deti universidade de aveiro
departamento de electrónica, telecomunicações e informática

Software Engineering

1st Semester

2024/2025

Individual Assignment Report

José Miguel Costa Gameiro, 108840

Table of Contents

Table of Acronyms

- **IDP** - Identity Provider;
- **AWS** - Amazon Web Services;
- **PR** - Pull Request;
- **INVEST** - Independent, Negotiable, Valuable, Estimable, Small, Testable;
- **VPC** - Virtual Private Cloud;
- **EC2** - Elastic Compute Cloud;
- **RDS** - Relational Database Service;
- **ALB** - Application Load Balancer;
- **AAA** - Authentication, Authorization, Accounting;
- **JWT** - Json Web Token.

Table of Images

1. Introduction

The individual project consists in developing a fully functional software, by following the agile methodology and deploying it using **AWS**.

The software must include a web application, a RESTful API and integration with a relational database, it also must have an authentication, authorization, and accounting system using an **IDP**.

The main idea of the solution to be implemented is to create a **To-Do List Application** that follows this functionalities:

- Each user must authenticate in the system;
- The tasks created by a user can only be visible to them;
- Users should be able to add tasks with a title and description;
- Users can mark tasks as completed;
- Users can edit or delete existing tasks;
- Users can set deadlines for tasks;
- Provide options to sort tasks by creation date, deadline, or completion status;
- Allow filtering tasks by category or completion status;
- Users should be able to assign priorities (like low, medium, high) to tasks;

2. Agile Process

To manage the software project effectively, I implemented an Agile methodology using ***Jira*** as the primary tool for project organization and tracking. The initial step was to define the Definitions of **Done** and **Ready**, providing clear guidelines for when a user story was prepared for development and when it could be considered complete.

With these definitions in place, the work was organized into **Epics** (broad categories of functionality), each containing detailed **User Stories** that described specific user requirements. For every user story, an **Acceptance Criteria** was defined using the **Given-When-Then syntax**, a story point value was estimated and assigned and a prioritization was also attributed for the story, it could have the following values:

- Lowest;
- Low;
- Medium;
- High;
- Highest.

The development was structured into **two-weeks sprints**, during which tasks were prioritized, and tracked to ensure consistent progress toward the project goals. In total **5** sprints were created and concluded with success.

In terms of branch organization, it follows a feature branch approach, where it exits two main branches named “**main**” for the production environment and “**dev**”, and for each user story a new branch is created based on the code that relies in the dev branch. When the development of the feature is finished a pull request is made from the feature branch to the **dev** one, and after accepting it and merging the changes the feature branch is deleted. When the time to deploy arrives a **PR** is created from the **dev** branch to the **main** one. Some rulesets were created for each type of branch, the rules for the **main** and **dev** are similar where it is not possible to commit directly into these branches, only through a **PR**, this must not have any merge conflicts and if a new commit is added then it analyzes if there exists any merge conflicts again. For feature branches only one rule is applied which is forbid push forces.

2.1 Epics, User Stories and Tasks

The user stories were grouped into five epics, each representing a key functionality of the application and also containing multiple subtasks with a description of what needs to be done. Three tasks were also created, cause they are not considered has functionalities of the system, instead they are configurations, like dockerize the components of the software or configure the deployment, so they were created has tasks and not user stories or epics (they can be accessed through this links EIP-24, EIP-48 and EIP-49).

2.1.1 Task Ownership

- **User Story 1:** As a user, I want to authenticate in the system (EIP-7);
- **User Story 2:** As a user, I want to be able to see the tasks that I created so that I can know which tasks I must complete (EIP-8).

2.1.2 Task Management

- **User Story 3:** As a user, I want to be able to add tasks with a title and description (EIP-9);
- **User Story 4:** As a user, I want to be able to mark tasks has completed so that I can know which tasks are completed and which are not (EIP-10);
- **User Story 5:** As a user, I want to be able to edit or delete existing tasks (EIP-11).

2.1.3 Task Deadlines

- **User Story 6:** As a user, I want to be able to set deadlines for a task (EIP-12).

2.1.4 Sorting and Filtering

- **User Story 7:** As a user, I want to be able to sort tasks by creation date, deadline or completion status (EIP-13);
- **User Story 8:** As a user, I want to be able to filter tasks by completion status and prioritization (EIP-14).

2.1.5 Task Prioritization

- **User Story 9:** As a user, I want to be able to assign priorities (like low, medium or high) to a new or existing task (EIP-15).

2.2 Definitions of Ready and Done

For the definitions of Ready and Done a checklist was created and each time a new user story was added, this checklist would appear which contained this two definitions

2.2.1 Definition of Ready

A user story is considered ready if:

- It met all **INVEST** principles;
- Acceptance criteria was clearly defined;
- It was estimated in story points;
- It was appropriately prioritized in the product backlog.

2.2.2 Definition of Done

A user story is considered done/completed if:

- It is fully developed;
- Relevant documentation is completed;
- It was tested locally (To aid this some pictures were included in each user story to showcase the functionality developed in **Jira**);
- It met all acceptance criteria during testing.

3. Architecture

3.1 Main Architecture

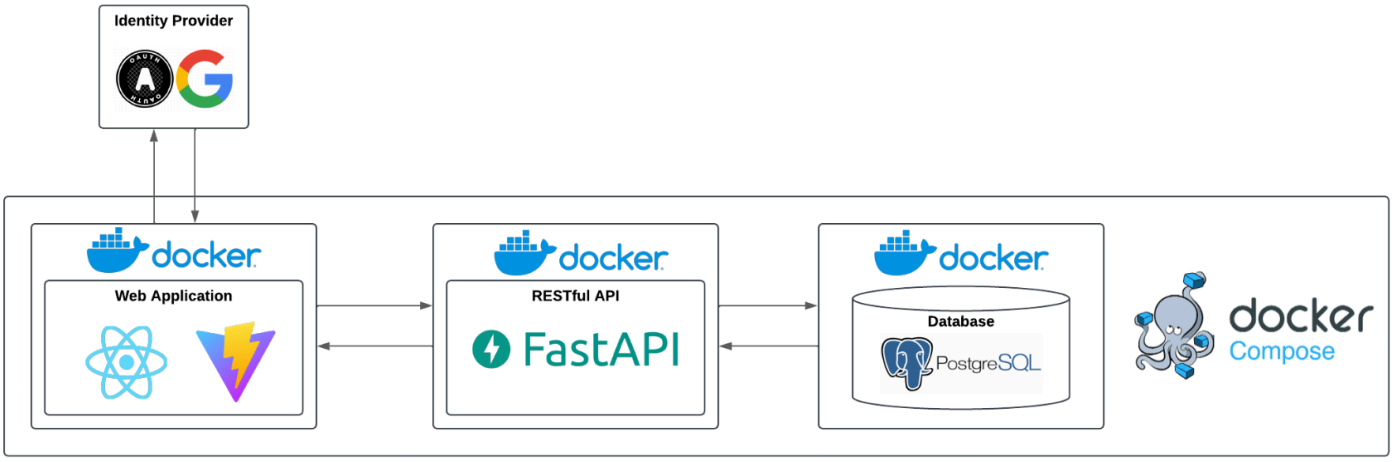

Fig. 1 - Main Architecture representation

The main architecture is described in the image above and has the following components:

- **IDP**: The application leverages third-party identity providers such as Google OAuth2.0 to handle user authentication and authorization securely;
- **Web Application:** The frontend is built using the React and Vite frameworks, and containerized with Docker, using the ***node:18.17.0-alpine*** image;
- **RESTful API:** The backend service is implemented using FastAPI, a high-performance web framework for building APIs with Python, it is also containerized to maintain an isolated and portable environment, using the ***python:3.12.3*** image;
- **Database**: A PostgreSQL database is used to store the application's data securely and It is managed in its own Docker container, using the ***postgres:15*** image.

Docker Compose is utilized to orchestrate all containers described above.

3.2 Database Diagram

The database model consists of a single table named **Task**, it wasn't necessary to create another table to store user information because it was unnecessary, due to the need to only have one attribute associated with a task which is the user email. It has the following attributes:

- **id:** The primary key of the table, uniquely identifying each task;
- **title:** A short and descriptive title for the task. This column is indexed to allow for faster searches;
- **description:** A brief description of the task's purpose or details;
- **is_completed:** Indicates whether the task has been completed. Defaults to False (meaning that it hasn't been completed when created);
- **creation_date:** Records the date and time when the task was created;
- **deadline:** Stores the deadline for completing the task. Defaults to null if no deadline is specified;
- **priority:** Stores the priority level of the task, it can either be "*High*" "*Medium*" or "*Low*". Defaults to null if no priority is assigned;
- **user_email:** Stores the email of the user associated with the task.

Fig. 2 - Database Diagram

3.3 Deployment Architecture

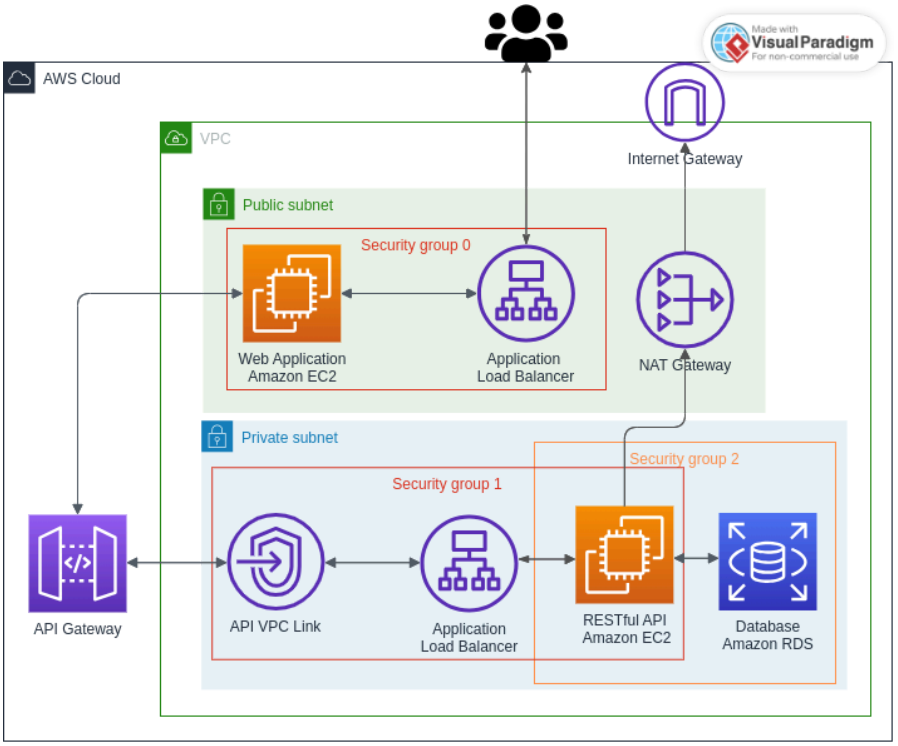

Fig. 3 - Deployment Architecture representation

The proposed architecture demonstrated in the picture above for deploying the software in AWS is designed to ensure high availability, security and scalability.

Most of the components in the architecture are hosted within a **VPC**, enabling the creation and configuration of both private and public subnets. This ensures that instances without external access requirements remain isolated. The architecture spans to two Availability Zones (us-east-1a and us-east-1b) to enhance high availability and provide fault tolerance in case on instance becomes unavailable

The private subnet hosts an Amazon RDS instance with a PostgreSQL database, designed to prevent external access and mitigate the risk of data loss or corruption. Additionally, an **EC2** instance running the RESTful API, developed with FastApi, is also located in the private subnet for similar security reasons. However, this instance requires internet access to perform two tasks: downloading the Docker image containing the API and running it. To facilitate this, both a NAT and Internet Gateway were configured. Neither the **RDS** nor the **EC2** instances have public IP addresses, only private ones, and both belong to the same security group. This setup ensures that the **EC2** instance can communicate with the database for information storage. To enable this communication, an inbound rule was added to allow TCP traffic on port 5432, the default port for PostgreSQL.

To enable external interaction with the RESTful API, an **ALB**, a **VPC** Link and an API Gateway were configured. The **ALB** and **VPC** Link are located within the private subnet,

while the API Gateway operates outside the **VPC**. The **VPC** Link facilitates secure communication between the API Gateway and resources within the targeted **VPC**, as HTTP API private integration methods only permit access through a **VPC** Link to private subnets. The **ALB** is set up as an internal load balancer with a target group that includes the private **EC2** instance. It routes requests using private IP addresses to resources in the private subnet. When the **ALB** receives a request from an HTTP API endpoint, it evaluates the listener rule to determine the appropriate protocol and port. The target group then forwards the request to the **EC2** instance for processing.

The public subnet hosts a NAT Gateway, as previously mentioned, along with two additional components: an **EC2** instance running the web application in a Docker container and an internet-facing **ALB** to facilitate access to the application. The public **ALB** is configured with an HTTPS listener and uses a server certificate provided by the class instructor. The **ALB** terminates the frontend connection by decrypting incoming client requests before forwarding them to the targets. Both the **EC2** instance and the **ALB** are part of the same security group. A target group was created to route HTTP traffic from the **ALB** to the **EC2** instance.

4. Implementation

In this section of the report some details of the implementation phase are described, like the authentication system, security in the frontend and backend.

4.1 Authentication

As previously mentioned, the authentication mechanism was implemented using Google's OAuth to simplify the development of an **AAA** system. To achieve this, the frontend project incorporated the @react-oauth/google library, and a Google login component was added to the login page. Setting up this component required the following steps:

1. Creating a new project in the Google Cloud Console;
2. Configuring an OAuth consent screen;
3. Creating and configuring a new web client ID.

Once configured, a client ID was generated and integrated into a GoogleOAuthProvider to enable Google-based authentication. The Google Login component was then embedded in the **LoginPage**, utilizing the **onSuccess** and **onError** callback functions provided by the component. These callbacks handle the login process as follows:

- **On Success**: A **JWT** token containing user information is provided. Using the jwtDecode function from the jwt-decode package, the token is decoded, and key details (such as the user's name, picture URL, and the encoded JWT token) are stored in a **Zustand** store. This token is later used for communication with the RESTful API;
- **On Error**: If an error occurs during the login process, an appropriate error message is displayed to the user.

In the backend, a function was implemented to validate **JWT** tokens provided by Google. This function utilizes the verify_oauth2_token method from the google-auth Python library. The validation process involves the following steps:

- Confirm if the token is correctly signed by Google;
- Check if the aud value in the token matches the application's client ID;
- Verify that the iss value is either accounts.google.com or https://accounts.google.com;
- Ensure the token has not expired, based on the exp timestamp, which indicates the token's validity for one hour.

If the function completes these checks without encountering errors, the token is deemed valid, and the decoded token information is returned. Otherwise, the function raises an error, indicating the token is invalid.

4.2 Application Security

In terms of security some mechanisms were implemented:

- **Protected routes**: In the web application, the **HomePage** serves as the main interface where users can interact with the RESTful API to create tasks, complete them, and perform other actions. Since these actions require associating tasks with a user's email or retrieving their tasks, the user must be logged in. To enforce this authentication requirement, a **RequiredAuth** component was created. This component checks the authentication status store in the user state, managed using Zustand. If the **isLoggedIn** value is **true**, the user is considered authenticated and can proceed to access the app. Otherwise, an alert is displayed with the message "You must log in first", and the user is redirected to the login page;
- **Protected endpoints**: In the RESTful API, all the endpoints require a JWT Token provided by Google to validate the user and retrieve their email. To streamline this process, a custom decorator called **authenticated** was created. This decorator extracts the access token from the request headers and stores it in a variable. If no token is found, an HTTP exception is raised with a 401 status code and the

message, “Unauthorized, you must be logged in to access this resource”. For each endpoint, the **validate_credential** (described earlier, using the verify_oauth2_token method), if the token passes validation, the user’s requested action is executed, else another HTTP exception is raised with a 401 status code and the message “Invalid credential”.

4.3 API Documentation

All the endpoints developed are documented using Swagger, when the EC2 instance is running the documentation can be seen through this url https://f9xqys4fz4.execute-api.us-east-1.amazonaws.com/docs. However the instance can not be running instead in the repository there’s a **docs.json** file in the backend directory, to visualize it simply go to this link https://editor.swagger.io/, open the tab that says “**File**” and select the option “**Import file**”, then select the JSON file and the documentation will be displayed in the screen.

5. Future Work and Solution Links

In terms of future work, this software needs some improvements like:

- Add more informations to each endpoint to have a more detailed documentation for the RESTful API;
- Fix an error in the edit task, in the web application, cause when the button edit is the tasks information only appears if the button is clicked one time, then modal is closed and clicked again. And if another action needs to be edited then it will appear the information relative to the previous selected task;
- Add more error handling in the frontend, when the jwt token expires no action appears in the frontend and it should redirect the user to the login page to perform the login action again.

Link for the github:

https://github.com/zegameiro/ES_Individual_Proj

Link for the Jira workspace:

https://ua-team-swpr9qbn.atlassian.net/jira/software/projects/EIP/list

Deployed Solution:

- **API:** https://f9xqys4fz4.execute-api.us-east-1.amazonaws.com
- **Web Application:**
  - https://es-ua.ddns.net (For this link to work you must follow the tutorial in the README.md of the github repository or use the other one)
  - https://taskflow-frontend-load-balancer-506252942.us-east-1.elb.amazonaws.com (This one will say that is insecure to continues just click in the option to proceed)

6. References

- https://docs.aws.amazon.com/elasticloadbalancing/latest/application/create-https-listener.html
- https://aws.amazon.com/pt/blogs/compute/configuring-private-integrations-with-amazon-api-gateway-http-apis/
- https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/USER_CreateDBInstance.html
- https://docs.aws.amazon.com/vpc/latest/userguide/vpc-example-private-subnets-nat.html
- https://blog.logrocket.com/guide-adding-google-login-react-app/
- https://developers.google.com/identity/gsi/web/guides/verify-google-id-token?hl=pt-br#python