IAP Assignment

Guilherme Amorim, MEI, 107162

Date: December 10, 2024

Contents

1 Introduction **4**

2 Development **4**
2.1 Sprint 1: Setup and Initial Development . . . 5
2.2 Sprint 2: User Interface (UI) functionalities and Authentication . . . 6
2.3 Sprint 3: Final Deployment . . . 7
2.3.1 Database (DB) . . . 7
2.3.2 Application Programming Interface (API) . . . 7
2.3.3 UI . . . 7

3 Deployment **8**
3.1 Securing the React Frontend with Hyper Text Transfer Protocol Secure (HTTPS) . . . 8
3.2 Securing the Backend API with HTTPS . . . 8

4 Architecture **9**
4.1 UI . . . 9
4.2 API and DB . . . 9

5 Conclusion **10**

List of Figures

1 Sprint 1 overview . . . 5
2 Sprint 2 overview . . . 6
3 Sprint 3 overview . . . 7
4 Architecture Diagram . . . 9

Glossary

AAA	Authentication, Authorization, and Accounting
ALB	Application Load Balancer
API	Application Programming Interface
AWS	Amazon Web Services
DB	Database
EC2	Elastic Compute Cloud
IDP	Identity Provider
HTTP	Hyper Text Transfer Protocol
HTTPS	Hyper Text Transfer Protocol Secure
RDS	Relational Database Service
SCP	Secure copy Protocol
SSL	Secure Sockets Layer
UI	User Interface

1 Introduction

The goal of this project was to design, develop, and deploy a task management application, using Amazon Web Services (AWS). The project aimed to provide an experience in building a web-based solution that integrates an API, a user-friendly UI, and a DB, including the implementation of Authentication, Authorization, and Accounting (AAA) mechanisms, using Cognito as the Identity Provider (IDP), to allow assimilate secure practics.

This document outlines the development process, from the initial planning phase following an Agile workflow with defined epics and user stories, to the final deployment of the solution on AWS.

2 Development

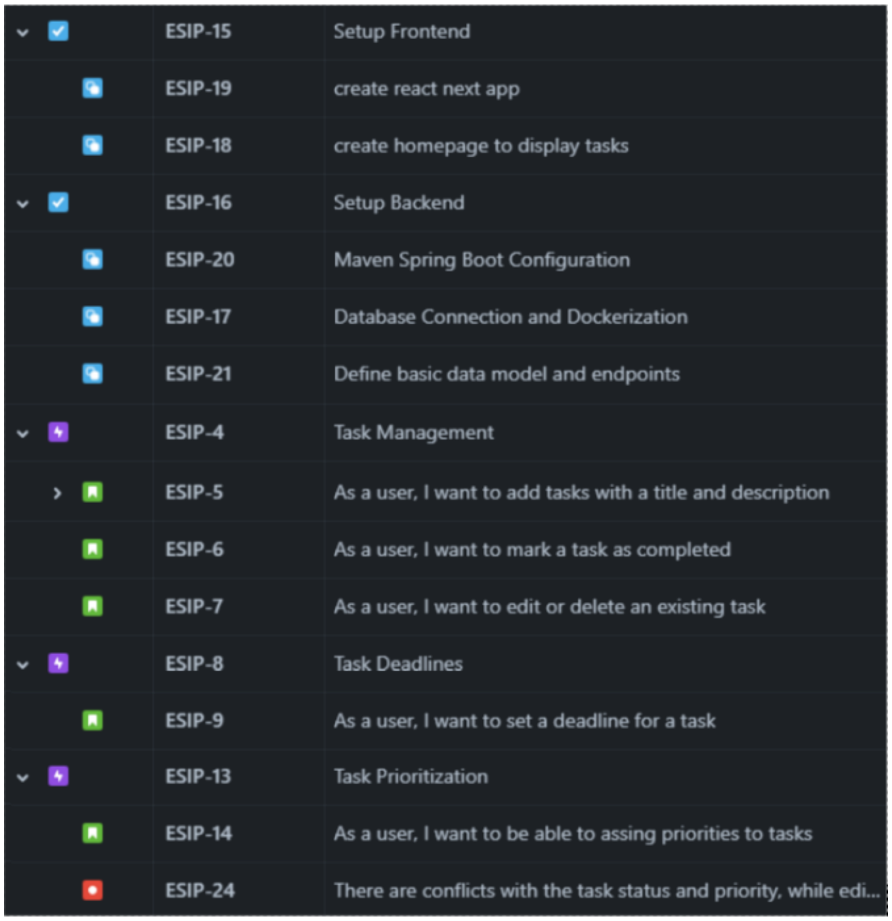
The development process was guided by an Agile Workflow approach using JIRA. First, I started by defining the essential epics for the pretended solution:

- **Task Ownership**: Each user must only have access to his tasks
- **Task Management**: The user should add, edit or delete a task
- **Task Deadlines**: The tasks can have deadlines
- **Sorting and Filtering**: The application should be accessible and provide the user a way to sort and filter the tasks
- **Task Prioritization**: The tasks can be more or less important.

From these epics, I defined the user stories and the corresponding tasks. The project was divided into 3 sprints.

2.1 Sprint 1: Setup and Initial Development

This one was focused on setting up the development environment and initial frontend and backend functionalities, that included the epics **Task Management**, **Task Deadlines**, and **Task Prioritization**.



ESIP-15	Setup Frontend
ESIP-19	create react next app
ESIP-18	create homepage to display tasks
ESIP-16	Setup Backend
ESIP-20	Maven Spring Boot Configuration
ESIP-17	Database Connection and Dockerization
ESIP-21	Define basic data model and endpoints
ESIP-4	Task Management
ESIP-5	As a user, I want to add tasks with a title and description
ESIP-6	As a user, I want to mark a task as completed
ESIP-7	As a user, I want to edit or delete an existing task
ESIP-8	Task Deadlines
ESIP-9	As a user, I want to set a deadline for a task
ESIP-13	Task Prioritization
ESIP-14	As a user, I want to be able to assing priorities to tasks
ESIP-24	There are conflicts with the task status and priority, while edi...

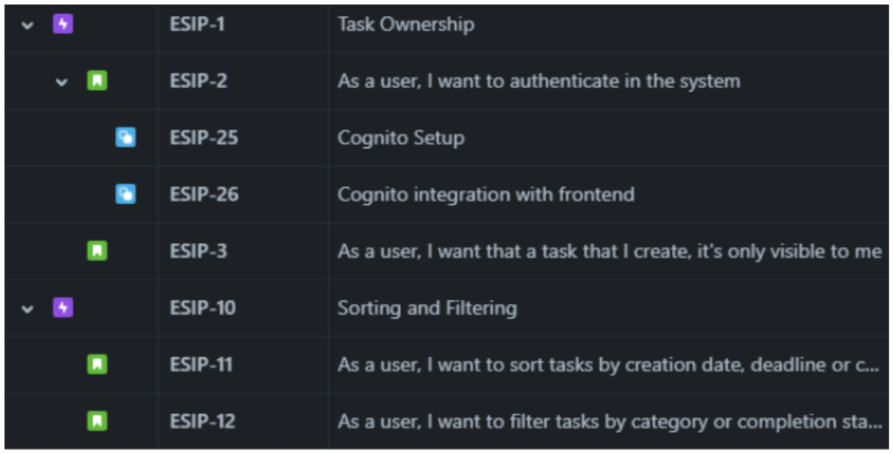
Figure 1: Sprint 1 overview

The first step was to design a small prototype of the UI, using React. Therefore, I could have a better idea of the system and help me to develop the API. It was created with Spring Boot and connected to a Docker instance running Postgres.

After these tasks were done, it was time to start implementing user stories – 5 in total, counting 5 story points. There were a total of five user stories, estimated at five story points, all related to the CRUD functionalities for tasks. During this phase, a bug was identified: while editing a task, conflicts were discovered between the variables managing the task's status and priority.

2.2 Sprint 2: UI functionalities and Authentication

The second sprint addressed the epics **Task Ownership** and **Sorting and Filtering**.



ESIP-1	Task Ownership
ESIP-2	As a user, I want to authenticate in the system
ESIP-25	Cognito Setup
ESIP-26	Cognito integration with frontend
ESIP-3	As a user, I want that a task that I create, it's only visible to me
ESIP-10	Sorting and Filtering
ESIP-11	As a user, I want to sort tasks by creation date, deadline or c...
ESIP-12	As a user, I want to filter tasks by category or completion sta...

Figure 2: Sprint 2 overview

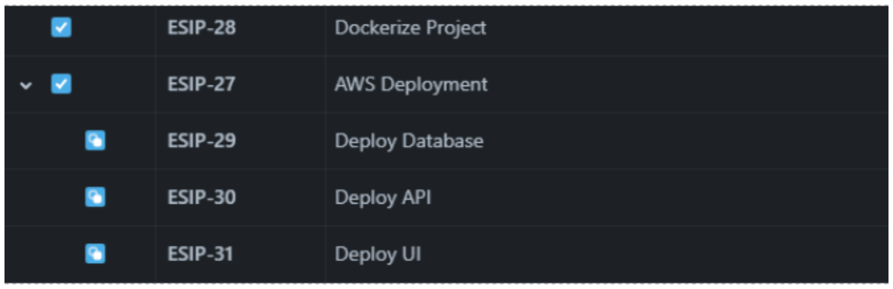
There were developed some functionalities at frontend level, to allow the user to sort and filter the tasks. After that, I focused on the authentication topic, using Cognito.

In total, 4 user stories were completed, with an estimate cost of 8 story points.

At the end of this sprint, all core functionalities had been successfully developed and Cognito authentication was fully integrated in the local environment.

2.3 Sprint 3: Final Deployment

The goal of the third and last sprint was to deploy the developed solution using AWS services.



✓	ESIP-28	Dockerize Project
✓	ESIP-27	AWS Deployment
	ESIP-29	Deploy Database
	ESIP-30	Deploy API
	ESIP-31	Deploy UI

Figure 3: Sprint 3 overview

Before initiating the AWS deployment, a preliminary task was created to virtualize the entire project. This step had the objective of validating that the application could run as a unified solution without any issues. As everything went well, I proceeded to the deployment.

2.3.1 DB

For the DB layer, Relational Database Service (RDS) was chosen with PostgreSQL 17 as the engine. The primary reason for choosing this service was the ease of creating the DB, once it simplifies the process of setting up a DB with just a few configurations.

2.3.2 API

The backend, built with Maven Spring Boot, was deployed using Elastic Beanstalk. This service was selected for its ability to simplify application deployment: the backend was packaged as a JAR file using the Maven command: *mvn clean package*. The resulting JAR file was then uploaded to Elastic Beanstalk, where it was automatically deployed to an Elastic Compute Cloud (EC2) instance. This service was ideal for deploying the Spring Boot API as it abstracted the complexity of manually configuring the infrastructure.

2.3.3 UI

Initially, Amplify was considered for deploying the React Next app, once that many articles and websites recommended this way due to its direct integration with modern web frameworks, like React.

However, Amplify is not available in the AWS Learner Labs environment, which restricted its usage for this project. Following the guidance from Professor Rafael, the decision was made to use EC2 for the deployment instead.

To deploy the React application, the project directory was copied from the local machine to the EC2 instance using Secure copy Protocol (SCP). Once transferred, the necessary modules were installed directly on the EC2 instance and the application was started, running the app on port 3000. To ensure that the application was accessible on standard Hyper Text Transfer Protocol (HTTP) ports, I also installed nginx and configured it as a reverse proxy to redirect requests on port 80 to the React application running on port 3000 of the EC2 instance's localhost.

3 Deployment

The foundation for the deployment strategy has been explained in the previous section. However, due to certain challenges encountered during the deployment process, adjustments were required. These challenges and their respective solutions are detailed above:

3.1 Securing the React Frontend with HTTPS

Although the React application was accessible through the public domain of the EC2 instance, the login functionality was not working. This was because Cognito requires secure communication over HTTPS to handle authentication processes. To resolve this issue, an Application Load Balancer (ALB) was introduced in front of the EC2 instance to enable HTTPS access.

An Secure Sockets Layer (SSL) certificate was registed on the ALB for the domain https://es-ua.ddns.net. The ALB was then configured to expose HTTPS traffic and forward the requests to the EC2 instance running the React application on port 80.

3.2 Securing the Backend API with HTTPS

Like the frontend, the API was initially accessible only over HTTP through the public domain of the EC2 instance managed by the Elastic Beanstalk environment hosting the Spring Boot API. Although this setup allowed the API to function, it presented a compatibility issue once that now, the UI was secured with HTTPS and only could make requests to HTTPS endpoints.

To address this, an API Gateway was introduced to act as an intermediary. It was configured to expose the API over HTTPS, redirecting requests to the HTTP endpoint of the EC2 instance running the API.

4 Architecture

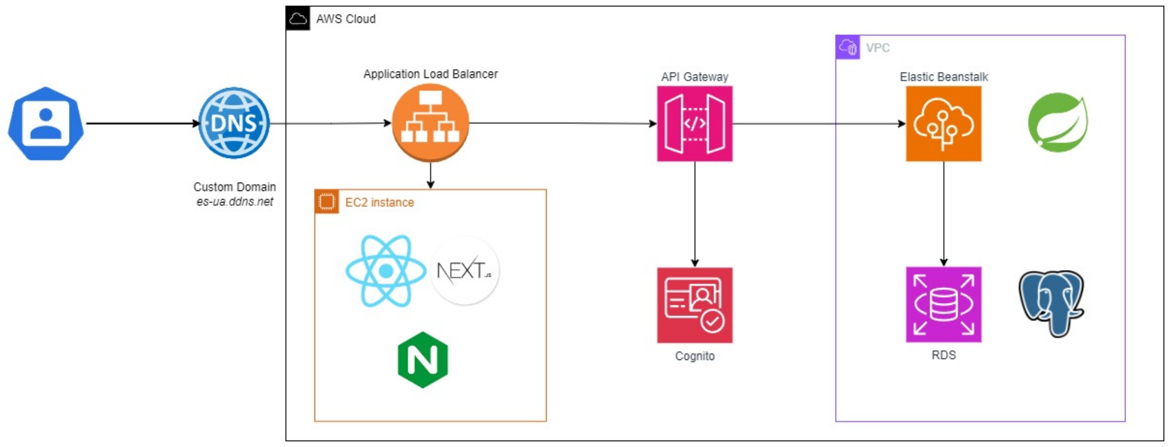


Figure 4: Architecture Diagram

The architecture of the solution integrates multiple AWS services. So, let's recap the components of the overall solution. The architecture is composed of three primary layers:

- UI - a React Application responsible for handling user interactions;
- API - Spring Boot application that processes logic and communicates with the DB.
- DB – PostgreSQL Instance that stores and manages data.

4.1 UI

The React application runs locally on port 3000 within the EC2 instance. Nginx is configured to redirect requests from port 80 to the application running on port 3000. To ensure HTTPS, an ALB uses an SSL certificate for the domain https://es-ua.ddns.net to forward HTTPS requests to the UI instance, at port 80.

While the use of Nginx was maintained in this solution, it is worth noting that the ALB could have been configured to forward traffic directly to port 3000 of the EC2 instance. In this case, Nginx would not be necessary, however, since Nginx was already implemented, I decided to keep it in place to avoid code refactoring.

4.2 API and DB

To deploy the API and DB, I created a VPC hosting the components of the application.

The Spring Boot API was deployed using Elastic Beanstalk, which automatically provisioned an EC2 instance to run the backend service. All the API requests were routed through an API Gateway that exposes the API over HTTPS, redirecting the requests to the HTTP endpoint in the instance. This component was very important to enable communication between the frontend and the backend, since it's only possible to make requests to a HTTPS API from a HTTPS host. For the DB, I chose RDS with PostgreSQL 17 as the engine, that receives requests from the API using its internal endpoint.

5 Conclusion

This project was an interesting experience, as it gave me the opportunity to work in a topic of high importance in software development: the process of deploying a complete solution, as well as a reinforcement of my knowledge about Agile development methodologies.

That said, I believe that with more time, I could have focused more on implementing stronger security practices, which would be the next step for this project. For example, in the current AWS environment, there are no restricted access configurations to URLs, and public and private networks are not defined. Additionally, some sensitive information, such as usernames, passwords, and URLs, is explicitly present in the code, which poses a potential security risk.

Since the primary goal of this project was to apply Agile practices and learn about AWS deployment, I chose to focus on this topics and, consequently, security was not prioritized. However, if the project were intended for production, this aspect would need to be addressed rigorously, once that for a real-world application, the security of data is critical.

It is also worth noting that the use of the AWS Learner Labs environment introduced some constraints to the project. It came with limitations, such as restricted access to certain services and configurations. These limitations influenced the design choices and prevented the exploration of potentially better alternatives for some aspects of the solution.