universidade de aveiro deti
departamento de eletrónica, telecomunicações e informática

universidade de aveiro theoria poiesis praxis

INDIVIDUAL ASSIGNMENT REPORT

SOFTWARE ENGINEERING

DOER

Daniel Madureira | 107603

1st semester

2024/2025

INDEX

INTRODUCTION

The objective of this individual project is to develop a fully functional **Task Management software** employing **agile methodologies**, as well as to gain experience in **designing**, **developing** and **deploying** this software project to the **Amazon Web Services** (AWS).

The main idea behind the project is to build a Task Management software, similar to a **ToDo List application**. The software app must include a **web user interface** (UI), a **RESTful API** that interacts with a **relational database**. It also must have external **authentication**, **authorization** and **accounting** (AAA) mechanisms using an **identity provide**r (IdP). Moreover, the solution must be robust and secure.

The solution must include a ToDo List application with the following **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 (e.g., low, medium, high) to tasks.

AGILE METHODOLOGY

To approach this project as effectively as possible, I adopted an Agile methodology, with Jira serving as my primary tool for organizing and managing the work.

The first step in applying the Agile methodology was to understand the problem I needed to solve and break it down into smaller, manageable pieces by creating **epics**.

After gaining clarity on what needed to be done, I defined **how the work would be carried out** by establishing the **Definition of Ready (DoR)** and the **Definition of Done (DoD)**. These definitions ensured that all tasks were well-prepared before development and met the necessary standards upon completion.

With the "what" and "how" clearly defined, I proceeded to organize the work into smaller, actionable tasks—**user stories**. These user stories were written using the "As a [user], I want to [action] so that [benefit]" format and were aligned with the established criteria.

Each user story was complemented by **acceptance criteria**, which provided a clear, testable, and measurable set of conditions that needed to be met for the story to be considered complete. The acceptance criteria followed the **Given-When-Then** format.

Once the user stories were created, they were assigned **story points** to estimate effort and complexity. The user stories were then prioritized based on their importance and urgency, following a hierarchy of:

- **Lowest**: Nice-to-have tasks with minimal impact.
- **Low**: Non-critical items that can be addressed eventually.
- **Medium**: Important tasks that should be completed in due course.
- **High**: Critical tasks essential for core functionality.
- **Highest**: Urgent tasks requiring immediate attention.

The user stories were then integrated into the **backlog**, awaiting their turn to be included in a sprint. I conducted **weekly sprints** as much as possible to maintain a **rapid pace** of development. In total, **four weekly sprints** were completed, along with **one two-week sprint**.

BRANCHES

For version control, I utilized **GitHub** and **Git** to effectively manage the project's codebase. By integrating GitHub with Jira, I was able to make a smart use of branches. The workflow revolves around a single **main** branch, from which **feature branches** are created for each **user story**.

Feature branches are regularly updated through pull requests (PRs) to the **main** branch, ensuring that new features are incorporated in an organized manner. Occasionally, hotfixes may be applied directly to the **main** branch when they do not justify the creation of a separate branch.

EPICS, USER STORIES AND TASKS

As mentioned earlier, the work was divided into **Epics** and **User Stories** to ensure structured progress and clarity in execution. Each **Epic** focuses on a core area of the project, and the **User Stories** are broken down into actionable tasks, some of which contain **sub-tasks** for further refinement of complex work.

In total, the project counts with **five epics** and **seven user stories** as shown below:

Task Ownership

- This epic focuses on user authentication, allowing users to securely log into the system and access their personal task list.
    - **DOER-6: User login**: This user story ensures users can log into the system, granting them access to their tasks and providing a personalized experience.

Task Management

- This epic is about managing the core functions of tasks in the system. The user stories related to this epic include:
    - **DOER-28: Add tasks**: This user story allows users to create new tasks and define key details, such as the task name, description, and priority.
    - **DOER-29: Edit Task**: This story enables users to modify existing tasks, correcting mistakes or updating details as needed.
    - **DOER-31: Complete tasks**: This allows users to mark tasks as completed, helping to track what work has been finished.

Task Deadlines

- The focus of this epic is to implement deadline features for tasks. This will help users set and manage deadlines effectively, ensuring timely completion of tasks.
    - **DOER-27: Deadlines**: This user story allows users to set deadlines for their tasks, keeping track of due dates and staying on schedule.

Sorting and Filtering

- This epic involves the ability to sort and filter tasks, improving the user experience by allowing users to organize tasks based on various criteria such as date, priority, or status.
    - **DOER-26: Sorting and filtering**: This user story enables users to filter and sort their tasks, providing an intuitive way to focus on the most relevant tasks at any given moment.

Task Prioritization

- This epic addresses the ability to prioritize tasks, ensuring that users can focus on high-priority items first.
    - **DOER-30: Assign priority**: This user story allows users to assign priority levels to tasks, helping them to manage their workload and prioritize more critical tasks effectively.

In addition to user stories, **tasks** were also used to manage **developer-centric** activities such as backend processes, deployment, and documentation. These tasks were not directly related to user functionality but were essential for the completion of the project.

DEFINITION OF READY AND DONE

The Definition of Ready (DoR) and Definition of Done (DoD) are essential components of an agile project lifecycle. They help maintain consistency in the work being done and ensure that each task meets the necessary standards for progress and completion. For this project, the DoR and DoD are as follows:

Definition of Ready

A user story is considered **Ready** for development when the following criteria are met:

- **INVEST Principles**: The user story adheres to the **INVEST** criteria.
- **Clear Acceptance Criteria**: The user story must have well-defined and testable acceptance criteria.
- **Story Points Estimated**: The user story must have story points assigned to it.
- **Appropriately Prioritized**: The user story must be appropriately prioritized within the product backlog.

Definition of Done

A user story is considered **Done** when the following conditions are met:

- **Fully Developed**: The feature or functionality described in the user story is fully developed and integrated with the project.
- **Relevant Documentation Completed**: All relevant documentation is completed and up-to-date.
- **Local Testing Completed**: The user story must be tested locally by the developer to ensure it functions as expected.
- **Acceptance Criteria Met**: During testing, the user story must meet all the defined acceptance criteria.

ARCHITECTURE

MAIN ARCHITECTURE

The system may be described as:

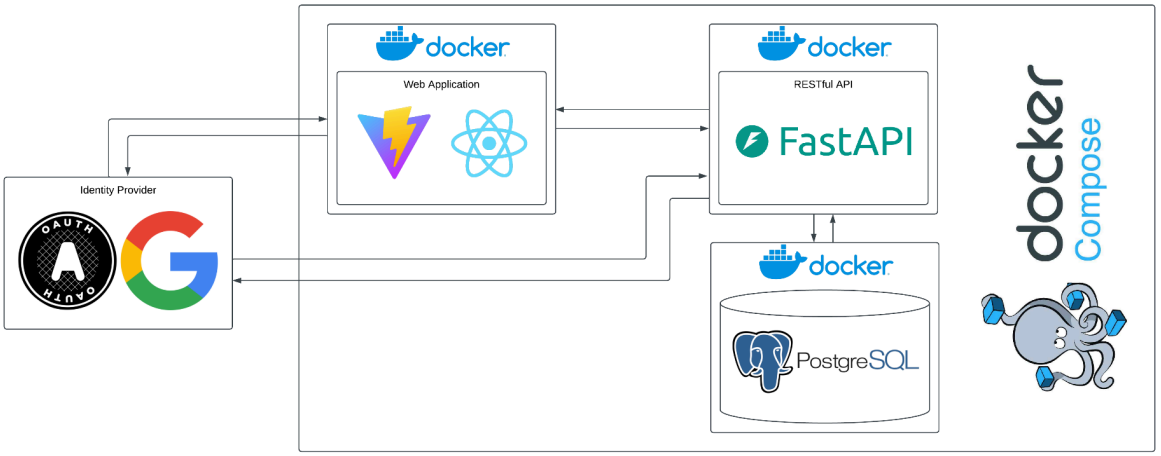

Fig.1 - Database table diagram

The architecture of my application can be divided into several interconnected parts. each fulfilling a specific purpose. The components are as follows:

- **IdP**: The application leverages an **external IdP** for AAA mechanisms, specifically **Google IdP** with the OAuth standard. This integration enables secure, token-based access control and seamless user authentication. The issued tokens are used to verify the user's identity when interacting with the API, ensuring secure and efficient communication.

- **WebApp**: The web application serves as the user interface (UI) of the system, built with **React** and **Vite** as the build tool. It enables end-users to interact with the application, handling user authentication via the IdP and communicating with the API to retrieve and manage data, such as user tasks, from the database. This component uses a `node:22-slim` docker image.
- **API**: The backend is implemented using **FastAPI**, acting as the core for business logic and data processing. It provides RESTful endpoints to the frontend, validates tokens with the IdP for secure operations, and interacts with the database to manage tasks. This includes creating, updating, deleting, and retrieving tasks, with support for sorting and filtering. This component uses a `python:3.12.7-alpine` docker image.
- **Database**: The application uses a PostgreSQL relational database to store and manage data, including user information and tasks. It interacts with the API to execute CRUD operations and ensures consistent and reliable data storage. This component uses a `postgres:15` docker image.
- **Docker & Docker compose**: To simplify deployment and ensure consistent environments, the entire application is containerized using **Docker**. **Docker Compose** is utilized to orchestrate the various components into a seamless system.

DATABASE ARCHITECTURE

The database for this application consists of a single table, **Tasks**. This approach was chosen because there is no need to store additional user information, such as profile pictures or usernames, as these details are already included in the access token provided by the IdP.

The table includes a **user_email** field, which ensures that each task is associated with a specific user. This guarantees that users only have access to their own tasks, enhancing security and simplifying the data model. The model has the following attributes:

- **id**: An integer field that serves as the primary key and index for the task.
- **title**: A varchar field to store the title of the task, indexed for faster lookup.
- **description**: A varchar field for a brief description of the task.
- **timestamp**: A bigint field that stores the creation timestamp of the task.
- **updated_at**: A bigint field that stores the timestamp of the last update made to the task.

- **deadline**: A bigint field to store the deadline of the task, in Unix timestamp format.
- **priority**: A varchar field to store the priority of the task.
- **status**: A varchar field to store the current status of the task—.
- **completed**: A boolean field to indicate whether the task has been completed or not.
- **user_email**: A varchar field that associates the task with a specific user based on their email.

Fig.2 - Database table diagram

DEPLOYMENT ARCHITECTURE

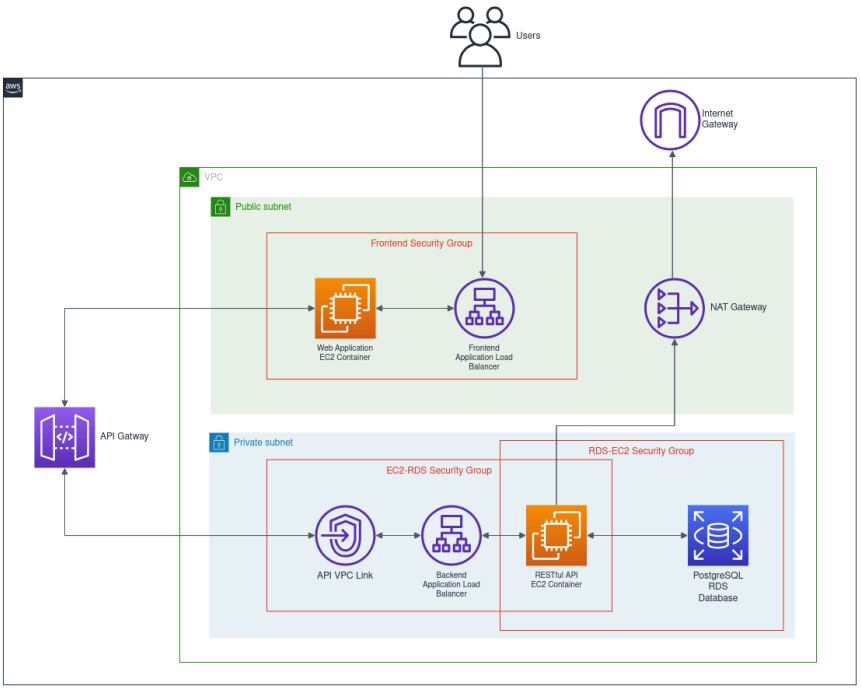

Fig.3 - Deployment architecture diagram

The deployment architecture described above represents how the application is deployed on AWS. This architecture ensures a scalable, secure, and reliable deployment of the application, with proper separation of concerns between frontend and backend services.

Most of the components of the application are hosted within an **Amazon Virtual Private Cloud (VPC)**, where both private and public subnets are created. This architecture ensures that instances remain isolated and only interact with the necessary components, improving both security and performance. The infrastructure is spread across two different Availability Zones (us-east-1a and us-east-1b), ensuring high availability and fault tolerance in case of failure of any instance or component.

Inside the VPC, there are two main subnets: a **private subnet** and a **public subnet**, which can be viewed as the "*backend*" and "*frontend*" subnets, respectively.

The **private subnet** hosts an **Amazon Relational Database Service (RDS) instance** with a **PostgreSQL** database. The database is hosted in the private subnet as it is only accessed by the backend API, so it remains isolated from external access for security reasons. This isolation ensures that sensitive data is not exposed to the public internet, protecting it from unauthorized access.

Within the private subnet, there is also an **Amazon Elastic Compute Cloud (EC2) instance** running the **RESTful API**. This EC2 instance is responsible for processing incoming API requests and interacting with the database to handle CRUD operations.

The API instance communicates with the internet through a **Network Address Translation (NAT) Gateway** in the public subnet, which allows the EC2 instance to validate tokens with the **IdP**. This configuration allows the backend API to communicate externally while remaining isolated from direct public access.

Inside the private subnet, two **security groups** control the communication between the EC2 instance and the RDS database. The **EC2-RDS security group** defines rules that allow inbound **TCP traffic** on port *5432* (the default PostgreSQL port) from the EC2 instance to the database. Similarly, the **RDS-EC2 security group** allows outbound **TCP traffic** from the RDS instance on port *5432* to the EC2 instance.

An **internal Application Load Balancer (ALB)** is configured within the private subnet to manage traffic between the EC2 instance running the API and clients accessing the API. The ALB is set up with a target group that includes the EC2 instance. When the ALB receives an incoming request from the API Gateway, it forwards the request to the appropriate EC2 instance based on the listener rule, determining the appropriate protocol and port for communication. Additionally, the **VPC Link** allows secure communication between the **API Gateway** (located outside the VPC) and the private subnet, ensuring that HTTP requests from the public API Gateway can access the private EC2 instance securely, without exposing the internal resources to the public internet.

In the **public subnet**, a **NAT Gateway** is hosted to enable outbound internet access for the resources in the private subnet, more specifically to the EC2 instance running the backend API. The **NAT Gateway** allows the private EC2 instance to connect to the internet by routing outbound traffic through the **Internet Gateway**, while still keeping the backend instance hidden from direct external traffic. Also in the public subnet, an **EC2 instance** is deployed to run the **frontend application**. This EC2 instance hosts the user interface, allowing end-users to interact with the system. It communicates with the backend API located in the private subnet via the API Gateway.

The **public ALB** is configured with an **HTTPS listener** and serves as the entry point for external users accessing the web application. It terminates the incoming connections using a **server certificate** provided by the class instructor. After decrypting the requests, the ALB forwards them to the target EC2 instance in the public subnet, enabling communication between the frontend and backend.

The ALB and the EC2 instance hosting the frontend are part of the same security group, and a route is set up to allow **HTTP traffic on port 80** from the ALB to the EC2 instance.

The use of both private and public subnets, along with the configuration of NAT and Internet Gateways, ensures that sensitive backend services like the database and API remain isolated while still providing external access through the frontend.

IMPLEMENTATION

In this section I'll outline some details of the implementation that I consider important steps on the development of this application or that I consider extra features, such as user authentication, application protection, user feedback, data visualization and documentation.

USER AUTHENTICATION

To simplify the implementation of AAA mechanisms, the system uses **Google OAuth**. The frontend application integrates authentication functionality using the @react-oauth/google library.

To leverage the GoogleIdP, a project was first created in the **Google Cloud Console**. Within the console, the **OAuth consent screen** was configured, followed by the creation of a **web client ID**. This setup included specifying the authorized JavaScript origins and redirection URLs. Upon completion, the Google Cloud Console generated a **client ID**, which was then used in the frontend.

The client ID was integrated into a `<GoogleOAuthProvider />` component that encapsulates the entire application. The **Login Page** includes a `<GoogleLogin />` component, configured with *onSuccess* and *onError* callback functions to handle responses from the IdP. If authentication is successful, the IdP returns a **JSON Web Token** (JWT). This token is stored in a **Zustand store** (*userStore*), and using the jwt-decode package, the token is decoded to extract user information, which is also saved in the *userStore*. Once authenticated, the user gains access to the dashboard. In the event of a failed authentication, an error message is displayed to the user via a toast notification in the bottom-right corner of the screen.

When the user interacts with the API, the JWT is included in the **Credential header** of the request. On the backend, the token is validated using the google-auth Python library, which communicates with the Google IdP to verify its authenticity. If the token is valid, the API processes the request; otherwise, a **401 Unauthorized** response is returned with an "Invalid Token" message.

APPLICATION PROTECTION

To ensure the application is secure, several security mechanisms have been implemented:

- **Protected Routes**: In the web application, certain pages, such as the **dashboard**, are protected routes, meaning they are only accessible to authenticated users. If an unauthenticated user attempts to access the */my* route (dashboard), the application redirects them to the **login page** with an error message indicating they must log in first. Conversely, the **landing page** and **login page** are not protected and can be accessed by anyone.
  To implement this functionality, the *userStore* in the frontend maintains a user state that includes an *isLoggedIn* flag. This flag is initially set to *false*. When a user successfully logs in, the flag is updated to *true*, granting them access to protected pages like the dashboard. The `<RequireAuth/>` component handles this verification by checking the *isLoggedIn* flag. If the flag is *false*, the component redirects the user to the login page; otherwise, the user can proceed to the dashboard.
- **Protected Endpoints**: The **RESTful API** enforces security by verifying the token included in the *Credential* header of each request. A custom decorator, *@authenticated*, was created to handle this validation. If a token is not present in the request headers, the endpoint raises a **401 Unauthorized** error with an appropriate message.
  If a token is found, the decorator calls the *validate_credential* function from the **google-auth** Python library (discussed earlier) to validate the token with the IdP. If the token is invalid, a **401 Unauthorized** response is returned with an error message. If the token is valid, the request proceeds, allowing authorized users to access the requested resource.

USER FEEDBACK

A key feature of the application is its ability to provide users with immediate feedback through the **Toast** component. This component displays messages with a visual indicator based on the status of the feedback, ensuring clarity and enhancing user experience. The color scheme for the Toast messages is as follows: **red** for errors, **orange** for warnings, **blue** for informational messages, and **green** for success notifications.

For example, when an unauthenticated user attempts to access the **dashboard** page, the application redirects them to the **login page** and displays an error Toast notification, informing them that they need to log in first. This mechanism helps users understand what went wrong and guides them toward corrective action.

I found that providing consistent and clear feedback significantly reduces user errors and improves overall usability. The Toast system plays a critical role in standardizing the application's feedback mechanism. It is utilized not only for error handling but also for confirming successful actions, such as when a user creates or updates a task. This ensures that users are always aware of the outcome of their actions, making the application more intuitive and user-friendly.

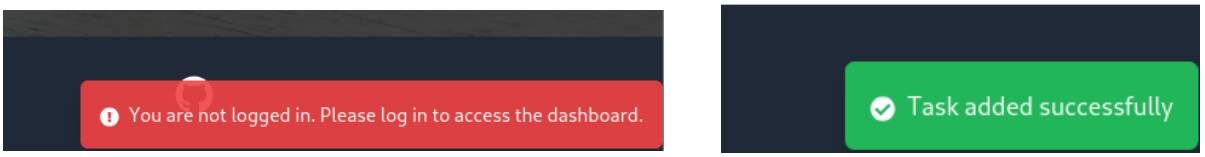

Fig.4 - Toast user feedback

DATA VISUALIZATION

A significant aspect of this application is its versatile and powerful task views. The application offers three distinct views that users can choose from: **Gallery View**, **List View**, and **Kanban View**.

1. **Gallery View**: This is the default view, where each task is displayed as a card containing all relevant details. It provides a visually appealing layout, ideal for users who prefer a graphical interface.
2. **List View**: This view displays tasks in a tabular format, with each task represented as a single row. This format is particularly useful for users familiar with software like Jira, offering an organized, spreadsheet-like interface.

3. **Kanban View**: This view categorizes tasks by status into three distinct Kanban columns. It is designed for users who prefer task tracking through a workflow-oriented approach.

These views were made possible by decoupling task data from the UI components responsible for rendering the views. To achieve this, I utilized a **Zustand store** called *taskStore* to centralize and manage shared task-related information, including the selected view type, sorting and ordering preferences, and status or priority filters. This decoupled approach allowed the application to **dynamically adapt** the task display without tightly **binding the data** with specific UI components.

Previously, methods for editing and deleting tasks were directly embedded within UI components, such as the **Card** component in the Gallery View. By abstracting these functionalities and passing them as arguments to the UI components, I was able to make the application easily to **support additional views**. This decoupled approach not only streamlined the implementation of the existing views but also ensured **scalability**, enabling future view additions with minimal refactoring.

Another critical decoupling involved handling filtering and sorting on the **backend** rather than the **frontend**. This approach improves **performance** by minimizing the workload on client devices, ensuring smooth operation even when managing **large datasets**. It is also more efficient since only the filtered and sorted data is transmitted, **optimizing** network usage and reducing **response times**. Processing on the backend ensures consistency, providing identical results for the same filters and sorting criteria across all users. Furthermore, it enhances scalability and security by centralizing data processing. This makes it easier to **add new filters** or **criteria** and ensures that sensitive data remains secure, with only authorized information sent to the client.

DOCUMENTATION

All the endpoints developed for this project are documented using Swagger. The API documentation is accessible through the */docs* endpoint at the link https://jn1kiib3wl.execute-api.us-east-1.amazonaws.com/docs, provided the EC2 instance hosting the API is running.

In cases where the API instance is down or inactive, the documentation can still be accessed using a pre-generated JSON file named *doer_api_docs.json*, located in the *docs* folder of the project. This file can be loaded into the https://editor.swagger.io/ by navigating to **File > Import File** and selecting the JSON file.

FUTURE WORK

Despite all the work completed, some features remain unfinished due to time constraints. These include:

- **Status view on Kanban:** Displaying tasks grouped by their current status within the Kanban board.
- **Grouping tasks by parameters:** Allowing users to organize tasks based on specific attributes such as priority or deadline.
- **Drag-and-drop functionality on Kanban:** Enabling users to intuitively reorder tasks within and across status columns.
- **Enhanced error handling on the frontend:** Improving the system's ability to manage unexpected errors and provide clearer feedback to users.

LINKS

Github link:
https://github.com/Dan1m4D/DOER-ES

Jira dashboard link:

https://es-individual-project-107603.atlassian.net/jira/software/projects/DOER/summary

Deployed solution:

- Web UI:
  - HTTPS (follow tutorial on project README.md): https://es-ua.ddns.net/
  - HTTP (load balancer link without https listener):
    - http://alb-doer-frontend-570455431.us-east-1.elb.amazonaws.com/
- API documentation: https://jn1kiib3wl.execute-api.us-east-1.amazonaws.com/