System architecture

Description of the architecture

Style of the architecture

The chosen architectural style for this system is a multi-layered client-server architecture based on the Model-View-Controller (MVC) design pattern. This style was selected for its capacity to separate business logic, data, and presentation, making maintenance and development easier. The MVC structure organizes the application into three core components: model, view and controller.

- Model handles data while interfering with database. It isn't aware of a view and controller components.
- View manages the user interface and is also responsible for rendering HTML and CSS. Doesn't contain any logic also isn't aware of a model and controller components. Knows about the model component only because it is rendering data from it.
- Controller is the "brain" of the whole application, it connects the model and the view component. It acts as a mediator, processing user input and interfacing between the model and view.

Subsystems

The system is divided into three main subsystems: web server, web application, database.

- Web server is the backend system component, implemented with Spring Boot framework in Java, providing support for HTTP-based communication. It handles the processing of requests from the frontend and routes them to the appropriate service. It also manages application logic, interferes with the database, and it is responsible for coordinating communication between backend and frontend.

- Web application is a software system that is divided into **frontend** and **backend**. **Frontend** is responsible for displaying content to the user. It contains a graphical interface coded in TypeScript using React framework, with whom users are able to interact. It provides client's side validation of data before sending it to the server via HTTPS protocol. Responsive design provides convenient user experience across different platforms. **Backend** is the part of the system responsible for processing incoming requests and executing various system actions. To achieve separation of concerns, it is organized into controllers, services, and repositories. Controllers are responsible for handling incoming HTTPS requests and providing clients with appropriate responses. Services' main purpose is to efficiently process requests from frontend. Repositories manage the backends interaction with the database, abstracting away database complexities and providing a simple and consistent way for services to communicate with data.

- Database securely stores data and communicates with repositories in backend. It uses relational scheme, and it is implemented in Postgre SQL. Data can be quickly inserted, removed or overwritten with this scheme.

Network protocols

Our application uses HTTP/HTTPS, RESTful API, Web Socket, JDBC, and gRPC protocols.

Main data flow

The user initiates interaction with the application via the frontend and sends an HTTP/HTTPS request to the backend server through the RESTful API. The backend server receives the request, routing it through controllers that manage the handling flow. Controllers call the appropriate methods in services, where business logic is implemented, including data validation and calculations. If data storage or retrieval is needed, the service calls repository methods that access the database through the JDBC protocol. Repositories fetch, save, or update data in the database and return it to the service layer. After processing, the backend server sends responses through the RESTful API, which may include confirmations of successful processing, requested data, or error messages. For real-time interactions, feedback to users is sent via Web Socket. The frontend application receives the response and displays the updated information to the user. The user interface updates based on the received data, allowing the user to continue interacting with the system.

Hardware-software requirements

Our application is accessible to a broad user population, so no excessive hardware is needed. The application can be accessed on Windows, macOS and Linux operating systems. It can be run with 8GB RAM with a quad-core processor.

Our team is using Visual Studio Code and IntelliJ IDE. PostgreSQL is being used for the database setup and communication. Backend is configured with Spring Boot framework in Java that allows efficient REST API development. Frontend is displayed with React and JavaScript which provide dynamic and responsive interface. Docker is being used for the deployment and configuration of the application.

Choice explanation of the selected architecture

Multi-Layered MVC Architecture

The multi-layered MVC design offers distinct layers that divide the application into clear functional segments: model, view, and controller. This division simplifies development and allows each layer to evolve independently. Separation of concerns is what makes this model so powerful. Each layer has a defined purpose: the model handles data and interactions with the database, the view manages the user interface, and the controller connects and gives instructions to other layers. Developers can independently access each layer, thus making it easier to implement functionalities and detect errors. Bugs in one layer cannot interfere with another layer, which makes it easier for developers to update, change, maintain and test programs.

HTTP/HTTPS

As an established and well-understood protocol, HTTP ensures compatibility with a wide range of web technologies and frameworks, making integration straightforward. Its simple request-response model supports high performance in web interactions, facilitating efficient, reliable communication between client and server. HTTPS is a standardized and secure protocol. It enables safe data exchange and greatly improves application's overall security.

Java

Java is known for its stability, performance, and cross-platform compatibility, allowing reliable operation across a wide range of systems and devices. Java's extensive library support and active community contribute to faster development and easier problem-solving. Its object-oriented paradigm makes Java an excellent programming language for model component.

Spring Boot

Spring Boot complements Java by offering a comprehensive framework tailored for high-performance applications. Key benefits of Spring Boot include simplified configuration and automated setup, which enable faster application development. Spring Boot provides an easy design of RESTful APIs and integrates well with various databases and external systems. It also provides excellent scalability and modularity for future development, making it easy to expand and adapt the application.

React

React was chosen for the frontend due to its flexibility and efficiency, making it a perfect fit for building dynamic, responsive user interfaces. Known for its component-based architecture, React allows developers to create reusable UI components that enhance both productivity and consistency across the application. One of React's key strengths is its virtual DOM, which efficiently updates and renders only the components that change, providing fast, smooth user interactions. React's scalability and adaptability are perfect for applications that may grow in complexity, as its modular structure enables addition of new features and UI adjustments.

PostgreSQL

Relational database scheme ensures that data can be managed dynamically and effectively, with frequent insertions, updates and deletion. It organizes data into tables with predefined schemas which provide consistency and integrity. Separating data inside a dynamic system enables its fast manipulation and fetching across the platform. PostgreSQL supports the use of SQL, which is a very powerful and standardized language for querying and manipulating data. It allows developers to express complex queries in a clear, understandable way and take advantage of features like joins, aggregations, and filtering to retrieve and manipulate data across multiple tables.

System organization on higher level

The business trip management system is implemented using a **client-server architecture** and consists of several key components that work together to deliver the system's functionality.

1. Client-Server Architecture

Component	Description
Client	The client is a **web application** developed using React. It provides a graphical user interface (GUI) for users to perform actions like submitting trip requests, tracking approvals, and managing expenses. It communicates with the backend via REST APIs using HTTP requests and JSON responses.

Component	Description
Server	The server is a **backend application** built with Spring Boot. It processes client requests, implements business logic, and performs operations such as querying the database, managing authentication, and handling approvals and notifications.

2. Database

Feature	Description
Type	Relational database (PostgreSQL).
Purpose	Persistent storage of structured data, ensuring data integrity and enabling complex queries.
Stored Data	User information (e.g., login credentials, roles), trip details (e.g., destinations, dates, statuses), expense data (e.g., costs, receipts), and relationships between entities.

3. File System

Feature	Description
Purpose	Stores uploaded files, such as receipts or supporting documents for expenses.
Integration	The backend interacts with the file system for uploading, retrieving, and managing files. Metadata, such as file paths or URLs, is stored in the database.

4. Graphical User Interface (GUI)

Feature	Description
Type	Responsive web application built with React.
Features	Allows users to: • Create and manage trip requests • Track trip statuses • Submit expenses with receipts • View reports and notifications
Integration	Communicates with the backend via RESTful APIs. When a user performs an action, the client sends an HTTP request to the server, which processes it and sends a response for rendering.

Organization of the application

The business trip management application is organized into multiple layers, ensuring clear separation of responsibilities and facilitating maintenance and scalability.

1. Frontend and Backend Layers

Layer	Description
Frontend	The frontend is a **React-based web application** that handles the **presentation layer**. Its main responsibilities include: • Providing an intuitive and responsive user interface (UI). • Handling user interactions (e.g., form submissions, navigation). • Making HTTP requests to the backend via REST APIs. • Rendering data received from the backend (e.g., trip details, expense reports). The frontend ensures that users can seamlessly interact with the system, focusing on user experience.
Backend	The backend is implemented in **Spring Boot** and represents the **business logic and data processing layer**. Its responsibilities include: • Handling client requests and routing them to the appropriate services. • Implementing business rules (e.g., trip approval workflows, expense calculations). • Interacting with the database for CRUD (Create, Read, Update, Delete) operations. • Managing authentication and authorization. • Providing RESTful API endpoints to the frontend. The backend acts as the backbone of the application, coordinating between the frontend and other components like the database and file storage.

2. MVC Architecture

The backend follows the **MVC (Model-View-Controller)** design pattern to separate the application logic into distinct layers:

Component	Description
Model	The Model layer represents the **data and business logic**. It includes: • Entities: Objects that map directly to database tables (e.g., User, Trip, Expense). • Repositories: Interfaces for interacting with the database, typically using JPA (Java Persistence API). • Services: Business logic implementations (e.g., trip approval workflows, expense report calculations).
View	The View layer in this architecture is primarily handled by the **frontend**. The backend provides data to the frontend in the form of JSON responses through REST APIs, which the frontend uses to render dynamic views.

Component	Description
Controller	The Controller layer is responsible for **handling HTTP requests** and **coordinating between the Model and View layers**. For example: • Receives API calls from the frontend. • Validates request data. • Invokes the appropriate service methods to process data. • Returns responses to the frontend. This ensures a clean separation of concerns.

Frontend-Backend Interaction

1. The **frontend** sends HTTP requests (e.g., GET, POST) to the backend for actions such as creating a trip, fetching expense reports, or uploading receipts.
2. The **backend controllers** handle the requests, process them using the **services**, and fetch or update data via the **repositories**.
3. The processed data is returned to the frontend as a structured JSON response, allowing the UI to update dynamically.
4. The frontend ensures data is displayed in an intuitive manner, reflecting the backend's responses in real-time.

This architecture ensures scalability, maintainability, and a clear separation of responsibilities between the frontend and backend components.

Database

Description of the database

The system is based on the use of a relational database implemented in PostgreSQL, where entities are modeled as tables, each with a unique name and a set of attributes. The decision to use a relational database is driven by the need for efficient data management, especially for handling business trips, expense reports, and user roles. The relational structure allows for easy modeling of real-world processes, enabling better organization, tracking, and reporting. The database ensures data security, as well as efficient access, storage, insertion, modification, and retrieval of data for further processing. The database of this application includes the following entities:

- **Countries**
- **Company**
- **Departments**
- **Users**
- **User_Roles**
- **Roles**
- **Trips**
- **Trip_Statuses**
- **Expense_Categories**

- **Expense_Subcategories**
- **Expense_Reports**
- **Expense_Report_Items**
- **Receipts**

Table Descriptions

Countries Table

Field	Type	Description
`code`	Primary Key	Unique country code.
`eur_daily_wage`	Decimal	Daily wage rate in euros.
`continent`	VARCHAR	Name of the continent.
`name`	VARCHAR	Country name.

Roles Table

Field	Type	Description
`id`	Primary Key	Unique identifier for each role.
`name`	VARCHAR	Name of the role (e.g., Employee, Accountant).

Departments Table

Field	Type	Description
`id`	Primary Key	Unique identifier for each department.
`name`	VARCHAR	Name of the department.

Users Table

Field	Type	Description
`id`	Primary Key	Unique identifier for each user.
`department_id`	Foreign Key	Links to **Departments** to associate users with departments.
`has_registered`	Boolean	Indicates if the user has completed registration.
`email`	VARCHAR	User's email for authentication.
`password_hash`	VARCHAR	Encrypted password.
`first_name`	VARCHAR	User’s first name.
`last_name`	VARCHAR	User’s last name.

Field	Type	Description
`iban`	VARCHAR	Bank account information for processing payments.
`provider`	VARCHAR	Third-party provider used for registration.
`provider_id`	VARCHAR	ID from the third-party provider.
`registration_hash`	VARCHAR	Hash used for registration status.

User_Roles Table

Field	Type	Description
`user_id`	Foreign Key	Links to **Users**.
`role_id`	Foreign Key	Links to **Roles**.

Company Table

Field	Type	Description
`id`	Primary Key	Unique identifier.
`eur_cost_per_km`	Decimal	Cost per kilometer in euros.
`location_coord_lat`	Decimal	Latitude coordinate of the company location.
`location_coord_lon`	Decimal	Longitude coordinate of the company location.
`address`	VARCHAR	Company street address.
`city`	VARCHAR	Company city.
`country_code`	VARCHAR	Company's home country code.
`iban`	VARCHAR	Company IBAN for transactions.

Trips Table

Field	Type	Description
`id`	Primary Key	Unique identifier for each trip.
`user_id`	Foreign Key	Links to **Users**.
`coordinates_lat`	Decimal	Latitude of the trip destination.
`coordinates_lon`	Decimal	Longitude of the trip destination.
`created_at`	Timestamp	Timestamp for when the trip was created.
`datetime_from`	DateTime	Start date and time of the trip.
`datetime_to`	DateTime	End date and time of the trip.

Field	Type	Description
`city`	VARCHAR	Destination city.
`country_code`	VARCHAR	Destination country code.
`reason`	VARCHAR	Reason for the trip.
`request_number`	VARCHAR	Unique request number.

Trip_Statuses Table

Field	Type	Description
`id`	Primary Key	Unique identifier for each status change entry.
`trip_id`	Foreign Key	Links to **Trips**.
`created_at`	Timestamp	Timestamp of the status change.
`message`	VARCHAR	Status message (e.g., Approved, Rejected).
`status`	VARCHAR	Current status.

Expense_Categories Table

Field	Type	Description
`id`	Primary Key	Unique identifier for each category.
`name`	VARCHAR	Name of the expense category.

Expense_Subcategories Table

Field	Type	Description
`id`	Primary Key	Unique identifier.
`expense_category_id`	Foreign Key	Links to **Expense_Categories**.
`name`	VARCHAR	Name of the subcategory.

Expense_Reports Table

Field	Type	Description
`id`	Primary Key	Unique identifier for each report.
`trip_id`	Foreign Key	Links to **Trips**.
`eur_total_cost`	Decimal	Total cost of expenses in euros.
`created_at`	Timestamp	Timestamp for when the report was created.

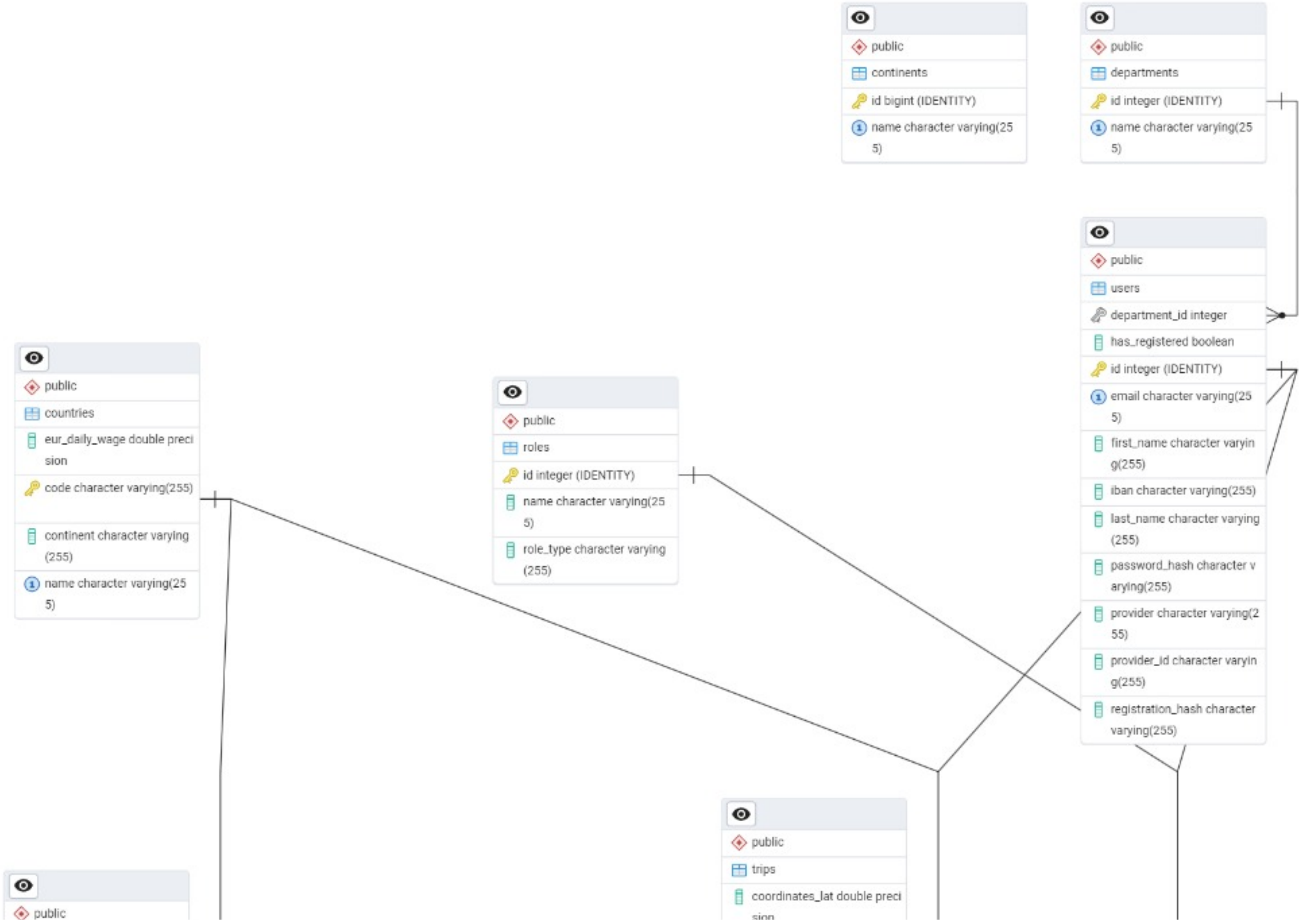
Expense_Report_Items Table

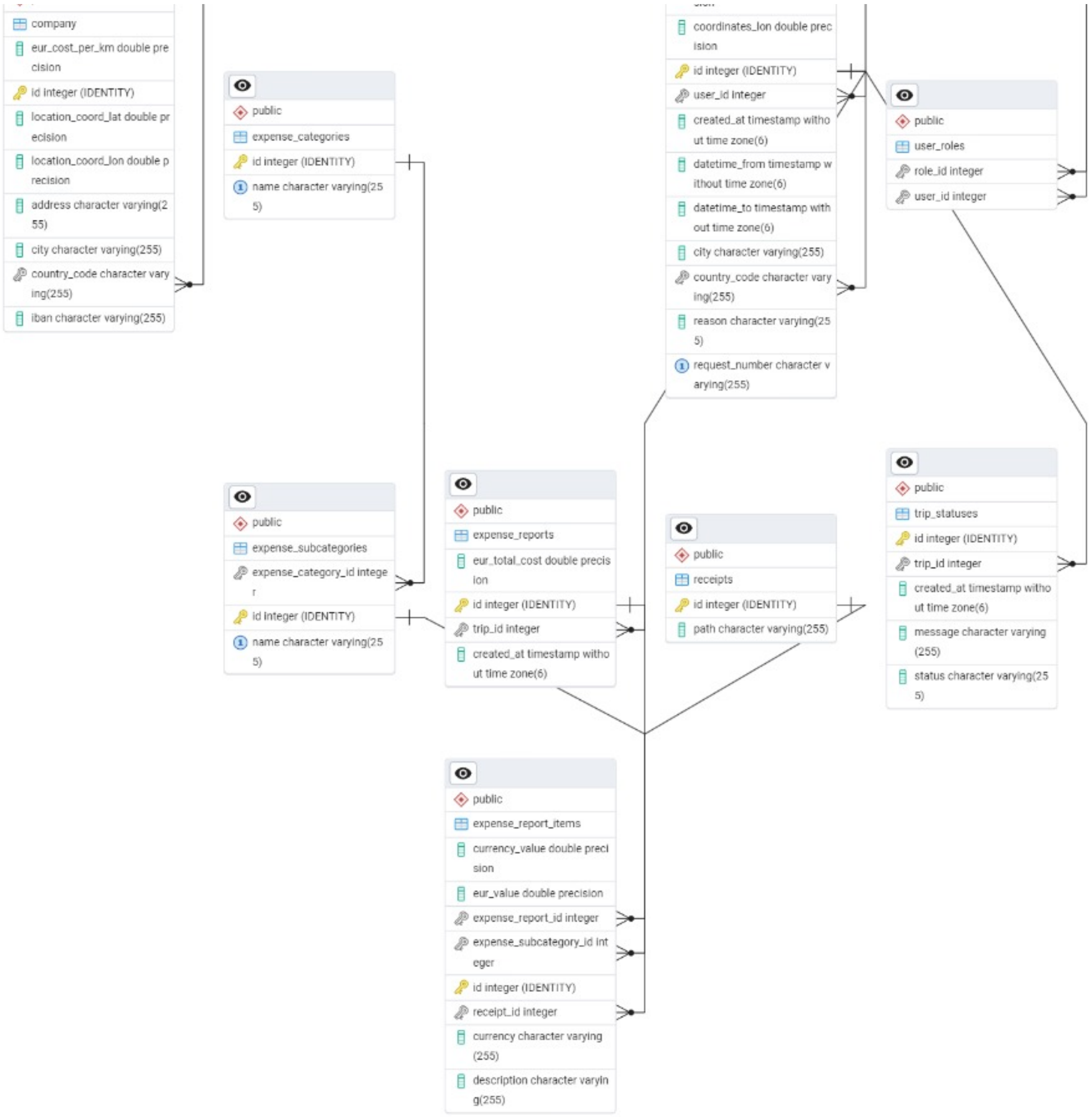
Field	Type	Description
`id`	Primary Key	Unique identifier for each expense item.
`expense_report_id`	Foreign Key	Links to **Expense_Reports**.
`currency_value`	Decimal	Value of the expense in the original currency.
`eur_value`	Decimal	Converted value in euros.
`currency`	VARCHAR	Original currency of the expense.
`expense_subcategory_id`	Foreign Key	Links to **Expense_Subcategories**.
`receipt_id`	Foreign Key	Links to **Receipts**.
`description`	VARCHAR	Description of the expense item.

Receipts Table

Field	Type	Description
`id`	Primary Key	Unique identifier for each receipt.
`path`	VARCHAR	File path or location of the uploaded receipt in PDF format.

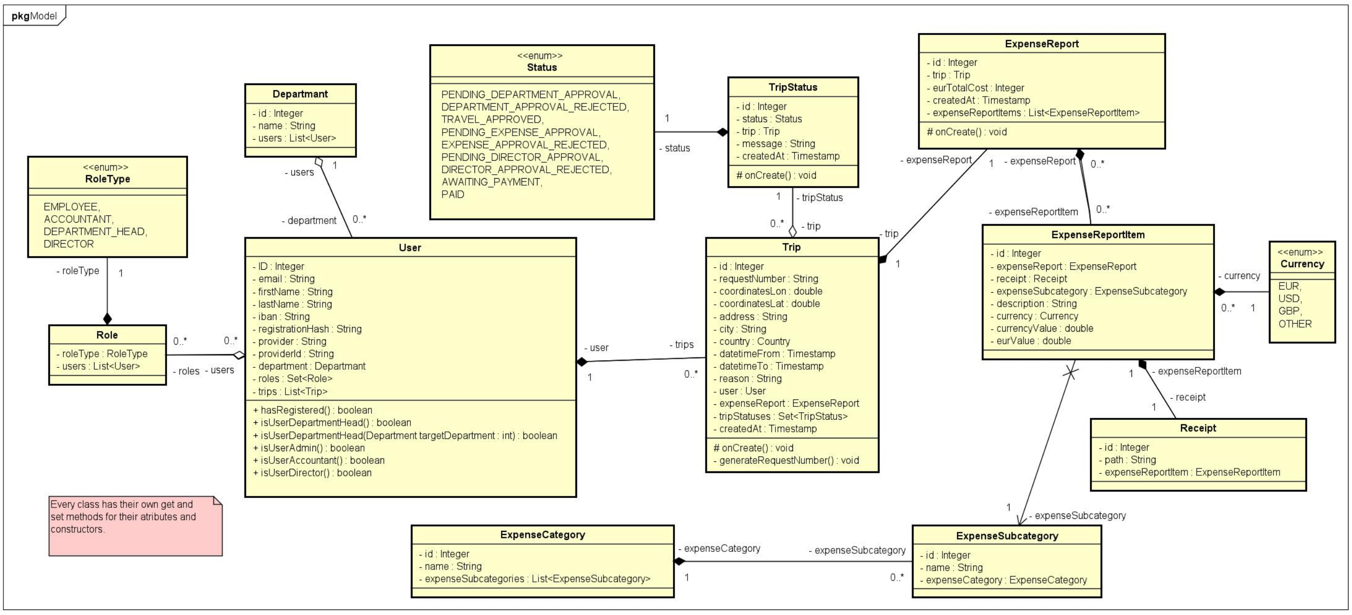
Database Diagram


public
continents
id bigint (IDENTITY)
name character varying(255)
public
departments
id integer (IDENTITY)
name character varying(255)
public
users
department_id integer
has_registered boolean
id integer (IDENTITY)
email character varying(255)
first_name character varying(255)
iban character varying(255)
last_name character varying (255)
password_hash character varying(255)
provider character varying(255)
provider_id character varying(255)
registration_hash character varying(255)
public
countries
eur_daily_wage double precision
code character varying(255)
continent character varying (255)
name character varying(255)
public
roles
id integer (IDENTITY)
name character varying(255)
role_type character varying (255)
public
trips
coordinates_lat double precision
public

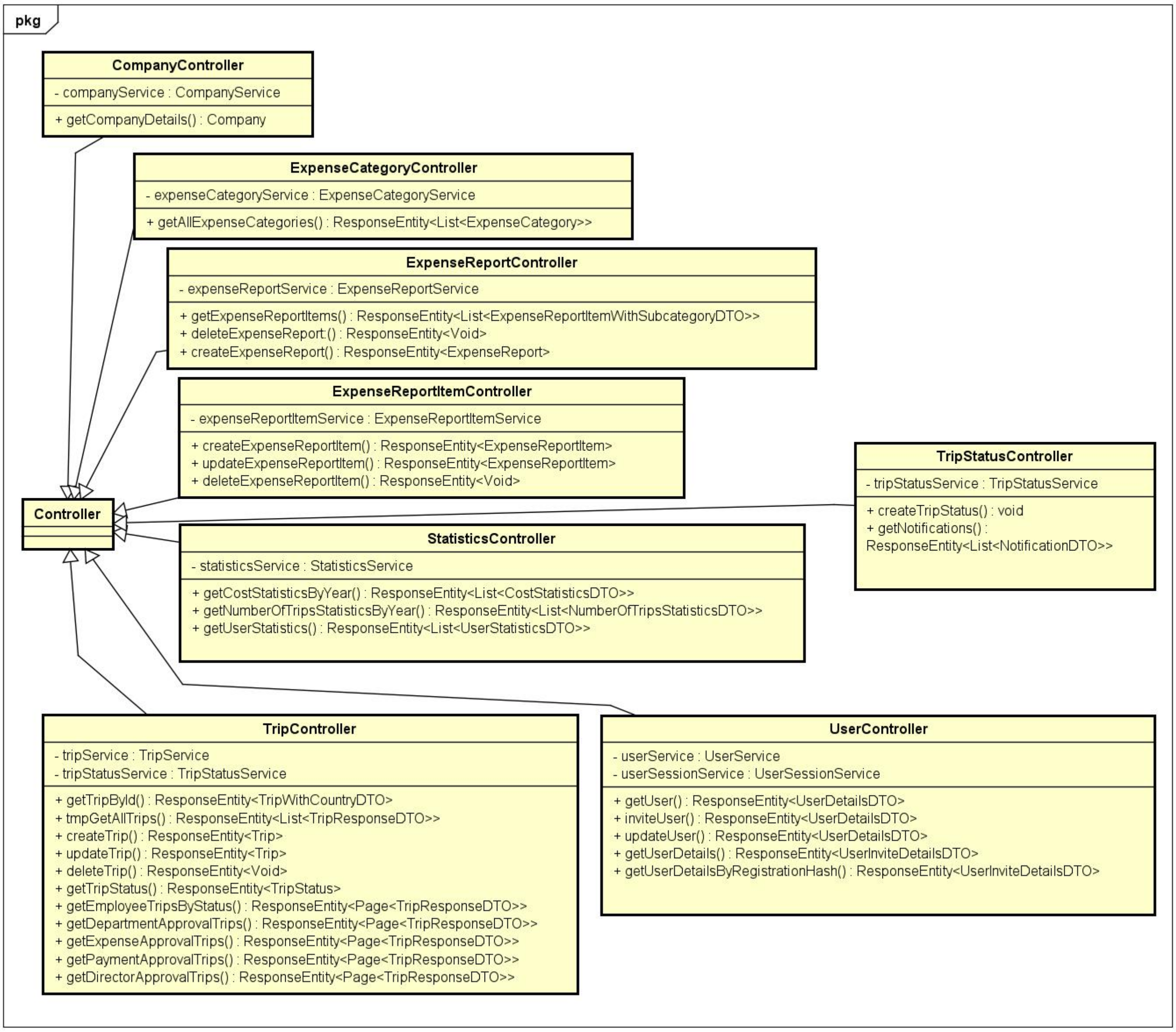

company
eur_cost_per_km double precision
id integer (IDENTITY)
location_coord_lat double precision
location_coord_lon double precision
address character varying(255)
city character varying(255)
country_code character varying(255)
iban character varying(255)
public
expense_categories
id integer (IDENTITY)
name character varying(255)
coordinates_lon double precision
id integer (IDENTITY)
user_id integer
created_at timestamp without time zone(6)
datetime_from timestamp without time zone(6)
datetime_to timestamp without time zone(6)
city character varying(255)
country_code character varying(255)
reason character varying(255)
request_number character varying(255)
public
user_roles
role_id integer
user_id integer
public
expense_subcategories
expense_category_id integer
id integer (IDENTITY)
name character varying(255)
public
expense_reports
eur_total_cost double precision
id integer (IDENTITY)
trip_id integer
created_at timestamp without time zone(6)
public
receipts
id integer (IDENTITY)
path character varying(255)
public
trip_statuses
id integer (IDENTITY)
trip_id integer
created_at timestamp without time zone(6)
message character varying (255)
status character varying(255)
public
expense_report_items
currency_value double precision
eur_value double precision
expense_report_id integer
expense_subcategory_id integer
id integer (IDENTITY)
receipt_id integer
currency character varying (255)
description character varying(255)

Class Diagram

Model Class Diagram

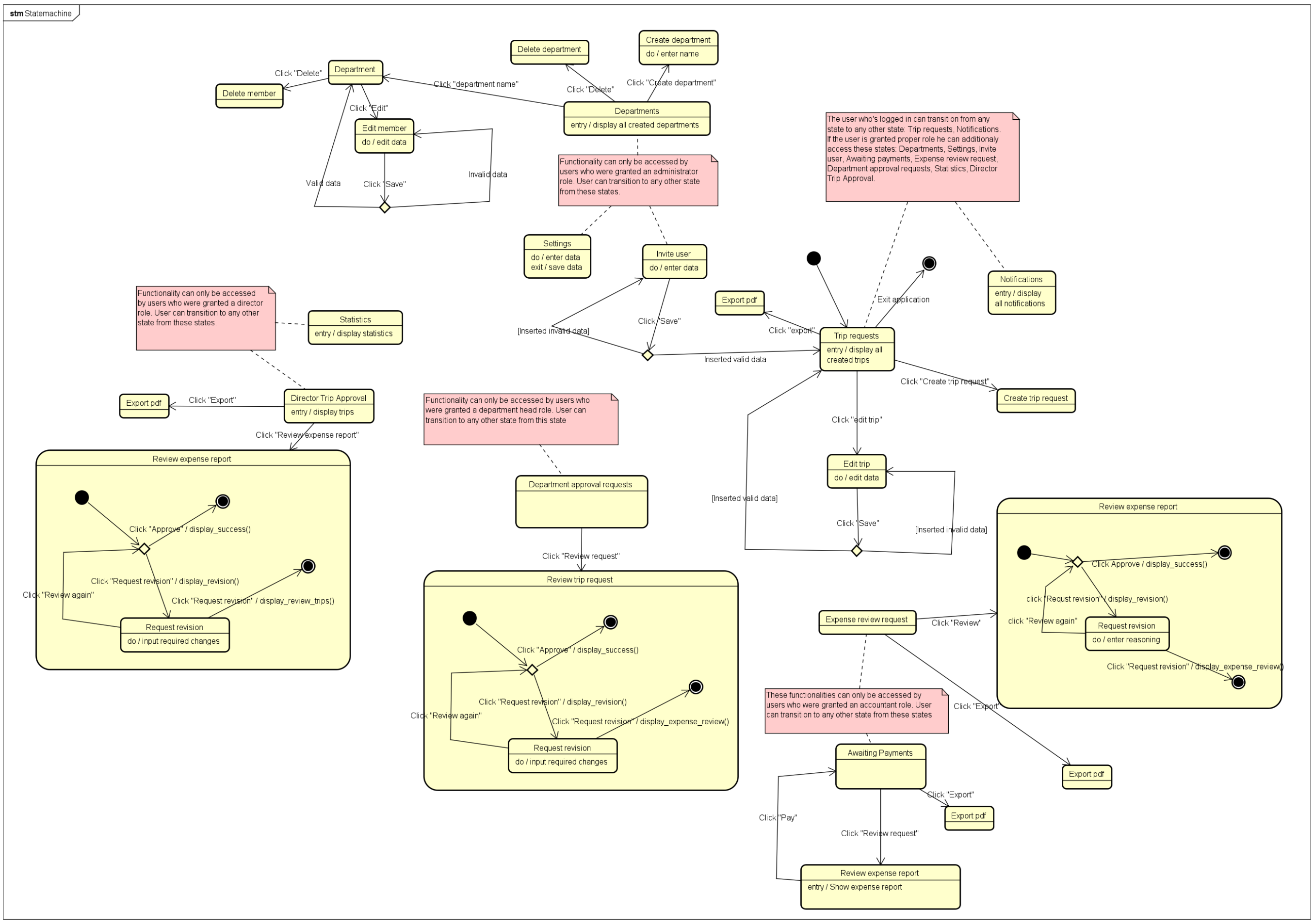


Controller Diagram


pkg
CompanyController
- companyService : CompanyService
+ getCompanyDetails() : Company
ExpenseCategoryController
- expenseCategoryService : ExpenseCategoryService
+ getAllExpenseCategories() : ResponseEntity<List<ExpenseCategory>>
ExpenseReportController
- expenseReportService : ExpenseReportService
+ getExpenseReportItems() : ResponseEntity<List<ExpenseReportItemWithSubcategoryDTO>>
+ deleteExpenseReport() : ResponseEntity<Void>
+ createExpenseReport() : ResponseEntity<ExpenseReport>
ExpenseReportItemController
- expenseReportItemService : ExpenseReportItemService
+ createExpenseReportItem() : ResponseEntity<ExpenseReportItem>
+ updateExpenseReportItem() : ResponseEntity<ExpenseReportItem>
+ deleteExpenseReportItem() : ResponseEntity<Void>
Controller
TripStatusController
- tripStatusService : TripStatusService
+ createTripStatus() : void
+ getNotifications() : ResponseEntity<List<NotificationDTO>>
StatisticsController
- statisticsService : StatisticsService
+ getCostStatisticsByYear() : ResponseEntity<List<CostStatisticsDTO>>
+ getNumberOfTripsStatisticsByYear() : ResponseEntity<List<NumberOfTripsStatisticsDTO>>
+ getUserStatistics() : ResponseEntity<List<UserStatisticsDTO>>
TripController
- tripService : TripService
- tripStatusService : TripStatusService
+ getTripById() : ResponseEntity<TripWithCountryDTO>
+ tmpGetAllTrips() : ResponseEntity<List<TripResponseDTO>>
+ createTrip() : ResponseEntity<Trip>
+ updateTrip() : ResponseEntity<Trip>
+ deleteTrip() : ResponseEntity<Void>
+ getTripStatus() : ResponseEntity<TripStatus>
+ getEmployeeTripsByStatus() : ResponseEntity<Page<TripResponseDTO>>
+ getDepartmentApprovalTrips() : ResponseEntity<Page<TripResponseDTO>>
+ getExpenseApprovalTrips() : ResponseEntity<Page<TripResponseDTO>>
+ getPaymentApprovalTrips() : ResponseEntity<Page<TripResponseDTO>>
+ getDirectorApprovalTrips() : ResponseEntity<Page<TripResponseDTO>>
UserController
- userService : UserService
- userSessionService : UserSessionService
+ getUser() : ResponseEntity<UserDetailsDTO>
+ inviteUser() : ResponseEntity<UserDetailsDTO>
+ updateUser() : ResponseEntity<UserDetailsDTO>
+ getUserDetails() : ResponseEntity<UserInviteDetailsDTO>
+ getUserDetailsByRegistrationHash() : ResponseEntity<UserInviteDetailsDTO>

Dynamic application behaviour

Statechart



The user who's logged in can transition from any state to any other state: Trip requests, Notifications. If the user is granted proper role he can additionaly access these states: Departments, Settings, Invite user, Awaiting payments, Expense review request, Department approval requests, Statistics, Director Trip Approval.

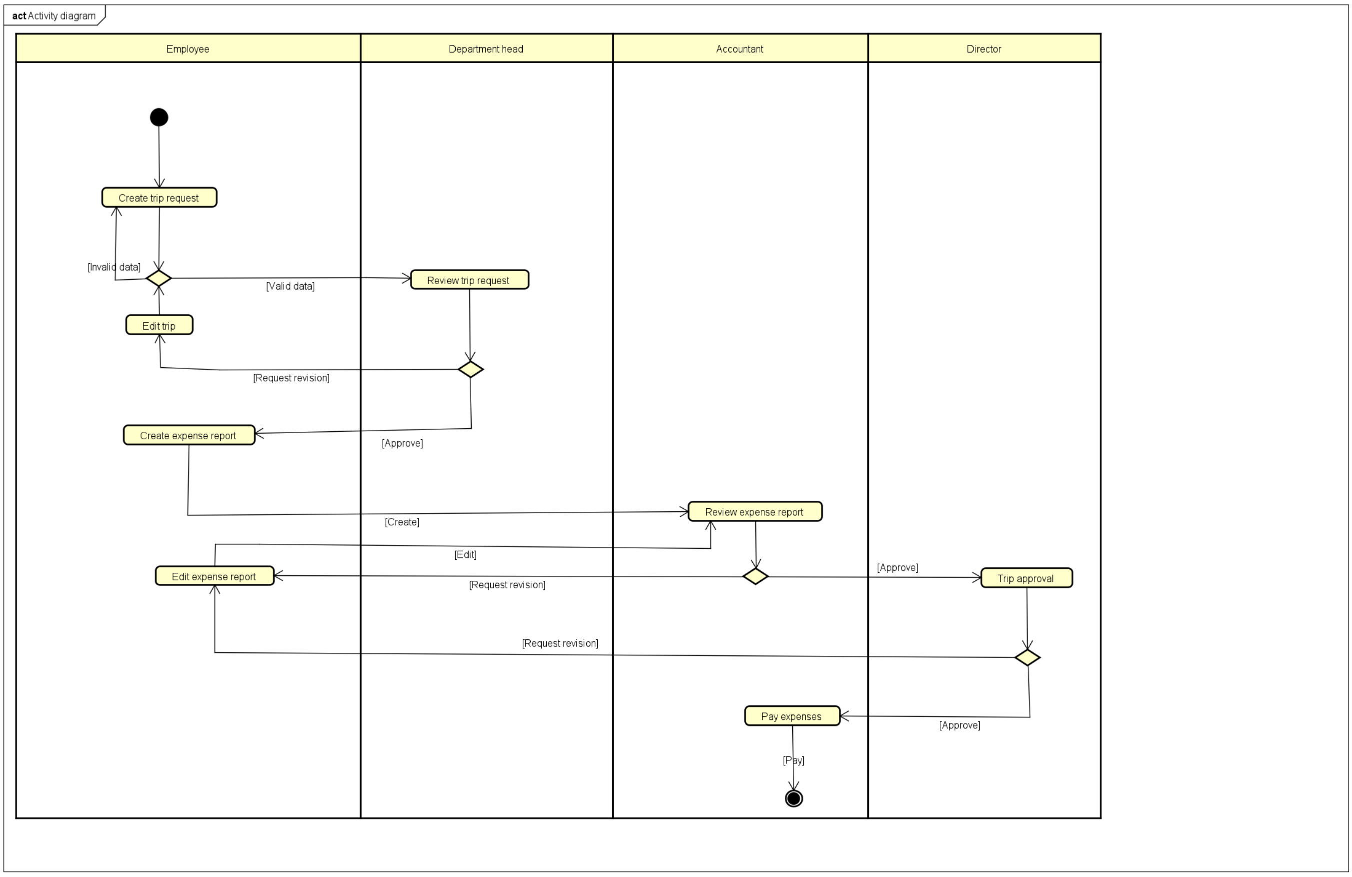
Functionality can only be accessed by users who were granted an administrator role. User can transition to any other state from these states.

Functionality can only be accessed by users who were granted a director role. User can transition to any other state from these states.

Functionality can only be accessed by users who were granted a department head role. User can transition to any other state from this state

These functionalities can only be accessed by users who were granted an accountant role. User can transition to any other state from these states

Activity Diagram


act Activity diagram
Employee
Department head
Accountant
Director
Create trip request
[Invalid data]
[Valid data]
Review trip request
Edit trip
[Request revision]
Create expense report
[Approve]
Review expense report
[Create]
[Edit]
Edit expense report
[Request revision]
[Approve]
Trip approval
[Request revision]
Pay expenses
[Approve]
[Pay]