ES Individual Project report

Bruno Páscoa, brunopascoa03@ua.pt

November 14, 2024

Abstract

Report for the ES Individual Project

1 Introduction

This document reports the results of the work performed regarding the Individual Assignment for the ES course. For this project, I was tasked to build a full-stack "To-do List" full-stack (api and ui) solution and deploy it in the cloud via AWS.

2 Development rules

(Note: both of these definitions are only meant to be applied

2.1 Jira issue nomenclature

In order to separate the different type of work that needed to be done, I have used the following Jira issue types:

- **Epics:** these were used to divide the remaining issues into 3 main categories:
  - Authentication
  - Task Management
  - AWS Setup
  - Documentation
- **User stories:** Designate tasks that require actual coding and bring value to the customer. Only user stories are subjected to the definitions of done and ready, as they are the only ones that require any coding.
- **Tasks:** Any type of work that does not involve direct coding. However, despite not producing any actual code, these tasks are still important and still require time, which is the reason I have decided that they should still be recorded.
- **Sub tasks:** Used to divide other issues into smaller steps. These are used exclusively for adding clarity and tracking progress within the same user story (as splitting the user story would lead to a lot of interdependence).

2.2 Definition of Ready

- Story must have an assigned epic
- Story must have a story point estimation (used scale: 1: should be able to do multiple in one day, 3: should be able to complete in a day even if bugs appear, 5: may take more than one day)
- Story must have a description (in the "as a... I want to... in order to..." format)

- The code must not be developed in the main branch (creating a new branch with the relevant issue key is recommended but not required).

(Note: the final rule does not force to use a separate branch per issue, as some issues either were developed simultaneously either for ui layout issues (filtering and sorting) or because it made testing the ui easier (adding and listing issues).

2.3 Definition of Done

Before pull request:

- Code must be "functional" (as in, the code must not have any unexpected errors and must do what the tasks asks)

Before merge:

- Code must pass all tests (minimum requirement is manual testing, but automated testing must be done whenever possible)
- API Documentation must match the new additions

After merge:

- AWS containers (if any exist at the time) must be updated and the developer must confirm that they are running properly.

2.4 Testing methodology

In terms of testing, I have concluded that automated ui testing is unnecessary for this small scale project, however manual testing is required.

Moreover, any task that is considered "automatable" (as in, any task that is either read-only or whose changes can be automatically reverted, which excludes only authentication) should have unit tests created for it that can either target the services directly or do a more integrated approach where the API is called directly.

As the focus of this subject is deploying to AWS and not testing, I have also decided not to test the API in isolation (with mocked services).

3 Development

3.1 Sprint overview and user stories implemented

(Note: as this project was made using Agile template and, because of that, using story points as metric is not possible)

3.1.1 Sprint 1: Basic setup and planning

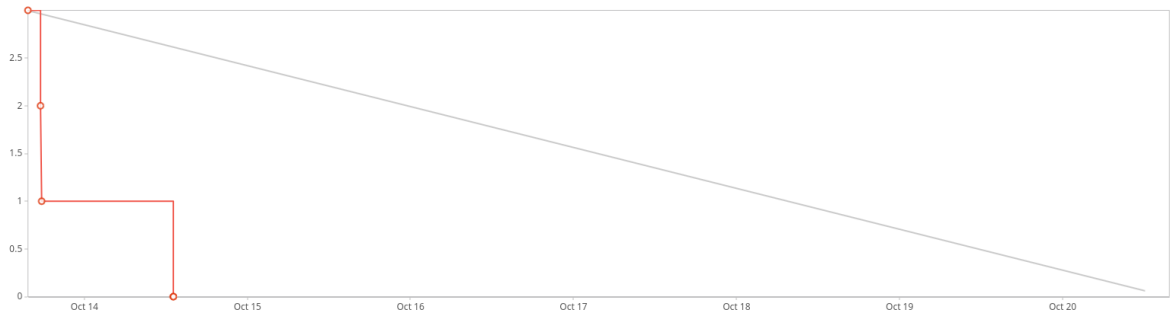

Figure 1: Sprint 1 burn down chart

Overview: as this sprint was mostly for initial setup, there were no user stories implemented, however, having a basic plan in mind helped a lot when it came

3.1.2 Sprint 2: Adding and listing tasks

Issue key	Story name	Description	Story points
KAN-1	Allow users to authenticate using Cognito	As a user, I want to authenticate, so that I can interact with my tasks	1
KAN-2	Allow users to see their tasks	As a user, I want to see the tasks I created, so that I know what needs to be done	1
KAN-3	Allow users to add tasks	As a user, I want to add new tasks, so that I can check them later	3

Table 1: Sprint 2 user stories

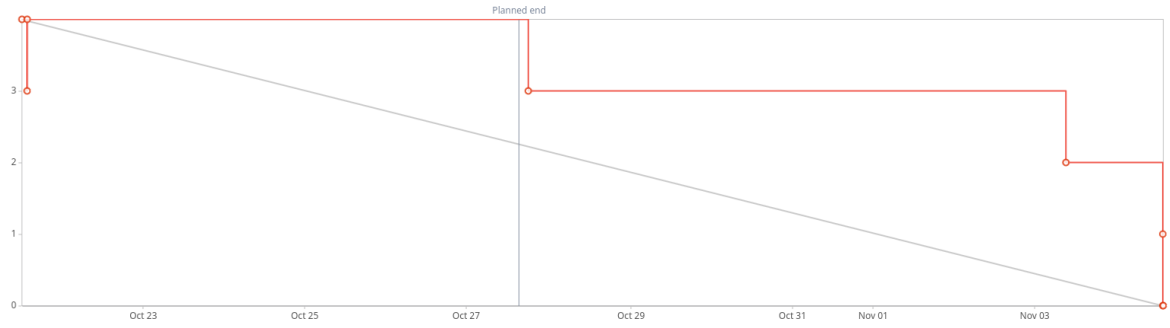

Figure 2: Sprint 2 burn down chart

Overview: Overall, I believe that this sprint went well, however, I did have to take 1 more week than expected, as I realized I would not meet the deadline for another course if I didn't leave less priority work behind. (Note: The drop at the beginning was due to accidentally marking an issue as "done" instead of "in progress" and then swiftly reverting that mistake)

3.1.3 Sprint 3 (current): Finalizing

Issue key	Story name	Description	Story points
KAN-4	Allow users to edit/delete tasks	As a user, I want to edit and delete tasks, so that I can fix mistakes	1
KAN-5	Allow users to mark tasks as completed	As a user, I want to mark tasks as completed, so I know distinguish what I have already done from what I still need to do	1
KAN-6	Allow users to sort tasks by creation date, deadline or completion status	As a user, I want to sort tasks, so that I can find the task I want more easily	1
KAN-7	Allow users to filter tasks by category or completion status	As a user I want to filter tasks by category or completion status, so that I can find the tasks I need more easily	1

Table 2: Sprint 3 user stories

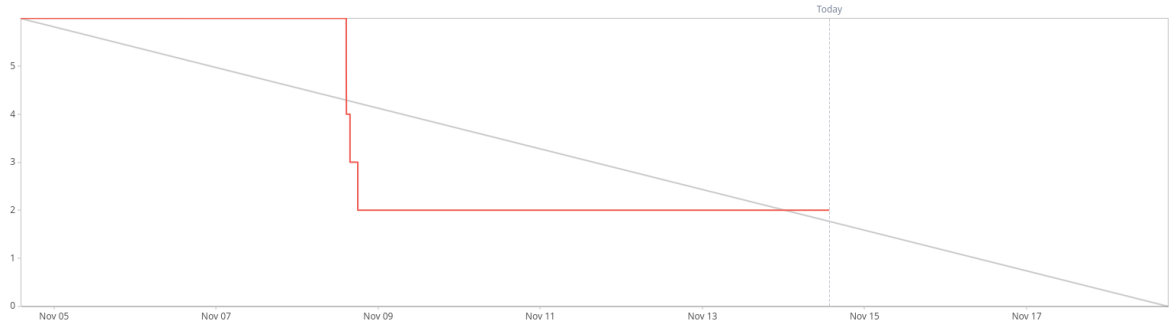

Figure 3: Sprint 3 burn down chart

Overview: at least so far, I can confidently say that I did all that I needed in order to deliver a good product. The sharp decrease seen in the burn down chart can be explained by the fact that none of the tasks took a lot of time, allowing me to do all 4 in a single day of work. (Note: the sprint is not considered over as I still need to finish the documentation and demo tasks)

3.2 AWS architecture and design choices

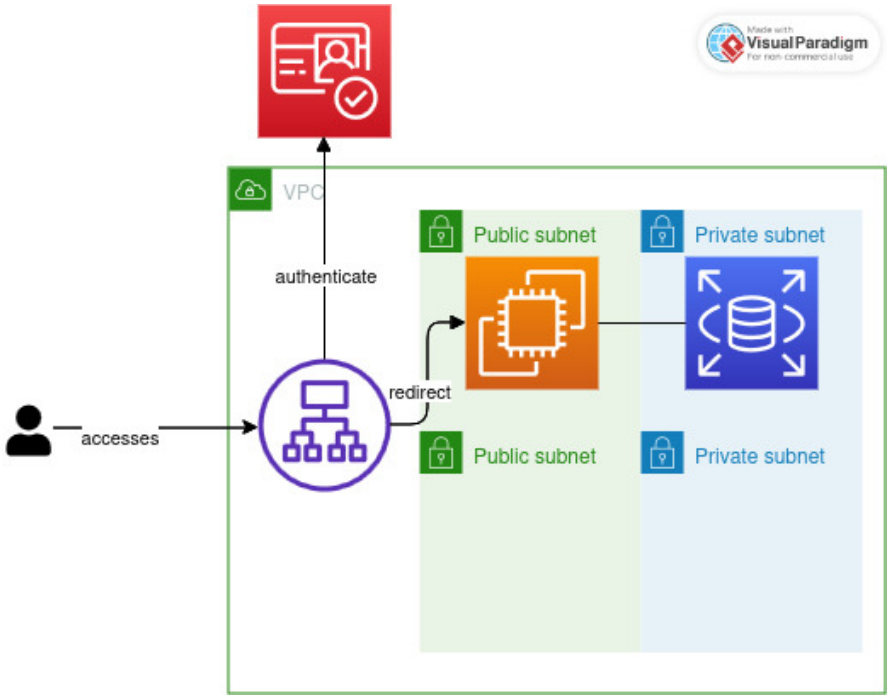

Figure 4: AWS architecture

In terms of AWS architecture, I have used the following services:

- **Application Load Balancer:** provides redirecting to the EC2 instance, as well as automating the use of Cognito.
- **Cognito:** chosen OAauth2 IDP for authentication.
- **EC2:** chosen container to run the flask server. This was chosen over ECS because I already had some experience with EC2 from the group project, as well as the fact that the VSCode SSH extension allows me to use my usual VSCode interface not only to deploy, but also to fix any last-minute errors without having to rebuild anything.
- **RDS:** chosen database service. It stands out for basically working as MySQL on the cloud allowing me to switch between using a local database for testing and a cloud-based one almost seamlessly
- **VPC:** service required by most of the above for maintaining networks. I have chosen to create 2 public networks for the EC2 instances and 2 private for the RDS replicas (although, for reasons explained below, I have only used one of each).

3.3 Database architecture

Figure 5: Database architecture

While I initially considered modeling a class for the user, the fact that Cognito made that unnecessary, making the "Task" table the only one required to be stored.

The task table fields themselves weren't open to a lot of interpretation, as the requirements clearly spelled out what was needed.

3.4 Main workflows

To give examples of the main workflow, I have selected 2 representative workflows: authentication and adding a task

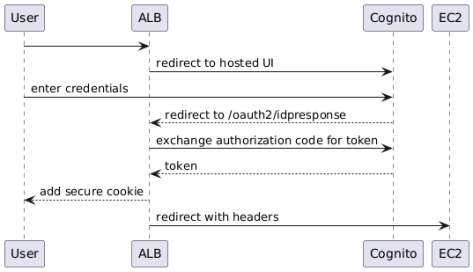

Figure 6: Authenticating with cognito

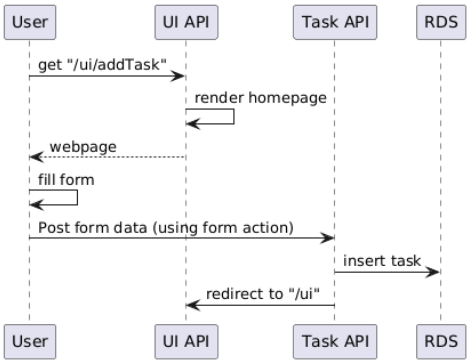

Figure 7: Add a new task

3.5 API Documentation

The API itself can be split into 3 different groups: auth, ui and tasks

- **auth:** legacy endpoint that was previously used for handling authentication via Cognito Hosted UI before I discovered that ALB can deal with the the authentication without needing for manual redirects. Despite this, the endpoints should still be functional and are sometimes still used for local testing.
- **ui:** endpoints that deal with the rendering and return of the web pages
- **tasks:** this endpoint deals with the actual task-related functionality (such as listing, adding new tasks,...) and can be used either indirectly via the ui or directly via a rest calls.

The full documentation can be found here. (this API was generated automatically via Flask-restx and then manually hosted via Swagger Hub, the API can also be accessed via the ALB's root ("/") endpoint)

4 Limitations

Due to the limitations posed by the learner labs restricted access, as well as their budget limits, I had to take the following decisions:

- As there's only 1 ui and 1 API service, I decided to serve both using the same Flask server in order to reduce the amount of EC2 instances to half
- While originally I had 2 replicas of the EC2 instances under the same ALB for redundancy, due to the costs nearing the maximum budget, I had to reduce this number to 1 instance only
- Only 1 RDS instance exists at a time, and no replication was done, in order to maintain budget

Were learner lab to have a higher budget and less restrictions (or if I also didn't have to take the Group Project's spending into account), these decisions would likely not have been made.

5 Additional notes

- Definitions of Done and Ready are also defined in the Project Pages section of Jira.
- The remaining documents describe the architecture of the project at project's start and may have slight differences compared to the current architecture, as I learned more about AWS during this project's development and adjusted accordingly.
- All diagrams used can also be viewed in the GitHub repository under docs/
- If authenticating via the ALB, the "log out" button may not work correctly (as the credentials, as the Flask can use either its own credentials or the ALB's).

6 Related Resources

- GitHub repository
- Jira Board
- Application root endpoint (displays API documentation)
- Application home page
- Swagger Hub API documentation
- Overleaf report