TUNIS BUSINESS SCHOOL
UNIVERSITY OF TUNIS

Summer Internship Report

Prod. Incidents Management System Project

InciManage

Full-stuck Development & Business Intelligence

authored by

Amal Jawahdou

supervised by

Eng. Marwa Dridi

directed by

Eng. Malek Ben Kamel

22 October, 2024

Declaration of Academic Ethics

I, Amal Jawahdou, declare that this written report, submitted for the summer internship project, is my own work that reflects my own ideas in my own way, and that any external ideas or words included in this paper have been properly cited and acknowledged respectively.

I affirm that I have adhered to respect all the academic ethics of honesty and integrity. And, I have ensured that all the mentioned information in this report is free from misguidance, fallacies, and misrepresentation.

I am fully cognizant that any violation of these ethical principles may lead to serious disciplinary actions, and I am committed to accept the consequence of any breaches.

Acknowledgement

I would like to thank my supervisor, my manager and every individual inside and outside the enterprise that supported me during the internship and contributed to the success of this project. Without their help and guide it was almost impossible to fulfill what I've fulfilled.

I want to express my deepest gratitude for all the team members, the IT stuff and the colleagues for my making this experience unforgettable.

An experience full of special moments and skilled persons from whom I've learned a lot in both the professional live and the daily one.

Your advice and support will always serve as a guiding light and a source of motivation for me. The lessons I have learned and the encouragement I have received will drive me to continuously seek improvement and strive for the best possible results in my future endeavors.

Abstract

This report will present details about my summer internship at Sofrecom Tunisie, it will describe the valuable experience I have got at this enterprise in which I have strengthen not only my hard skills on coding and using various programming languages and tools to built a useful web application, but also my soft skills.After 2 months of perseverance and dedication leveraging my professional experience was for sure a goal and has been reached.

Incidents such as bugs and run time errors has been always frustrating, specially occurring during production phase when it become more urgent to manage and resolve those issues. Yet, having them placed all in one platform will surely help managing the incidents more quickly and efficiently. In addition to that the need to visualize and analyze all the incidents in one dashboard has become also crucial that it change radically the process of incident management and make more smooth, efficient and fast which will leverage the resolving time laps and accelerate the production phase and give the producers the ability to focus more on improving their product services and serving better user experience to their customers.

InciManage is the web application I have built to offer those services via the integration of a spring-boot REST-full web service and nebular angular library for the front-end.

Executive Summary

The main purpose of this internship was to develop a web application utilizing various tools and frameworks, including Angular, Spring Boot, PostgreSQL, and Nebular, to create a comprehensive incident management system. The primary objective was to streamline the process of tracking, reporting, and resolving incidents within an organization.

Throughout the development, several key goals were achieved:

- **Frontend Implementation**: The application's frontend was developed using Angular and integrated with Nebular for UI components, enabling a user-friendly interface. Features such as authentication (with JWT), dynamic role-based menus, and form handling were implemented for optimal user interaction.
- **Backend Development**: The backend was built using Spring Boot, with the creation of RESTful APIs for CRUD operations on incidents, users, and applications. Additionally, JWT authentication was added to secure the web service.
- **Database Integration**: PostgreSQL was used for database management, with tables representing incidents, users, roles, and applications. Complex relationships between these entities were managed using JPA annotations.
- **Incident Management Logic**: The core functionality of the application includes creating, updating, and resolving incidents, with fields for severity, resolution time, and assigned users. Charts and summaries were added to display incident statistics effectively.
- **Business Intelligence and Dashboarding**: To further enhance the system, Business Intelligence (BI) where ETL (Extract, Transform, Load) processes were implemented enabled the creation of dynamic dashboards that provide insights into incident trends, performance metrics, and other key data points, allowing for data-driven decision-making.

The success of this internship lies in the fact that the built system has the potential to enhance productivity and incident resolution efficiency, offering a scalable solution for real-world production environments.

Contents

Chapter 1

Introduction

This chapter provides a general overview of the internship's context, the organization where the work was conducted, and the problem the project aims to solve. The objectives of the internship are also outlined, including the mission and vision for the developed incidents management web application.

1.1 Organization

This internship was conducted at **Sofrecom Tunisie**[8], a company specializing in consulting and engineering services. Sofrecom provides innovative solutions for telecom operators and other businesses, focusing on digital transformation, IT systems, and business process improvements.

1.2 Context

In the context of IT operations, incidents are inevitable. Effective management of these incidents is crucial for ensuring continuous service delivery and minimizing downtime. The focus of this project is to build a robust incidents management system tailored to streamline the reporting, assignment, and resolution of IT-related issues. The application is expected to cater to the needs of various roles within the organization, from end-users reporting issues to administrators overseeing incident resolution.

1.3 Existing Solutions

Currently, various tools and software solutions exist for managing IT incidents, such as JIRA[3], ServiceNow[7], and Zendesk[9]. These platforms offer features like ticketing systems, incident tracking, and automation for incident resolution. However, many of these tools come with complexities that make them less user-friendly for non-technical

users. Moreover, they often require significant customization to meet the specific needs of different organizations, which can result in high implementation costs and inefficiencies.

1.4 Problem Statement and Motivation

The main problem to solve is the lack of an efficient, scalable, and user-friendly system for managing IT incidents. Existing tools either lack the functionality needed for effective communication between different roles or are not intuitive enough for regular use. This results in delayed resolutions, decreased productivity, and a lack of accountability. The motivation behind this project is to create a system that not only addresses these issues but also enhances organizational efficiency by providing a smooth workflow for incident tracking and resolution.

1.5 Objectives

The general objective of this internship is to build a complete, efficient, and user-friendly incidents management web application with the following mission and vision:

Mission:

The created system aims to assist in managing IT incidents by providing an intuitive platform for users to report and assign incidents easily. It ensures that every incident is tracked from reporting to resolution, promoting transparency and accountability.

Vision:

The vision is to provide an efficient and scalable web application for managing incidents during the production phase. By ensuring robustness and scalability, the system aims to enhance organizational productivity and improve the accountability of incident resolution processes.

Chapter 2

Back-end Development

This chapter covers the back-end development aspects of the project, focusing on database management, web service creation, and API endpoint testing. It provides an overview of the design, implementation, and testing processes for the back-end components of the application.

2.1 Database Management

Database management is crucial for ensuring data integrity, efficiency, and scalability in the application. This section provides a comprehensive overview of the database schema, design, and key performance metrics involved in managing the application's data. It includes various diagrams to visualize and understand the data structure and relationships.

2.1.1 Data Storage

Data storage is a critical component of the application, determining how and where data is stored. PostgreSQL has been chosen for this project due to its robustness, scalability, and support for complex queries and data integrity.

PostgreSQL Overview

PostgreSQL[6] is an open-source relational database management system known for its advanced features and performance.

It supports:

- ACID compliance for transaction safety.
- Advanced indexing techniques to speed up queries.
- Extensibility and support for custom data types and functions.

Data Storage Implementation

The application uses PostgreSQL to store all data related to incidents, users, and applications. This choice ensures that the system can handle large volumes of data efficiently while maintaining high performance.

2.1.2 Database Design

The database design is fundamental in defining how data is stored, related, and accessed. This subsection outlines the database schema, including tables, fields, and relationships between entities.

The Entity-Relationship Diagram (ERD) is used to visualize the data structure, illustrating how different data entities interact and relate to one another.

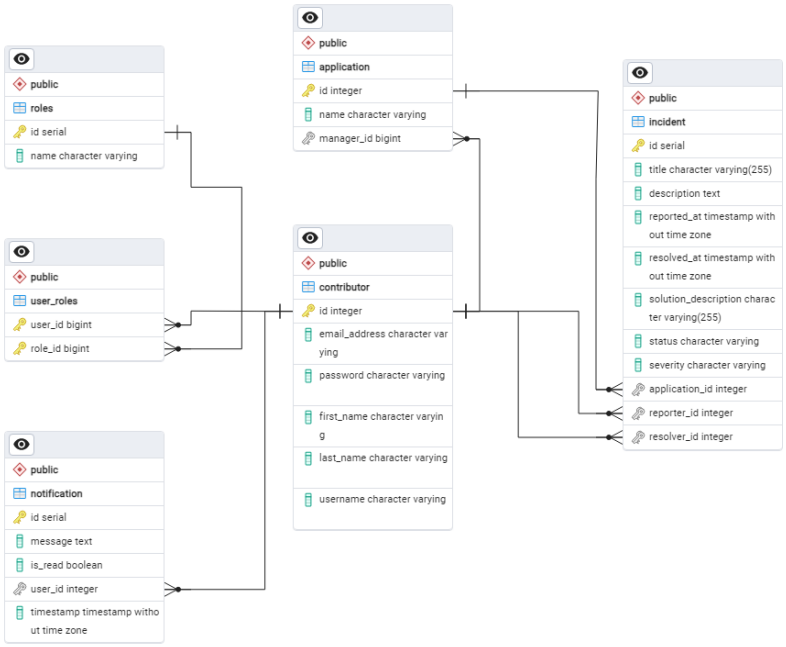

Figure 2.1: Entity-Relationship Diagram (ERD) of the Database

Tables and Fields

The database consists of several key tables, each with specific fields. For instance:

- **Incident** table: Contains fields such as `id`, `title`, `description`, `status`, `reportedBy`, and `resolvedBy`.
- **Contributor** table: Contains fields such as `id`, `username`, `email`, and `role`.
- **Application** table: Includes fields like `id`, `name`, and `managerId`.

Relationships

The relationships between these tables are defined to ensure data integrity. For example:

- `Incident.reportedBy` and `Incident.resolvedBy` are foreign keys referring to the `User` table.

- The `Application` table is related to the `Incident` table through a foreign key `applicationId`.

2.1.3 Class Diagram

The class diagram provides a detailed view of the application's data model from an object-oriented perspective. It depicts the classes, their attributes, methods, and the relationships between them, helping to understand the back-end system's design.

Classes and Relationships

The class diagram includes:

- **Incident** class: Attributes include `id`, `title`, `description`, `status`, `reportedBy`, and `resolvedBy`.
- **User** class: Attributes include `id`, `username`, `email`, and `role`.
- **Application** class: Attributes include `id`, `name`, and `managerId`.

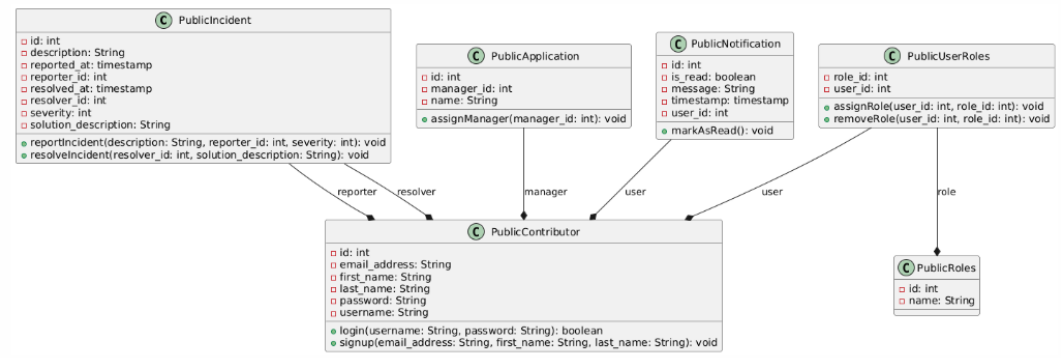

Figure 2.2: Class Diagram of the Application

2.1.4 Use Case Diagram

The use case diagram illustrates the interactions between users and the system, capturing various scenarios of user interaction. It helps in identifying the functional requirements and user roles within the application.

Actors and Use Cases

The use case diagram includes:

- **Actors**: User, Admin, Manager
- **Use Cases**: Report Incident, View Dashboard, Assign Incident, Resolve Incident, View All Incidents, etc.

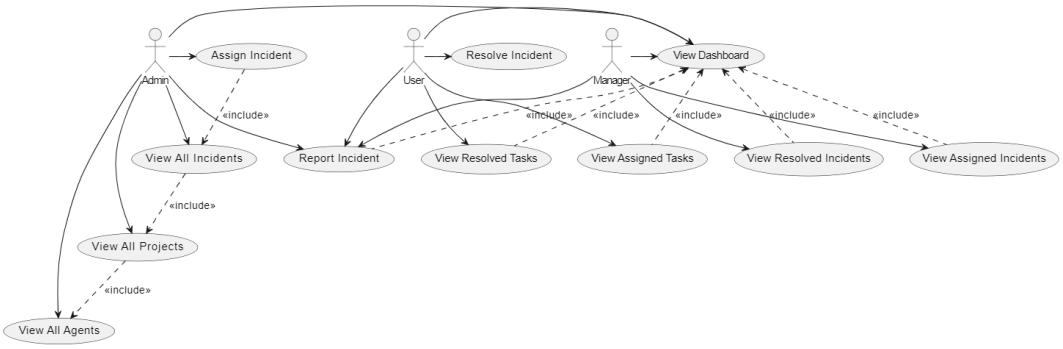

Figure 2.3: Use Case Diagram of the Application

2.1.5 Key Performance Indicators

Key Performance Indicators (KPIs) are metrics used to evaluate the performance and efficiency of the database and its queries. Monitoring KPIs helps in maintaining the system's health and performance.

Important KPIs

Some relevant KPIs include:

- **Query Execution Time**: Measures how long it takes to execute a query.
- **Database Response Time**: Time taken by the database to respond to requests.
- **System Throughput**: Number of transactions processed per unit of time.

Performance Monitoring Tools

Tools such as PostgreSQL's built-in performance monitoring and third-party tools like pgAdmin can be used to track these KPIs.

2.2 Web Service Creation

Web service creation involves developing the back-end logic and APIs that interact with the front-end application. This section discusses the implementation of RESTful services using Spring Boot, including service design, security measures, and the tools and technologies used.

2.2.1 Service Design

The web services were designed following RESTful principles to ensure scalability and maintainability. The services are structured to interact with the database through JPA repositories, and the business logic is handled by service classes.

Key design patterns used include:

- **Controller-Service-Repository Pattern:** This pattern separates concerns by dividing the application into controllers (handling HTTP requests), services (containing business logic), and repositories (interacting with the database).

The services communicate with the PostgreSQL database to manage incidents, applications, and user data.

2.2.2 Security Measures

Security is implemented using Spring Security to protect sensitive information and ensure only authorized users can access specific resources. Key security measures include:

- **Authentication:** JSON Web Tokens (JWT) are used for secure user authentication.
- **Authorization:** Role-based access control is implemented to restrict access based on user roles.
- **Data Encryption:** Sensitive data is encrypted both at rest and in transit.

2.2.3 Dependencies

The project uses the following dependencies:

- **Spring Boot Starter:**
  - `spring-boot-starter`: Core dependency for Spring Boot applications.
  - `spring-boot-starter-web`: Provides web development features.
  - `spring-boot-starter-data-jpa`: For working with JPA and Hibernate.
- **Security:**
  - `spring-boot-starter-security`: For security features including authentication and authorization.
  - `spring-security-config`: Configuration support for Spring Security.
  - `jjwt-api`, `jjwt-impl`, `jjwt-jackson`: JWT libraries for token management.
- **Database:**
  - `postgresql`: PostgreSQL database driver.
- **Testing and Development:**
  - `spring-boot-starter-test`: Testing support for Spring Boot applications.
  - `spring-boot-devtools`: Development tools for improved productivity.
- **Others:**

- `springfox-boot-starter` and `springfox-swagger-ui`: For Swagger API documentation.
- `springdoc-openapi-ui`: OpenAPI documentation support.
- `lombok`: Reduces boilerplate code.
- `jackson-databind`: For JSON serialization and deserialization.

2.3 API Endpoints Testing

Testing the API endpoints is essential to ensure that the web services function correctly and fulfill the application's requirements. This section outlines the process used to test the endpoints, focusing on the practical approach taken during the development.

2.3.1 Testing Approach

The API endpoints were tested using a tool called the BRUNO testing tool. Bruno is a Fast and Git-Friendly Opensource API client, aimed at revolutionizing the status quo represented by Postman, Insomnia and similar tools out there[2]. This tool was used to verify that each endpoint performs as expected, returning the correct responses and handling errors gracefully.

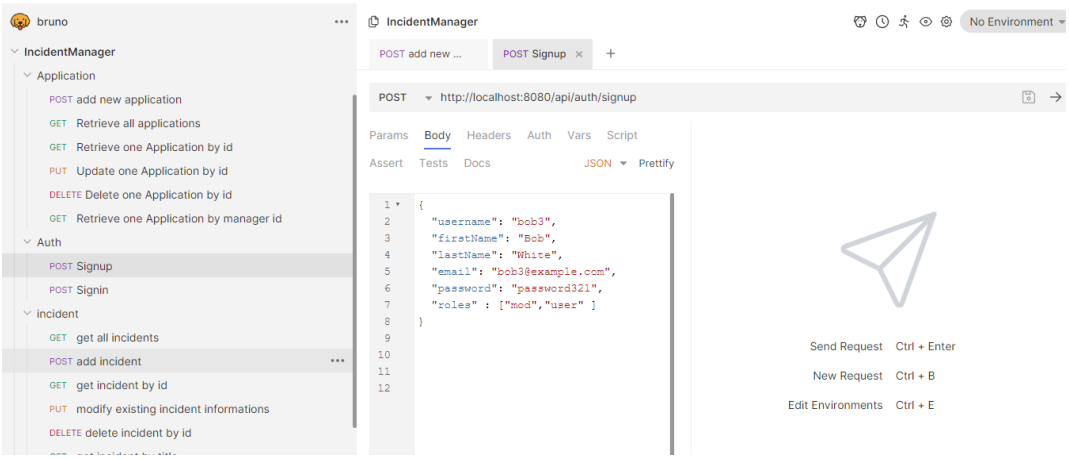

Figure 2.4: API Testing

2.4 API Endpoints

This section describes the API endpoints created for managing applications, incidents, users, authentication, and notifications. Each endpoint is tested using the BRUNO testing tool.

2.4.1 Application Endpoints

- **POST /api/saveApp** Adds a new application.

  Request Body: `name: String (required), managerId: Integer (required)`

  Response: `Application` object with `id`, `name`, `managerId`.

- **GET /api/findAllApplications** Retrieves all applications.

  Response: `List<Application>` with fields `id`, `name`, `managerId`.

- **GET /api/findAppById**

  Retrieves an application by ID.

  Request Parameter: `id: Integer (required)`

  Response: `Application` object with fields `id`, `name`, `managerId`.

- **GET /api/findAppByManagerId**

  Retrieves applications by manager ID.

  Request Parameter: `id: Integer (required)`

  Response: `List<Application>` with fields `id`, `name`, `managerId`.

- **PUT /api/updateApplication/id**

  Updates an application by ID.

  Request Parameters: `id: Integer (required)`

  Request Body: `name: String (optional), managerId: Integer (optional)`

  Response: `Application` object with updated fields.

- **DELETE /api/deleteApplication**

  Deletes an application by ID.

  Request Parameter: `id: Integer (required)`

  Response: `MessageResponse` with success message.

2.4.2 Authentication Endpoints

- **POST /api/auth/signin** Authenticates a user using their email and password.

  Request Body: `email: String (required), password: String (required)`

  Response: `JwtResponse` - Includes `jwt` token, `userId`, `username`, `email`, and `roles`.

- **POST /api/auth/signup** Registers a new user in the system.

  Request Body: `username: String (required), email: String (required),`

  `password: String (required), firstName: String (optional),`

`lastName: String (optional), role: Set<String> (optional)`

Response: `MessageResponse` - Success message indicating user registration status.

2.4.3 Incident Endpoints

- **POST /api/saveIncident** Adds a new incident.

  Request Body: `Incident` object.

  Response: `Incident` object.

- **GET /api/findAllIncidents** Retrieves all incidents.

  Response: `List<Incident>`.

- **GET /api/findContributorsByIncidentId** Retrieves contributors related to a specific incident.

  Request Parameter: `id: Integer (required)`

  Response: `List<Object[]>` - Contributors.

- **GET /api/findIncidentById** Retrieves an incident by its ID.

  Request Parameter: `id: Integer (required)`

  Response: `Incident` object.

- **GET /api/findIncidentByResolverId** Retrieves incidents assigned to a resolver.

  Request Parameter: `resolverId: Integer (required)`

  Response: `List<Incident>`.

- **GET /api/findIncidentByTitle** Retrieves incidents by title.

  Request Parameter: `title: String (required)`

  Response: `List<Incident>`.

- **GET /api/findIncidentByAppId** Retrieves incidents associated with a specific application ID.

  Request Parameter: $\texttt{application}_id : Integer(required)$

  `Response:List<Incident>`.

- **PUT /api/updateIncident/id** Updates an existing incident by its ID.

  Request Parameters: `id: Integer (required)`

  Request Body: `Incident` object.

  Response: `Incident` object.

- **DELETE /api/deleteIncident** Deletes an incident by its ID. **Request Parameter:**

  `id: Integer (required)`

  Response: Success message.

- **POST /api/predict** Predicts the incident type based on the description.

  Request Body: `description: String (required)`

  Response: Predicted incident type as `String`.

2.4.4 Notification Endpoints

- **GET /api/unread** Retrieves all unread notifications for a user.

  Request Parameter: `userId: Integer (required)`

  Response: `List<Notification>` - Unread notifications.

- **POST /api/mark-as-read** Marks a specific notification as read.

  Request Parameter: `notificationId: Integer (required)`

  Response: Success message.

- **POST /api/save** Saves a new notification.

  Request Body: `Notification` object.

  Response: Success message.

2.4.5 User Endpoints

- **POST /api/saveUser** Adds a new user.

  Request Body: `User` object.

  Response: The created `User` object.

- **GET /api/findAllUsers** Retrieves all users.

  Response: A list of `User` objects.

- **GET /api/findUserById** Retrieves a user by their ID.

  Request Parameter: `id: Integer (required)`

  Response: The `User` object with the given ID.

- **GET /api/findUserByEmail** Retrieves a user by their email.

  Request Parameter: `email: String (required)`

  Response: The `User` object with the given email.

- **PUT /api/updateUser/id** Updates an existing user by their ID.

  Request Parameter: `id: Integer (required)`

  Request Body: Updated `User` object.

  Response: The updated `User` object.

- **DELETE /api/deleteUser** Deletes a user by their ID.

  Request Parameter: `id: Integer (required)`

  Response: A success message.

- **GET /api/exists/userRole** Checks if a user has a specific role.

  Request Parameters: `userId: Integer (required)`, `roleId: Integer (required)`

  Response: A boolean indicating if the user has the role.

- **GET /api/UsersByRoleId** Retrieves all users with a specific role ID.

  Request Parameter: `roleId: Integer (required)`

  Response: A list of `User` objects with the specified role.

2.5 Results and Observations

All endpoints were tested to ensure they met the desired functionality. The testing process revealed that the endpoints correctly handled various scenarios, such as creating, retrieving, updating, and deleting records. Any issues encountered were resolved during testing.

Chapter 3

Front-end Development

This chapter details the development of the front-end for the Incident Management System using Angular, Nebular, and other technologies. It includes discussions on Uuser Interface design, component structure, and the integration with the back-end services.

3.1 Technologies and Frameworks

The front-end of the application was developed using modern web technologies and frameworks:

- **Angular:** A web framework that empowers developers to build fast, reliable applications.[1]
- **Nebular:** A customizable UI component library that provides a robust and responsive design system.[4]
- **TypeScript:** The primary language for developing Angular applications, providing static types and powerful tooling.
- **HTML/SCSS:** Used for structuring the UI and styling components.

Also, the front-end of the application was developed using the ngx-admin template, a popular Angular dashboard template designed for building complex and responsive admin interfaces. This template provided a structured layout, numerous UI components, and pre-built themes, which significantly accelerated the development process.

- **ngx-admin template:** The most popular admin dashboard based on Angular 9+ and Nebular with Eva Design System support. Free and Open Source for personal and commercial purposes. [5]

3.2 Component Structure and Layout

The application follows a modular component structure to ensure reusability and scalability.

3.2.1 Main Components

- **Login Component:** Displays the login form.
- **Sign Up Component:** Displays the sign up form.
- **Incident List Component:** Displays a list of incidents retrieved from the API.
- **Incident Form Component:** Allows end-users to create or edit incidents.
- **Dashboard Component:** Displays user-specific information, such as notifications and incident assignments.
- **Application Management Component:** Allow managers to manage the incidents linked to their applications.
- **Tasks Management Component:** Allow developers and testers to manage the incidents assigned to them.
- **Incident Management Component:** Allow admins to manage all incidents.
- **Notification Component:** Allow all the application users to receive notifications.

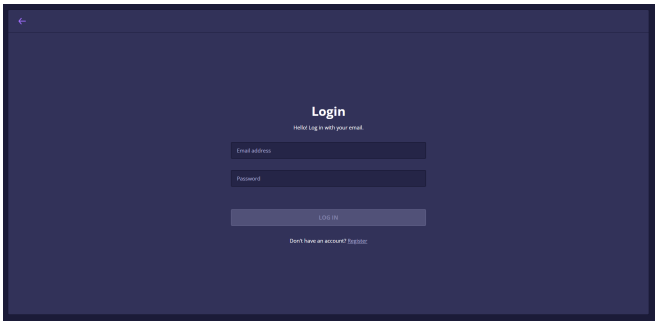

Figure 3.1: Login Form

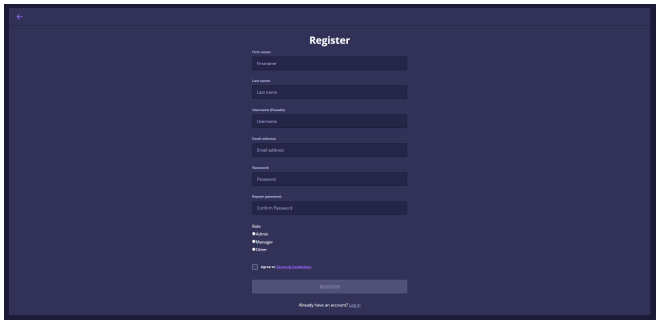

Figure 3.2: Registration Form

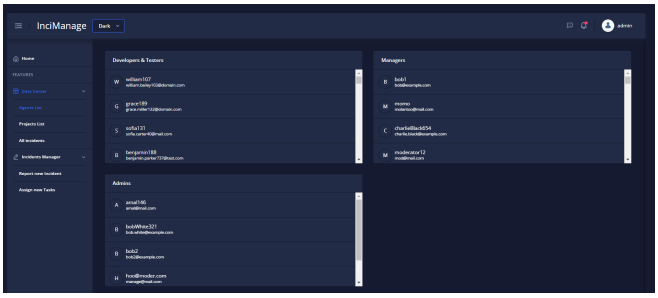

Figure 3.3: Agents List

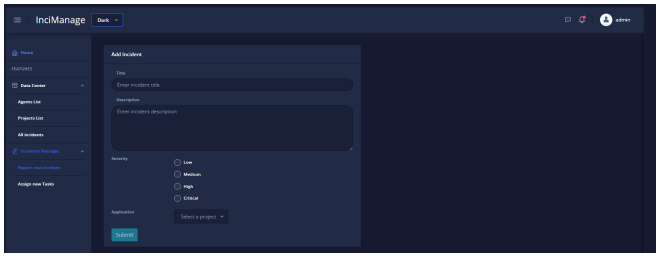

Figure 3.4: Incident Reporting Form

3.2.2 Routing

Angular's routing module was used to navigate between different views and components. The application includes routes such as:

- /auth/login – User login page.
- /auth/register – User sign up page.
- /pages/dashboard – Incidents management dashboard.
- /pages/forms/inputs – Displays the form to report new incident.
- /pages/tables/assigned-tasks – Displays the User's assigned tasks list page.

3.3 User Interface Design

The design of the application prioritizes a user-centric approach, with a focus on delivering an intuitive and visually appealing experience. By leveraging the Nebular design system and ngx-admin template, the interface provides seamless navigation, modern aesthetics, and responsive layouts across all devices.

Figure 3.5: Admin Dashboard in Dark Theme

3.3.1 Theme and Styling

The application features a polished, consistent design by utilizing Nebular's pre-built themes—both dark and light variants. These themes were customized with accent colors, typography adjustments, and consistent styling across the entire application.

Special attention was given to contrast and readability, ensuring that users could comfortably interact with the application for extended periods.

- **Primary Colors:** The primary color scheme includes shades of blue, green and gray, chosen to evoke professionalism and clarity. Accent colors, such as green and red, were used sparingly to indicate important statuses like successful operations or errors.
- **Typography:** Fonts were carefully selected to maintain clarity and hierarchy within the interface. A mix of bold headers and regular body text helps guide the user's eye through the application's various components.
- **Icons and Buttons:** Icons from Nebular's icon pack were used to reinforce actions and improve navigation clarity. Buttons were designed with clear call-to-actions and hover effects to provide immediate feedback to users.

To further enhance the visual experience, graphical elements such as shadows, borders, and smooth transitions were applied. These subtle design touches contribute to a more engaging and modern look, reinforcing the professional tone of the application.

3.3.2 Graphics and Visual Enhancements

To enrich the user interface, various graphical elements were integrated into the design:

- **Charts and Graphs:** Integrated with ngx-admin, the front-end incorporates visually appealing graphs to represent data such as incident severity, types, and resolutions. Using libraries like ECharts, these graphs dynamically update based on user input and offer smooth transitions, tooltips, and legends for enhanced interactivity.
- **Dashboard Layout:** The dashboard layout features visual cards that display key metrics and summaries of incidents, with the ability to quickly access detailed views. Icons and progress indicators add visual context to the numbers.
- **Notification Badges:** Notification badges were implemented using `nb-action` with a badge dot mode to alert users of pending notifications. These dots are color-coded to reflect urgency, adding to the application's usability.
- **Animated Transitions:** Subtle animations were applied to various components, such as dropdown menus, sidebar toggles, and modal windows, contributing to the smooth user experience.

3.3.3 Responsive Design

The application is fully responsive, ensuring that users can access it on any device, from large desktop monitors to smaller mobile screens. The flexible grid system provided by ngx-admin automatically adapts layouts to different screen sizes, while key elements such as navigation menus and content cards scale fluidly.

Graphics like charts, cards, and tables are designed to reflow based on screen resolution, making the application intuitive whether used on a desktop or on-the-go.

3.4 Integration with Back-end Services

The front-end communicates with the back-end through RESTful API endpoints, facilitating seamless interaction between the client and server. Angular services were created to manage HTTP requests, ensuring efficient data retrieval and submission while abstracting the complexity of API calls from the UI components. These services are responsible for handling data binding across components, enabling real-time updates and synchronization with the backend. Leveraging Angular's HttpClient module, the services ensure that requests are handled asynchronously, allowing the application to remain responsive even during intensive data operations. Additionally, proper error handling and observables were implemented to manage state changes and ensure a smooth user experience.

3.4.1 HTTP Services

- **Incident Service:** Manages API calls related to incident operations (create, read, update, delete).
- **Authentication Service:** Handles user-related API calls that are the login and registration.
- **User Service:** Handles other user-related API that calls user data retrieved from the back-end such as user role and assiociated tasks.
- **Application Service:** Manages the application data retrieved from the back-end.
- **Notification Service:** Manages the notifications data retrieved from the back-end.

3.4.2 State Management

Angular's built-in services and observables were used to manage state across the components, ensuring a seamless flow of data between the user interface and the back-end.

3.5 Error Handling and Validation

Front-end validation was implemented for all forms using Angular's form validation mechanism. Error handling strategies were also put in place to manage API errors, ensuring that users receive clear feedback when actions fail.

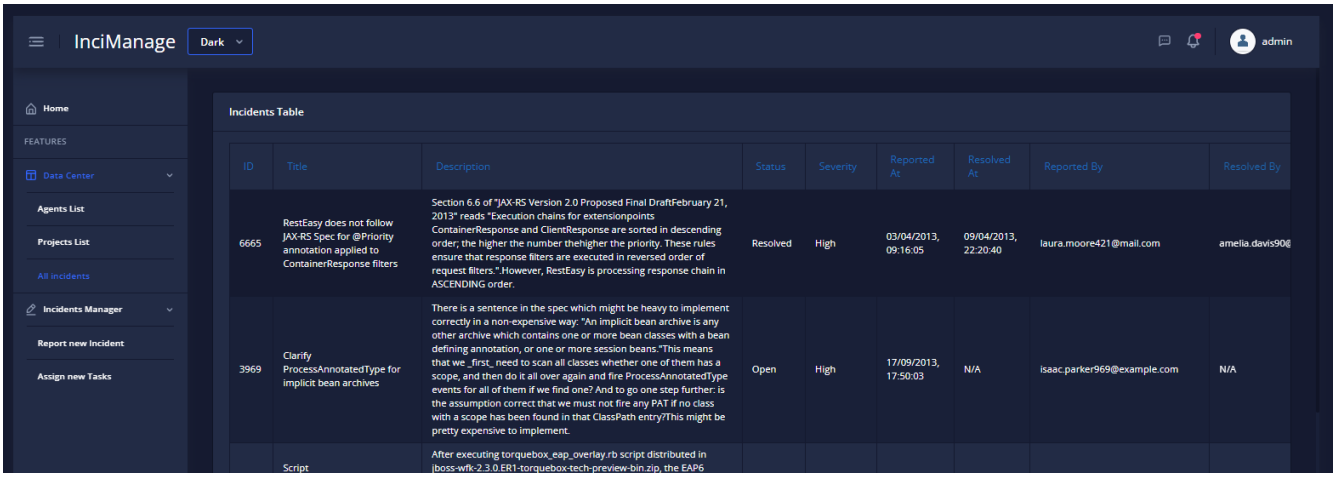

Figure 3.6: Integration of Incident Service to View all Incidents

3.6 Challenges and Solutions

Several challenges were encountered during development, such as handling asynchronous operations, managing state efficiently, and ensuring cross-browser compatibility. Solutions included using Angular's reactive forms, leveraging services for state management, and testing the application across multiple browsers.

Chapter 4

Business Intelligence

This chapter discusses the Business Intelligence (BI) aspects of the project, including the chosen dataset, the ETL (Extract, Transform, Load) process, and the visualization of insights through interactive charts integrated into the UI.

4.1 Data Gathering

The dataset used for analysis is the *Jira Issue Reports* dataset, which was obtained from Kaggle. This dataset is valuable for incident management and analysis due to its comprehensive records of issue reports and associated metadata.

4.1.1 Dataset Description

The *Jira Issue Reports* dataset provides detailed information about issues reported within a Jira system. It includes various attributes related to each issue, making it suitable for a range of analytical purposes. The dataset is structured to facilitate both exploratory data analysis and reporting.

4.1.2 Source

The dataset was sourced from Kaggle, a well-known platform for data science and machine learning competitions. The specific dataset can be accessed at: `https://www.kaggle.com/datasets/antonyjr/jira-issue-reports-v1`. This source is reputable and commonly used in the field of IT service management for various analyses and research.

4.1.3 Format

The dataset is provided in CSV (Comma-Separated Values) format, which is widely used for storing tabular data. CSV files are easy to work with and compatible with most data analysis tools and libraries.

4.1.4 Content

The dataset includes the following key fields:

- **Issue ID:** A unique identifier for each issue.
- **Key:** An other unique identifier for the issue.
- **Project name:** The name of the affected project.
- **Repository name:** The name of the affected project repository.
- **Resolution:** An indicator for the resolution state of the incident.
- **Type:** Indicates the type of the reported issue.
- **Updates:** The date when the issue has been updated.
- **Votes:** Number of votes for this issue.
- **Watchers:** Number of watchers for this issue.
- **Title:** A brief description of the issue.
- **Description:** A detailed description of the issue.
- **Priority:** The severity level of the issue (e.g., critical, major, minor).
- **Status:** The current status of the issue (e.g., open, in-progress, resolved).
- **Created:** The date when the issue was reported.
- **Resolved:** The date when the issue was resolved.
- **project:** An identifier for the project related to the issue.
- **Reporter ID:** An identifier for the user who reported the issue.
- **Assignee ID:** An identifier for the user assigned to resolve the issue.

4.1.5 Relevance

The dataset is highly relevant to IT service management as it captures various aspects of incident reporting and resolution. By analyzing this dataset, one can gain insights into issue trends, the effectiveness of response times, and the overall performance of the incident management process. This data is instrumental in identifying areas for improvement and making data-driven decisions to enhance service delivery and support.

The comprehensive nature of this dataset allows for a wide range of analyses, including incident frequency analysis, severity distribution, and performance metrics, which are crucial for effective incident management and operational efficiency.

4.2 ETL Process

The ETL process was implemented using Python, leveraging powerful data manipulation libraries such as Pandas and Numpy. The process was divided into three main stages:

4.2.1 Extract

In this phase the data was extracted from the CSV file derived from the data source. Below is the code used for data extraction:

```
import pandas as pd
import psycopg2
# Extraction of data from the CSV file
csv = pd.read_csv(r"..\ jira_issues .csv")
# Convert the 'created' column to datetime
csv['created'] = pd.to_dateime(csv['created'], format='%Y-%m-%d-%H:%M:%S', errors='coerce')
# Filter the data to keep only rows where the 'created' date is in the year 2013
dataset = csv[csv['created'].dt.year == 2013]
```

Listing 4.1: Data Extraction

4.2.2 Generate

In this phase the data was generated to fill the gap in the chosen dataset and match database requirements. It's hard and almost impossible to find a dataset that matches 100% the database architecture and for that reason generating some generic data was a task that must be done.
for this reason a series of data generation processes has been followed:

1. **Assigning Roles to Users :**

```
import pandas as pd
from extraction import dataset
# Get unique apps in the incidents
uniqueApps = dataset['project_name'].unique()
nbApps = uniqueApps.size
managersIds = []
# Add roles for all new users
with open(r'..\data\synthetic_users_roles.csv', 'w', newline='') as file:
    writer = csv.writer(file)
    i = 65; j = 1
    # Add default role for all new users
    for _ in range(3099):
        i = i + 1
        user_id = i
        role_id = j
        writer.writerow([
            user_id,
```

```
            role_id
        ])
    i = 100 ; role_id = 2
    # Add manager role for random new users
    for _ in range(nbApps):
        if (dataset['assignee_id'].eq(i).any()) :
            i = i + 14
        else:
            user_id = i
            managersIds.append(i)
            i = i + 14
        writer.writerow([
            user_id,
            role_id
        ])
```

Listing 4.2: Users Missing Data Generation

2. **Assigning fake names, emails and passwords to Users:**

```
import csv
import bcrypt
import random
import string
# Common names and domains
first_names = ["John", "Jane", "Alice",..]
last_names = ["Smith", "Johnson", "Williams",..]
domains = ["example.com", "test.com", "mail.com", "email.com", "domain.com"]
# Function to generate a realistic email
def generate_email(first_name, last_name):
    return f"{first_name.lower()}.{last_name.lower()}{random.randint(1,-999)}@{random.choice(domains)}"
# Function to generate a random string
def generate_string(length):
    return ''.join(random.choices(string.ascii_lowercase + string.digits, k=length))
# Function to hash a password using bcrypt
def hash_password(password):
    return bcrypt.hashpw(password.encode('utf-8'), bcrypt.gensalt()).decode('utf-8')
# List of existing email addresses to exclude
exclude_emails = ["user@example.com", "test@email.com",..]
# List of existing usernames to exclude
exclude_usernames = ["user1", "admin",..]
# Set of emails to check for duplicates
generated_emails = set(exclude_emails)
generated_usernames = set(exclude_usernames)
# Open a CSV file to write the data
with open(r'..\data\synthetic_users.csv', 'w', newline='') as file:
    writer = csv.writer(file)
    i = 65
    # Generate 4000 rows of data
    for _ in range(4000):
        i = i + 1
        id = i
        first_name = random.choice(first_names)
        last_name = random.choice(last_names)
        email = generate_email(first_name, last_name)
        # Ensure email is unique
        while email in generated_emails:
            email = generate_email(first_name, last_name)
        # Add the email to the set of generated emails
        generated_emails.add(email)
        password = hash_password(generate_string(10))
        username = f"{first_name.lower()}{random.randint(1,-300)}"
        while username in generated_usernames:
            username = f"{first_name.lower()}{random.randint(1,-300)}"
        generated_usernames.add(username)
        writer.writerow([
```

```
            id,   # No quoting
            email,   # Quoting email
            password,  # No quoting
            first_name,   # Quoting first_name
            last_name,   # Quoting last_name
            username # Quoting username
        ])
print("CSV file generated successfully.")
```

Listing 4.3: Users Missing Data Generation

3. **Adding associated application id and manager id to the existing application :**

```
import pandas as pd
import csv
from user_roles_generation import managersIds
# Get unique apps in the incidents
uniqueApps = pd.read_csv(r'..\data\transformed_dataset.csv')['project_name'].unique()
nbApps = uniqueApps.size
# Add roles for all new users
with open(r'..\data\synthetic_apps.csv', 'w', newline='') as file:
    writer = csv.writer(file)
    i = 22
    j = 0
    for _ in range(nbApps):
        i = i + 1
        app_id = i
        app_name = uniqueApps[j]
        app_manager = managersIds[j]
        j = j + 1
        writer.writerow([
            app_id,
            app_name,
            app_manager
        ])
```

Listing 4.4: Application Missing Data Generation

4.2.3 Transform

The transformation phase involved cleaning the data, handling missing values, normalizing fields, and performing data enrichment where necessary.

Pandas provided efficient data structures for manipulating tabular data, while Numpy was used for numerical computations.

For this purpose a pipeline of transformations has been followed:

1. Dropping unneeded columns from the dataset.
2. Replacing the values of the status and severity values with ones matching the database base fields types.
3. Dropping the rows where some columns values are not in the specified rules.
4. Reassign the the reporter and resolver id.
5. Auto generate the solution description based of the type of the issue.

6. Drop the project name column and replace it with the project id to match the database architecture.

7. Rearrange the columns.

8. Get sample dataset from the filtered data and Save it in a support.

Below you can find the code used for the data transformation process:

```
import pandas as pd
import numpy as np
from extraction import dataset
# Delete not useful data from the CSV resource
columns_to_drop = ['id', 'project', 'key', 'repositoryname', 'votes', 'watchers', 'resolution']
dataset.drop(columns=columns_to_drop, inplace=True)
# Replace the priority values
priority_replacements = {
    'Major': 'High',
    'Critical': 'Critical',
    'Minor': 'Medium',
    'Optional': 'Low'
}
dataset['priority'] = dataset['priority'].replace(priority_replacements)
# Filter out rows where 'priority' values are not in the specified replacements
valid_priorities = ['High', 'Critical', 'Medium', 'Low']
dataset = dataset[dataset['priority'].isin(valid_priorities)]
# Refactor Status values
status_replacements = {
    'Coding-In-Progress': 'In_Progress',
    'Closed': 'Resolved',
    'Done': 'Resolved',
}
dataset['status'] = dataset['status'].replace(status_replacements)
# Filter out rows where 'status' values are not in the specified replacements
valid_status = ['Resolved', 'In_Progress', 'Open']
dataset = dataset[dataset['status'].isin(valid_status)]
# Additional filtering based on 'status' and 'assignee_id'
dataset = dataset.loc[~(
    ((dataset['status'].isin(['Resolved', 'In_Progress'])) & (dataset['assignee_id'].isnull())) |
    ((dataset['status'].isin(['Resolved', 'In_Progress'])) & (dataset['reporter_id'].isnull())) |
    ((dataset['status'] == 'Open') & (dataset['assignee_id'].notnull())) |
    ((dataset['status'] == 'Open') & (dataset['reporter_id'].isnull())) |
    (dataset['description'].isnull())
)]
# Reassign the reporter_id and assignee_id
def generate_random_id(low, high):
    return np.random.randint(low, high + 1)
dataset['reporter_id'] = dataset['reporter_id'].apply(
    lambda x: int(generate_random_id(50, 4050)) if pd.notnull(x) else np.nan
)
dataset['assignee_id'] = dataset['assignee_id'].apply(
    lambda x: int(generate_random_id(50, 4050)) if pd.notnull(x) else np.nan
)
# Filter out rows based on 'type'
valid_type = ['Error', 'Fail', 'Not-working', 'Slow', 'Bug']
dataset = dataset[dataset['type'].isin(valid_type)]
# Add new column 'solutionDescription' with short solutions based on 'description'
def suggest_solution(description):
    if 'error' in description.lower():
        return 'Check-error-logs-and-stack-trace.'
    elif 'fail' in description.lower():
        return 'Verify-configurations-and-retry.'
    elif 'not-working' in description.lower():
        return 'Restart-the-application-and-check-settings.'
    elif 'slow' in description.lower():
        return 'Optimize-performance-and-check-resource-usage.'
    elif 'bug' in description.lower():
```

```
        return 'Review-code-for-bugs-and-apply-fixes.'
    else:
        return 'Investigate-the-issue-thoroughly.'
dataset['solutionDescription'] = dataset['description'].apply(suggest_solution)
dataset.drop(columns=['type'], inplace=True)
# Add new app_id column
def fetch_app_id(app_name):
    app_id = df[df['name'] == app_name]['id']
    return app_id.iloc[0] if not app_id.empty else None
dataset['app_id'] = dataset['project_name'].apply(fetch_app_id)
dataset['app_id'].fillna(1, inplace=True)
dataset['app_id'] = dataset['app_id'].astype(int)
dataset.drop(columns=['project_name'], inplace=True)
# Add new 'id' column
dataset['id'] = range(len(dataset))
# Reorder the columns
final_columns = ['id', 'title', 'description', 'created', 'resolved', 'solutionDescription', 'status', 'priority', 'app_id', 'reporter_id', 'assignee_id']
dataset = dataset[final_columns]
# Save the filtered data to a new CSV file
output_csv_path = r"..\data\transformed_dataset.csv"
sampled_dataset = dataset.sample(n=2800, random_state=1)
sampled_dataset.to_csv(output_csv_path, index=False)
```

Listing 4.5: Data Transformation

4.2.4 Load:

The transformed data was loaded back into the database engine for storing and further usage in the visualisation step. The ETL process ensured that the data was consistent, clean, and ready for analysis, forming the foundation for the BI components of the project.

```
import pandas as pd
from sqlalchemy import create_engine
conn = create_engine('postgresql+psycopg2://postgres:system@localhost:5432/IncidentManager')
#Import new users
newUsers = pd.read_csv(r'..\data\synthetic_users.csv')
newUsers.to_sql("contributor",con=conn,if_exists='append',index=False)
#Import new users Roles
newUsersRoles = pd.read_csv(r'..\data\synthetic_users_roles.csv')
newUsersRoles.to_sql("user_roles", con = conn, if_exists ='append',index=False)
#Import new projects
newProject = pd.read_csv(r'..\data\synthetic_apps.csv')
newProject.to_sql("application", con = conn, if_exists ='append',index=False)
#Import new incidents
newIncidents = pd.read_csv(r'..\data\transformed_dataset.csv')
newIncidents.to_sql("incident", con = conn, if_exists ='append',index=False)
```

Listing 4.6: Data Loading

4.3 Data Modeling

In this step, the main purpose was to to structure the data, deriving the most suitable primary components for the best efficient analysis and reporting result.

Those primary components include the Fact Table, Dimensions, Measures, and the chosen schema.

This schema was the best we can find to derive the most valuable insights.

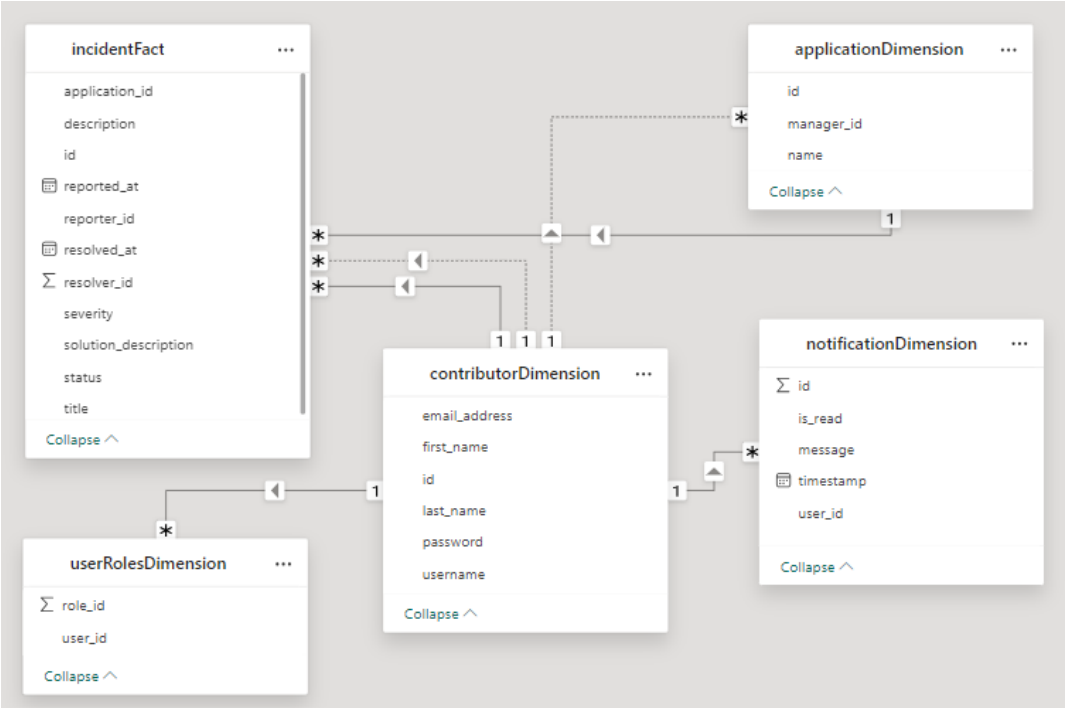

Figure 4.1: Snowflake Schema of the dataset

- **Fact Table:**

  The central element of our data id the Incident table which captures essential information about the reported issues that forms the core of our analytical focus providing valuable insights into the incident resolution and the project performance.

- **Dimensions:**

  Dimension tables represents the path for more oriented, deep and unique data analysis.

  1. **Contributor dimension:** Indicates the involved party information such as the email, username and so on.
  2. **Application dimension:** Provides insights about the corresponding application to the incident.
  3. **Notification dimension:** Indicates either the involved parties has been notified or not.
  4. **User Roles dimension:** Contains brief indicators for the involved parties roles.

- **Measures:**

  Our analysis requires quantitative metrics, known as Measures, to quantify performance and trends. In the context of our data model, the main key measure was the incident **id** , while it's not explicitly a numerical

value, it could potentially be used as a count of incidents in function of many other categorical attributes such as the severity and the status.

- **Schema Choice: Snowflake Schema:**

  I opted for a Snowflake Schema to organize our data efficiently. This schema's normalized structure facilitates easy maintenance and reduces data redundancy. It fosters a clear separation between dimensions and the central fact table, promoting scalability and flexibility in our analytical adventure.

4.4 Visualization and Charts

In the Incident management web application, data is a key driver of insights and helps with decision making. To effectively convey these findings, the UI has an interactive dashboard with a visually appealing dashboard that offers users with an in-depth summary of key statistics and trends. This was accomplished through the incorporation of advanced charting libraries such as Chart.js and Nebular's in-built charting capabilities which permit customizable and dynamic data visualizations.

The various charts incorporated into this dashboard include bar charts, line charts, pie charts or scatter plots each representing different aspects of incident data. Through interacting with these visualizations, they can filter or drill down into the real-time data for further analysis. For instance, severity distribution charts display critical incidents instantly while time based trend graphs enables users to monitor how incidents have evolved over specific periods. Furthermore, even intricate datasets are presented within the UI in such ways that are easily understood, empowering users to make better decisions that are driven by evidence. The uniformity in color schemes and legends across all analyses ensures cohesion in user experience allowing insights be interpreted at a glance easily.

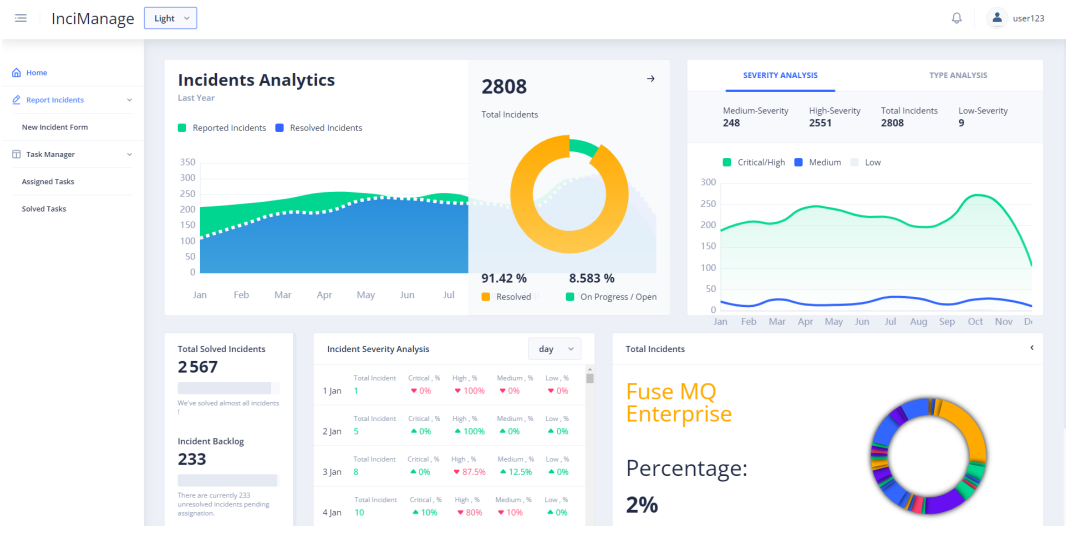

Figure 4.2: Default User Dashboard View

- **Severity and Status Distribution:** Line and bar charts illustrating the distribution of incidents by severity and status.

Figure 4.3: Severity Analysis Chart

Figure 4.4: Type Analysis Chart

- **Application-Based Insights:** Pie and Line charts showing the number of incidents per application, helping to identify applications with higher incident rates.

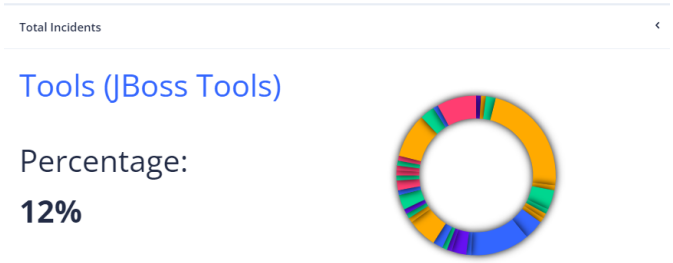

Figure 4.5: Application Analysis Chart

These charts not only enhance the visual appeal of the UI but also provide key stakeholders with the ability to monitor and make data-driven decisions. The integration of BI elements in the frontend allows users to quickly identify trends and areas requiring attention, leading to a more proactive approach in incident management.

Chapter 5

Gemini Powered Chat-bot

In this part of project I've tried to leverage the user experience and offer some extra services for better problem solving and support. For this reason the Gemini API was the best solution as it was at the same time free and easy to integrate. This integration process was composed of this series of simple steps :

1. **Getting a Gemini AI API Key from the official Google AI Studio website.**

2. **Integrating the Gemini text generation service**

```
// gemini-ai.service.ts

import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { GoogleGenerativeAI } from '@google/generative-ai';

Codiumate: Options | Test this class
@Injectable({
  providedIn: 'root'
})
export class GeminiAiService {
  private generativeAI: GoogleGenerativeAI;
  private apiKey = 'A--------------------------';  // Keep your API key secure

  constructor(private http: HttpClient) {
    this.generativeAI = new GoogleGenerativeAI(this.apiKey);
  }

  Codiumate: Options | Test this method
  async generateText(prompt: string): Promise<string> {
    const model = this.generativeAI.getGenerativeModel({ model: 'gemini-pro' });
    const result = await model.generateContent(prompt);
    const response = await result.response;
    return response.text();
  }
}
```

Figure 5.1: Integration of the Gemini-AI text generation service

3. **Building the angular Chat-bot component**

4. **Empowering the Chat-bot responses with the controlled Gemini AI service**

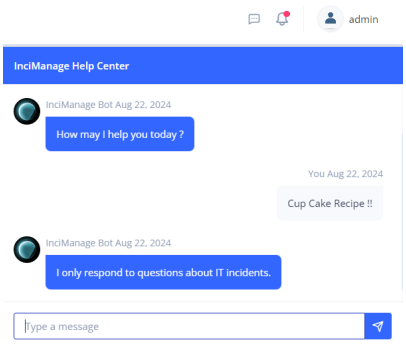

Figure 5.2: Integration of the Chat-bot In the Interface

I can't deny the fact that it has take me a lot of work and perseverance to successfully finish the integration of the Gemini ai powered chat-bot or as we have called it the "InciManage Help Center" but it was a good step to boost the performance of the platform and guide users.

Chapter 6

Conclusion and Future Enhancements

Conclusion

In conclusion, my summer internship at Sofrecom Tunisie provided invaluable experience in both technical and soft skills, allowing me to develop a fully functional web application for incident management. The project's success was driven by the integration of modern frameworks such as Angular, Spring Boot, and Nebular, resulting in a robust system that efficiently manages incidents from reporting to resolution. Through this project, I have honed my ability to solve real-world problems, particularly those related to IT incident tracking, resolution efficiency, and business intelligence.

The **InciManage** system effectively addresses the challenges of existing incident management tools, providing an intuitive interface, comprehensive reporting features, and seamless incident resolution workflows. The system's scalable architecture ensures that it can be adapted to various production environments, ultimately improving productivity and enabling quicker response times for incident resolution.

Future Enhancements and Perspectives

Looking ahead, several enhancements could be introduced to further improve the **InciManage** application:

- **Enhanced Security Features**: Implementing more advanced security protocols, such as multi-factor authentication (MFA) and role-based access controls (RBAC), would further protect sensitive data and improve the system's overall security posture.
- **Machine Learning for Predictive Analysis**: Integrating machine learning algorithms that could help predict future incidents based on historical data. This would allow organizations to proactively manage potential problems before they occur, reducing downtime and improving overall efficiency.

- **Mobile Application Development**: Developing a mobile version of the application would enable users to report, track, and resolve incidents from their smartphones, offering greater flexibility and accessibility.
- **Integration with Other Enterprise Tools**: Expanding the application's compatibility with other enterprise tools such as Slack, Microsoft Teams, and third-party monitoring systems would allow for more seamless communication and incident management across various platforms.

By implementing these enhancements, the **InciManage** application could evolve into a more comprehensive and powerful tool for incident management, offering organizations an even greater ability to streamline their operations and improve incident resolution processes.

References

[1] *Angular*. URL: https://angular.dev/.

[2] *Bruno*. URL: https://www.usebruno.com/.

[3] *JIRA: Issue and Project Tracking*. URL: https://www.atlassian.com/software/jira.

[4] *Nebular*. URL: https://akveo.github.io/nebular/.

[5] *ngx admin template*. URL: https://akveo.github.io/ngx-admin/.

[6] *PostgreSQL*. URL: https://www.postgresql.org/.

[7] *ServiceNow: IT Service Management*. URL: https://www.servicenow.com/products/itsm.html.

[8] *Sofrecom Tunisia*. URL: https://www.sofrecom.com/en/about-us/our-company.html.

[9] *Zendesk: Customer Service Software*. URL: https://www.zendesk.com/.