



**Politecnico
di Torino**

Politecnico di Torino

Computer Engineering

A.y. 2023/2024

Graduation session October 2023

**Development of a Web Application
for Project Management using
Amazon Web Services and
Microfrontends**

Supervisor

Luigi De Russis

Company Supervisor

Massimo Pacileo

Candidate

Alberto Baroso

Abstract

Project management involves monitoring a wide range of company assets: resources must be efficiently assigned to projects, which in turn must be completed in time to meet customer release deadlines. This thesis presents the comprehensive analysis, design, implementation, testing, and deployment of a web application aimed at replacing old project management processes based on Excel spreadsheets and email communications. The application's purpose is to streamline managers' and project managers' workflows by automating tasks, validating inputs, and allowing faster communication. The requirements were initially gathered by observing the existing solution and assessing the concerns of its users. The resulting design and implementation leveraged a serverless backend powered by Amazon Web Services. The front end was initially entirely developed in Flutter and later evolved into a microfrontends architecture. Finally, this thesis presents code quality metrics such as static analysis and coverage from automated tests, and it compares the performance of monolithic and microfrontend applications using Web Vitals.

Table of Contents

List of Tables	III
List of Figures	IV
Glossary	VI
1 Introduction	1
1.1 Context	1
1.2 Goals	2
1.3 Thesis structure	2
2 Analysis	4
2.1 Existing solution	4
2.2 Microfrontends	5
2.3 Requirements	6
2.3.1 Application users	6
2.3.2 Functional requirements	6
3 Technologies and services	9
3.1 Employed technologies	9
3.2 Comparison of technological alternatives	13
3.2.1 Database	13
3.2.2 RESTful VS GraphQL APIs	19
4 Design	20
4.1 Project structure	20
4.2 System architecture	21
4.3 Back end	22
4.3.1 Business logic	22
4.3.2 Lambda layers	23
4.3.3 Dead-letter queue	23
4.3.4 DynamoDB streams	24
4.4 Front end	24
4.4.1 Monolith	24
4.4.2 Microfrontends	24

4.5	Database	25
5	Implementation	28
5.1	Project management	28
5.2	Project setup	29
5.2.1	Git branches and Amplify environments	29
5.2.2	Amplify CI/CD	30
5.2.3	Application build	31
5.2.4	Application deployment	33
5.3	User Interface	35
5.4	Authentication	41
5.4.1	Custom user sign-up and sign-in forms	42
5.4.2	Corporate Single Sign-On with OAuth flow	42
5.5	Authorization	45
5.5.1	Pre-signup lambda trigger	45
5.5.2	Post confirmation lambda trigger	45
5.5.3	Manual user group assignment	47
5.6	CRUD operations	49
5.6.1	Resources	50
5.6.2	Business units and cities	52
5.6.3	Work planning and finalization	52
5.7	Microfrontends	55
6	Evaluation	57
6.1	Testing	57
6.1.1	Unit test	57
6.1.2	End-to-end test	58
6.1.3	Chaos test	59
6.1.4	Static code analysis	60
6.2	Monolith and Microfrontends performance	60
6.2.1	Web Vitals	60
6.2.2	Lighthouse reports	61
7	Conclusions	64
7.1	Results	64
7.2	Future Developments	65
	Sitography	69

List of Tables

2.1	Resource management user stories	7
2.2	Customer management user stories	7
2.3	Contact person management user stories	7
2.4	Release management user stories	7
2.5	Project management user stories	8
2.6	Project planning user Stories	8
3.1	DynamoDB and Amazon RDS comparison	14
3.2	Estimated size of entities	15
3.3	Cost comparison of DynamoDB and RDS	16
3.4	REST and GraphQL comparison	19
5.1	Roles and permissions	46
6.1	BluePlan line coverage and technical debt metrics	57
6.2	Monolith and microfrontend homepage Web Vitals comparison . . .	62
6.3	Monolith and microfrontend customer list Web Vitals comparison .	62
6.4	Monolith and microfrontend resource form Web Vitals comparison .	62

List of Figures

1.1	Blue Reply logo	1
4.1	Overall system architecture	21
4.2	GraphQL queries and mutations sequence diagram	22
4.3	Lambda invocation with GraphQL resolver sequence diagram	23
4.4	Database schema	26
5.1	Amplify environemnts relation with Git branches in Monolithic app	29
5.2	Amplify environments relation with Git branches in microfrontends	30
5.3	Amplify CI/CD pipeline steps	30
5.4	Application build architecture	31
5.5	Deployment of microfrontends	33
5.6	Retrieval of microfrontends	34
5.7	Home page UI	35
5.8	Customer list UI	36
5.9	Customer form UI	37
5.10	Release form UI	37
5.11	Contact person form UI	37
5.12	Project list UI	38
5.13	Project form UI	38
5.14	Resource list UI	39
5.15	Resource form UI	39
5.16	Project selection UI	40
5.17	Business unit list UI	40
5.18	Business unit form UI	41
5.19	Authentication flow with OAuth Code Grant	44
5.20	Pre-signup and Post confirmation lambda triggers	47
5.21	User group propagation from DynamoDB to Cognito	48
5.22	AssignUserToGroup sequence diagram	49
5.23	Creation of Resource with Projects	50
5.24	Update of Resource with Projects	51
5.25	Holidays retrieval and store	53
5.26	Microfrontends schema	55
6.1	Test Users Manager	59

Glossary

AOT Ahead-Of-Time	JIT Just-In-Time
API Application Programming Interface	LCP Largest Contentful Paint
AWS Amazon Web Services	PK Partition Key
CD Continuous Delivery	REST REpresentational State Transfer
CDN Content Delivery Network	RTT Round Trip Time
CI Continuous Integration	SDK Software Development Kit
CLS Cumulative Layout Shift	SK Sort Key
CSS Cascading Style Sheets	SPA Single Page Application
DBMS Database Management System	SSO Single Sign-On
DLQ Dead-Letter Queue	TBT Total Blocking Time
DOM Document Object Model	TCP Transmission Control Protocol
FCP First Contentful Paint	UI User Interface
FID First Input Delay	UX User Experience
FK Foreign Key	
HTTP HyperText Transfer Protocol	
IaC Infrastructure as Code	

Chapter 1

Introduction

1.1 Context

Project management is a crucial process for businesses, its goal is to efficiently allocate resources to deliver results to customers within deadlines. It requires appropriate supporting tools to provide managers and project managers with a complete overview of all company assets. Blue Reply [1] decided to develop an internal web application, **BluePlan**, to upgrade the old project management process based on Excel files and email communications.

Blue Reply is one of the main technology consulting companies of the Reply group. They collaborate with their customers to help them intercept the main technological trends, design and implement innovative solutions that differentiate themselves on the market.



Figure 1.1: Blue Reply logo

Blue Reply has worked with the biggest Italian and international companies belonging to the Finance (Banking, Credit, and Insurance), Manufacturing, Telco & Media, and Retail sectors thanks to which they have built their twenty-year expertise in the fields of Artificial Intelligence, Low Code, Cloud Computing, Microservices & DevOps, Internet Of Things, Big Data & Analytics, Cloud Native Architectures and eCommerce.

Project management is made challenging by projects having their unique historical context: some follow an agile methodology while others use the waterfall approach, the size of the teams vary greatly, and customers have different needs

from the accounting and reporting point of view. A custom-made web application can help provide a standard vision of all these heterogeneous projects.

1.2 Goals

The project has three main objectives:

- Develop a user-friendly web application to centrally and efficiently manage company resources and projects. This will improve efficiency by automating tasks and it will provide managers with a standardized view of various project metrics across clients.
- Design and set up a flexible architecture to support future functionalities, allowing the application to become a multi-purpose dashboard that can be used to manage other aspects of the business.
- Explore the possible usage of microfrontends to implement the user interface.

The development of a custom web application provides two major advantages over using existing solutions: it can be tailored to the specific business process inside Blue Reply and it also avoids big expenses for third-party management software licenses.

1.3 Thesis structure

The realization of the **BluePlan** application is described in chapters following the phases of the software development lifecycle. It is important to acknowledge that although they are described separately, these phases frequently overlap due to the iterative and agile nature of the work. The full division of chapters and their content is reported in the following list:

- Chapter 2, **Analysis**: The problems arising from the usage of Excel spreadsheets are explored in deeper detail. Users' concerns and requests are formalized into functional requirements. An overview of the current state of the art of microfrontends architecture is provided.
- Chapter 3, **Technologies**: The main technologies employed throughout the project are briefly described, then the rationale behind the choice of DBMS, database schema, and API type are explained.

- Chapter 4, **Design**: The overall architecture is laid out defining the major system components. The cooperation of services on the AWS back end is then explained as well as the definition of the UI using Flutter and microfrontends. Finally, the data model to support the use cases is delineated.
- Chapter 5, **Implementation**: Explores how the project itself was managed, how authentication and authorization are enforced, and the implementation of CRUD operations for the main database entities.
- Chapter 6, **Evaluation**: Results obtained from static code analysis and various types of automated tests are displayed and commented on. A performance comparison is then performed between the monolithic and the microfrontends implementations.
- Chapter 7, **Conclusions**: A summary of the overall development results is presented, explaining the findings and future works.

Chapter 2

Analysis

The analysis phase is essential for identifying the problems that the web app must address and the goals to be achieved. In this step, user and technical requirements are evaluated to define the scope and limitations of the project.

2.1 Existing solution

Currently, planning and finalizing work hours is performed with the support of Excel files. At the end of each week, managers and project managers need to send a message to all employees to remind them to compile a spreadsheet with all the hours they worked on every activity. Each month, these files are combined into a summary. This manual work is highly error-prone. Additional complexity arises from the lack of standard project names across all Excel files, this increases the probability of misunderstandings.

A web application to support this workflow would help mitigate human errors, automate tasks, and result in significant time savings. The app would also provide a better user experience to all workers and help management roles by giving them a more centralized view of everything happening in the business units. An initial implementation would mainly focus on project management, but defining a *microfrontends architecture*, described in section 2.2, will allow for a wide range of further developments:

- Collection of data about projects and work to perform analysis and optimize resource management.
- Recommendation system to help management roles easily identify the available and most suitable resources for a given project.
- Moving additional company processes to the app to automate them.
- Dashboard with graphs and automatic report generation.

2.2 Microfrontends

Microfrontend [2] is an architectural style representing an evolution of microservices architecture where the typical front-end monolith is split into smaller, more manageable components. Each application has its own continuous delivery pipeline. This modular approach not only simplifies the development process but also improves the maintainability and deployability of the individual codebases.

Central to this architectural pattern is the presence of a container application that addresses cross-cutting concerns like authentication and defines the page layout by composing front ends together. The various front ends can be assembled at different steps of the deployment flow:

- **Build-time integration:** Each application is published as a separate package . The container is responsible for composing the applications into a single one. This solution requires redeploying every time a new change occurs.
- **Run-time integration:** Microfrontends are assembled during application execution. It can be performed at various points in the client-server communication flow:
 - **Server-side rendering:** The server retrieves and organizes the front ends into a view later served to the client.
 - **Edge-side rendering:** The view is put together at the CDN level.
 - **Client-side rendering:** Microfrontends are assembled directly on the browser using iframes, web components, or embedding `<script>` tags.

Microfrontends offer numerous advantages, among them:

- **Autonomous teams:** Teams own full-stack applications, this results in faster feature delivery and improved maintainability.
- **Isolation:** Developments and deployments are independent of other applications.
- **Technology diversity:** Each team can choose the most suitable technology stack to support their requirements.

The microfrontends architecture increases the overall system complexity and poses various challenges. Visual consistency must be enforced among the various applications and CSS conflicts may arise. Shared libraries can act as styleguides that harmonize the appearance of the different components but special care must be taken to avoid multiple downloads of the shared dependencies.

2.3 Requirements

An initial set of core requirements has been drafted after inspecting the structure of the Excel files and after gathering insights from the managers and project managers using them. Due to the agile nature of the work, many requirements have been further refined throughout the weekly sprints.

A few technical requirements have been defined by the product owner itself: the use of Amazon Web Services as the cloud computing platform, and particularly the use of AWS Amplify to manage the cloud resources. For the front end side, no constraints were imposed other than having a couple of pages already designed using Flutter. The choice would be to either continue using this technology or find a solution to integrate it with other frameworks.

2.3.1 Application users

The application is intended to be utilized by three main user categories:

- **Managers:** can manage all aspects of the application.
- **Project Managers:** Assign *Resources* to projects, plan and approve *Resources* work.
- **Resources:** functional and technical analysts who can view the planned effort and report their finalized work.

Access to the platform must be limited to the business unit employees. A mechanism for user registration, authentication, and authorization must be devised.

2.3.2 Functional requirements

The following tables formalize all the actions that users of the application will perform. User actions have been grouped based on which database entity they are associated with. For each action, it's reported its intended actors and the estimated number of monthly read or write operations.

Tables 2.1, 2.2, 2.3, and 2.4 show that the *Manager* is the only role with full control over the company base assets. They have permission to perform all CRUD operations on *Resources*, *Customers*, *Contact persons*, and *Releases*, while *Project Managers* can only read these entities but not modify them.

Actors	User Story	Operations
Manager	Add a new resource	Write ~ 10/Month
Manager, PM	View all resources	Read ~ 1000/Month
Manager	Modify resource information	Write ~ 1/Month
Manager	Delete a resource	Write ~ 1/Month

Table 2.1: Resource management user stories

Actors	User Story	Operations
Manager	Create a new customer	Write ~ 10/Month
Manager, PM	View all customers	Read ~ 1000/Month
Manager	Modify customer information	Write ~ 1/Month
Manager	Delete a customer	Write ~ 10/Month

Table 2.2: Customer management user stories

Actors	User Story	Operations
Manager	Create a contact person for a customer	Write ~ 10/Month
Manager, PM	List all customer contact persons	Read ~ 100/Month
Manager	Modify contact person information	Write ~ 1/Month
Manager	Delete a contact person of a customer	Write ~ 10/Month

Table 2.3: Contact person management user stories

Actors	User Story	Operations
Manager, PM	Create a release for a customer	Write ~ 10/Month
Manager	List all customer releases	Read ~ 100/Month
Manager	Modify information of a release	Write ~ 1/Month
Manager	Delete a release of a customer	Write ~ 10/Month

Table 2.4: Release management user stories

Table 2.5 shows that both *Managers* and *project managers* are the allowed to perform all CRUD operations related to projects. To allow easier retrieval, projects can also be filtered by values of the entities they are associated with.

Actors	User Story	Operations
Manager, PM	Create a new project associated to a customer's release	Write \sim 100/Month
Manager, PM	List all projects	Read \sim 1000/Month
Manager, PM	Filter projects by: customer, release, project manager, and contact person	Read \sim 100/Month
Manager, PM	Modify project information	Write \sim 10/Month
Manager, PM	Delete a project	Write \sim 10/Month

Table 2.5: Project management user stories

Table 2.6 describes the system operations necessary to support two main actions:

- Allowing *Managers* and *PMs* to define the *Planned Work* of a *Resource* for a *Project* in a given month.
- Allowing the consequent finalization of the time a *Resource* has spent working on a given *Project* and month.

Actors	User Story	Operations
Manager, PM	View all projects to plan	Read \sim 100/Month
Manager, PM	Define planned hours for a resource in a month	Write \sim 100/Month
Manager, PM	View all resources assigned to a project to plan	Read \sim 1000/Month
Manager, PM	View summary of project allocation for each month	Read \sim 1000/Month
Manager, PM	View the finalized hours of each resource for each month	Read \sim 1000/Month
Manager, PM	Modify the finalized hours of a resource for a month	Write \sim 100/Month
Resource	View assigned projects	Read \sim 100/Month
Resource	View finalized work for previous months	Read \sim 100/Month
Resource	Report finalized hours for a project in a month	Write \sim 100/Month

Table 2.6: Project planning user Stories

Chapter 3

Technologies and services

3.1 Employed technologies

This section briefly showcases the frameworks, languages, tools, and services used while developing the application, defining a few key features for each of them.

Flutter

Flutter [3] is an open-source user interface (UI) software development kit (SDK) created by Google. It's designed for creating high-performance cross-platform applications, such as mobile, web, and desktop, using a single codebase. Flutter utilizes a widget-based development style, where the entire UI is composed of small, self-contained, and reusable building blocks called widgets.



Dart

Dart [4] is a programming language developed by Google, designed to build scalable applications. It's the primary language for developing applications using the Flutter framework. Dart utilizes a strong typing system and it includes a just-in-time (JIT) compiler which enables fast iterative development and hot reloading. In addition, Dart can be compiled into native machine code using ahead-of-time (AOT) compilation, resulting in highly optimized performance for production-ready applications.



JavaScript

JavaScript is an interpreted programming language widely used in web development. JavaScript can be employed for both front end and back end logic, in the context of this project, JavaScript was utilized for implementing the back end Lambda functions and in the front end to create dynamic and interactive content.



JavaScript logo

Angular

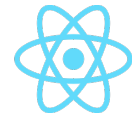
Angular [5] is a TypeScript-based framework used for building scalable web applications. It allows defining reusable components and it exploits the principles of dependency injection. Finally, Angular two-way data binding ensures efficient synchronization between data and the UI.



Angular logo

React

React [6] is a JavaScript library geared towards building user interfaces focused on rendering efficiency. It employs a component-based architecture that encourages the creation of encapsulated, reusable UI elements. React introduces a virtual *Document Object Model* (DOM), which optimizes rendering performance by minimizing direct manipulation of the actual DOM.



React logo

Vue

Vue [7] is a progressive JavaScript framework designed to build user interfaces. Vue employs a declarative syntax and component-based architecture. Vue's reactivity system efficiently tracks changes in data and automatically updates the UI.



Vue logo

Node.js

Node.js [8] is a cross-platform asynchronous JavaScript runtime environment built on top of the Chrome V8 Engine [9]. It allows the execution of JavaScript outside of traditional web browsers, in particular in this project it has been used to run Lambda functions on AWS.



Node.js logo

GraphQL

GraphQL [10] is a query language and runtime for APIs that was developed by Facebook. It provides a flexible and efficient approach to data fetching and manipulation, offering a powerful alternative to traditional RESTful APIs. Furthermore, GraphQL offers a strongly typed schema system that serves as a contract between the client and the server.



GraphQL logo

The schema defines the available types and operations, providing a standardized way for clients to interact with the API. In the context of the project, GraphQL was utilized to enhance the efficiency and flexibility of data retrieval and manipulation. GraphQL offers several operations:

- **Query:** Operations with no side effects, used for reading and fetching data.
- **Mutation:** Operations with side effects, handle creation, update, and deletion of data.
- **Subscription:** Long-lasting operation where the server pushes real-time updates to the client.

Webpack

Webpack [11] is a static module bundler that provides optimization features and helps manage dependencies in JavaScript applications. It can bundle files, reduce their size, and apply optimizations like tree shaking and code splitting. A *bundle* is a set of static assets that can be served to clients. Webpack works by building a dependency graph from one or more software modules and then combining them into one or more bundles.



Webpack logo

Single SPA

Single SPA [12] is a JavaScript framework to manage multiple microfrontends. It acts as a top-level router that renders applications based on the URL. It also allows defining the graphical structure of the various frontends.



Single SPA logo

Amazon Web Services

Amazon Web Services [13] constitute the architectural backbone of the application and provide essential functionalities. The list of all Amazon Web Services employed during the development of the application is shown below. Each service is characterized by an icon and a short description of the features it provides.



Lambda

Serverless computing service that allows executing functions on the cloud.



Amplify

Libraries, UI components, and a CLI for full-stack application development.



Elastic Container Registry

Store, manage, and deploy container images.



Identity and Access Management

Access control for services and resources.



S3

Cloud object storage for storing and retrieving any amount of data.



Cognito

Application users' identity and access management.



CodeCommit

Source control service for hosting private Git repositories.



CloudFormation

Provisioning and management of collections of AWS resources.



CodeBuild

CI service for compiling, running tests, and producing deployment packages.



AppSync

Serverless APIs for accessing and manipulating data using GraphQL.



DynamoDB

Flexible Key-value and document NoSQL database.



CloudWatch

Monitoring service for tracking metrics and saving logs about AWS resources.



API Gateway

Allows creating, publishing, monitoring, securing, and maintaining APIs.



CloudFront

Content Delivery Network (CDN) for fast web content distribution.

Jira

Jira [14] is a project management tool for agile software development. The primary feature utilized was its Kanban board, which enabled the visualization and tracking of task progress. Through the utilization of Jira’s Kanban board, the team created distinct columns representing various stages of the project development cycle. These columns encompassed “To Do”, “In Progress”, “Done”, and “Blocked”.



Jira logo

Confluence

Confluence [15] is a comprehensive documentation tool that provides a centralized space where team members can create, edit, and collaborate on project-related documentation, including requirements, environment setup, project specifications, design documents, and implementation details.



Confluence logo

SonarQube

SonarQube [16] is a tool designed to support continuous code quality management and static code analysis. It automatically inspects programs helping developers identify issues early in the development lifecycle. SonarQube provides a range of static analysis rules and metrics that assess the presence of code smells and vulnerabilities.



SonarQube logo

3.2 Comparison of technological alternatives

The main technological choices are outlined in this chapter, highlighting their strengths and weaknesses and comparing them to their counterpart. Finally, for each major decision, it is explained the rationale behind the chosen tools.

3.2.1 Database

Relational VS Non-relational

When it comes to *Database Management Systems* (DBMSs) AWS offers a wide range of alternatives. The main *Relational DBMS* (RDBMS) solution is *Amazon Relational Database Services* (RDS), while for NoSQL the main option is

DynamoDB. To determine the most suitable type of DBMS, the services were compared, evaluating them based on multiple non-functional requirements.

	DynamoDB	Amazon RDS
Database type	Non-relational, key-value	Relational
Scalability	Highly scalable with automatic horizontal scalability	Vertically scalable, horizontal scalability with read replicas
Availability	Automatic data replication over three physical nodes	Optional, database replicas at additional cost
Performance	Single-digit millisecond latency	Higher latency compared to DynamoDB
Join	Application-level joins performed with AppSync and GraphQL	Full support for join operations
Transaction	Read or Write transactions with conditional checks	Full support for complex transactions
Data Model	Flexible schema	Fixed schema
Maintenance	Updates to the data model are transparent to the end user when using schema versioning	Requires downtime for updating production database schema
Cost	1 UP* per GB 9.96×10^{-7} UP* per RRU 4.98×10^{-6} UP* per WRU	0.89 UP* per GB 0.13 UP* per hour of database instance uptime

Table 3.1: DynamoDB and Amazon RDS comparison

The cost requirement requires a deeper examination. When comparing the two services, the following terminology is used:

Read Request Unit (RRU): a single read request (up to 4 KB)

Write Request Unit (WRU): a single write request (up to 1 KB)

Write Capacity Unit (WCU): one read request per second (up to 4 KB)

Read Capacity Unit (RCU): one write request per second (up to 1 KB)

***Unit Price (UP):** used to perform a currency-independent comparison. 1 UP is equivalent to the cost of 1 GB of storage for DynamoDB. The database instance price for RDS corresponds to the least-performant AWS EC2 instance available for the Postgres DBMS. All costs have been computed using the AWS Pricing Calculator [17].

DynamoDB offers a Free Tier of type *Always Free* that includes: 25 WCU, 25 RCU (enough to handle 200 million requests per month), and 25 GB of indexed data storage. Even though RDS also provides a Free Tier, it has a temporal limit of one year. Therefore, the following calculations do not take it into account.

The following application insights have been computed using the number of monthly user actions estimated in subsection 2.3.2:

Monthly read requests: ~ 6000

Monthly write requests: ~ 700

To know the size of the data that the application should handle, an estimate of the amount of data stored in a 10-year range is computed as the sum for all entities of their average record size multiplied by the number of entities of that kind. A very high-level approximation of the record size is obtained as the sum of the length in bytes of potential field names and values.

Entity	Amount	Approximate Record Size	Total Size
Resource	200	~ 128 B	~ 25.60 KB
Customer	20	~ 64 B	~ 1.28 KB
Contact Person	20	~ 16 B	~ 0.32 KB
Business Unit	10	~ 32 B	~ 0.64 KB
Project	20 / Month	~ 256 B	~ 5.12 KB / Month
Release	20 / Month	~ 32 B	~ 0.64 KB / Month
Planned work	30 * 200 / Month	~ 32 B	~ 192.00 KB / Month
Finalized Work	30 * 200 / Month	~ 32 B	~ 192.00 KB / Month

Table 3.2: Estimated size of entities

Estimated yearly database size (without considering indexes):

Constant data items: $\sim 27.84KB$

Time-dependent data items: $\sim 389.76 * 12KB$

Total $\approx 4.70MB$

The limited size of the storage required to run the application can be completely covered by the DynamoDB free tier, while it represents a rather small cost for RDS.

DBMS	Yearly Space cost	Monthly Uptime cost	Operations cost	Yearly cost
DynamoDB	0 UP	-	0 UP	0 UP
RDS	0.004 UP	62.4 UP (16h/day, 30 days/month)	-	748.80 UP

Table 3.3: Cost comparison of DynamoDB and RDS

DynamoDB offers limited support for transactions and join operations when compared to RDS, but this is a rather negligible drawback that can be accounted for with a suitable database schema definition. On the other hand, DynamoDB comes with a considerable cost advantage and future updates to the application schema and use cases will most likely remain within its free tier. Moreover, DynamoDB built-in data replication ensures high availability, whereas RDS would require the setup of multiple availability zones at additional cost (not considered in the above comparison). These non-functional requirements drive the decision to opt for DynamoDB as the chosen BDMS.

Single-table VS multi-table schema

DynamoDB design principles recommend minimizing the number of tables used. Possibly using as little as a single table. This clashes with Amplify's support for multi-table design. Amplify in fact automatically provides table-level authorization and other useful features that would need to be implemented by scratch if using a single-table design. An analysis of the pros and cons of each solution has been performed to identify the best alternative:

Single table

The single-table design requires the definition of a unique table that can host all entities. A careful analysis must be performed to determine the most suitable way of organizing the entities and where to use embeddings (copies of the same data in multiple locations).

Advantages

- + Best performances as reading data doesn't require join operations
- + Retrieve and update data with a single read or write operation
- + Limited use of transactions as updates on a single document are atomic by default.

Disadvantages

- Higher implementation complexity, as each entity composing the table must be manually managed with custom lambda functions.
- Uses a denormalized schema, leading to data duplication and possibly to multiple updates to the document when an entity changes.
- Complex to maintain, as adding, removing, or updating an entity to the schema can affect all other entities.
- No default support from Amplify for CRUD operations and authorization

One table for each entity

This strategy follows the traditional relational schema design, where each entity becomes a separate table. This leads to the creation of many tables and many relationships between them.

Advantages

- + Simple to comprehend, as it uses a normalized schema.
- + Easily maintainable: adding new entities to the schema doesn't affect the existing tables.
- + As each entity corresponds to a document, the maximum size of 400KB per document enforced by DynamoDB is never exceeded.
- + Simple to set up: Amplify automatically implements CRUD operations and provides support for authorization.

Disadvantages

- Lower performances due to the presence of joins.
- The number of read and write operations is higher than in the single-table design.
- Necessitates transactions for operations working on multiple entities.

Multiple Tables

As it's usually the case, the best solution lies in the middle. In this scenario multiple entity tables are combined into a single one, thus resulting in fewer tables being used. This allows exploiting the advantages of both previous solutions. The rule used to merge entities is "Items that are accessed together should be stored together".

Advantages

- + Easily maintainable: adding new entities to the schema doesn't affect the existing tables.
- + The number of read and write operations is lower than in the previous solution.
- + As records are still relatively small, the maximum size of 400KB per document is never exceeded.
- + Simple to set up: Amplify automatically implements CRUD operations and provides support for authorizing users.

Disadvantages

- The number of read and write operations is higher than in the single-table design.
- Lower performances due to the presence of joins, but with a limited number of joins with respect to the previous solution.
- Uses a denormalized schema, leading to data duplication, but only where data won't need to be updated if the main entity changes.
- Necessitates transactions for operations working on multiple tables.

To decide which solution is the best fit, a few considerations must be taken into account: the application will be mainly used in the context of business units, thus the number of employees will be relatively small. Due to the limited amount of data this application must handle, the high performances achievable with the single-table approach do not compensate for the increased implementation and maintenance complexity. The comparison between using one table for each entity and the multi-table option reveals that the former offers limited advantages. On the other hand, the multi-table solution provides better performance, although it does introduce a minor development overhead. Despite this small additional difficulty, the multi-table approach has been chosen due to having the best overall efficiency-complexity trade-off.

3.2.2 RESTful VS GraphQL APIs

AWS Amplify offers two approaches to enable communication between the front end and the back end: REST and GraphQL. They both are API architectural styles that enable the exchange of data between services and applications in a client-server model. GraphQL is also a query language and a runtime for fulfilling those queries. Numerous similarities exist between REST and GraphQL, including their underlying use of the HTTP protocol, statelessness, language and database neutrality, and adherence to a resource-based design. An analysis of their differences has been conducted to determine the most suitable option for the project.

	REST	GraphQL
Data access	Multiple endpoints	Single endpoint
Actions	HTTP verbs	Queries, mutations, and subscriptions
Data structure	Data has a fixed structure defined by the server. This can lead to the N+1 problem, underfetching, and overfetching	Returns data in a flexible structure defined by the client, allowing it to request only necessary data
Data type	Weakly typed. The client must interpret the returned data	Strongly typed, types are defined in a contract between client and server
AWS Services	API Gateway and Lambda	AppSync, optionally Lambda

Table 3.4: REST and GraphQL comparison

GraphQL's ability to provide a flexible data structure defined by the client allows a precise and efficient retrieval of the required data with fewer network round trips when compared to REST. As a result of improved performance and simpler communication between the front end and back end, GraphQL is considered the best solution for the project requirements.

Chapter 4

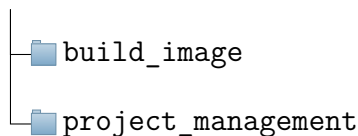
Design

This chapter showcases the process of translating the requirements into a comprehensive plan for the web application, acting as a bridge between the analysis and implementation phases. It explores the architectural decisions, user interface design choices, database schema definition, and overall system design, laying the foundation upon which the application is built.

4.1 Project structure

The project repository has been structured following the monorepo approach. Monorepo is a software development strategy in which the code for multiple projects is stored within the same repository. Each project is represented as a directory within the monorepo, encompassing its source code, dependencies, and configuration files. Initially, only two sub-folders had been defined: one contained the Flutter application and the other stored the dockerfile for building the image necessary to compile the app in the cloud. In the second phase, a directory for each microfrontend has been added.

Monolith



Microfrontends



4.2 System architecture

The overall system architecture is represented in Figure 4.1. On the back end, a few services offer solutions to cross-cutting concerns: CloudWatch is used to log and observe all events happening on the cloud services, Simple Queue Service is used to send unprocessable messages to a Dead-letter queue, and CloudFormation provides *Infrastructure as Code* (IaC), which involves converting source code into cloud resources. User authentication is handled by one Amazon Cognito user pool and one identity pool. Cognito interacts with the Azure Active Directory to obtain user information.

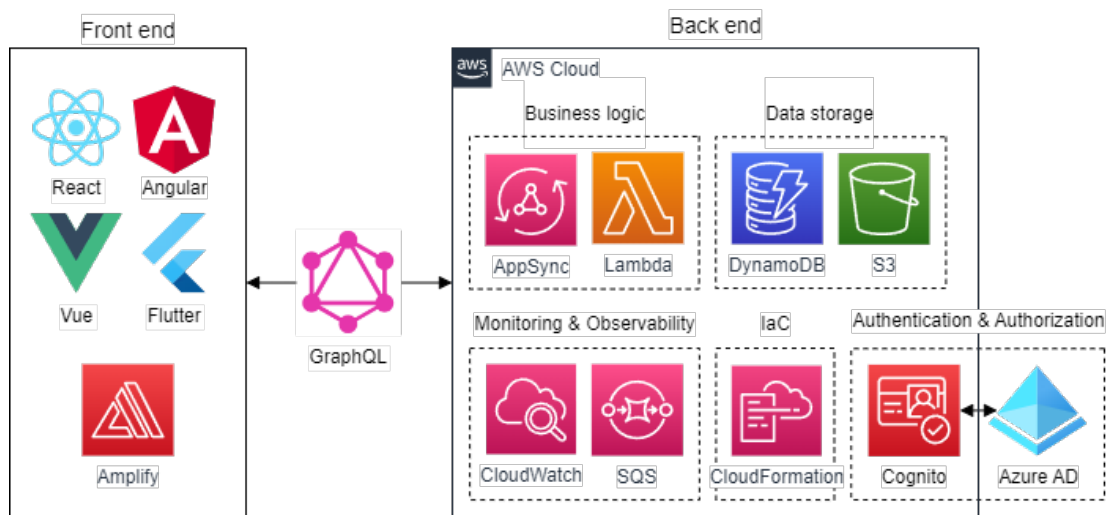


Figure 4.1: Overall system architecture

Communication between the UI and cloud resources happens by means of GraphQL queries and mutations.

Each microfrontend has its own presentation layer, while the same back end is shared among all the front ends. Each application follows a layered architecture in which three main tiers can be identified:

- **Presentation:** Static content (HTML, CSS, JavaScript, and other assets) hosted on S3 and downloaded to the browser when the app is run.
- **Logic:** built using Lambda functions and AppSync triggered by GraphQL queries and mutations. DynamoDB streams and CloudWatch rules are also used to trigger functionalities whenever specific events occur.

- **Data:** The application itself is hosted in an S3 bucket, while the data is stored in DynamoDB tables.

4.3 Back end

The application leverages a *Serverless architecture*, a way to run applications using cloud services without having to manage the underlying infrastructure. Lambda functions are employed whenever a user action requires custom validation rules or it involves multiple entities. In the latter case, DynamoDB transactions are used to enforce an atomic behavior.

4.3.1 Business logic

The application logic is mainly stored within Lambda functions. Some of them are triggered by events such as user registration, or updates in database tables, but most are directly invoked as a result of user interactions on the front end. Time-based invocation is also used for the generation of new holiday instances every year.

Simple CRUD operations are automatically implemented by Amplify and AppSync as GraphQL queries and mutations, as shown in Figure 4.2.

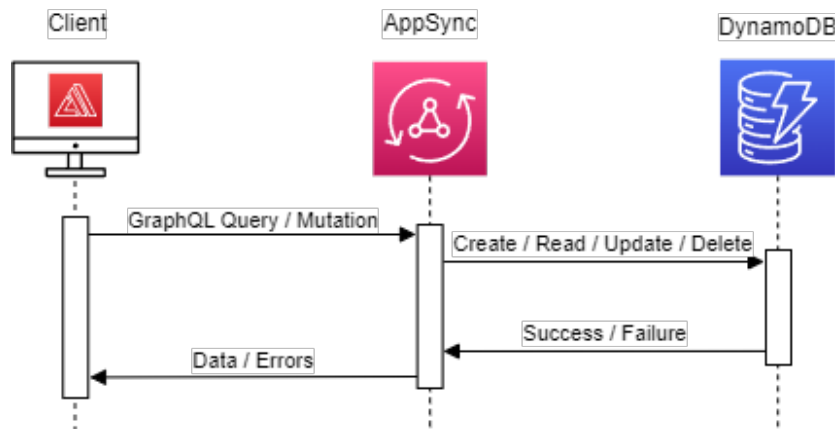


Figure 4.2: GraphQL queries and mutations sequence diagram

A Lambda function can be used to implement advanced functionalities that operate on multiple database entities or require transactional behavior. To preserve

continuity with the previous solution, GraphQL mutations are still employed. Mutations act as AWS Lambda resolvers, invoking the functions. A visual representation of this scenario is portrayed in Figure 4.3.

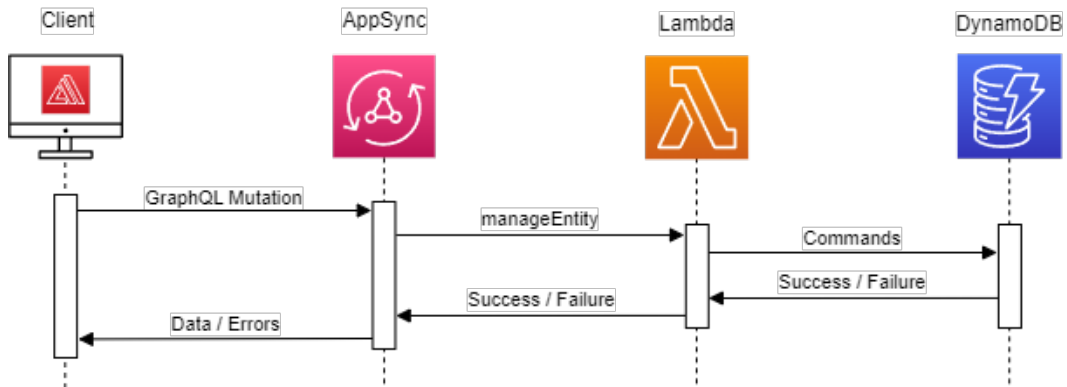


Figure 4.3: Lambda invocation with GraphQL resolver sequence diagram

4.3.2 Lambda layers

A Lambda Layer is a collection of scripts and libraries that can be shared between Lambda functions. They provide advantages such as code reuse and separation of core function logic from dependencies.

Two main groups of functionalities that can benefit multiple Lambdas have been identified and aggregated into Layers:

- **Database utils:** Provide common logic for interacting with DynamoDB, like the retrieval of one or more items by their IDs or by a secondary index.
- **User utils:** Provide shared utility functions for interacting with Cognito services, such as: retrieving/creating/deleting users and listing/assigning/removing user groups.

4.3.3 Dead-letter queue

A Dead-Letter Queue (DLQ) is a type of message queue with the responsibility of temporarily storing messages that the system cannot process due to errors. A global DLQ has been set up for the entire project. Messages are sent to this queue after failures in Lambda executions.

The advantages of using a dead-letter queue include the possibility to perform analysis on the errors, debugging and troubleshooting, and issuing alerts to administrators when issues arise.

4.3.4 DynamoDB streams

DynamoDB streams are a Change Data Capture (CDC) mechanism that detects changes (inserts, updates, deletes) made to items within a DynamoDB table and makes them available to other AWS services as an ordered sequence for further processing in real-time. In the context of this project, a DynamoDB stream has been utilized for synchronizing data between a DynamoDB table and the Cognito User pool.

4.4 Front end

An initial plan for the UI encompassed the use of Flutter to develop a monolithic front end. Later in the project, the UI was re-designed into microfrontends.

4.4.1 Monolith

The Flutter application is composed of a navigation bar, a homepage with all the links grouped by functionality, and all the screens needed to support CRUD operations. Specifically, each of the *Resource*, *Project*, *Business Unit*, *Customer*, *Contact Person*, and *Release* entities have:

- A creation and an update form.
- A delete button with a confirmation popup.
- A list of all existing instances in the database.

4.4.2 Microfrontends

A new user interface has been designed as the composition of many single-page applications (SPAs). The original Flutter application has been stripped of its navigation bar and homepage, which became independent applications of their own. To demonstrate the ability of microfrontends to be technology-independent, each application has been developed using a different library or framework. In particular, the overall UI is composed of the following microfrontends:

- **Container / Root:** Plain HTML and JS, uses the Single-SPA library to define the routing and layout of the various front ends.

- **Navbar:** Vue application showing the current page name, and information about the authenticated user.
- **Home:** Implemented in React, shows all the menu categories and their links.
- **Project management:** The original Flutter application.
- **Styleguide:** Shared CSS and JavaScript components available to all other microfrontends.

Routing

The container is responsible for rendering the individual components whenever the URL matches the ones defined by the applications themselves. Each application would perform individual routing as if they were running standalone.

Shared libraries

The same libraries might be used by more than one microfrontend. If their size is considerable, it is better to explicitly mark these dependencies as shared. Doing this allows the reuse of the same dependency, avoiding multiple downloads, thus resulting in better performance.

A way to achieve this is through Import Maps [18], a browser specification that allows the use of an alias instead of the full URL when importing a dependency in a JavaScript file on browsers.

4.5 Database

Following the approach decided in Table 3.2.1 of combining multiple models into a single database table whenever possible, the composite entities have been defined:

- **Customer:** contains a list of *Contact Persons* and one of the *Releases*. The target release date and the email of the internal and external contact persons are replicated in the *Project* entity.
- **Resources:** contains employee's personal information and an array of all the vacations and working hours of the *Resource*.
- **Monthly work:** aggregates all daily work for a given month, employee, and project in a single table record.

The database schema is depicted in Figure 4.4. The primary key of each table is composed of a **Partition Key** (PK) and optionally a **Sort Key** (SK).

Partition key: Attribute on which a hashing function is applied, the results determine the bucket in which the record is placed. All records with the same partition key will end up in the same bucket.

Sort Key: Optional set of arguments used to sort items within the same partition.

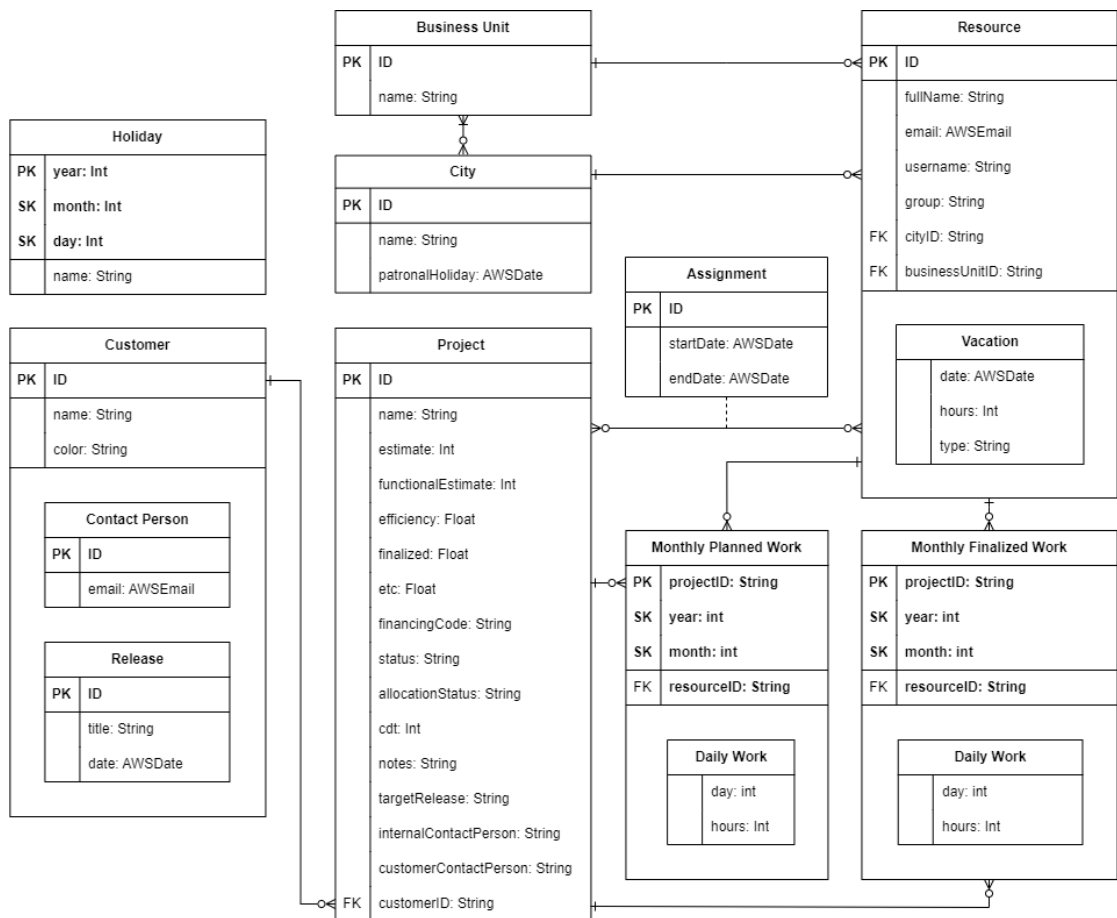


Figure 4.4: Database schema

As theory clashes with reality, the need to minimize the number of tables encounters the limitations enforced by the Amplify framework. In particular, the *Monthly Work* entity has been split into *Monthly Planned* and *Monthly Finalized* as this allows for simpler fine-grained authorization controls: *Resources* are allowed to visualize both of them but they they are only allowed to update the latter.

DynamoDB allows querying data only by their primary key by default. A secondary index must be defined to access data using fields not part of the primary key. A secondary index has been defined on the *email* attribute of the *Resource* entity to allow efficiently querying employees when knowing only their email address.

Chapter 5

Implementation

This chapter explores the implementation choices that turned the project design into a fully functional web application. It delves into how the source code was managed, how the CI/CD pipeline was set up, how authentication and authorization were enforced, and it explores the implementation details for project management and work planning business logic.

5.1 Project management

The project was managed with an agile-like approach similar to what the Scrum framework proposes. The team consisted of:

- A manager acting as the product owner who provided requirements and feedback, ensuring that the result was aligned with the project goals.
- A project manager who defined tasks, established timeframes, and developed the application mockups.
- Four developers (including myself). One of them focused on front end development in the initial part of the project, while the other two handled minor UI-related tasks at first and then contributed to the implementation of the work-planning interface.
- An AWS expert who provided implementation tips and insights about working with Amazon Web Services.

The time frame dedicated to this project was 6 months. This time was split into one-week sprints. At the end of each sprint, a one-hour meeting was held to review completed work, provide a demo of the newly added features to the product owner, and plan tasks for the next week. The short sprints and weekly meetings ensured that plans could be quickly adjusted based on feedback and evolving needs.

The work was organized using a Kanban board on Jira. Tasks were divided into four columns: “To-Do”, “In Progress”, “Done”, and “Blocked”.

5.2 Project setup

5.2.1 Git branches and Amplify environments

The source code is stored in a Git repository hosted on AWS CodeCommit. The repository has been organized into the following Git branches:

- **main**: Production-ready code, the final application accessed by the end-user.
- **test**: Used to evaluate the application quality before moving the code to the production environment.
- **dev**: Main development branch, used to merge all individual development branches.
- **developers'**: Feature-specific or individual branches for local development.

An **Amplify hosting environment** is a service that hosts the front end of the application by serving the static assets to the users.

An **Amplify back end environment** is a container for all of the cloud capabilities of the application, such as API, auth, and lambda functions.

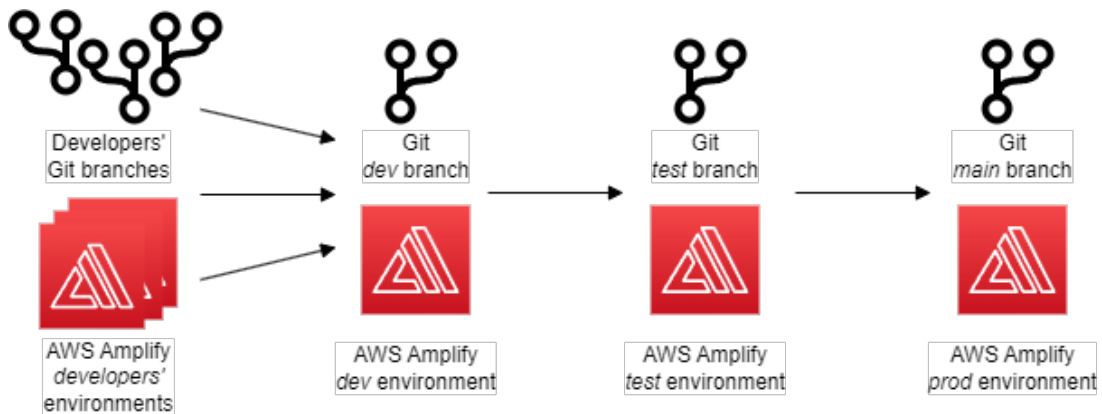


Figure 5.1: Amplify environments relation with Git branches in Monolithic app

In the monolithic scenario, there is a single Amplify application. Each hosting environment has a corresponding back end environment and the application is built and deployed every time a commit is pushed on the corresponding branch. A visual representation of the relation between git branches and Amplify environments is

provided in Figure 5.1.

In the context of multiple applications, each microfrontend has its own Amplify application. In each of these, there is a hosting environment and optionally a back end environment for every Git branch (the back end is optional as some UI may not need to use server-side features). Following the monorepo paradigm, Git branches are shared among all microfrontend applications.

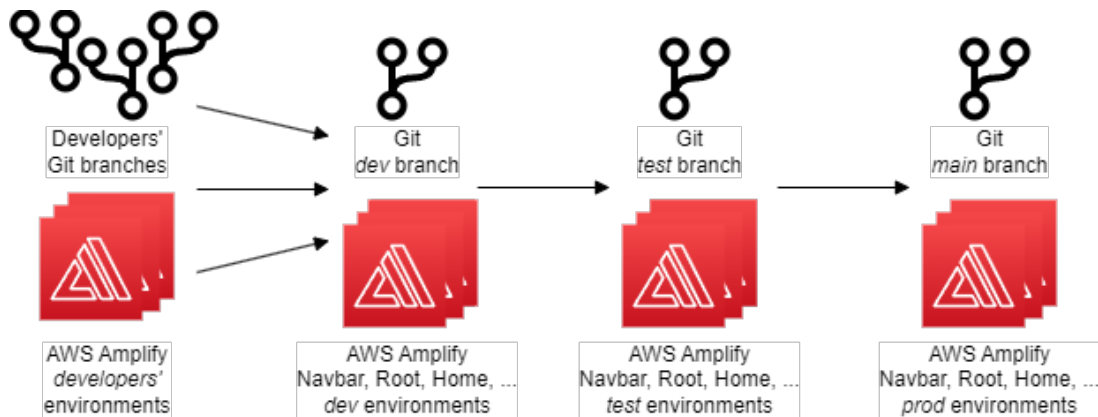


Figure 5.2: Amplify environments relation with Git branches in microfrontends

5.2.2 Amplify CI/CD

Amplify backend environments are set up to automatically trigger the Amplify Continuous Integration / Continuous Delivery (CI/CD) process whenever a new commit is pushed to the branch associated with the environment.



Figure 5.3: Amplify CI/CD pipeline steps

Figure 5.3 shows the different phases composing the Amplify CI/CD pipeline:

- **Provision:** A Docker image is hosted on an isolated instance with 4 vCPU and 7GB of memory.

- **Build:** The Git repository is cloned from CodeCommit and the commands defined in the build phases of the `amplify.yml` file are executed. A more detailed description of this phase is provided in subsection 5.2.3.
- **Test:** Automatic unit and end-to-end tests are run using the newly built application, the pipeline fails if one or more tests are unsuccessful.
- **Deploy:** The artifacts generated by the previous steps are uploaded to the S3 bucket of their Amplify application.

5.2.3 Application build

The build process goal is to obtain an executable version of an application that can be deployed. The sequence illustrated in Figure 5.4 outlines the steps to obtain a working application starting from its source code.

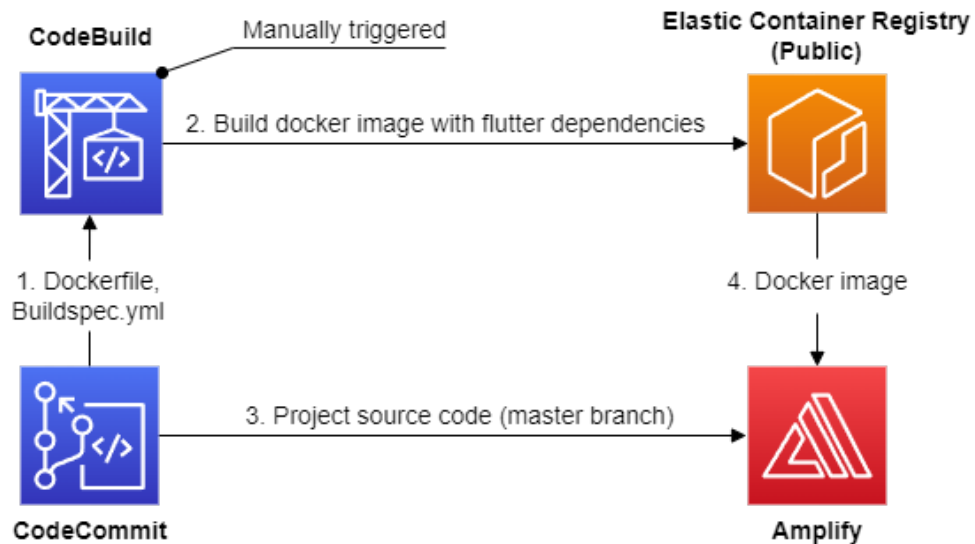


Figure 5.4: Application build architecture

1. The Dockerfile and the Buildspec.yml file are retrieved from the CodeCommit Git repository.
2. The CodeBuild pipeline is responsible for generating a new Docker image and uploading it to the Elastic Container Registry.
3. When a new commit triggers an application build, Amplify retrieves the project source code from CodeCommit.

4. Amplify uses the previously built Docker image to build the application.

Build image

A suitable Docker image containing all the necessary dependencies for building an application is required to compile the front end.

AWS provides some predefined Docker images to be used but none of the available ones had the necessary dependencies to compile the Flutter application. To overcome this issue, a custom Docker image has been constructed and hosted on the Elastic Container Registry (ECR). The Docker image is built only once and is re-used by Amplify every time the Flutter application must be compiled.

To generate the Docker image and upload it to the Elastic Container Registry a build process must be manually triggered from the Build Project on AWS CodeBuild. The Build Project takes care of fetching the source files from the main branch of the repository on CodeCommit, in particular the `Dockerfile` and the `buildspec.yml`, and using them to obtain a Docker image.

Dockerfile

The Dockerfile contains the sequence of commands needed to create a Docker image possessing all the necessary dependencies to build an application within the AWS Amplify CI/CD flow. One of the predefined Dockerfiles provided by Amplify has been updated to install additional dependencies.

Buildspec.yml

The `buildspec.yml` file contains the commands that CodeBuild runs to build a Docker image from the `Dockerfile`. The commands are executed in three different phases:

- `pre_build`: Obtain the credentials and log in to the Registry.
- `build`: The `Dockerfile` is used to build an image, to which a tag is added.
- `post_build`: The built Docker image is uploaded to the public Elastic Container Registry.

The Amplify workflow is then modified to use the newly published image during the build phase.

The build steps for each application are defined in the *Amplify.yml* file. They are split into **frontend** and **backend** sections. Each section is further partitioned into **preBuild**, **build**, and **postBuild**. These phases are responsible respectively for installing the project dependencies, building the application, and deploying the output of the build process.

The output is a bundle, a JavaScript file containing all the necessary code to execute the application. The filename of the bundle follows the format **app.[HASH].js**, where *[HASH]* is the result of a hashing function applied to the content of the file. Hashing is useful for caching the file, as the hash changes only when the bundle content is updated.

5.2.4 Application deployment

For the initial standalone Flutter application, the deployment meant uploading the output of the build phase to an S3 bucket and serving its content to the clients requesting it.

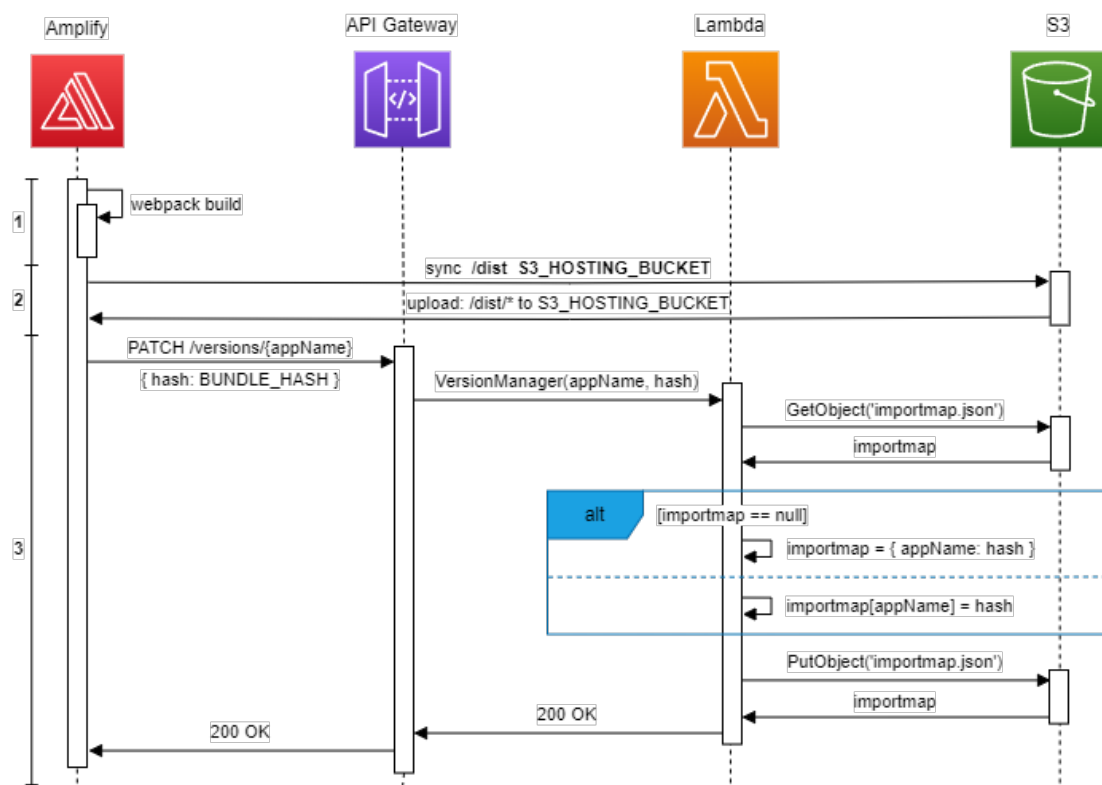


Figure 5.5: Deployment of microfrontends

In the microfrontends approach, instead, the build output of each application is still uploaded to an S3 bucket, but only the root microfrontend content is directly served to the client. All other applications contained in the various S3 buckets are lazy-loaded from the root, which has to know where the microfrontends are located. The latest version of each application is saved in a file called **importmap.json**, stored in the S3 bucket of the root microfrontend.

The graph reported in Figure 5.5 shows the steps automatically performed each time a microfrontend is updated:

1. The front end and back end are built as described in the previous section.
2. The bundle obtained from the build process, located in the `/dist` folder, is uploaded to the S3 bucket associated with the microfrontend being deployed.
3. An HTTP PATCH request is sent, along with the microfrontend symbolic name and the *hash* value computed in step 1, to an API Gateway endpoint hosted on the root microfrontend back end. Requests to this endpoint trigger the *MicrofrontendVersionManager* Lambda function, which is responsible for retrieving the **importmap.json** from the S3 bucket, updating the entry corresponding to the invoking microfrontend with the latest *hash* value (or creating a new **importmap.json** if this file is not present yet), and save the updated file to the S3 bucket.

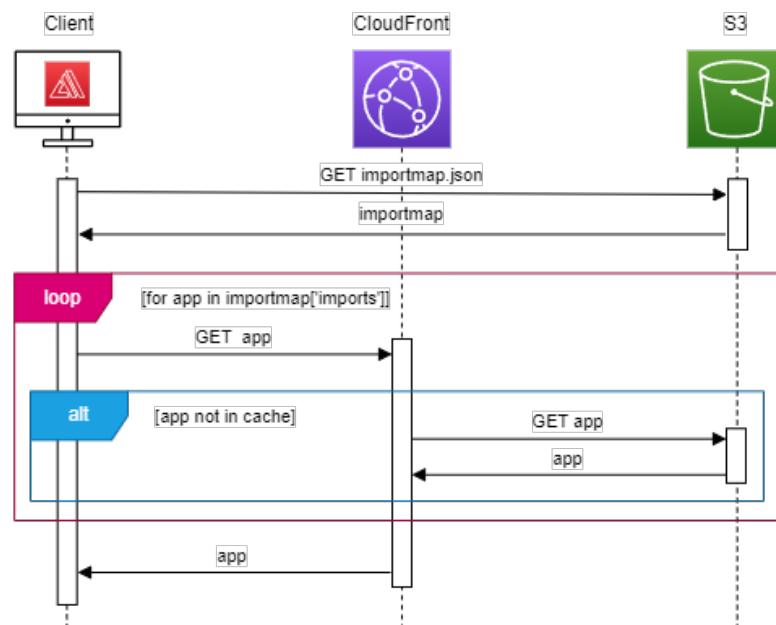


Figure 5.6: Retrieval of microfrontends

The users access the application functionalities by navigating to the URL on which the root application is hosted. As described in Figure 5.6, the client first requests the **importmap.json** file, which is never cached.

The content of the **importmap.json** file follows this structure:

```
1 {
2   "imports": {
3     "@blueplan/home": "[HOME-CLOUDFRONT]/main.[HASH].js",
4     "@blueplan/navbar": "[NAVBAR-CLOUDFRONT]/js/app.[HASH].js",
5     "@blueplan/root": "[ROOT-CLOUDFRONT]/main.[HASH].js",
6     "@blueplan/project-management": "[PROJECT-MANAGEMENT-
7     CLOUDFRONT]/main.[HASH].js"
8   }
9 }
```

[HASH] is a sequence of alphanumeric characters, and *[CLOUDFRONT]* is the CloudFront Distribution URI where the S3 Bucket contents of each microfrontend are cached.

Once the client receives the import map, it fetches each application by issuing a corresponding request to CloudFront. Microfrontend bundles are cached by CloudFront, if the application is already in the cache it is returned immediately, otherwise, it is first retrieved from its S3 bucket.

5.3 User Interface

Home page UI

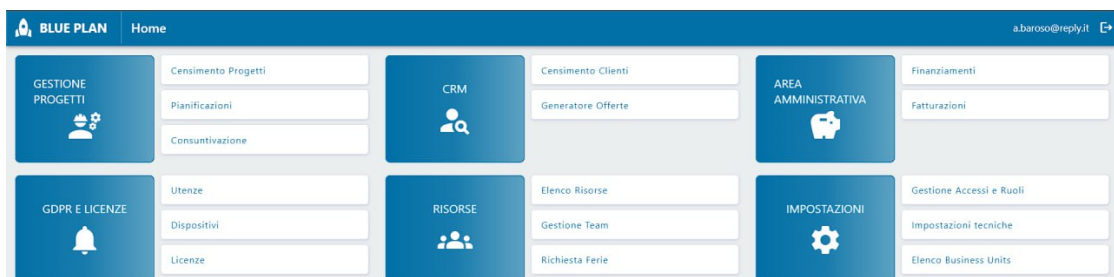


Figure 5.7: Home page UI

The landing page presented to the user after the login is the home page, visible in Figure 5.7.

In the uppermost part of the home, the navbar displays the logo, the application name, and the title of the current page on its left side, while on the right the user can see their username and a log-out button. In the lower part of the home page, a series of links are grouped in sections based on their functionalities.

The active links on the homepage redirect to the *Project* lists [Figure 5.12], list of *Resources* [Figure 5.14], *Customer* list [Figure 5.8], and *Business Unit* [Figure 5.17] list. All other buttons are currently not enabled but will be used for future functionalities.

Customers, releases, and contact persons UI



	NOME CLIENTE	NUMERO PROGETTI	
<input type="radio"/>	Customer #1	3	
<input type="radio"/>	Customer #2	0	
<input type="radio"/>	Customer #3	2	

Figure 5.8: Customer list UI

Figure 5.8 shows the list of all the clients, each described by a name and a color. Clicking on an entity from a list will direct the user to the corresponding update form, pre-populated with the entity’s values. Additionally, each update form page includes a “Remove” button to delete the selected instance.

Figure 5.9: Customer form UI

The *Customer* update page [Figure 5.9] presents the *Customer* information and an editable list of *Releases* and *Contact persons*. Clicking on the list items redirects respectively to the Release form [Figure 5.10] and to the Contact Person form [Figure 5.11].

Figure 5.10: Release form UI

Figure 5.11: Contact person form UI

Projects UI

The list of *Projects*, reported in Figure 5.12, displays information and statistics about the activities and it can be filtered specifying any combination of *Customer*, *Release date*, *Project manager*, or *Contact Person*. Selecting a *Customer* automatically restricts the options of the other dropdown fields to the values associated with the chosen *Customer*. An average of the metrics of all the activities is shown right below the filters.

Implementation

FILTRI						
Cliente	Release	Project Manager		Referente Cliente		
Tutti i clienti	Tutte le release	Tutti i project manager		Tutti i referenti		
	STIMA	LAV.	ETC	EFF	REFERENTE	RELEASE
TOTALI	564	147	190	40.24%		
Customer #5 - Aggiornamento pagina prodotti						
	100	50	50	0%	referente@customer5.com	mag. 2022
Customer #1 - Pagina vendite						
	120	30	30	- %	referente@customer1.com	mag. 2023
Customer #3 - Project name						
	10	0	0	- %	referente@customer3.com	mag. 2023

Figure 5.12: Project list UI

The *Project* form is reported in Figure 5.13. Creating or updating a *Project* requires selecting a *Customer*, *Release date*, *Project Manager*, and *Contact Person* among those available from their respective dropdowns. The form also allows to define a name and initial values for the estimate, finalized, and ETC attributes.

Cliente	Nome
Customer #1	Pagina vendite
Release	Stima
15/05/2023 - v1 gestione	120
Project Manager	Consuntivato
Green M. (PM)	30
Referente Cliente	ETC
referente@customer1.com	30

Cancella Salva

Figure 5.13: Project form UI

Resources UI









RUOLO	EMAIL	NOME	
Analista Funzionale	mark.red@reply.it	Mark Red	
Project Manager	miriam.green@reply.it	Miriam Green	
Analista Tecnico	richard.orange@reply.it	Richard Orange	
Manager	julie.brown@reply.it	Julie Brown	
Manager	rachel.blue@reply.it	Rachel Blue	
Manager	jimmy.pink@reply.it	Jimmy Pink	

Figure 5.14: Resource list UI

The Resource list [Figure 5.14] displays all Managers, Project Managers, Technical analysts, and Functional Analysts of the company. For each employee, its role, work email, and full name are shown.

Nome	Cognome	Email
Mark	Red	mark.red@reply.it
Business Unit	Sede	Ruolo
Products & Claims	↓ Sede di riferimento	↓ Analista Funzionale

Progetti assegnati  Filtra

Customer #1 - Pagina Previsioni e Analisi Mercato Assegnato dal 11/07/2023 al 28/03/2024 

Aggiungi progetto




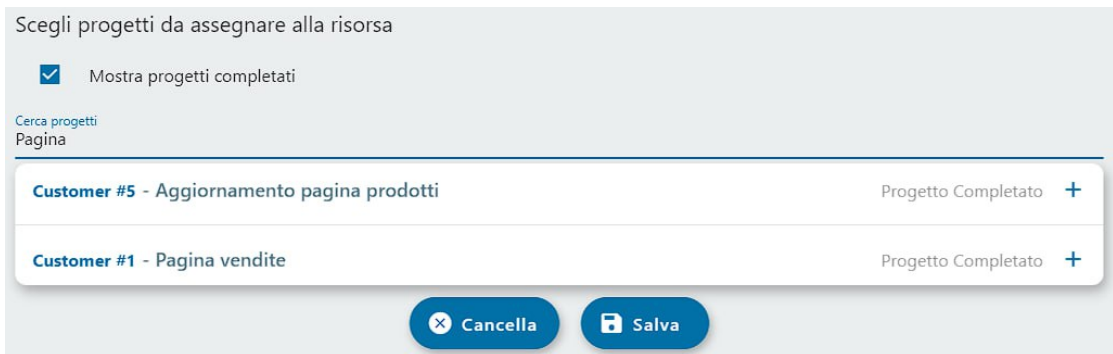
 Cancella
 Salva
 Elimina

Figure 5.15: Resource form UI

The *Resource* form, shown in Figure 5.15, not only allows the definition of the employee's personal and business details, but also provides the possibility of assigning the *Resource* to an existing project.



Scegli progetti da assegnare alla risorsa

Mostra progetti completati

Cerca progetti
Pagina

Customer #5 - Aggiornamento pagina prodotti	Progetto Completato +
Customer #1 - Pagina vendite	Progetto Completato +

Figure 5.16: Project selection UI

The assignment of *Projects* is performed by the form reported in Figure 5.16. An auto-completion dropdown allows to search for *Projects* by their name or customer name. Completed activities (those with a *Release* date in the past) can be filtered out by selecting a checkbox. Clicking on a *Project* includes it in the “Assigned Projects” section, where it’s possible to define from which date to which other date the activity is considered assigned to the employee.

Business units UI



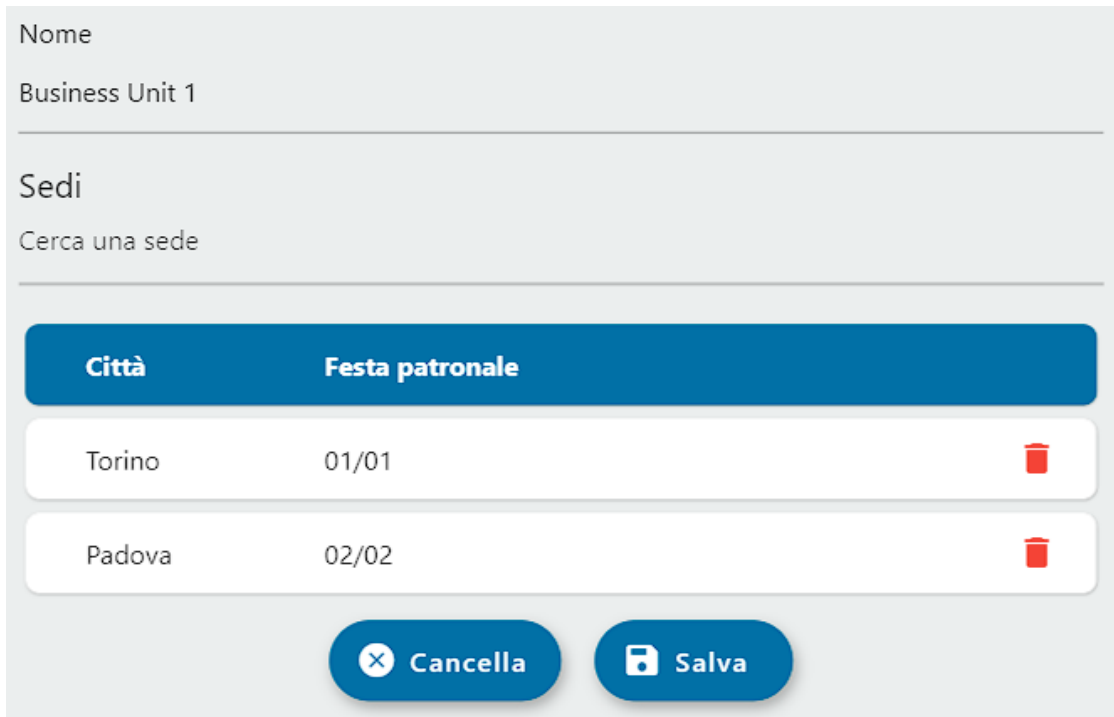
Business Unit	
Business Unit 1	
Business Unit 2	
Business Unit 3	

Figure 5.17: Business unit list UI

Each item in a list presents a red trash can icon that when clicked shows a pop-up prompting the user to confirm or cancel the deletion of the entity.

A floating action button is located in the bottom right corner of each list. It enables the user to navigate to the creation form for the entity. The creation form is identical to the update one, except its inputs start from a blank state.

The *Business unit* list, reported in Figure 5.17, simply shows the name of each BU. The form to create or update a *Business Unit*, visible in Figure 5.18, contains a field to specify the BU name, and a list of all the cities where it has its offices.




Città	Festa patronale	
Torino	01/01	
Padova	02/02	

Figure 5.18: Business unit form UI

5.4 Authentication

The application's target users are managers, project managers, and analysts of a business unit. A solution must be devised to allow only company employees to sign up and access the service. The authentication process has been implemented using a couple of functionalities from the Amazon Cognito identity platform:

- *Cognito User Pool*: Collection of application users that supports user registration and sign-in, as well as provisioning identity tokens for signed-in users.

- *Cognito Identity Pool*: Store of user identity data that provides temporary AWS credentials for users who are unauthenticated and for authenticated users possessing a token.

An important aspect to take into account while devising a suitable implementation is that within the organization, each employee owns unique user credentials. User account data is stored within an instance of Azure Active Directory (Azure AD) [19]. A dedicated instance of Azure AD, referred to as a **tenant**, serves as an entity representing the company inside the Azure environment.

As a result of an initial analysis, two solutions have been identified to implement the authentication process.

5.4.1 Custom user sign-up and sign-in forms

A custom registration flow would require users to fill out a sign-up form with their personal information, verify the provided email address, and log in by submitting a sign-in form.

The constraint of allowing only company employees to register could be enforced by requiring the use of their work email while registering to the platform. An additional whitelist would be required to permit the usage of the application only to a selected group of workers.

AWS Amplify facilitates the development of a user interface to support this scenario by providing and managing ready-made components for sign-up and sign-in forms. The back end would make use of an Amazon Cognito user pool for storing user identities and managing access privileges.

Although effective, this authentication strategy provides poor User Experience (UX) and results in a duplication of the corporate user database that can lead to inconsistencies between the two data sources.

5.4.2 Corporate Single Sign-On with OAuth flow

An alternative approach would rely on the existing set of corporate users by integrating a **Single Sign-On (SSO)** strategy with the user Azure Active Directory. This solution leverages the Azure AD and both the Cognito user pool and identity pool. An *application* is required within the Azure AD Reply's tenant to handle the user authentication process. The OAuth 2.0 [20] protocol is fundamental for allowing communication between all these components.

OAuth 2.0 is an industry-standard authorization framework that enables third-party applications to obtain limited access to users' resources without accessing their credentials. The OAuth protocol specifies several **grant types**: secure and standardized sequence of steps for implementing SSO across different systems. Among the possible grant types, the OAuth **Authorization Code Grant** [21] flow was used.

This scenario, although more complex, has been selected as the preferred one as users benefit from a better UX since they are not required to explicitly register to the application and manage new credentials. If a corporate auth session is already active in the browser, the login process becomes completely transparent to the user. A further benefit of this approach is the automatic exploitation of the two-factor authentication already set up with Azure AD. However, as this solution would allow every employee with valid credentials to automatically have access to the application, an additional mechanism to grant access only to manually selected users must be put in place.

OAuth Code Grant flow with Amplify

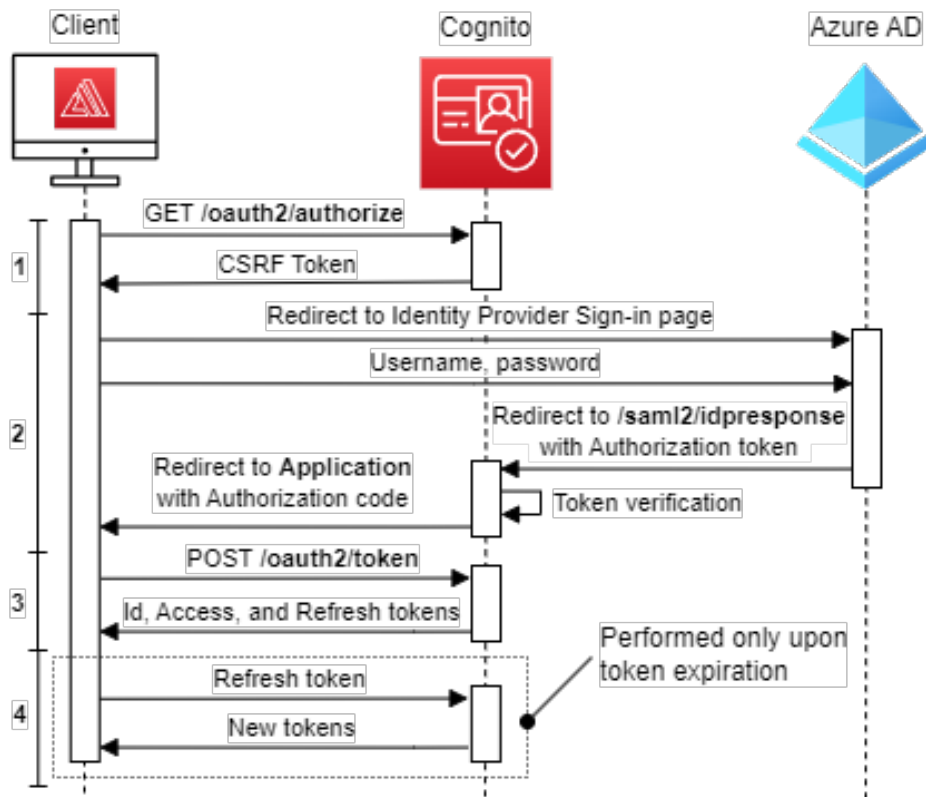


Figure 5.19: Authentication flow with OAuth Code Grant

1. The client initiates the auth flow by requesting a *Cross-Site Request Forgery token* from the Cognito Authorization Server as a security measure.
2. The user is redirected to the Identity Provider (Microsoft Azure AD) authentication page where they are requested to enter their credentials. Upon successful log-in the user is redirected to the Cognito Authorization Server, which will verify the *Authorization token* returned by Azure AD and will consequently return an *Authorization token* to the application.
3. The application exchanges the *Authorization token* for the *Id, Access, and Refresh tokens*.
4. Amplify automatically refreshes tokens if they expire while using AWS services.

The Auth Code Grant flow makes use of different tokens and endpoints, this is their meaning:

Tokens

Access token

String that the client can use to make requests to the resource server.

ID token

Contains information about what occurred when the user authenticated, and optionally information about the user itself.

Refresh token

String that allows the client to obtain a new *Access token* without user involvement.

Endpoints

`/oauth2/authorize`

Redirects users to the login page for their **Identity Provider (IdP)**.

`/oauth2/token`

Exchanges the authorization code for the Access, Identity, and Refresh tokens.

5.5 Authorization

Cognito allows the invocation of Lambda functions in response to actions such as user sign-up, confirmation, and sign-in. These functions, called Lambda triggers, can be executed before or after one of these actions. To fulfill the requirement of allowing only whitelisted users to access the application and to provide a better user experience, pre-signup and post confirmation lambda triggers have been set up.

5.5.1 Pre-signup lambda trigger

The pre-signup trigger receives user information before it is added to the user pool and can veto the user creation. This function was employed to automatically confirm users whose email address ends in `@reply.*`. The acceptance of the user is necessary to activate the post-confirmation lambda.

5.5.2 Post confirmation lambda trigger

This trigger receives the attributes describing the user, such as their email and username, after it has been confirmed. The purpose of this function is to link the Cognito user with the application *Resource* entity. The connection is established using the employee email address as it's the only information the two user representations have in common.

New resources are created by Managers through the application and each resource has a role associated with them. This characteristic affects the operations they will be able to perform and the UI that will be presented to them. Roles have been managed by exploiting Cognito Groups, which represent collections of users with common permissions.

Table 5.1 presents an overview of the roles defined within the application, showing the corresponding read (R) or write (W) permissions for each table in the database.

Role	Resource	Customer	Release	Contact Person	Business Unit	Planned Work	Finalized Work	Project	Holiday	City
Admin	W	W	W	W	W	W	W	W	W	W
Manager	W	W	W	W	W	W	W	W	R	R
Project Manager	W	R	W	W	R	W	W	W	R	R
Technical Analyst	R	R	R	R	R	R	W	R	R	R
Functional Analyst	R	R	R	R	R	R	W	R	R	R
Unauthorized	-	-	-	-	-	-	-	-	-	-

Table 5.1: Roles and permissions

Permissions for each database table are defined in the GraphQL schema and are automatically handled by Amplify and AppSync. *Unauthorized* users will be shown a message prompting them to ask their manager to authorize their access. Managers can enable new users by creating a *Resource* entity with their work email address.

A challenge that arose while implementing this solution was that Managers might try to create a *Resource* for an account that has never logged in to the application, and therefore their user might not yet exist in the Cognito user pool. To avoid responding to the Manager with an error message, a delayed group assignment has been devised within the post-confirmation trigger. Additionally, the lambda function must handle the case in which the registering user has no corresponding record in the *Resource* table and as a consequence, they will be assigned to the *Unauthorized* role. Figure 5.20 summarizes the operations performed by the two Lambda triggers.

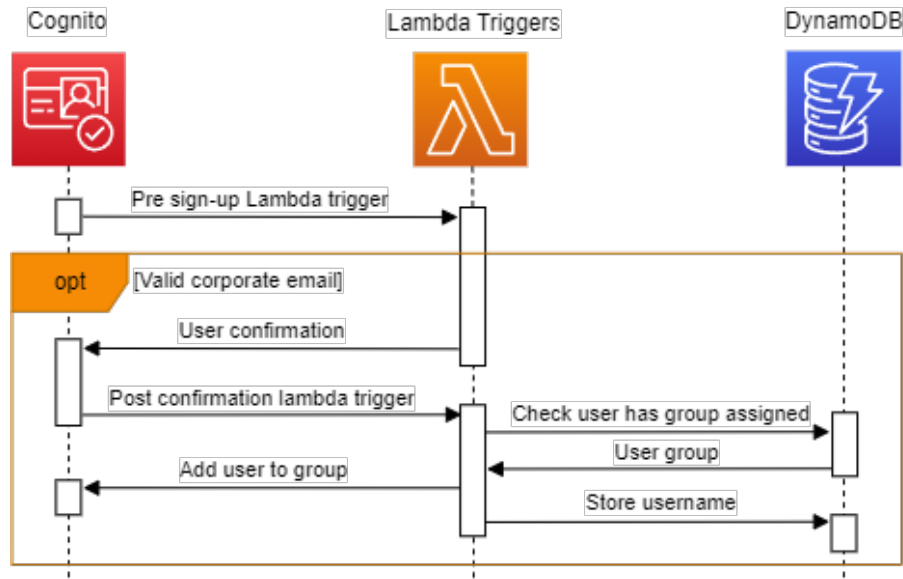


Figure 5.20: Pre-signup and Post confirmation lambda triggers

5.5.3 Manual user group assignment

After the initial user group assignment, that can happen either when the user first logs in or when the *Resource* entity is created, the group to which a resource belongs can be manually updated or set to *Unauthorized* as a consequence of the deletion of the resource.

Updating the role of the *Resource* and the group to which they are assigned are two separate operations that can independently fail and must be atomically executed. A DynamoDB stream [4.3.4] has been employed as a trigger for the **UserGroupAssigner** to replicate updates from the DynamoDB *Resource* table to the Cognito user pool to avoid inconsistencies. These changes are performed in an eventually consistent way.

As shown in Figure 5.21 the Managers can perform write operations of *Resources* using GraphQL API calls, which AppSync forwards to DynamoDB. The stream then triggers the Lambda and depending on the type of operation (Create, Delete, Update) it passes to the function respectively the new record, the old one, or both of them. The *Resource* representations contained in the records are called Images. The flow pictured below shows that if the resource email has been updated, then the group of the user associated with the old email is set to *Unauthorized*. In case of creation and updates, the group is updated to the one contained in the new

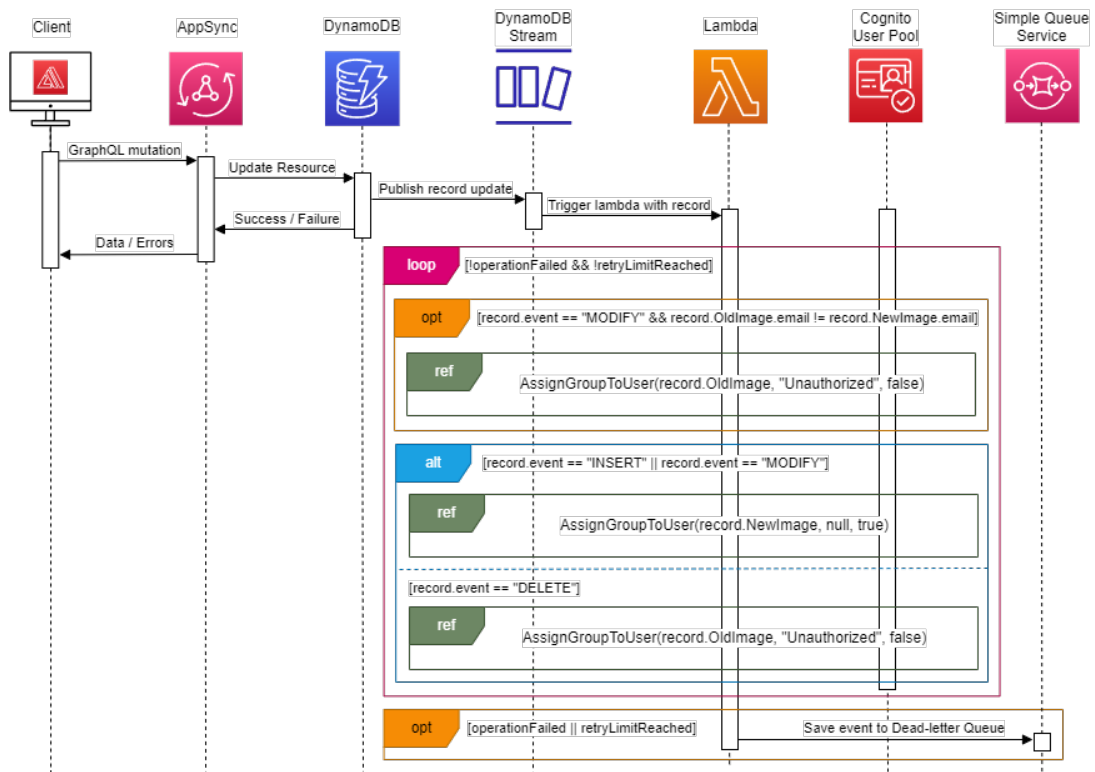


Figure 5.21: User group propagation from DynamoDB to Cognito

image, while for deletion the group is once again set to *Unauthorized*.

Upon failure the Lambda is automatically retried up to 10 times, if this limit is exceeded, a message is sent to the global Dead Letter Queue [4.3.3] for debug purposes.

The actual group assignment is delegated to the function shown in Figure 5.22. It is responsible for retrieving the username of the *Resource* when this is not already present in the image it receives, and optionally saving it in the DynamoDB table for easier access during future operations. The username is necessary for retrieving all currently assigned groups to a user, removing them, and associating the new group that was provided as a parameter.

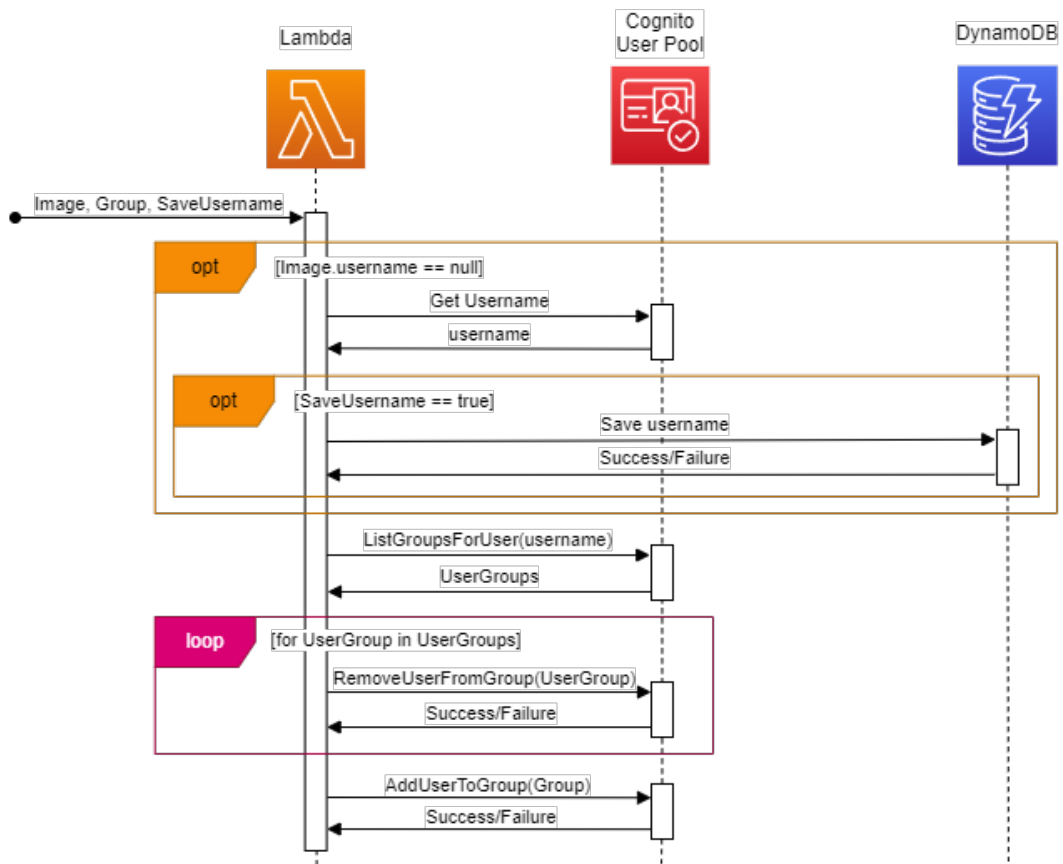


Figure 5.22: AssignUserToGroup sequence diagram

5.6 CRUD operations

All CRUD operations regarding individual models, such as *Customer*, *Release*, *Contact Person*, and *Project* entities, are handled with standard GraphQL queries and mutations, as illustrated in Figure 4.2. Some actions involve multiple entities, such as the case of *Business units* with *Cities* and *Resources* with *Projects*. The transactional support required by these operations is provided by a Lambda function as described in Figure 4.3 sequence diagram.

The CRUD operations for each of the previously mentioned database entities are supported by a respective front end page. Each entity has: one page where all instances are listed, a form to create a new instance, a form to modify the information of an existing one, and an action button to remove it. The functions developed to handle the more sophisticated CRUD scenarios are described in the following sections.

5.6.1 Resources

Resources represent all company employees. They are associated with a *Business Unit* and a *City*. While inserting and updating *Resources* with the *Project Manager*, *Technical analyst*, or *Functional analyst* roles, it's possible to add and assign projects to them. Due to the necessity of updating multiple entities when working with *Resources*, a transaction is employed to ensure data consistency.

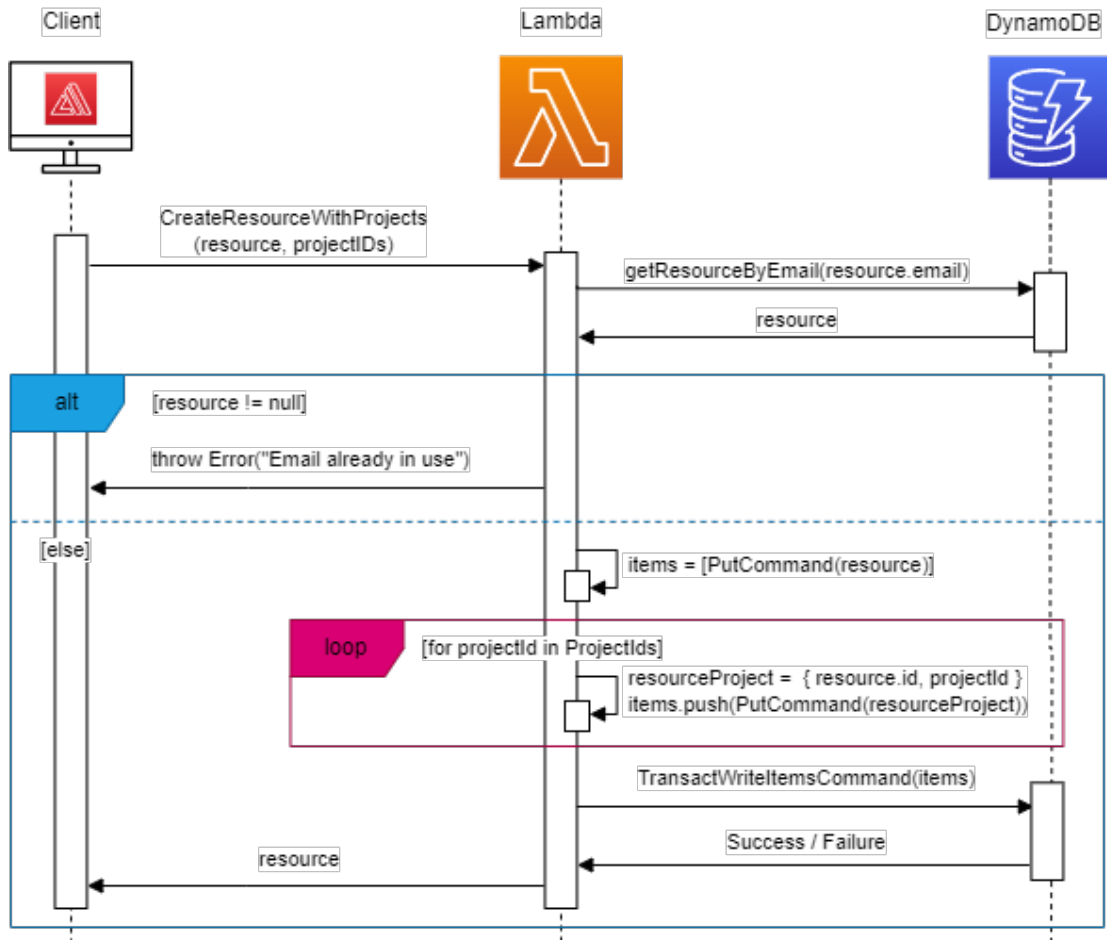


Figure 5.23: Creation of Resource with Projects

Figure 5.23 describes the process of inserting a new *Resource* with assigned *Projects*. First, a check on the existing entities is performed to ensure a *Resource* with the specified email doesn't exist yet, then the *Projects* are mapped into bridge-table entries, finally, these entries, as well as the *Resource* itself, are marshaled into DynamoDB PUT `TransactWriteItems` [22] which are later committed to the database inside a transaction.

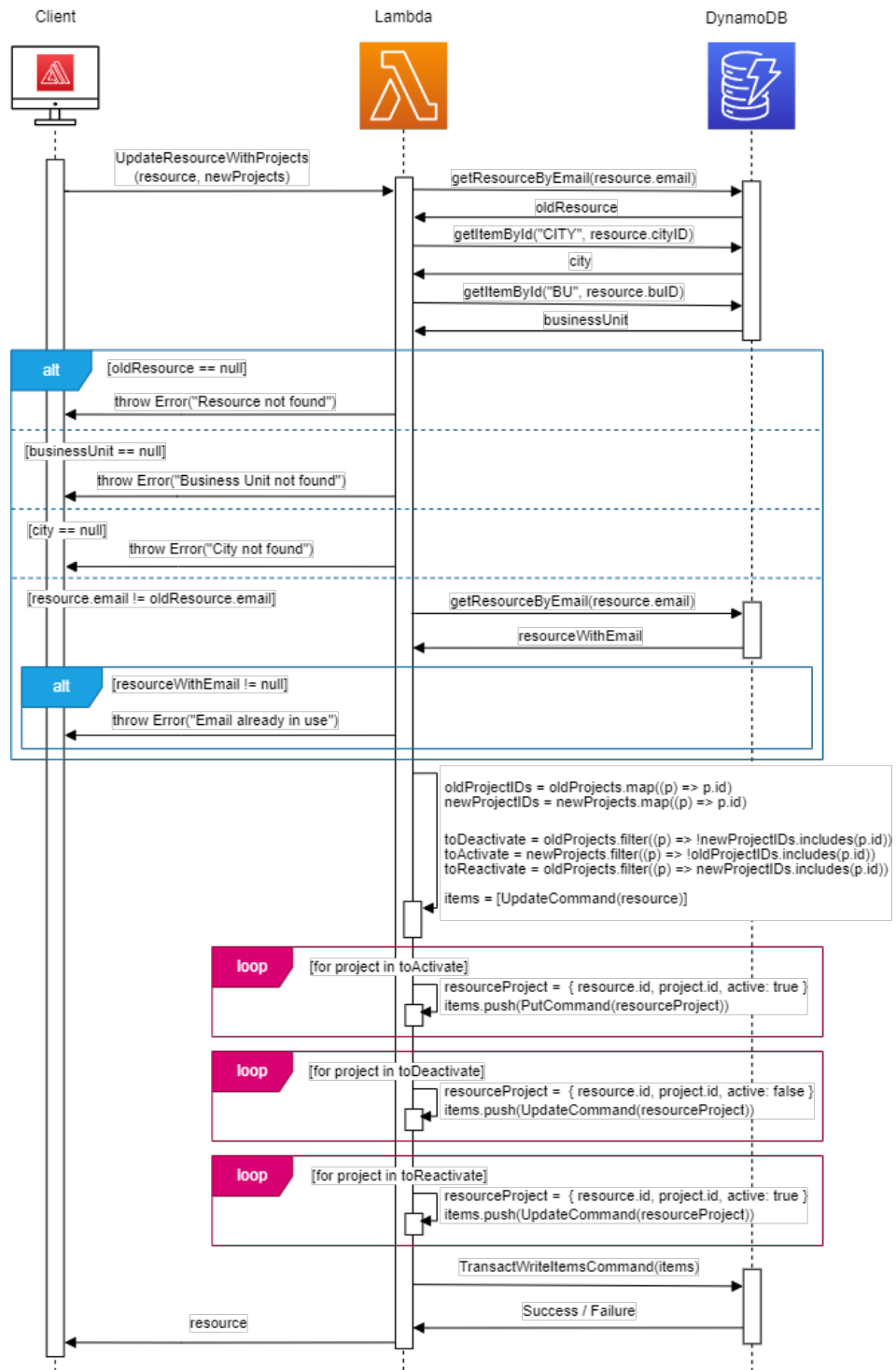


Figure 5.24: Update of Resource with Projects

As Figure 5.24 shows, a single Lambda function is responsible for changing a *Resource*'s personal information as well as updating the *Projects* assigned to them. The parameters required by the Lambda include the set of all *Projects*, called **NewProjects**, on which the employee must be actively working. The function retrieves the set of all *Projects* previously assigned to the *Resource* and stores them in a list called **OldProjects**. The projects belonging to these two groups are then divided into three categories:

- *Projects to activate*: Present in **NewProjects** but not in **OldProjects**, a new record must be inserted in the **Assignment** table.
- *Projects to deactivate*: Present in **OldProjects** but not in **NewProjects**, their *Active* flag is set to false. This approach ensures data is never removed from the **Assignment** table so that it can be used for analysis in the future.
- *Projects to reactivate*: Present in both **OldProjects** and **NewProjects**, their *Active* flag must be set to true, as they could have been previously deactivated.

5.6.2 Business units and cities

Italian *cities* are statically stored in their homonym DynamoDB table. Each city can house zero or more *Business Unit* headquarters, and each *Business Unit* can have offices in many different cities. This many-to-many relationship is addressed with the second strategy explained in subsection 4.3.1: a custom Lambda function is responsible for performing CRUD operations inside a transaction.

5.6.3 Work planning and finalization

Resources can work and report their finalized work only on days that are not weekends, local holidays (such as their city patronal feast), or national holidays. In addition, worker-specific paid leave and vacations must be taken into account.

National holidays

Even though generating all Italian holidays, including Easter, using an algorithm would be feasible, this method would require human intervention in the event holiday dates were to change. The use of publicly available APIs to retrieve the list of holidays, such as Nager Date [23], has shown to be a more robust solution as changes to holidays are delegated to the maintainers of the open-source API. The latter technique is also more flexible if the application has to handle different countries in the future.

A Lambda function is responsible for invoking the holidays API and caching the results in the database *Holiday* table. A Custom resource has been added to the lambda's CloudFormation stack to invoke the function on every new deployment. This ensures the Holiday table is automatically initialized on all new environments with holidays from the current and next year.

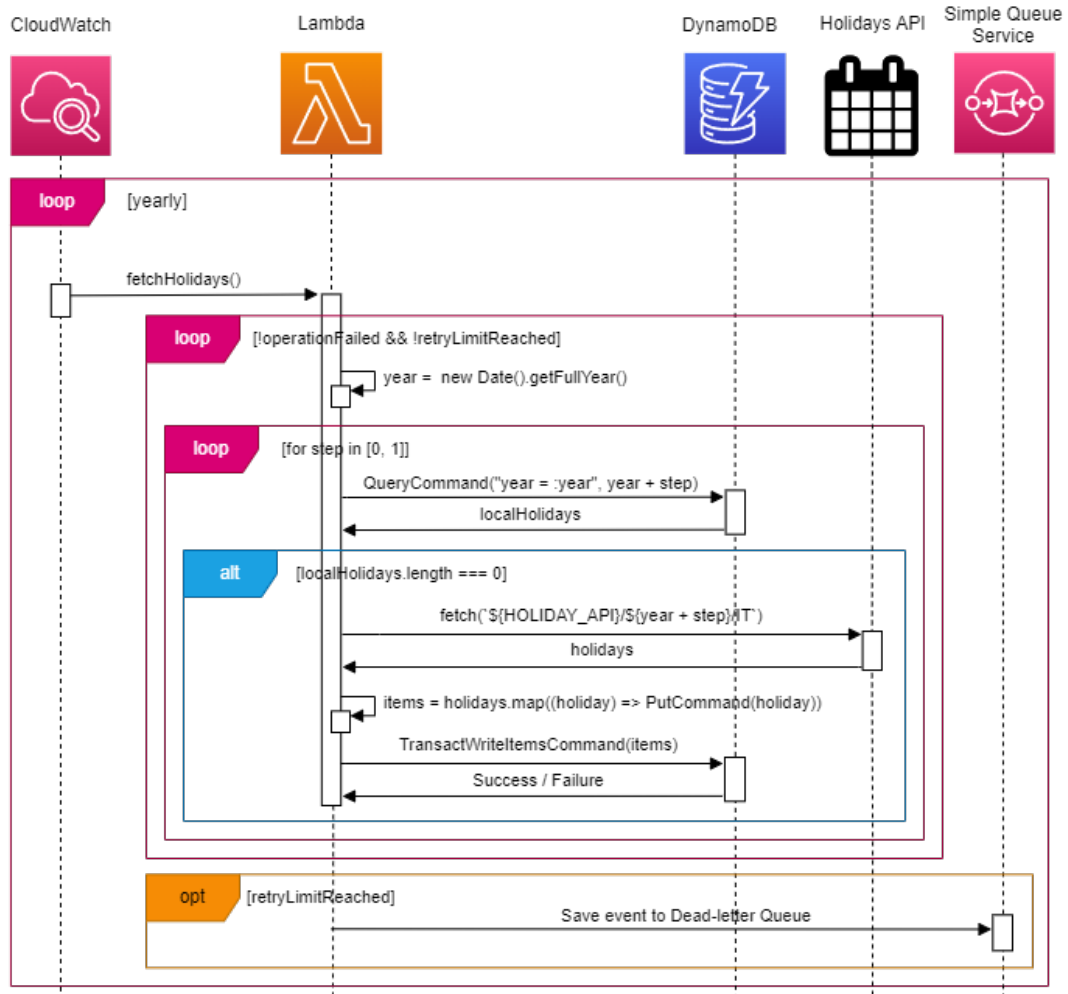


Figure 5.25: Holidays retrieval and store

As shown in the function sequence diagram in Figure 5.25 a **CloudWatch rule** has been set up to recurrently invoke the function every year. The function is set to be called every January, after the first execution it will only fetch the holiday dates for the next year. AWS automatically handles possible failures by re-trying the execution up to 3 times. If the retry limit is exceeded, an error message is sent to the global Dead-letter queue.

Patronal feasts

Resources cannot finalize or have work planned on the day on which their office patronal feast is held. Analysts collaborating on the same project might be working from different headquarters, and thus have different patronal feasts. The patronal feast of each resource is easily retrieved with GraphQL in the same query used to fetch their details.

Paid leave and vacations

Workers can independently request paid leave and vacation days. This information is saved in the Resource DynamoDB object as an array and it's retrieved in the same query as when fetching the resource details.

Future functionalities

Although the data model has been set up to handle work planning and finalization, CRUD operations on *PlannedWork* and *FinalizedWork* are currently lacking UI support and essential backend checks to ensure everything is consistent. To implement these checks, it will be necessary to define a Lambda function responsible for retrieving all *Holidays*, *Patronal feasts*, and *Paid leave/vacations* to ensure no work is scheduled or performed in these timeframes. Additional checks are required to ensure a daily maximum of 8 hours is assigned to each *Resource* across all *Projects* they are working on.

5.7 Microfrontends

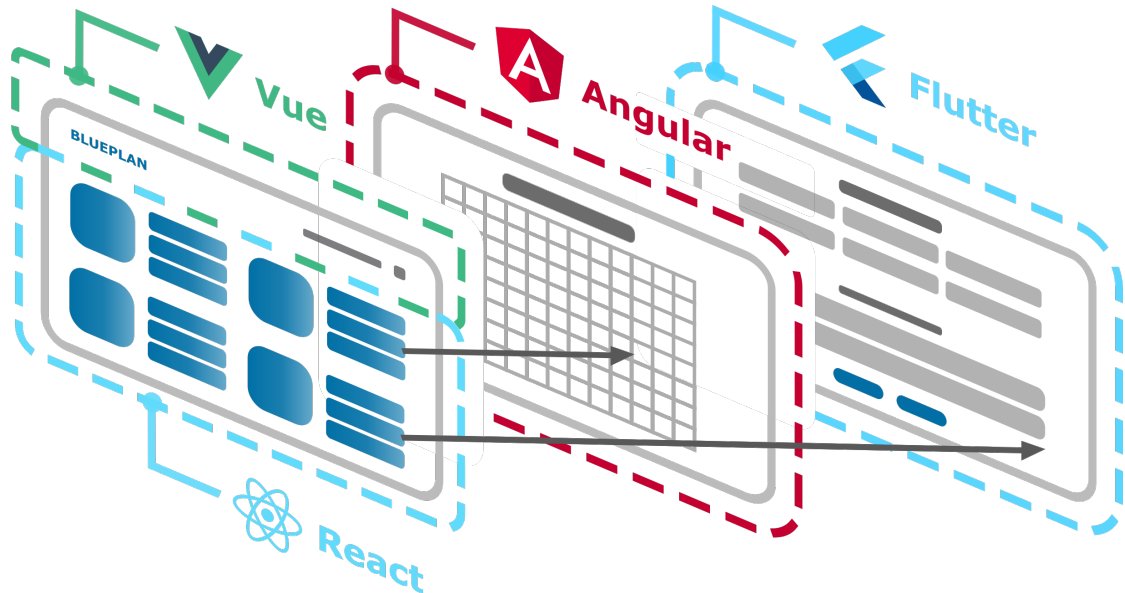


Figure 5.26: Microfrontends schema

The microfrontends architecture has been embraced in this application mainly to explore its usability for future projects. To showcase the feasibility of integrating different frameworks, the navbar and the homepage of the monolithic UI in Flutter have been replaced respectively by a Vue and a React implementation. As shown in Figure 5.26, in the future new functionalities could be easily implemented as separate microfrontends, such as an Angular one.

The various applications have been integrated into a unique front end following *The Recommended Setup* [24] of the Single-SPA [12] framework. Single-SPA is responsible for lazy-loading the individual UI fragments based on the current URL of the page and structuring them according to a user-defined layout.

Composition

The root microfrontend establishes for which URLs and in which part of the DOM each application is rendered. BluePlan has been set up to load:

- The Navbar at the top of the page for all paths.
- The Home page below the navbar, only when the path is `/`.

- The wrapped Flutter app is loaded underneath the navbar for all URLs except the one for the Home page.

Authentication

The root microfrontend is responsible for providing solutions to cross-cutting concerns to all other applications, such as performing user authentication. It interacts with Cognito resources and the Azure Active Directory as described in subsection 5.4.2. The authentication tokens are automatically saved by Amplify in the browser's local storage, every JS-based microfrontend can access them directly. For the Flutter app, the tokens are retrieved by a Dart method which invokes a JavaScript function that directly accesses the local storage. This cooperation is possible due to Flutter's JavaScript interoperability [25].

Performance

CloudFront cache policy ensures microfrontend bundles are efficiently served to the clients requesting them. Application load time can be further improved by configuring WebPack to consider large libraries as external [26]. This results in dependencies not being directly included in the output bundles, thus reducing their overall size and improving download time. Libraries used by multiple microfrontends are set as external, this way the browser downloads them only once and reuses them whenever needed.

The bundler expects the libraries marked as external to exist as an in-browser module. To use dependencies as in-browser modules they must be defined in an import map in the root frontend. This import map links bare import specifiers to libraries hosted on the **unpkg** [27] and **jsDelivr** [28] Content Delivery Networks.

To ensure the correct behavior on browsers not yet supporting import maps, SystemJS [29] has been used to provide polyfill-like behavior for import maps and in-browser modules.

Chapter 6

Evaluation

6.1 Testing

Automated software testing is essential to ensure the reliability of applications. It allows detecting issues during development and facilitates identifying regressions by verifying that new code changes do not break existing functionalities.

A comprehensive testing approach has been used throughout the development of the *BluePlan* application, ranging from the simple evaluation of a small portion of code with *unit tests* to integrating multiple aspects of the application through *end-to-end tests*. Lambda functions have also been tested by means of *Chaos tests*.

As a result of running tests, a coverage report in the LCOV format is generated. The obtained line coverage, the number of code smells, and vulnerabilities are reported in Table 6.1 for all applications. The microfrontend styleguide app is not included in the assessment as it is currently empty.

Application	Covered lines	Line coverage	Code smells	Vulnerabilities
Monolith	2891/3339	86.6%	0	0
Microfronted Root	105/147	71.4%	0	0
Microfronted Home	127/130	97.7%	0	0
Microfronted Navbar	100/123	81.3%	0	0

Table 6.1: BluePlan line coverage and technical debt metrics

6.1.1 Unit test

Unit tests are assessments of small isolated components or functions to verify their correctness. They may require the use of mocks to simulate external dependencies of the components being tested. Mocks are custom objects used to simulate the

behavior of real dependencies. Unit tests have been performed on all microfrontend applications and have been added to the Test phase of the CI/CD pipeline.

Flutter provides native support for unit testing, while the Jest [30] framework was employed to perform unit tests on JavaScript-based applications.

6.1.2 End-to-end test

End-to-end (E2E) tests simulate real user interactions to make sure that the software behaves as expected in a production-like environment. They make use of real-world dependencies, such as AWS services, without mocking them. End-to-end tests have also been added to the Test phase of the CI/CD pipeline of each application.

In Flutter, E2E tests have been implemented using the Patrol framework [31]. Patrol builds on top of Flutter’s test tools to provide a simpler interface for retrieving and interacting with the widgets inside the page.

Test users

A Lambda function called *TestUsersManager* has been developed to enable the creation and deletion of users to perform authenticated actions during tests. As shown in Figure 6.1, the function takes an action (“CREATE” or “DELETE”) and an array of users as its parameters, each user is described by an email, a password, and a group. The Lambda creates a test user in the Cognito user pool and assigns it to the indicated user group, then it inserts a corresponding record in the *Resource* DynamoDB table.

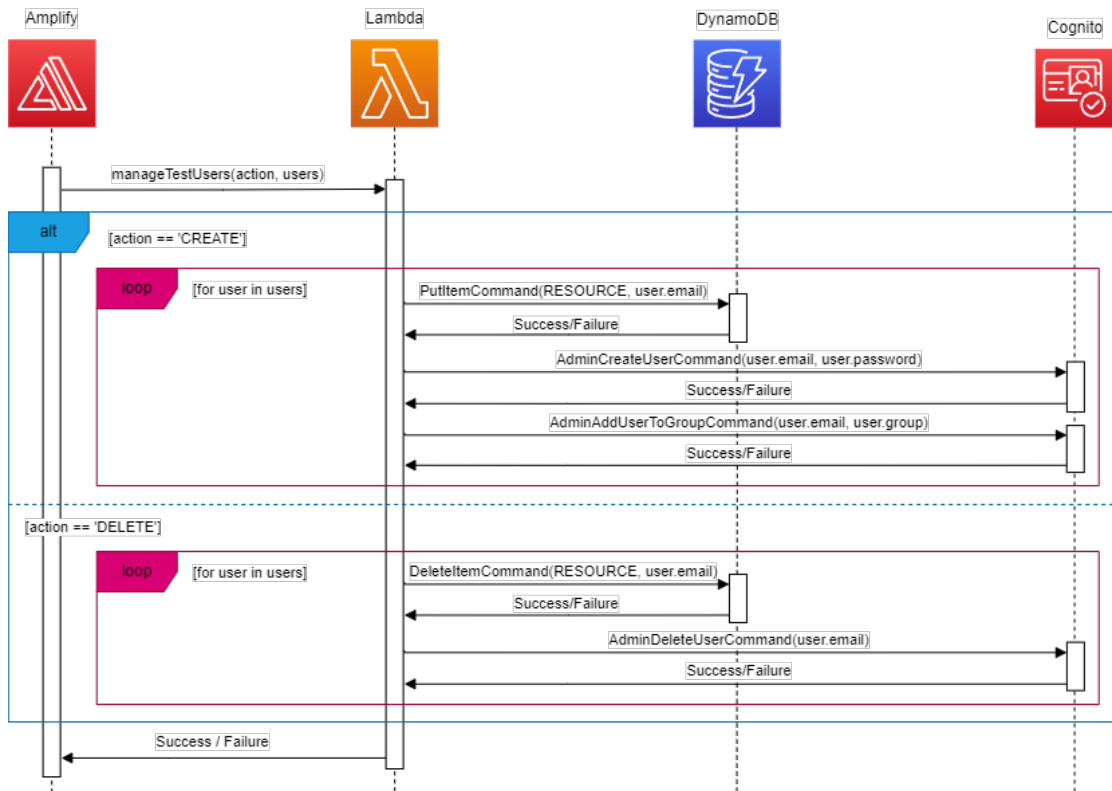


Figure 6.1: Test Users Manager

The Lambda can be executed in all non-production environments and it can be invoked only by the Amplify CI/CD process and by the developers who have been granted with the necessary permissions.

At the beginning of the end-to-end tests the Lambda is called to create the necessary test users, they are then deleted whenever the test suite is completed.

6.1.3 Chaos test

Chaos testing is the process of simulating failures in a given environment to assess if the system is capable of handling them correctly. Although the concept of chaos testing is usually mainly applied to microservices, in the context of this application it has been adapted to allow testing Lambda functions depending on other AWS services.

Due to the lack of an official AWS tool to simulate failures in a Lambda, a new version of the function has been created, where each call to an AWS service has been

wrapped into a method that has a 0.2 random probability of triggering an exception.

The expected outcome is that the Lambda is automatically re-tried until its retry limit is reached, and if that happens, then an error message is sent to the global Dead Letter Queue.

The Lambdas have been launched until all possible scenarios were covered, that is until all exceptions were launched and until the Lambda completed successfully due to it being retried after a failure.

6.1.4 Static code analysis

Static code analysis is an automated debugging technique that allows the detection of code smells and vulnerabilities in the source code of an application without the need to execute it.

The analysis of the source code has been performed using Flutter analyze [32] for Flutter and SonarQube [16] for JavaScript applications. The total amount of code smells and vulnerabilities has been consistently kept at 0, by inserting the code analysis as a mandatory step in the CI/CD pipeline.

6.2 Monolith and Microfrontends performance

The performance of the monolithic application entirely written in Flutter, and the one using microfrontends, have been compared using Web Vitals [33].

6.2.1 Web Vitals

Web Vitals are metrics ideated by Google to provide a quantitative assessment of web pages performance, interactivity, and visual stability. These aspects are evaluated by the following metrics:

- **Largest Contentful Paint (LCP)** [34]: the point in time when the page's main content has likely loaded. LCP helps evaluate the *perceived load speed* of the web page. To achieve a good user experience, LCP must occur within 2.5 seconds since the web page starts loading.
- **First Contentful Paint (FCP)** [35]: the time elapsed between the initial loading of the page and the moment when the browser renders the first DOM element. FCP also measures the *perceived load speed*. An FCP of less than 1.8 seconds is necessary to provide a good UX.

- **Time To Interactive** (TTI) [36]: It's the earliest moment after First Contentful Paint when the page is reliably ready for user interactivity. TTI measures *load responsiveness* and it's useful for identifying when a page looks interactive but in reality, it is not. TTI should be below 5 seconds.
- **Total Blocking Time** (TBT) [37]: measures the amount of time between First Contentful Paint and Time to Interactive where the main thread was blocked enough to cause a delay in input response. TBT helps measure a page load responsiveness. To provide a good user experience Total Blocking Time should be less than 200 milliseconds.
- **Cumulative Layout Shift** (CLS) [38]: Google defines CLS as a measure of the *largest burst* of *layout shift scores* for every unexpected layout shift occurring during the entire time the web page is loaded. CLS is used to evaluate the website's *visual stability*. A CLS value of 0.1 or less is needed to maintain a good user experience.

A *layout shift* is a change in the position of a visible element from one rendered frame to the next.

A *burst of layout shifts* is the occurrence of one or more individual layout shifts with less than 1 second in between them and within a time frame of a maximum of 5 seconds.

The *largest burst* is the one with the highest cumulative of all layout shifts within the time frame.

6.2.2 Lighthouse reports

Lighthouse [39] is a tool developed by Google for inspecting and improving web performance. It has been used to collect the previously mentioned Web Vitals metrics for the monolithic and microfrontends approaches.

The information collected by Lighthouse is defined as *Lab data*, meaning that it has been generated by loading a page on a single device subject to specific network constraints. The tests have been performed with a network throttling of 40ms TCP Round Trip Time (RTT) and a simulated throughput of 10,240 kb/s.

Web Vitals have been collected for three representative pages of each application: the home, the *Customers* list, and the *Resource* creation form. Table 6.2, Table 6.3, and Table 6.4 showcase the obtained results color-coded according to the convention adopted by Lighthouse [40]: 0 to 49 (red): Poor, 50 to 89 (orange): Needs Improvement, 90 to 100 (green): Good. The value of each metric is scored on a range from 0 to 100 by looking at where the value falls on a distribution derived

from the performance metrics of real websites in the HTTP Archive [41]. The overall score is then computed as a weighted average of the metric scores.

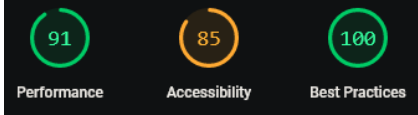
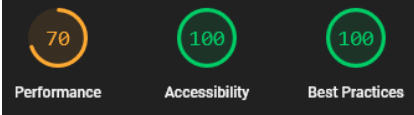
	Monolith	Microfrontends
Home		
LCP	0.6 s	1.4 s
FCP	0.3 s	0.8 s
TTI	0.8 s	0.8 s
TBT	205 ms	0 ms
CLS	0.000	0.246

Table 6.2: Monolith and microfrontend homepage Web Vitals comparison

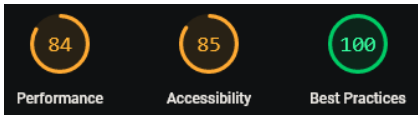
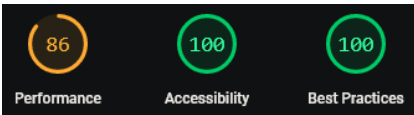
	Monolith	Microfrontends
Customer list		
LCP	0.9 s	1.7 s
FCP	0.3 s	0.9 s
TTI	0.8 s	1.2 s
TBT	303 ms	140 ms
CLS	0.000	0.001

Table 6.3: Monolith and microfrontend customer list Web Vitals comparison

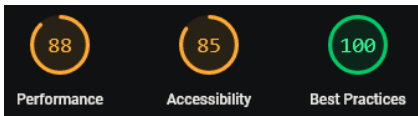
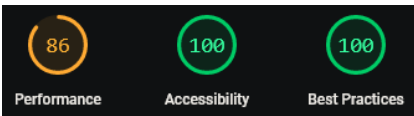
	Monolith	Microfrontends
Resource form		
LCP	0.7 s	1.6 s
FCP	0.3 s	0.8 s
TTI	0.8 s	2.7 s
TBT	260 ms	144 ms
CLS	0.000	0.001

Table 6.4: Monolith and microfrontend resource form Web Vitals comparison

In terms of the Largest Contentful Paint and First Contentful Paint, the monolithic Flutter app consistently outperforms the microfrontends version across all pages, thus resulting in a better loading speed perceived by the users. Time to Interactive remains constant for the monolithic application while it varies significantly for the microfrontends implementation. This suggests that the former may provide more predictable and responsive interactions. The stability of content is equally good across applications, with the notable exception of the microfrontends Home page, where further improvements are needed to provide a more stable page load. The microfrontends-based app obtains lower TBT values, resulting in fewer delays due to main thread activities. Its counterpart has higher TBT on all pages, resulting in higher latency for the end user.

In summary, the monolithic approach demonstrates slightly superior performance compared to the microfrontends-based application. It delivers faster loading times and better interactivity. On the other hand, the microfrontended application is better at providing lower Total Blocking Time and the two approaches obtain comparable results for the CLS metric.

The observed behavior can be explained by recognizing that the Flutter application is fully incorporated into the microfrontends app, which, in turn, necessitates downloading additional frameworks such as Vue and React. Optimal performance could have been achieved by adopting the same framework across all microfrontend applications.

However, the difference between the overall performance of the two approaches is marginal. The microfrontends architecture proved to be a valid option, worth adopting if the development teams can benefit from the advantages it offers.

Chapter 7

Conclusions

7.1 Results

This thesis has provided a comprehensive description of the software development lifecycle of the BluePlan web application, covering each phase from initial analysis to design, implementation, and evaluation. BluePlan empowers managers to efficiently keep track of the company's business units, employees, customers, and projects and provides project managers with a comprehensive overview of project deadlines and how resources are assigned to projects.

The back end infrastructure has been developed exploiting many AWS functionalities while the UI was written initially entirely using Flutter and later it has been transformed into a microfrontends architecture. The performance and accessibility of the two front-end implementations have been evaluated and compared by means of Lighthouse reports using Web Vitals as metrics. These assessments have revealed that both monolithic and microfrontend approaches are not only accessible but also efficient, although there is still room for improvement.

Requirements have been only partially completed because although the back end has already been designed to handle working hours finalization, the user interface is not ready to support this functionality. All other requirements have been fulfilled as planned.

The development process followed standards and practices to achieve high code quality, including technical debt management and comprehensive testing that led to extensive code coverage. This ensured that no code smells or vulnerabilities were introduced, ultimately guaranteeing the robustness and reliability of the BluePlan web application.

BluePlan will play a significant role in streamlining project management operations and improving productivity within the organization. Due to its flexible architecture, the application lays a solid foundation for future functionalities.

7.2 Future Developments

As the application starts replacing the old project management process, the workflow can be further optimized by adjusting it based on user feedback.

The web application also has a large potential in terms of future functionalities. The microfrontend architecture provides flexibility to the developer teams allowing them to work on independent features at their own pace, and without interfering with others work.

New functionalities could include:

- **Recommendation:** Assisting Managers and PMs in the decision of which *Resources* should be assigned to a *Project* by prompting the most suitable candidates based on their availability and skills.
- **Improved Logging:** Add advanced logging of events with *Amplify analytics* to collect data about application usage to improve its monitorability.
- **Analytics:** Examine completed projects data to gather insights about what could have been improved to achieve higher efficiency in future activities.
- **Reporting:** Improved dashboard with charts for a better panoramic view on company resources and automatic PDF reports generation.

Acknowledgements

I would like to thank my family and friends for accompanying me during this journey and for always supporting me.

Thanks to all the professors that I have had the opportunity to meet during these years of university and in particular to my supervisor, Professor Luigi De Russis, for the support and advice he provided me during the writing of this thesis.

A big thank you to all the colleagues I had the pleasure of meeting throughout my internship at Blue Reply.

Last but not least, I would like to give a big shoutout to all the amazing people at the Mu Nu Chapter of IEEE-HKN.

Thank you.

Sitography

- [1] *Blue Reply*. URL: <https://reply.com/blue-reply> (cit. on p. 1).
- [2] *Microfrontends*. URL: <https://martinfowler.com/articles/micro-frontends.html> (cit. on p. 5).
- [3] *Flutter*. URL: <https://flutter.dev> (cit. on p. 9).
- [4] *Dart*. URL: <https://dart.dev> (cit. on p. 9).
- [5] *Angular*. URL: <https://angular.io> (cit. on p. 10).
- [6] *React*. URL: <https://react.dev> (cit. on p. 10).
- [7] *Vue*. URL: <https://vuejs.org> (cit. on p. 10).
- [8] *NodeJS*. URL: <https://nodejs.org> (cit. on p. 10).
- [9] *V8 JavaScript Engine*. URL: <https://v8.dev> (cit. on p. 10).
- [10] *GraphQL*. URL: <https://graphql.org> (cit. on p. 11).
- [11] *Webpack*. URL: <https://webpack.js.org> (cit. on p. 11).
- [12] *Single SPA*. URL: <https://single-spa.js.org> (cit. on pp. 11, 55).
- [13] *Amazon Web Services*. URL: <https://aws.amazon.com> (cit. on p. 12).
- [14] *Jira*. URL: <https://atlassian.com/software/jira> (cit. on p. 13).
- [15] *Confluence*. URL: <https://atlassian.com/software/confluence> (cit. on p. 13).
- [16] *SonarQube*. URL: <https://www.sonarsource.com/products/sonarqube> (cit. on pp. 13, 60).
- [17] *AWS Pricing calculator*. URL: <https://calculator.aws> (cit. on p. 14).
- [18] *Import Maps*. URL: <https://github.com/WICG/import-maps> (cit. on p. 25).
- [19] *Azure Active Directory*. URL: <https://azure.microsoft.com/en-us/products/active-directory> (cit. on p. 42).
- [20] *OAuth 2.0*. URL: <https://oauth.net/2> (cit. on p. 42).
- [21] *OAuth 2.0 Authorization Code Grant*. URL: <https://oauth.net/2/grant-types/authorization-code> (cit. on p. 43).

- [22] *DynamoDB TransactWriteItems*. URL: https://docs.aws.amazon.com/amazondynamodb/latest/APIReference/API_TransactWriteItems.html (cit. on p. 50).
- [23] *Nager Date public holidays API*. URL: <https://github.com/nager/Nager.Date> (cit. on p. 52).
- [24] *Single SPA Recommended Setup*. URL: <https://single-spa.js.org/docs/recommended-setup> (cit. on p. 55).
- [25] *Flutter-JavaScript interoperability*. URL: <https://dart.dev/web/js-interop> (cit. on p. 56).
- [26] *Webpack Externals*. URL: <https://webpack.js.org/configuration/externals> (cit. on p. 56).
- [27] *Unpkg CDN*. URL: <https://www.unpkg.com> (cit. on p. 56).
- [28] *jsDelivr CDN*. URL: <https://www.jsdelivr.com> (cit. on p. 56).
- [29] *SystemJS module loader*. URL: <https://github.com/systemjs/systemjs> (cit. on p. 56).
- [30] *Jest*. URL: <https://jestjs.io/> (cit. on p. 58).
- [31] *Patrol testing framework*. URL: <https://patrol.leancode.co> (cit. on p. 58).
- [32] *Flutter analyze*. URL: <https://dart.dev/tools/analysis> (cit. on p. 60).
- [33] *Web Vitals*. URL: <https://web.dev/vitals> (cit. on p. 60).
- [34] *Largest Contentful Paint*. URL: <https://web.dev/lcp> (cit. on p. 60).
- [35] *First Contentful Paint*. URL: <https://web.dev/fcp> (cit. on p. 60).
- [36] *Time to Interactive*. URL: <https://web.dev/tti> (cit. on p. 61).
- [37] *Total Blocking Time*. URL: <https://web.dev/tbt> (cit. on p. 61).
- [38] *Cumulative Layout Shift*. URL: <https://web.dev/cls> (cit. on p. 61).
- [39] *Lighthouse*. URL: <https://developer.chrome.com/docs/lighthouse> (cit. on p. 61).
- [40] *Lighthouse performance scoring*. URL: <https://developer.chrome.com/en/docs/lighthouse/performance/performance-scoring> (cit. on p. 61).
- [41] *HTTP Archive*. URL: <https://httparchive.org/> (cit. on p. 62).