFAULT AS IDENTITY NOT NULL,
	address_description varchar(255) NULL,
	postal_code int4 NULL,
	city_id int8 NULL,
	CONSTRAINT address_pkey PRIMARY KEY (id),
	CONSTRAINT fk_city FOREIGN KEY (city_id) REFERENCES city(id)
);
```

- Degree Table

The degree table stores information about degrees and references the institution table.

```
CREATE TABLE "degree" (
	id int8 GENERATED BY DEFAULT AS IDENTITY NOT NULL,
	title varchar(255) NULL,
	institution_id int8 NULL,
	CONSTRAINT degree_pkey PRIMARY KEY (id),
	CONSTRAINT fk_institution FOREIGN KEY (institution_id) REFERENCES 
institution(id)
);
```

- Education Table

The education table stores education details and references the degree and sector tables.

```
CREATE TABLE education (
	id int8 GENERATED BY DEFAULT AS IDENTITY NOT NULL,
	end_date date NULL,
	start_date date NULL,
	degree_id int8 NULL,
	sector_id int8 NULL,
	CONSTRAINT education_pkey PRIMARY KEY (id),
	CONSTRAINT uk_degree UNIQUE (degree_id),
	CONSTRAINT fk_degree FOREIGN KEY (degree_id) REFERENCES 
"degree"(id),
	CONSTRAINT fk_sector FOREIGN KEY (sector_id) REFERENCES sector(id)
);
```

- Resume Table

The resume table stores resume details and references the address, country, and language tables.

```
CREATE TABLE resume (
	id int8 GENERATED BY DEFAULT AS IDENTITY NOT NULL,
	it_career_start_date date NULL,
	birth_date date NULL,
	email varchar(255) NULL,
	family_name varchar(255) NULL,
	formatted_name varchar(255) NULL,
	given_name varchar(255) NULL,
	phone_number varchar(255) NULL,
	summary varchar(255) NULL,
	address_id int8 NULL,
	nationality_id int8 NULL,
	primary_language_id int8 NULL,
	CONSTRAINT resume_pkey PRIMARY KEY (id),
	CONSTRAINT fk_address FOREIGN KEY (address_id) REFERENCES
address(id),
	CONSTRAINT fk_country FOREIGN KEY (nationality_id) REFERENCES
country(id),
	CONSTRAINT fk_language FOREIGN KEY (primary_language_id) REFERENCES
"language"(id)
);
```

- Resume Certificate Table

The resume_certificate table links resumes to certificates.

```
CREATE TABLE resume_certificate (
	resume_id int8 NOT NULL,
	certificates_id int8 NOT NULL,
	CONSTRAINT resume_certificate_pkey PRIMARY KEY (resume_id,
certificates_id),
	CONSTRAINT fk_certificate FOREIGN KEY (certificates_id) REFERENCES
certificate(id),
	CONSTRAINT fk_resume FOREIGN KEY (resume_id) REFERENCES resume(id)
);
```

- Resume Educations Table

The resume_educations table links resumes to educations.

```
CREATE TABLE resume_educations (
      resume_id int8 NOT NULL,
      educations_id int8 NOT NULL,
      CONSTRAINT resume_educations_pkey PRIMARY KEY (resume_id,
educations_id),
      CONSTRAINT fk_education FOREIGN KEY (educations_id) REFERENCES
education(id),
      CONSTRAINT fk_resume FOREIGN KEY (resume_id) REFERENCES resume(id)
);
```

- Skill Table

The skill table stores skills and references the resume table.

```
CREATE TABLE skill (
      id int8 GENERATED BY DEFAULT AS IDENTITY NOT NULL,
      description varchar(255) NULL,
      resume_id int8 NULL,
      CONSTRAINT skill_pkey PRIMARY KEY (id),
      CONSTRAINT fk_resume FOREIGN KEY (resume_id) REFERENCES resume(id)
);
```

- Experience Table

The experience table stores work experience details and references multiple other tables including city, client, employer, experience_type, project, and resume.

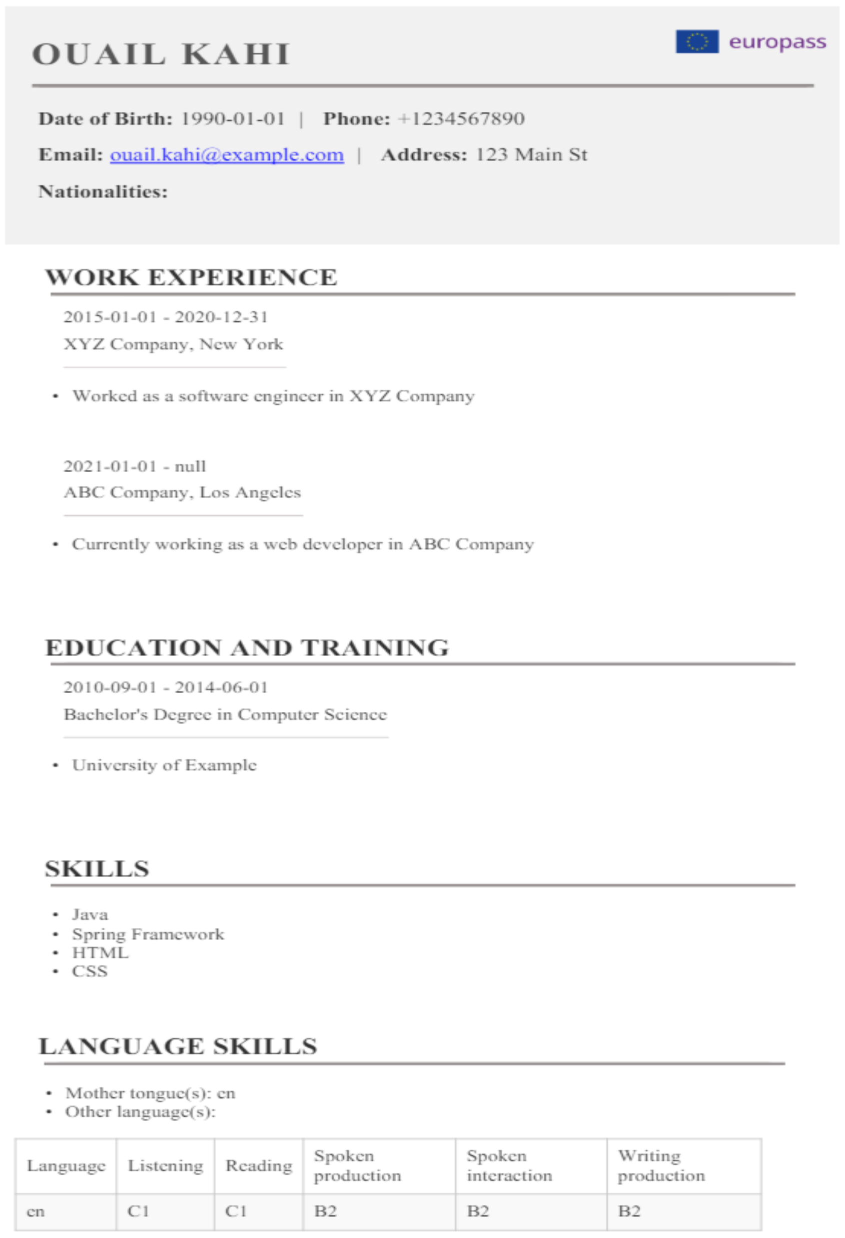
```
CREATE TABLE experience (
	id int8 GENERATED BY DEFAULT AS IDENTITY NOT NULL,
	description varchar(255) NULL,
	end_date date NULL,
	start_date date NULL,
	title varchar(255) NULL,
	city_id int8 NULL,
	client_id int8 NULL,
	employer_id int8 NULL,
	experience_type_id int8 NULL,
	project_id int8 NULL,
	resume_id int8 NULL,
	CONSTRAINT experience_pkey PRIMARY KEY (id),
	CONSTRAINT fk_city FOREIGN KEY (city_id) REFERENCES city(id),
	CONSTRAINT fk_resume FOREIGN KEY (resume_id) REFERENCES resume(id),
	CONSTRAINT fk_employer FOREIGN KEY (employer_id) REFERENCES
employer(id),
	CONSTRAINT fk_project FOREIGN KEY (project_id) REFERENCES
project(id),
	CONSTRAINT fk_experience_type FOREIGN KEY (experience_type_id)
REFERENCES experience_type(id),
	CONSTRAINT fk_client FOREIGN KEY (client_id) REFERENCES client(id)
);
```

- Language Proficiency Table

The language_proficiency table stores language proficiency levels and references the language and resume tables.

```
CREATE TABLE language_proficiency (
	id int8 GENERATED BY DEFAULT AS IDENTITY NOT NULL,
	interaction_level varchar(255) NULL,
	production_level varchar(255) NULL,
	listening_level varchar(255) NULL,
	reading_level varchar(255) NULL,
	writing_level varchar(255) NULL,
	language_id int8 NULL,
	resume_id int8 NULL,
	CONSTRAINT language_proficiency_interaction_level_check CHECK
(((interaction_level)::text = ANY ((ARRAY['A1', 'A2', 'B1', 'B2', 'C1',
'C2'])::text[]))),
	CONSTRAINT language_proficiency_listening_level_check CHECK
(((listening_level)::text = ANY ((ARRAY['A1', 'A2', 'B1', 'B2', 'C1',
'C2'])::text[]))),
	CONSTRAINT language_proficiency_production_level_check CHECK
(((production_level)::text = ANY ((ARRAY['A1', 'A2', 'B1', 'B2', 'C1',
'C2'])::text[]))),
	CONSTRAINT language_proficiency_reading_level_check CHECK
(((reading_level)::text = ANY ((ARRAY['A1', 'A2', 'B1', 'B2',
```

Here is how the CV is displayed



OUAIL KAHI

europass

Date of Birth: 1990-01-01 | **Phone:** +1234567890

Email: ouail.kahi@example.com | **Address:** 123 Main St

Nationalities:

WORK EXPERIENCE

2015-01-01 - 2020-12-31
XYZ Company, New York

- Worked as a software engineer in XYZ Company

2021-01-01 - null
ABC Company, Los Angeles

- Currently working as a web developer in ABC Company

EDUCATION AND TRAINING

2010-09-01 - 2014-06-01
Bachelor's Degree in Computer Science

- University of Example

SKILLS

- Java
- Spring Framework
- HTML
- CSS

LANGUAGE SKILLS

- Mother tongue(s): en
- Other language(s):

Language	Listening	Reading	Spoken production	Spoken interaction	Writing production
en	C1	C1	B2	B2	B2

VI. Conclusion

This project has been an invaluable learning experience, teaching us a great deal about modern software development practices and the intricacies of building a robust application. Throughout the development process, we leveraged various technologies such as Spring Boot, Spring Data JPA, and Jackson, which enhanced our understanding of how to efficiently process and store complex data.

Despite our efforts, we recognize that our solution is not perfect. There are areas where improvements can be made, especially in optimizing performance, enhancing error handling, and ensuring scalability. For instance, we could explore more sophisticated validation mechanisms, refine our service layer logic, and improve our database schema to handle larger volumes of data more effectively.

One significant aspect we encountered was the importance of clean and maintainable code. The use of DTOs to separate concerns and the implementation of a generic service class (MainService) were steps towards a more manageable codebase, yet we realized that there is always room to streamline and optimize our approach further.

In summary, this project has highlighted both our achievements and the areas needing improvement. The experience has reinforced the importance of iterative development, continuous learning, and adaptability in the face of challenges. As we move forward, we are equipped with the knowledge and insights gained from this project, ready to refine and expand upon our work to create solutions that are even more effective.

For the source code, here is the link: https://github.com/ouailkahi/Europras-cv