POLITECNICO DI TORINO

Master's Thesis in COMPUTER ENGINEERING

Politecnico di Torino

Design and Implementation of a Progressive Web Application

Supervisor
Prof. Giovanni MALNATI

Candidate
Yalda Sadat MOBARGHA

December 2025

Acknowledgements

First and foremost, I would like to express my sincere gratitude to Prof. Giovanni Malnati for giving me the opportunity to work on this project. This period has allowed me to better understand the path I want to pursue, and I am truly grateful for having spent it under the right guidance.

I also want to thank my friends and colleagues at Tonicminds, especially Giovasini, Nicolò, Matteo, Alessio, Alessandra, Luca, and Giuseppe, each of whom supported me throughout these months, taught me along the way, and made this time full of good moments and happy days.

I am deeply thankful to Gabriele, who has always supported me, taught me things that go far beyond academic work, and reminded me of the joy of learning.

My deep appreciation goes to Farnaz, who, despite the more than 3.5 kilometres distance between us, supported me through every exam, every challenge, and every moment of difficulty, always giving me strength and hope.

I want to thank my family, without whom starting this path would not have been possible.

Last but not least, I want to thank my partner, Alireza, who has been by my side from the very first day of this journey and has always been there for me in challenging moments.

I would also like to thank all my friends, whether mentioned here or not. It has been a memorable journey, and I'm grateful I got to share it with you.

Abstract

In the context of digital transformation, companies have to deal with their outdated approaches. Data management based on spreadsheets or fragmented desktop software often creates difficulties with consistency, traceability, and long-term maintainability. To solve these issues, companies are looking to adapt their internal workflows and information management systems towards more flexible and cross-platform technologies.

This thesis is focused on the development of a system based on a Progressive Web Application (PWA) that is designed to optimize and unify operations across all divisions of a large poultry company, replacing a spreadsheet system with a centralized, structured, and easily accessible platform. The developed application serves as a comprehensive, Business Process Model and Notation (BPMN) oriented task management system: it handles repetitive, process-driven activities, each managed by a defined workflow to ensure consistent and traceable execution across the organization. It enables users to record observations, plan and monitor production, manage sales, and control activities throughout the animals' lifecycle. Through standardized workflows and role-based access, employees can carry out their tasks in a consistent and guided manner, while managers have complete visibility over the overall process.

PWAs combine the accessibility of web technologies with the capabilities of native mobile applications. They offer significant advantages, as a single implementation effort can deliver both web and mobile experiences, resulting in reduced overall development, maintenance costs, and deployment simplification. Furthermore, PWAs support offline functionality, caching strategies, push notifications, and automatic updates, ensuring reliability and usability even in low-connectivity environments.

The system has been developed using a component-based web architecture and adopts a layered design: React delivers a modular UI; a Backend-for-Frontend mediates and optimizes interactions with enterprise services; Camunda orchestrates BPMN-defined processes, and notifications are delivered through cloud messaging. The modularity, maintainability, and operational reliability stemming from this approach have helped the development and deployment of the application, reducing the time necessary to validate its effectiveness with the actual stakeholders, that have been able to provide in a timely manner the necessary feedback to improve it further.

Table of Contents

List of Figures

Chapter 1

Introduction

This chapter introduces the general context and motivation behind the project, presents the case study company, and defines the problem that led to the development of the proposed system. It also outlines the main technological choices and the overall structure of the thesis.

1.1 Introduction

Digital transformation is reshaping how companies manage operations, data, and collaboration. Across industries, organizations are adopting digital systems to improve efficiency, transparency, and responsiveness. The agri-food sector is following the same path, moving away from manual procedures toward integrated platforms for process management and traceability.

This thesis describes the design and development of a web-based workflow management platform created within this broader digitalization context. It aims to show how modern web technologies can support structured, traceable, and scalable process management in an industrial environment.

1.2 Company introduction

Martini Alimentare is an Italian company active in the agri-food sector, mainly focused on the production and processing of fresh meat such as chicken and pork. It forms part of the wider Gruppo Martini, which operates across the entire supply chain — from livestock farming and breeding to slaughtering, processing, and the preparation of ready-to-cook meat products. This structure allows the group to oversee each stage of production internally and maintain consistent quality standards throughout. The company has implemented a traceability system that identifies the origin of each product throughout the supply chain; since 1999, meat

packages have included the name of the farm of origin. In 2014, Martini Alimentare established a new production site with a focus on research, development, and product innovation, reinforcing its presence in both domestic and international markets. Today, Martini represents a well-established enterprise within the Italian food industry, characterized by a vertically integrated model that combines farming, processing, and distribution within a single organization [1, 2].

1.2.1 Context and Motivation

As mentioned in the general introduction, in recent years, digital transformation has become a key factor for improving efficiency and transparency in industrial environments, including the agri-food sector. Many companies have gradually moved from manual documentation to digital systems that allow more accurate monitoring and data sharing across departments. Martini had already achieved a high level of process organization and traceability, supported by detailed documentation and well-defined workflows. However, most of this documentation was maintained through extensive Excel sheets, which, while complete and precise, made daily operations difficult to manage as the system grew in size and complexity. When data is distributed across multiple files or manually updated, tracking the progress of operations and maintaining consistency across different teams can require significant effort. Over time, this approach may introduce redundancy, reduce visibility, and make error detection or historical analysis less straightforward. While such systems often serve their purpose effectively, the growing need for integration, real-time access, and process automation encourages the adoption of more centralized and structured solutions. These considerations motivated the development of a modern, integrated web application that could preserve the accuracy of the existing documentation while improving usability, automation, and scalability across the company's processes.

1.2.2 Project Motivation

As mentioned in the Context and Motivation section, the increasing push toward digitalization has led many companies to replace traditional documentation systems with web-based tools that allow centralized data management and real-time collaboration. In this context, Martini aimed to modernize its internal workflow management and process documentation through a dedicated web application. The goal was to create a unified digital environment capable of improving process visibility, coordination, and traceability across different production stages. To achieve this objective, the company worked with two development teams. The first, a provider specialized in workflow automation, took care of modeling the business processes and supplying the BPMN-based logic used by the workflow

engine. Our team, working from another company, was responsible for building the surrounding software system — designing the user interface, implementing a Backend-for-Frontend (BFF) layer to link the application with the workflow engine, and managing deployment, configuration, and long-term maintenance. Through this collaboration, the project evolved into a complete web-based platform. The project, therefore, aimed to deliver a single, accessible platform that could unify and simplify process management across the organization, reducing manual effort and improving consistency and reliability in operational data handling.

1.2.3 Problem Definition and Requirements

During the requirements analysis, one of the first needs identified was that the system should be accessible from both desktop and mobile devices. Many employees perform their activities directly in the field—monitoring livestock, recording daily observations, or, in some cases, carrying out veterinary checks to assess the animals' health. For these roles, having a mobile interface was essential, allowing data to be entered immediately and accurately without the need to return to the office or rely on later transcription. Another key requirement concerned the integration of a real-time notification system. Field operators and staff needed to be promptly informed when new tasks were assigned or when the status of a process changed, ensuring that each activity could be carried out as soon as possible. This feature was crucial for maintaining coordination between different production stages and preventing delays in the workflow. In addition to these operational needs, it was also necessary to define what information should be collected for each activity and how it should be structured. Every step of the company's workflow required the recording of specific data—such as measurements, observations, or approval outcomes—that had to be accurately stored and made traceable over time. The new system, therefore, needed to support detailed digital forms for each process step, reflecting the information required in practice and ensuring consistency between users and departments. These requirements served as the foundation for the design of the application and guided the definition of its functional structure.

1.2.4 Production Process Overview

Based on the requirements collected during the initial analysis and the meetings held with the company's process managers, the overall workflow was organized into a set of well-defined operational areas. The result of this analysis was the identification of five main macro-processes, each representing a fundamental phase within the poultry production cycle. Together, these processes cover the complete path from animal selection to product delivery and form the structural basis for the workflows later implemented in the application.

- Planning (Pianificazione): The Planning phase covers the preparation activities preceding each production cycle, including the scheduling and authorization of chick purchases and their transfer to the farms. It ensures that all necessary approvals, health communications, and transport details are in place before the animals arrive, establishing the operational and regulatory foundation for the following stages of production.
- Maturation (Maturazione): The Maturation phase marks the start of the production cycle and focuses on preparing farms and animals for breeding. It includes activities such as environmental setup, initial health checks, and preventive veterinary treatments, ensuring proper sanitary conditions and readiness for the following stages.
- Breeders (Riproduttori) : This phase focuses on managing reproductive flocks and monitoring their productive and health status throughout the breeding cycle. It includes the registration of reproductive activities, scheduled health inspections, and control operations to ensure compliance with veterinary and production standards.
- Hatchery (Incubatoio): This phase manages all activities related to the handling and development of eggs. It includes the receiving of eggs from breeding farms, the start and monitoring of the incubation process, and the registration of hatching results.
- Fattening (Ingrasso): The Fattening phase covers the final stage of the production cycle, where chicks received from the incubation centers are grown until they reach the required market weight. During this period, operators record daily observations, feed consumption, and health data, while veterinary inspections ensure compliance with animal welfare standards. The phase concludes with the closure of the production cycle and the preparation of documentation for product shipment.

Each macro-process is further divided into a set of micro-processes that represent the specific operational steps required for that production phase. These micro-processes define the mandatory activities to be performed, the logical sequence of execution, and the dependencies between different tasks. This hierarchical organization allows the system to manage complex workflows in a coordinated and traceable way, ensuring consistency across all production sites and phases of the supply chain.

1.2.5 Users

Based on meetings with company representatives and the analysis of daily operations, user roles were identified. Each role is responsible for specific tasks within

the production process. Defining these roles is important for setting access levels, designing the user interface, and determining which data each user can view or modify in the system.

- Planner (Pianificatore): Manages production scheduling and coordination. Planners define start and end dates for each production phase, monitor progress, and ensure alignment between operations and the company's production plan.
- Technician (Tecnico): Responsible for recording operational data directly at the production sites. Technicians register observations, daily measurements, and activities related to the animals.
- Veterinarian (Veterinario): Is in charge of animal health and welfare across all production stages. Veterinarians perform inspections, monitor treatments, and validate the health data entered by technicians.
- Manager: Supervises overall performance and validates the main activities. Managers have access to progress data and reports, enabling them to make decisions and ensure that production targets are met.
- Admin: Validates and authorizes critical steps in the workflow, such as confirming purchase orders, verifying recorded data, or approving process completions.

In the next chapters, we will see how these roles influenced the design of the user interface, the definition of access controls, and the overall system architecture.

1.3 Design Choices and Rationale

Based on the requirements and the considerations described in the previous sections, as well as the client's request for a rapid delivery, the decision was made to develop a Progressive Web Application (PWA) rather than two separate applications for desktop and mobile. This approach made it possible to provide users with a single solution that behaves like a native mobile app while remaining fully accessible through any web browser. In addition to significantly reducing development and maintenance time, the PWA approach ensures easier updates, offline access, and cross-device consistency—key factors for operators working both in the field and in office environments.

The system was implemented using a modern web stack designed for flexibility, maintainability, and scalability:

- **React** was used to build a modular and reusable user interface capable of handling dynamic and interactive content efficiently.

- **TypeScript** provides static typing and improved code reliability, supporting large-scale development and long-term maintainability.
- **Redux Toolkit** simplifies state management and ensures predictable data flow across components, improving synchronization between user actions and application state.
- **Cloud Messaging** provides the infrastructure for real-time push notifications, ensuring timely updates and communication.
- **Camunda** is employed for workflow orchestration, allowing the execution of business processes modeled in BPMN.
- **Backend-for-Frontend (BFF)** serves as a dedicated service layer that mediates communication between the client and backend systems, optimizing performance and security.
- **Kubernetes** enables scalable, containerized deployment and simplifies the management of distributed services.

These technologies together provide a reliable, efficient, and extensible architecture that meets both the functional and operational needs of the project. They are described in more detail in the following chapters.

Chapter 2

Background and Related Work

This chapter provides an overview of the main technologies, frameworks, and methodologies that underpin the development of the proposed system. It introduces the fundamental concepts of Progressive Web Applications, state management, and workflow orchestration, along with related tools such as React, TypeScript, Redux Toolkit, and Camunda. The goal is to outline the theoretical and technological background that guided the architectural and implementation choices described in the following chapters.

2.1 Progressive Web Applications

Progressive Web Applications (PWAs) are web applications that use modern web capabilities to deliver an experience similar to native mobile or desktop apps. They combine the reach of the web with the reliability, performance, and user engagement features of native platforms. PWAs are built using standard web technologies—HTML, CSS, and JavaScript—but enhanced with progressive enhancement principles, meaning they work for every user regardless of browser choice or device constraints, and offer additional capabilities on browsers that support them [3, 4].

2.1.1 Features

PWAs are defined by a set of core features that distinguish them from traditional web applications. These include **installability**, **offline functionality**, **push notifications**, **background synchronization**, and **responsive design**. The

core idea is to make web applications reliable, fast, and engaging even in poor network conditions.

- **Reliability:** PWAs load instantly regardless of network conditions, thanks to caching strategies implemented through service workers. This ensures that essential assets and pages remain accessible even when the user is offline or the connection is unstable [5].
- **Performance:** By caching static resources and optimizing network requests, PWAs achieve loading speeds comparable to native applications. Techniques such as lazy loading and code splitting further enhance runtime efficiency [6].
- **Engagement:** PWAs support push notifications, background sync, and home screen installation, fostering continuous interaction and re-engagement with users. They also integrate with system features such as full-screen mode and app icons for a native-like appearance [3, 4].

A PWA is typically identified by three technical criteria: (1) it is served over HTTPS, ensuring secure communication; (2) it includes a *Web App Manifest* that describes metadata such as name, icon, and display behavior; and (3) it registers a *Service Worker*, the component responsible for offline access and background processing. These criteria allow browsers to recognize and promote web apps as installable and trustworthy [5, 3].

2.1.2 Installation

The installation process of a PWA allows users to add the application directly to their home screen or desktop, providing a launch experience similar to native applications. Installation is guided by the presence of the `manifest.json` file, which contains metadata including the app's name, icons, theme color, and display mode (e.g., standalone or fullscreen). When these requirements are met, supporting browsers display an installation prompt, typically through the *beforeinstallprompt* event. Once accepted, the PWA is installed as a shortcut that launches in its own window without browser UI elements, improving immersion and accessibility [4, 3].

2.1.3 Updates

PWAs handle updates differently from both traditional web pages and native applications. Because assets are often cached for offline access, the update cycle is managed through the service worker, which checks for changes in the application's resources and triggers a controlled update process. When a new version of the service worker is detected, it is downloaded in the background and activated only when no active clients (tabs) are using the old version. This prevents disruptions

while ensuring that users receive the latest features and fixes the next time they open the application [6, 5].

2.1.4 Service Worker

The *Service Worker* is the core technology enabling PWAs' advanced capabilities such as offline support, background synchronization, and push notifications. It is a JavaScript file that runs independently from the main web page, acting as a programmable network proxy between the application and the internet. Once registered, the service worker intercepts network requests, serving responses from cache or fetching from the network as needed, following strategies like *cache-first*, *network-first*, or hybrid approaches [5]. Service workers have a defined lifecycle consisting of installation, activation, and idle phases. During installation, they pre-cache essential resources; during activation, they clean up outdated caches; and during idle, they can listen for background events such as push messages. Because they operate independently of any open tab, service workers allow PWAs to handle notifications and background updates even when the app is closed [5, 3]. In modern frameworks like React with Vite, registering a service worker is handled automatically through plugins, which generate and inject the necessary scripts during build time. This integration abstracts away configuration complexity while maintaining compatibility with standard browser APIs [6, 7].

2.2 Notification systems

Notifications are short, time-sensitive messages that inform users about relevant events, updates, or required actions, even when an application is inactive or running in the background. They act as a communication channel between the system and the user, helping maintain engagement, awareness, and responsiveness. Modern mobile operating systems expose dedicated frameworks and APIs that allow applications to deliver alerts at the system level, often including sound, vibration, or visual cues to capture the user's attention [8, 9]. In mobile ecosystems, notifications represent an integral part of the user experience, enabling real-time interaction between users and applications beyond the active session.

2.2.1 user permissions

Modern mobile operating systems enforce strict permission models for displaying notifications, requiring explicit user consent before an application can send alerts. This permission-based design reflects broader privacy and usability principles, ensuring that users retain control over how and when they are interrupted [10, 11]. Upon installation or first launch, users are typically prompted to grant or deny

access to notifications; the decision can later be modified through system settings. Permission management has a direct impact on the effectiveness of notification strategies: studies show that opt-in rates vary significantly depending on the timing and context of the permission request [12]. Applications that transparently explain the value of notifications—such as reminders, workflow updates, or security alerts—tend to achieve higher acceptance and retention levels. Conversely, overuse or irrelevant messaging often results in users revoking permissions or disabling alerts entirely, undermining engagement and trust. To balance engagement with user autonomy, current design guidelines recommend implementing fine-grained controls such as per-category notification settings, silent or scheduled modes, and clear onboarding explanations for notification purposes.[11, 10].

2.2.2 Firebase Cloud Messaging (FCM)

Firebase Cloud Messaging (FCM) is a cross-platform messaging solution that lets you reliably send messages. How does it work?

An FCM implementation includes two main components for sending and receiving:[13]

- A trusted environment, such as Cloud Functions for Firebase or an app server on which to build, target, and send messages.

- An Apple, Android, or web (JavaScript) client app that receives messages via the corresponding platform-specific transport service.

Two message forms are common: notification messages, rendered by the OS, and data messages, handled by the app code; they can also be combined in a single send. Targeting can address a single token (one device), a topic (publish/subscribe groups for fan-out), or a condition (boolean expressions over topics).

2.3 SPAs vs MPAs

In the Front-end technologies, the actual presentation is driven by the HTML content, which is downloaded from the server, by providing the browser with the corresponding URL. The HTML file contains CSS formatting rules and JavaScript snippets that are interpreted by the browser in order to layout the page and control user interaction. Building on this basic infrastructure, different paradigms have been developed. Multi-page applications (MPA) and Single-page applications (SPA).

Multi-page applications (MPA) In MPA, each page maps to a distinct server URL. When the browser requests a page, the server renders a fresh HTML document—often fetching data from a database and applying business logic—and the browser's role is limited to presenting that content. Every user interaction that triggers navigation causes a full document reload: the browser unloads the previous page, constructs a new representation, and no interaction is possible during the transition (beyond canceling the request). This model typically yields slower navigation, increases server load, and manages application state on the server side—commonly via cookie-backed sessions.

Single-page applications (SPA) In a single-page application (SPA), the browser performs a one-time load of an HTML shell plus JavaScript and CSS assets, and afterwards updates the view dynamically without full page reloads. Client-side code renders content for the current URL, intercepts navigation events, and fetches data from the server (typically JSON via REST/HTTP) to update only the necessary parts of the UI. As routing and rendering are handled on the frontend, the server is largely stateless and focuses on providing APIs, while the browser maintains application state—commonly with libraries such as Redux, MobX, or React's Context API. This model yields smoother, app-like interactions (no blocking reloads), with typical client-managed states including shopping carts, multi-step wizards, user preferences, and table filters/pagination. In short, a single HTML document is loaded at startup, and JavaScript is responsible for dynamic updates and server communication throughout the session.

2.4 Frameworks and Build Tools

This section outlines the core choices in frontend, like React for the UI layer, TypeScript for type safety, and Vite for fast development.

2.4.1 React

React is an open-source JavaScript library for building interactive user interfaces. It was originally developed by Facebook and has since become one of the most widely used tools for developing modern web applications. React allows developers to create reusable components. Its declarative and component-based architecture simplifies the process of designing dynamic, data-driven views by abstracting the complexity of direct DOM manipulation [14]. React applications are often structured as Single Page Applications (SPAs), where the library efficiently updates only the necessary parts of the page through a virtual representation of the DOM, known as the Virtual DOM. React's ecosystem provides a rich set of extensions

and tools, including Hooks for managing state and side effects, React Router for navigation, and close integration with state management libraries such as Redux. Its flexibility, performance, and large community have contributed to its adoption by major companies and developers worldwide. Furthermore, React is fully compatible with TypeScript, supporting static type checking and improving maintainability in large-scale projects [15].

2.4.2 TypeScript

TypeScript is an open-source, statically typed superset of JavaScript developed by Microsoft. It extends JavaScript by adding optional static typing, interfaces, and advanced language features that enable early detection of errors at compile time, improving code quality and maintainability. TypeScript code is transpiled into standard JavaScript, ensuring compatibility with any environment that supports ECMAScript. The language introduces modern programming constructs such as enums, generics, and modules, providing a more structured and scalable foundation for large applications compared to plain JavaScript [16, 17]. TypeScript enhances developer productivity through features like type inference, code autocompletion, and robust refactoring tools. Its strong integration with popular frameworks such as React allows developers to define component props, state, and hooks with precise type annotations, reducing runtime errors and improving readability [18]. The use of TypeScript is now widespread in large-scale web development, where type safety and maintainability are crucial for long-term project reliability and collaboration.

2.4.3 Vite

Vite is a modern frontend build tool and dev server focused on speed and a streamlined developer experience. It serves source files via native ES modules for instant server start and fast hot module replacement during development, and uses an optimized Rollup-based pipeline to bundle and optimize assets for production. Vite is framework-agnostic (e.g., React, Vue, Svelte), supports TypeScript/JSX out of the box, and offers a rich plugin ecosystem, making it a lightweight alternative to traditional bundlers in many projects [19].

2.5 State management and Redux

This section explains why state management matters and how to approach it in practice. First, “application state” is explained, and then Redux is introduced as a predictable, one-way data-flow model for shared state.

2.5.1 State in Frontend Applications

In single-page applications (SPAs), application state refers to the data that controls rendering and behavior(authentication, user preferences, domain entities, loading/error flags). As applications develop, sharing this state across various components may result in duplication and updates that are difficult to understand. Centralized state management was created to make changes more explicit and predictable: instead of directly changing the state, an app describes changes as actions and pure reducer functions. Compute the next state based on the previous state plus the action. In Redux, the entire app's state is stored in a single store, and large apps create the root reducer from smaller feature reducers that operate on different sections of the state tree.

Local vs global state Local state works best for exclusive local concerns (form inputs, open/closed toggles). However, when several sections of the program need to read/write the same data, storing it in a single location provides one source of truth and unidirectional data flow. Redux defines this flow: Action → Reducer → Store → UI.

changes over time Early React apps used local setState and, where necessary, the Context API to transfer values between component trees. These tools are useful for simple, stable purposes, but when applications expand, Context alone can result in several re-renders. To handle growing complexity, the community embraced unidirectional data-flow designs that centralize information, describe changes as plain events, and compute updates using pure functions, boosting predictability and tooling. Among these approaches, within this ecosystem, Redux became widely adopted thanks to its single-store architecture and clear flow of actions and reducers. Over time, it has come to be regarded as a standard reference for managing predictable state changes in complex applications. Over time, the maintainers formalized established patterns into Redux Toolkit (RTK), which is now the recommended approach for building Redux logic.[20]

2.5.2 Redux

Redux is an open-source library used for managing application state predictably (global state). It is based on the reducer pattern, and it provides a central store to manage and access application state across different components. Redux follows a unidirectional data flow model. In other words, it serves as a centralized store for the state that needs to be used across your entire application, with rules ensuring that the state can only be updated in a predictable fashion. In general, Redux makes it easier to understand when, where, why, and how the state in the application

is being updated, and how the application logic will behave when those changes occur. Redux helps to write code that is predictable and testable, consequently resulting in an application that will work as expected. Like any tool, Redux has pros and cons. Some downsides are: more concepts need to be learned, more code to be written, and it also adds some indirection to the code, but it's a trade-off between short-term and long-term productivity.[21] The following sections outline Redux's core elements—stores, actions, and reducers—followed by an explanation of Redux Toolkit and how it simplifies their usage.

Store The store is a single source of truth and is the only object through which the state is read or updated. Its public API includes getState() (reading the current state), dispatch(action) (submitting an event to be processed), and subscribe(listener) (registering change listeners). State updates only occur when an action is sent and the root reducer computes the next state. (In Redux Toolkit, the store is created with configureStore, which wraps the low-level createStore and applies recommended defaults.)

Actions An event in the program is described by an action, which is a serializable plain object that typically has a string type and a payload. Actions are the only input to the store's update pipeline, and they are designed to be declarative representations of "what happened," allowing for predictable updates. (In the Redux Toolkit, action creators are often generated via createSlice, which assigns stable type strings.)

Reducers Reducers are pure functions that follow a pattern: (previousState, action) → nextState. They cannot trigger side effects or change the current state; instead, they must return a new state value based on the previous state and the action. Applications frequently combine many feature reducers into a root reducer used by the store, preserving Redux's unidirectional data flow.

Redux Toolkit

Redux Toolkit (RTK) is the official recommended approach for writing Redux logic. The @reduxjs/toolkit package wraps around the core Redux package, and contains API methods and common dependencies that are essential for building a Redux app. It follows best practices, simplifies most Redux tasks, and prevents common mistakes.[22] To be specific, RTK preserves Redux's core model (single store, actions, pure reducers, unidirectional flow) but replaces the complex, error-prone setup of "plain" Redux. Main APIs provided by Redux Toolkit:[20]

- **configureStore()** – Simplifies store setup with default middleware and DevTools support.

- **createSlice()** – Defines reducers and auto-generates corresponding action creators.
- **createReducer()** – Creates reducers using a case-mapping style, with *immer* for safe mutable updates.
- **createAction()** – Generates action creator functions from type strings.
- **createAsyncThunk()** – Handles async workflows by dispatching pending/-fulfilled/rejected actions.
- **combineSlices()** – Combines multiple slices into a single reducer (supports lazy loading).
- **createEntityAdapter()** – Provides helpers for managing normalized collections of items.
- **createSelector()** – Enables efficient memoized selectors for derived state.

2.5.3 RTK Query

RTK Query is a powerful data fetching and caching tool. It is designed to simplify common cases for loading data in a web application, eliminating the need to hand-write data fetching and caching logic yourself. It is an optional addon included in the Redux Toolkit package, and its functionality is built on top of the other APIs in Redux Toolkit.[23] The main features of RTK Query include: Over the last couple of years the React community has come to realize that "data fetching and caching" is a set of different concerns than "state management", and that trying to solve both problems with a single tool often leads to complexity and confusion. RTK Query focuses specifically on the data fetching and caching problem, providing a specialized solution that works well alongside Redux for state management. As an example in today's applications, tracking loading state, avoiding duplicate requests, supporting optimistic updates, and managing cache lifetimes are behaviors that need to be implemented. RTK Query provides a solution for these problems. To solve these problems, RTK Query took inspiration from other tools that have pioneered solutions for data fetching, like Apollo Client, React Query, Urql, and SWR, but adds a unique approach to its API design. Two concepts that are used in RTK Query are queries and mutations, which will be explained in the following:

Queries and mutations Queries are operations that fetch data from the server and cache it within the client. This is the most common use case for RTK Query.[24] RTK Query also caches the loading/error state of the result in the client. Since the most common type of query is an HTTP request, RTK Query

ships with fetchBaseQuery(...), which is a lightweight fetch(...) wrapper that automatically handles request headers and response parsing in a manner similar to common libraries like axios. Whenever a query hook is invoked, a check is made to determine whether data is already available, and the result is a cache hit or miss accordingly. If the data is already cached and valid, it is returned immediately without making a network request. If the data is not cached or has expired, the current state will be set to isLoading or IsFetching, and a network request is initiated to fetch the data from the server. Mutations are used to send data updates to the server and apply the changes to the local cache. Mutations can also invalidate cached data and force re-fetches.[25] The mutation lifecycle is slightly different from queries and involves clear steps that help to manage UI states effectively. It automatically generates a hook for each mutation. The mutation is triggered when the trigger function is invoked, and when invoked, the hook returns a tuple: the first element is the trigger function, the second contains mutation states and data. Upon triggering the mutation, the isLoading state attribute immediately becomes true, and a network request is asynchronously started. The asynchronous invocation may lead to two alternative outcomes, success or failure, based on which the state attributes are updated. The point to note is that, in order to impact cached data, when the mutation is defined, it is necessary to define a set of tags to be invalidated.

Caching behaviour When data is fetched from the server, RTK Query will store the data in the Redux store as a cache. When a request is attempted, if the data already exists in the cache, then that data is served, and no new request is sent to the server. Otherwise, if the data does not exist in the cache, then a new request is sent, and the returned response is stored in the cache. When a component first invokes a query, it subscribes to the corresponding endpoint, and when the last component subscribing to a query unmounts, the corresponding data will be removed from the cache after a given timeout.

2.6 Styling approaches

A design system defines the guidelines and building blocks that ensure an interface is consistent and scalable. It defines design tokens (colors, spacing, typography), theming (including light and dark variations), responsive breakpoints and layout constraints, and basic accessibility behaviors, which are then packaged into reusable components. Adopting a system saves duplication, unifies designers and developers around a single vocabulary, and accelerates delivery by organizing interaction patterns. In the below part, the 3 important styling approaches used in modern web development, including core CSS conventions, the utility-first framework

TailwindCSS, and the component-oriented library Chakra UI, are discussed.

2.6.1 CSS

Cascading Style Sheets (CSS) is a stylesheet language used to describe the presentation of a document written in HTML or XML (including XML dialects such as SVG, MathML, or XHTML). Each browser has its own default values for each element. These can be overridden by providing immediate values inside each element or by adding a set of rules that replace and integrate the browser default. CSS describes how elements should be rendered on screen, on paper, in speech, or on other media. CSS is among the core languages of the open web and is standardized across Web browsers [26] CSS resolves styles through the cascade, specificity, and inheritance, which together determine the final computed values for each element. Modern CSS adds finer control with cascade layers, nesting, and custom properties (variables) so teams can organize rules predictably, theme components, and expose design tokens without duplicating values.[27] Browsers apply a built-in user-agent stylesheet (default styles) to every element; author styles then override and extend these defaults via the cascade. A CSS rule has the form selector property: value; ... , where the selector identifies the elements to affect and the declarations are property–value pairs separated by semicolons. Elements can be targeted in many ways—by type, class, ID, attributes, pseudo-classes/pseudo-elements, and with combinators for structural relationships—and inline declarations or more specific selectors take precedence when rules conflict.

2.6.2 Tailwind CSS

Tailwind CSS is a utility-first CSS framework: instead of shipping pre-styled components, it provides low-level utility classes. This approach favors consistency and speed by encouraging reuse of a shared design vocabulary rather than writing ad-hoc selectors and component styles from scratch. Tailwind scans templates (HTML/JS/TS/JSX/TSX, etc.) to find class names, then generates only the styles that are actually used, and outputs a static CSS file. The framework is configured via a central "tailwind.config" file where the theme (colors, spacing, typography), breakpoints, plugins, and presets are defined.[28]

2.6.3 Chakra UI

Chakra UI is a simple, modular, and accessible component library for building React applications. It comes with pre-built, fully responsive components and a mobile-first styling approach, allowing layouts to adjust cleanly across devices with minimal coding. A unified theme system allows teams to set colors, fonts,

spacing, and breakpoints once and apply them consistently, and prop-based styling makes components modular and easy to adapt. Overall, Chakra UI allows you to quickly create clean, consistent UIs without being locked into a rigid design system.[29] It also includes first-class light/dark color-mode support via a provider and hooks, making theming and user preference persistence simple. Chakra styles its React components with Emotion (CSS-in-JS), and several interactive, stateful widgets are developed with Ark UI (a headless, accessibility-first layer based on Zag.js state machines for consistent behavior across frameworks). These choices, along with typed properties and clearly stated documentation, emphasize developer experience and simple interaction patterns, allowing for quick delivery of consistent UIs without locking teams into a rigid design system.

2.6.4 Chakra UI vs. Tailwind

Chakra UI and TailwindCSS offer two degrees of abstraction. Chakra is component-first, allowing you to build UIs from accessible React components and style them with style props backed by theme tokens and color modes, which speeds up consistent app development. Tailwind prioritizes utility: it generates low-level classes from a central theme configuration, allowing maximum visual control while leaving components and accessibility patterns to the developer. In short, Chakra optimizes speed-to-value for product UIs; however, Tailwind optimizes design freedom for teams that are comfortable composing from basic elements.

2.7 Backend for Frontend pattern

Backend-For-Frontend (BFF) is a layer that handles only the requirements that are specific to the interface, or in other words, it's a service that sits between the frontend client and the backend services. This pattern customizes the client experience for a specific interface without affecting other interfaces. It also optimizes performance to meet the needs of the frontend environment. Because each BFF service is smaller and less complex than a shared backend service, it can make the application easier to manage. Frontend teams independently manage their own BFF service, which gives them control over language selection, release cadence, workload prioritization, and feature integration. This autonomy enables them to operate efficiently without depending on a centralized backend development team.[30]

2.8 Workflow Management

Workflow management is the coordinated design, execution, and monitoring of business processes, defining how tasks, information, and responsibilities move between people and systems to achieve organizational objectives. In industry, structured workflows improve consistency, traceability, and operational efficiency while reducing manual errors and delays [31]. To support standardization and automation, the following sections introduce Business Process Model and Notation and the Camunda platform, which are tools for modeling and executing workflows [32].

2.8.1 BPMN

Business Process Model and Notation (BPMN) is an open standard that provides a graphical notation for modeling business processes in a standardized and comprehensible way. It enables both technical and non-technical stakeholders to visualize process flows using elements such as events, activities, gateways, and message flows. By offering a common language between business analysts and developers, BPMN improves communication, consistency, and automation across organizational processes [33].

2.8.2 Camunda

Camunda is an open-source process orchestration platform designed to automate and manage complex business workflows based on open standards such as BPMN and DMN(Decision Model and Notation). It enables organizations to model, execute, and monitor business processes by connecting human tasks, microservices, and external systems within a unified orchestration layer. The platform provides both a scalable workflow engine and a suite of supporting tools for process modeling, monitoring, and optimization. Through its modular architecture—comprising components such as Zeebe (workflow engine), Tasklist, Operate, and Modeler—Camunda offers flexibility for both cloud and on-premise deployments. It is widely adopted across industries to improve transparency, efficiency, and compliance in process-driven applications, ensuring that business logic is executed consistently and traceably across distributed systems [34, 35]. The most important components of Camunda are explained in the following:

TaskList

Tasklist is a ready-to-use application to rapidly implement business processes alongside user tasks. The user interaction with a task may involve making updates,

adding variables, filling out a Camunda Form, or simply reviewing and completing the task. Tasklist has two main pages:[36]

- Task List Page: to view and manage tasks.
- Processes Page: to start process instances.

Operate

Operate is a tool for monitoring and troubleshooting process instances running in Zeebe (the process/workflow engine, responsible for executing BPMN processes). In other words, Operate provides visibility into active and completed instances, variables, and incidents to support investigation and recovery.[37]

Modeler

Modeler is a tool to design and implement Business Process Model and Notation (BPMN) diagrams. It has 2 different versions: desktop and web-based. Web Modeler is suitable for collaborative modeling and, Desktop Modeler for local work and XML editing. It is also possible to implement details such as conditions within a gateway or service task implementation.[38]

2.9 Authentication and Keycloak

Authentication and authorization are related but distinct processes in an organization's identity and access management (IAM) system. Authentication verifies a user's identity. Authorization gives the user the right level of access to system resources. The authentication process relies on credentials, such as passwords or fingerprint scans, that users present to prove they are who they claim to be. Authentication is usually a prerequisite for authorization. A system must know who a user is before it can grant that user access to anything.[39] Identity and Access Management (IAM) governs the lifecycle of digital identities and the policies/technologies that ensure the right entities have the right access to the right resources at the right time.[40] In the following, Keycloak, an open-source IAM solution, is explained.

2.9.1 Keycloak

Authentication is the process of verifying who a user is, while authorization is the process of verifying what they have access to.[41] Keycloak is an open source Identity and Access Management (IAM) solution. It is based on standard protocols and provides support for OpenID Connect, OAuth 2.0, and SAML. Users authenticate

with Keycloak rather than individual applications; consequently, the applications itself doesn't have to deal with login forms, authenticating users, and storing users' credentials directly. Keycloak provides single sign-on (SSO) and single logout (SLO), whichmeans that users log in once to access different applications and services without needing to authenticate again for each one, and the same thing applies also for logout.[42] From a technical point of view, Keycloak organizes security within realms, which isolate users, credentials, and clients (applications/APIs). Access is granted via ID, access, and refresh tokens issued through standard OIDC/OAuth flows; token lifetimes and global sessions can be centrally managed. Beyond role checks, Keycloak offers authorization services with resources, scopes, permissions, and policy providers (RBAC/ABAC) for fine-grained access control. Integration is done through OIDC/SAML adapters or middleware (public, confidential, or bearer-only clients), while identity brokering and user federation connect external IdPs and directories (e.g., LDAP/AD). Operational features include clustering/HA, configurable password policies, themes for login/account UIs, and an Admin REST API and SPIs for extension.

Admin Console The admin console is the web-based interface for centrally managing all aspects of the Keycloak server. From this page, administrators can configure identity brokering and user federation, create and manage applications and services, and define fine-grained authorization policies.[42]

Account Management Console The account management console is another interface provided by Keycloak that allows users to manage their own account settings and gives them options such as updating profile information, changing passwords, and setting up two-factor authentication.[42]

2.10 Containerization and Deployment

Modern software deployment has evolved significantly with the adoption of containerization and orchestration technologies. Containerization enables applications to be packaged together with their dependencies and configurations inside lightweight, isolated environments that can run consistently across different platforms. This approach improves portability and reproducibility, making it possible to deploy the same application in development, testing, and production without environmental discrepancies [43]. It also simplifies distribution and supports the adoption of cloud-native and DevOps practices. On top of containerization, orchestration systems automate the deployment, scaling, and lifecycle management of these workloads, allowing applications to operate efficiently in dynamic or distributed infrastructures [44].

2.10.1 Docker

Docker is one of the most widely adopted platforms for containerization. It packages an application and its dependencies into a standardized image that can be executed consistently across environments. By abstracting the application from the host operating system, Docker reduces compatibility issues and facilitates deployment in heterogeneous infrastructures [45]. Compared to traditional virtual machines, containers are lighter, start faster, and consume fewer resources, providing an efficient model for scalable and portable software delivery [46]. When containerized applications grow in number or complexity, their coordination and lifecycle management require dedicated orchestration systems capable of automating deployment and scaling tasks.

2.10.2 Kubernetes

Kubernetes is an open-source orchestration platform for managing containerized applications at scale. It automates tasks such as deployment, scheduling, scaling, and recovery across distributed clusters of machines. Through a declarative configuration model, Kubernetes ensures that the system's actual state continuously aligns with the desired one defined by operators [44]. Kubernetes provides load balancing, self-healing, horizontal scaling, and multi-node coordination, forming the backbone of most modern cloud-native infrastructures [47]. Its configuration and operation, however, involve a considerable degree of complexity, which has motivated the creation of complementary tools such as **Helm**, used to simplify application management through reusable deployment templates.

Chapter 3

System Architecture

In this chapter, the overall architecture of the application is explained to give an overview of the different layers and components that make up the system.

3.1 General System Overview

The implemented system follows a modular and containerized architecture, designed to ensure scalability, maintainability, and clear separation of concerns. All services are deployed on a **Kubernetes** cluster, which manages their lifecycle, networking, and resource allocation. The main components of the architecture are: the **Progressive Web Application (PWA)** used by end-users, the **API Gateway**, the **Backend-for-Frontend (BFF)** service, the **Camunda 8** process engine and its workers, **Keycloak** for identity management, and **Firebase Cloud Messaging (FCM)** for asynchronous notifications. Each part of the system is connected as shown in Figure 3.1.

When a user accesses the application from a browser, the PWA provides the interface for interaction with the underlying business processes. The PWA communicates with the backend infrastructure exclusively through the **API Gateway**, implemented with **APISIX**, which acts as a single controlled entry point to the cluster, and is responsible for routing, load balancing, and enforcing authentication and authorization policies before forwarding requests to internal services. This modular structure allows the application to evolve over time, as each service can be updated or replaced independently without disrupting the entire system.

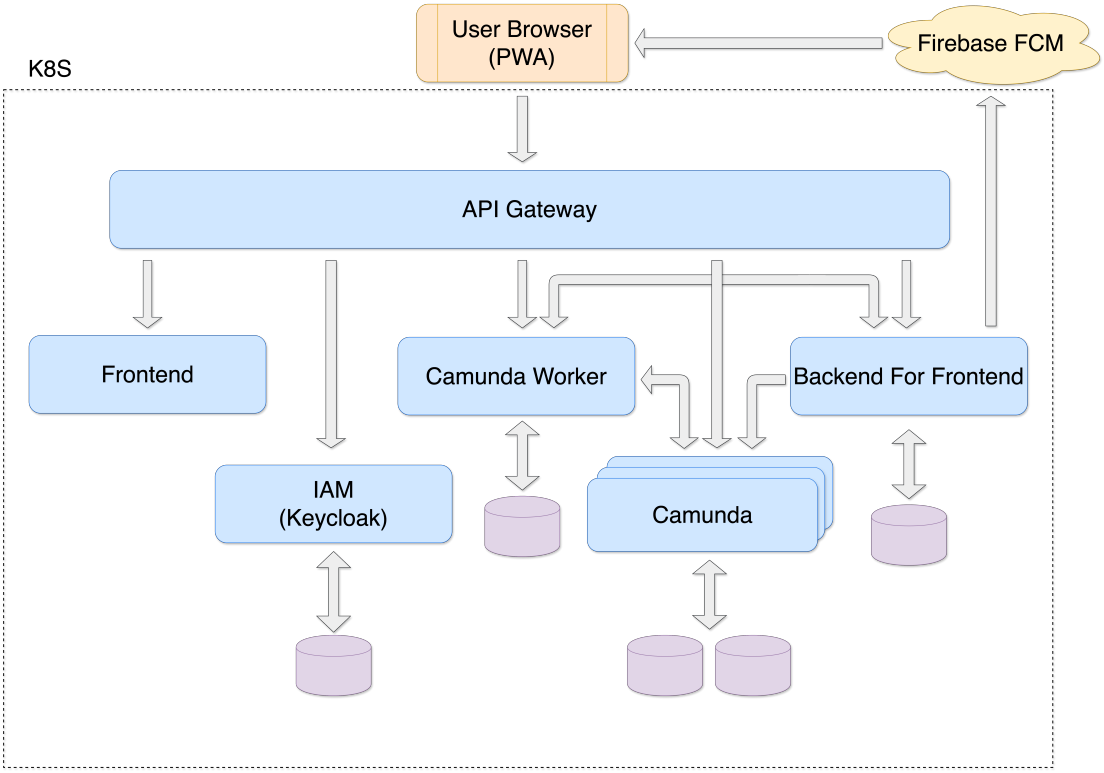

Figure 3.1: System Architecture

3.2 Frontend

The **frontend** of the system is implemented as a **Progressive Web Application (PWA)** developed in React and TypeScript. It provides a single code base that can be installed and used on both desktop and mobile devices, ensuring consistent behaviour across different platforms. The PWA handles all user interactions, renders dynamic forms, and communicates with the backend through the API Gateway using secure HTTPS connections.

Each interface component retrieves its data from the BFF through RESTful APIs. The frontend does not include business logic. It mainly serves as a presentation layer, showing workflow information, tasks, and notifications. All data on the interface, including process lists, forms, and summaries, is retrieved in real time from the backend. This approach ensures that the application always reflects the most recent process state maintained by Camunda.

From a structural perspective, the frontend is composed of modular components organized by feature domains. Routing is managed with React Router, and state management is handled through Redux Toolkit, which allows data such as the

logged-in user, roles, and notification settings to be shared across the interface. These will be explained in detail in Chapter 4.
Access to the application is controlled through authentication tokens issued by Keycloak. Depending on the assigned role, the interface dynamically adapts menus and permissions, ensuring that each user interacts only with the features relevant to their responsibilities. In practice, this allows different categories of users to access the same platform while maintaining strict separation of permissions.

3.3 API Gateway

At the center of the communication layer lies the **API Gateway**, implemented using **Apache APISIX**. It acts as the single controlled entry point to the Kubernetes cluster and is responsible for managing, securing, and optimizing all network traffic between external clients and internal services. Every request sent by the Progressive Web Application passes through the gateway before reaching the internal endpoints, ensuring that access control, routing, and logging are applied in a consistent and centralized manner.

The API Gateway serves multiple key purposes within the system architecture. First, it performs request authentication and authorization by integrating directly with **Keycloak**. Incoming requests are intercepted, and the embedded access tokens are verified against the Keycloak OpenID Connect configuration. Only validated requests are forwarded to the internal services, preventing unauthorized access and ensuring that all communications originate from authenticated users. In this way, the gateway acts as the first security boundary of the system. The details of the authentication flow are going to be further explained in the next sections.

Second, the gateway handles dynamic routing and service discovery. Each internal service, such as the BFF, Camunda, or notification handler, runs in its own Kubernetes pod. Its address can change as the cluster scales or updates. APISIX automatically resolves service locations and directs requests to the correct endpoint based on predefined routing rules. This layer hides the complexity of the underlying infrastructure from the frontend, enabling the PWA to communicate through one consistent URL no matter how the deployment is configured.

Another critical function of the API Gateway is to provide traffic management and observability. It supports fine-grained control over request rates and concurrency, enabling load balancing, rate limiting, and timeout policies that protect the backend from overload. In production, this becomes particularly important when multiple users operate concurrently or when large data payloads are exchanged during workflow operations. APISIX also generates detailed access logs and performance metrics that can be forwarded to monitoring tools for auditing and system diagnostics.

From a security point of view, the gateway is critical in ensuring isolation between the external network and internal services. The internal APIs of Camunda, Keycloak, and the BFF are not exposed to the internet. Instead, APISIX manages all incoming traffic through a controlled ingress configuration, which defines allowed routes and enforces the use of HTTPS for encrypted communication. This setup not only simplifies external access but also prevents unauthorized requests from reaching sensitive services within the cluster.

Additionally, the API Gateway simplifies integration between different modules by supporting a flexible plugin architecture. For instance, in this project, an *OpenID Connect plugin* was configured to handle authentication flow transparently, reducing the amount of security logic that each individual service must implement. Other plugins handle URL rewriting, redirection rules, and response normalization, ensuring consistent behavior across all endpoints.

The gateway also contributes to system scalability. By separating client communication from internal service topology, new microservices can be introduced or changed without affecting the client-side logic. In case of high traffic, the gateway can distribute requests across multiple instances of the same service, improving performance and fault tolerance.

In summary, the API Gateway represents the entry layer of the platform, providing secure, unified, and monitored access to all backend services. It enforces authentication through Keycloak, manages routing and load balancing, and contributes to both the reliability and observability of the system. Together with the BFF, it defines a clean and modular communication boundary between the external Progressive Web Application and the internal orchestration and process management services.

3.4 Authentication and Authorization

Keycloak manages all user authentication and authorization flows. When a user logs in, Keycloak issues a JWT token containing identity and role information. This token is included in all subsequent requests and verified by APISIX before forwarding them to other services. Each role—such as technician, veterinarian, manager, or planner—determines the accessible pages and available actions within the PWA. Keycloak itself is connected to its dedicated database for user, role, and session management, and exposes standard OpenID Connect and OAuth 2.0 endpoints used by the BFF and frontend during login or token refresh operations.

It is important to note that authentication data is never handled directly by the frontend; all validations are performed through the gateway and the BFF, ensuring a centralized and secure authorization model.

3.5 Backend-for-Frontend (BFF)

The **Backend-for-Frontend (BFF)** layer represents the communication point between the user interface and the backend services. Its main job is to give the frontend a simple, custom API that hides the complexity of the systems behind it. This lets the client app use one stable endpoint, while the BFF manages requests to different services like the Camunda process engine, Keycloak, and the notification system. This approach clearly separates the presentation layer from business logic. The frontend focuses on displaying data and managing the user interface, while the BFF takes care of tasks like combining requests, transforming data, and handling errors. In practice, the BFF offers REST endpoints tailored to the Progressive Web Application, reducing client-side processing and keeping data structures consistent throughout the system. All communication from the PWA goes through the API Gateway, which checks the user's authentication token and sends the request to the BFF. The BFF does more validation and decides what operation to perform. For example, if a user starts a new process, completes a task, or loads a form, the gateway securely sends the request to the Camunda engine using an internal API. The BFF then adjusts the response, filtering or restructuring the data based on the user's role and permissions before sending it back to the client. Besides interacting with Camunda, the BFF is also responsible for integrating with other platform services. It communicates with **Firebase Cloud Messaging** (FCM) to send push notifications to users. From a development perspective, the BFF also simplifies integration between frontend and backend teams. Changes in the internal APIs or process structure can mostly be handled at this layer without modifications on the client side. This makes it easier and faster to evolve the platform, as new workflows or features can be introduced by adjusting the BFF logic, leaving the user interface unchanged. In summary, the BFF serves as a bridge between the presentation layer and the business logic of the system. It ensures that data exchanged between the PWA and the backend services remains secure, consistent, and adapted to the user's context. By centralizing control over data flow and access rules, the BFF contributes to a modular and maintainable architecture, where each layer can evolve independently while preserving system integrity.

3.6 Process Management with Camunda

The orchestration of business processes is managed by **Camunda 8**, which executes the BPMN models designed during the analysis phase. Camunda receives process-related requests—such as starting a new workflow instance or completing a task—through the BFF. Each process instance is persisted in the **Camunda Database**, while execution metadata is indexed into **Elasticsearch**, which is used

internally by Camunda to provide fast retrieval and monitoring of process data.

In the architecture diagram, Camunda is represented as a composite block rather than a single module. This is because the platform is made up of multiple coordinated services, including the workflow engine and monitoring components, which work together to execute, store, and visualize process data. Although these services are internally connected, they function as one logical unit within the system.

It is important to note that, in the deployed solution, users never interact directly with Camunda. All communications occur through the BFF, which handles authentication, request validation, and data adaptation. Access to Camunda's native interfaces is limited to development and configuration tasks in order to support workflow testing and process validation. This separation ensures that the operational system remains secure and consistent, while preserving the flexibility of the underlying process engine.

3.7 Notifications and Firebase Cloud Messaging

Asynchronous communication is managed through **Firebase Cloud Messaging (FCM)**, which enables the delivery of notifications directly to the user's browser or device. This mechanism ensures that users are informed about important workflow events—such as the creation of a new task or a change in process status—without requiring manual refresh or polling.

The notification flow begins when a process event occurs in Camunda. The BFF, acting as the mediator, receives this event and sends a message request to Firebase through the **Firebase Admin SDK**. Firebase then handles message queuing, token management, and device targeting, delivering the notification to all registered clients associated with that specific task (or the group, such as all veterinarians).

On the client side, notifications are received by the service worker integrated within the PWA. This component runs in the background and displays push notifications even when the application is not open. When the user interacts with a notification, the PWA is automatically opened and redirected to the corresponding task or page, allowing immediate action.

It is important to note that the all the infrastructure behind FCM, including message storage and distribution servers, is managed entirely by Google. The application itself does not handle a dedicated database for push delivery or message queues. It only saves the basic subscription information it needs, such as user tokens, so that notifications can be linked to the correct accounts. This design choice reduces the amount of maintenance required, while providing reliable cross-platform support for real-time alerts.

3.8 Data Persistence and Databases

Each major service within the system architecture maintains its own dedicated database, ensuring clear separation of responsibilities and simplifying scalability and maintenance.

- **Keycloak Database:** stores user credentials, roles, and session information. It is deployed as a persistent volume within the cluster and managed by the Keycloak service itself.
- **Camunda Database:** holds process instance data, variables, and audit logs required for workflow execution. Camunda also uses Elasticsearch internally for indexing and querying process metadata, which supports its monitoring components.
- **BFF Database:** The system keeps track of each user and their corresponding Firebase registration tokens, along with a record of the notifications that have been sent or received. It also stores small pieces of metadata related to workflow activity, such as message timestamps. When a new notification is generated, the BFF saves the full version locally, while a lighter version is sent through Firebase Cloud Messaging. If the client needs the complete message, it can request it directly from the BFF. In this way, the database works as an operational log that complements Camunda's own storage, keeping the two systems aligned without duplicating business data.
- **Firebase Storage and Messaging Backend:** The notification subsystem relies entirely on the managed infrastructure provided by Google. All data related to message delivery and queue management is handled externally by the FCM service and is not hosted within the company's cluster.

This distributed persistence model prevents a single point of failure and ensures that each subsystem can evolve independently. Data flows between these databases only through controlled API interactions defined by the BFF, guaranteeing both consistency and data protection. Backups and access control are managed at the individual service level according to their operational requirements.

3.9 Deployment and Infrastructure

All of the components represented in the diagram are containerized and deployed in a single Kubernetes namespace. Each microservice (Frontend, BFF, Camunda, Keycloak) runs in its own pod, defined by independent deployment and service configurations. Ingress rules in APISIX manage secure HTTPS connections and

internal routing, while Kubernetes handles scaling, recovery, and configuration through declarative manifests.

This containerized design allows for more flexibility in maintenance while also simplifying versioning and horizontal scaling between environments. Isolating each component allows for independent updates and keeps defects within their respective service boundaries.

3.10 Summary

In summary, the system architecture is organized around a clear separation of responsibilities: the **frontend** for user interaction, the **BFF** for data mediation, **Camunda** for workflow orchestration, **Keycloak** for authentication, and **Firebase** for asynchronous messaging. The use of **APISIX** as a unified gateway ensures that communication remains secure and consistent, while Kubernetes provides a robust foundation for deployment and scalability. This architecture effectively combines modular design, process automation, and platform independence, aligning with the goals of digitalization and operational efficiency pursued by the company.

Chapter 4

Implementation

4.1 Client

The client side constitutes the front-end of the system, implemented with React, TypeScript, and Chakra UI, and designed to provide a responsive interface for all user roles. This section outlines the main elements of the client application, including its routing structure, state management with Redux Toolkit, theming configuration, and the dynamic form system based on Camunda JSON schemas.

4.1.1 Design

The design phase focused on creating an interface that is both visually consistent and user-friendly. Since the system is used by various employees, from technical staff working directly in the farms to managers, the design focused on clarity, accessibility, and efficient information display. The interface needed to adapt to different use contexts, ensuring that essential operations could be performed both from desktop computers in the office and from mobile devices in the field.

From the very beginning, the user experience (**UX**) design followed a goal-oriented approach; every screen and interactive element was planned to serve a specific operational need. For instance, the *planning cards* were designed to provide a concise overview of scheduled activities, highlighting deadlines and status indicators, while the *cycle cards* presented detailed production data and progress indicators. Each page was organized so that the most important information stands out. Users can quickly view details, update records, or complete tasks without unnecessary navigation.

The user interface (**UI**) design leveraged the component-based structure provided by React and Chakra UI. This made it possible to define consistent spacing, colors, and typography throughout the application, reinforcing visual uniformity and reducing cognitive load. Attention was given to contrast levels and component

visibility to ensure readability under different lighting conditions, particularly considering the use of mobile devices outdoors. Interactive components such as tables, buttons, and filters were designed with responsive layouts, so the interface remains clear and functional on smaller screens.

The design process was both iterative and collaborative throughout the project. After each design iteration or prototype, review sessions were held with company representatives to gather feedback on usability, clarity, and completeness of the displayed information. This feedback loop helped make sure the interface matched users' needs and fit real operational workflows. In several cases, design adjustments were made following direct input from field operators, who provided valuable insight into what information was most useful during their daily activities.

The final interface design, therefore, represents a balance between technical accuracy and user-friendliness. It integrates the structured logic of the BPMN-driven workflow with an interface that supports the user's tasks in an intuitive and efficient way. Examples of the implemented design, including both desktop and mobile views, are presented in the final section of this chapter.

4.1.2 Routing and Application Structure

High-Level Folder Structure The Frontend follows a modular folder organization to separate concerns and simplify maintainability. The `public/` directory contains static assets, including the application manifest and application logos. All source code is stored inside the `src/` directory, which is divided into sub-folders:

- `api/` – helper modules used to communicate with the Backend-for-Frontend (BFF).
- `components/` – reusable UI components shared across multiple pages (form fields, buttons, ...).
- `pages/` – the main page-level React components, each corresponding to a route in the router.
- `store/` – Redux Toolkit slices, RTK Query endpoints, and global application state.
- `utils/` – utility modules such as Firebase messaging integration and helper functions, component responsible for rendering dynamic Camunda JSON forms.
- Root files such as `App.tsx`, `Router.tsx`, and `main.tsx`, that configure the application and define the routing.

This structure keeps the project maintainable as it scales, isolates concerns, and makes the system easier to extend when new workflows or features are introduced.

Layout Structure Most routes are rendered inside a shared `Layout` component, which defines the global structure of the user interface. The layout includes the navigation bar, sidebar menu, and the notification area, ensuring that these elements remain consistent across all pages. Only the central content area changes when navigating between routes. This approach prevents code duplication and ensures a coherent user experience, as all operational sections, including cycles, tasks, summaries, and notification settings, reuse the same application framework.

The layout also integrates global behaviors such as color-mode styling, responsive design rules from Chakra UI, and user-session handling inherited from the top-level `App.tsx`. By delegating shared UI elements to a single component, the routing layer remains clean and focused solely on navigation logic.

Router Navigation in the application is managed using `React Router`, which defines all navigable paths, layout hierarchies, and access rules. The main configuration is centralized in the `Router.tsx` file, where routes are declared using the `createBrowserRouter` API. The router defines both public pages (such as the home or error screens) and private sections, accessible only to authorized users, depending on their assigned roles.

The base route renders a common layout that includes shared components such as the navigation bar, sidebar, and notification area. Each child route corresponds to one of the main pages of the application, such as task management, production cycles, or cycle summary.

```
const AppRouter = () =>
  createBrowserRouter([
    {
      path: '/',
      element: <Layout/>,
      children: [
        { path: '', index: true, element: <Home/> },
        { path: 'error', element: <ErrorScreen/> },
        { path: 'notifications', children: [
            { path: '', index: true, element: <Notifications/> },
            { path: ':notificationId', element: <NotificationPage/> },
        ]},
        { path: 'notification-settings', element: <NotificationSettings/> },
        { path: 'cycles', children: [
            { path: '', index: true, element: <Cycles/> },
            { path: ':cycleId', element: <CycleDetails/> },
            { path: ':cycleId/summary', element: <SummaryPage/>, children: [
                { index: true, element: <GeneralSummary/> },
                { path: 'feed', element: <FeedSummary/> },
                { path: 'in-out', element: <InOutSummary/> },
                { path: 'medicine', element: <MedicineSummary/> },
            ]}
```

```
        ]},
        { path: 'forms', children: [
            { path: '', index: true, element: <TasksPage/> },
            { path: ':taskId', element: <CamundaFormRenderer/> },
        ]},
      ]
    }
  ]);
```

This structure reflects the logical organization of the system. For example, the `cycles` section groups routes related to production cycles, each with a dedicated detail page and multiple subpages for summaries of feed, medicine, or in/out operations. The `forms` route is connected to the workflow engine and renders tasks dynamically using the `CamundaFormRenderer` described in the previous section.

Access control is enforced through the `RequireRole` component, which checks the user's assigned roles (retrieved from Redux) and prevents unauthorized access to restricted pages. If the current user does not have the required role, an error message is displayed instead of the protected content.

```
export const RequireRole = ({ allowedRoles }: RequireRoleProps) => {
  const roles = useSelector((state: RootState) => state.user.roles);
  const isAuthorized = roles.some(role => allowedRoles.includes(role));

  if (!isAuthorized)
    return <ErrorScreen error="Non sei autorizzato a visualizzare questa
    ↪  pagina"/>;

  return <Outlet/>;
};
```

This approach makes the routing modular and secure, while keeping navigation consistent throughout the app. Nested routes simplify component composition, allowing each section—such as notifications, cycles, or forms—to maintain its own layout and logic inside the global structure.

4.1.3 State Management with Redux and RTK Query

State management in the application is organized through Redux Toolkit, which centralizes all client-side data in a single store. Each slice manages a specific domain of the application, ensuring that updates are predictable and that information can be shared consistently across components. This structure was essential because the application operates on several concurrent data sources—user information, Camunda workflow data, notifications, and configuration parameters—while also requiring continuous synchronization with backend services through RTK Query.

The global store (`store.tsx`) integrates all slices and the RTK Query middleware. Each reducer represents an independent logical section of the state tree. The most relevant slices in the project are:

- **UserSlice** – stores the username and the list of roles associated with the authenticated user. This information is used to control interface visibility and enforce access permissions at the component level.
- **NotificationsSlice** – manages toast and in-app notifications, allowing messages to be displayed or cleared globally.
- **DataSlice** – used as a generic container for temporary or computed values shared between components.
- **SettingsSlice** – holds UI configuration data such as language, theme, or layout preferences.
- **ApiSlice** and **api.reducer** – expose endpoints and cache results handled through RTK Query, providing transparent integration with the backend.

A piece of the store configuration is shown below:

```
const store = configureStore({
  reducer: {
    user: userReducer,
    notifications: notificationsReducer,
    data: dataReducer,
    settings: settingsReducer,
    [api.reducerPath]: api.reducer,
  },
  middleware: (getDefaultMiddleware) =>
    getDefaultMiddleware().concat(api.middleware),
});
```

Each slice defines its own initial state and reducers. For instance, the `UserSlice` maintains a minimal user profile, enabling any component to read or update the username and roles. When the user logs out, the slice resets to its initial state, ensuring that no sensitive data remains in memory.

```
const userSlice = createSlice({
  name: "user",
  initialState: { username: '', roles: [] },
  reducers: {
    setUsername: (state, action) => { state.username = action.payload; },
    setRoles: (state, action) => { state.roles = action.payload || []; },
    logout: (state) => { state.username = ''; state.roles = []; }
  }
});
```

A more complex example is the `PlanningSlice`, which coordinates data used in the planning process. The state of this slice typically includes the list of all planned batches, the currently selected plan, and any filters applied by the planner. This structure was chosen because planning data is central to the system and must remain accessible across several views—such as plan details, summary dashboards, and validation screens. By keeping it in a dedicated slice, these views can update instantly when the planner modifies a schedule or when new data is fetched from the API.

Data loading and synchronization are managed through RTK Query. For example, the `PlanningApi` defines endpoints that retrieve the list of plans or the details of a specific plan. The hooks generated by RTK Query are then consumed directly inside React components.

```
export const planningApi = api.injectEndpoints({
  endpoints: (build) => ({
    getPlannings: build.query({
      query: () => ({ url: '/operator/plannings', method: 'GET' }),
      providesTags: ['Plannings']
    }),
    getPlanningById: build.query({
      query: (id) => ({ url: `/operator/plannings/${id}`, method: 'GET' }),
      providesTags: (_, __, id) => [{ type: 'Plannings', id }],
    }),
  }),
});
```

When a component calls the generated hook, such as `useGetPlanningsQuery()`, RTK Query automatically performs the request, caches the result, and keeps the data synchronized. This mechanism eliminates the need for manual request handling and avoids redundant calls, which is particularly useful in modules like planning and production summaries where several components depend on the same data.

In conclusion, Redux Toolkit and RTK Query together provide a clean separation between application domains and asynchronous data. Local slices manage authentication, settings, and UI state, while RTK Query handles communication with backend services. This organization ensures consistency, scalability, and efficient reuse of data across the entire application.

Note: To use Redux for development, it's necessary to have the React and Redux DevTools Extensions in browsers. [21]

4.1.4 Styling and Theming with Chakra UI

Light and Dark Theme Chakra UI provides built-in support for color mode management, allowing components to automatically adjust their appearance between light and dark themes. By default, the library includes a color mode context

and a system color mode manager that detects the user's operating system preference and applies the corresponding palette. Standard components—such as buttons, inputs, alerts, and typography—automatically adapt their background, border, and text colors using predefined design tokens. This means that most of the base styling transitions seamlessly between light and dark modes without additional configuration.

Custom Theme During development, it was decided to define specific custom colors for certain interface elements—such as table borders, panel outlines, and button states—to ensure consistent visual contrast across both light and dark themes and to better align with the application's visual identity rather than Chakra's default styling. To avoid repeating these style definitions in individual components, the corresponding values were centralized in the custom theme configuration file (`CustomTheme.tsx`), where they are exposed as semantic tokens and reused throughout the application. Below, a part of this file is shown, illustrating how these custom colors are defined:

```
const customConfig = defineConfig({
  theme: {
    semanticTokens: {
      colors: {
        bg: {
          DEFAULT: { value: { _light: "{colors.gray.100}", _dark:
          ↪  "{colors.gray.800}" } },
          panel:   { value: { _light: "{colors.gray.50}",  _dark:
          ↪  "{colors.gray.900}" } },
        },
        border: {
          table:   { value: { _light: "{colors.gray.300}", _dark:
          ↪  "{colors.gray.600}" } },
          outline: { value: { _light: "{colors.gray.400}", _dark:
          ↪  "{colors.gray.700}" } },
        },
        brand: {
          solid:    { value: "{colors.brand.500}" },
          contrast: { value: "{colors.brand.50}" }
        }
      }
    }
  }
});
```

With this setup, whenever a table, card, or panel component references `border.table` or `border.outline`, Chakra automatically applies the correct color for the current theme mode. This eliminates the need to define conditional style logic or inline

color values in each component and allows visual adjustments to be made globally from a single configuration file.

At runtime, Chakra's `Provider` component applies this configuration across the entire application, ensuring that all nested UI elements share the same design system and respond dynamically to color mode changes.

```
createRoot(document.getElementById('root')!).render(
  <ReduxProvider store={store}>
    <Provider>  {/* Custom Chakra Provider */}
      <App/>
      <Toaster/>
    </Provider>
  </ReduxProvider>
);
```

This approach combines the flexibility of Chakra UI's component system with centralized design control, resulting in a visually coherent interface that automatically adapts between light and dark themes without redundant code.

4.1.5 Form Generation Based on Camunda JSON

At the beginning of the project, each form was implemented as a separate React component. This approach provided flexibility and allowed for customized layouts and styles, but as the number of processes and forms increased, maintenance became increasingly complex. During development, new information and requirements continued to emerge from the workflow provider and the client. As a result, even while later forms were being developed, previously completed ones often required modifications—such as new fields, renamed variables, or adjusted logic. This constant evolution made it difficult to maintain consistency and slowed down overall progress. To address this issue, a more scalable solution was introduced: forms are now generated dynamically from the JSON schema automatically produced by Camunda. When a user opens a task, the frontend retrieves the corresponding form JSON through the API and passes it to a dedicated component called `CamundaFormRenderer.tsx`. This component parses the JSON schema and renders the form by composing reusable field components—such as `TextField`, `NumericField`, `RadioField`, or more complicated component types existing in Camunda, such as dynamiclist, expression, and . . . — based on the type definitions contained in the schema.

```
switch (type) {
  case "radio":
    return (
      <RadioField
```

```
        label={label || ""}
        name={componentPath || ""}
        value={value || ""}
        items={component.values || []}
        readOnly={readonly}
        required={required}
        onChange={onValueChange}
      />
    );

  case "expression":
    return (
      <ExpressionField
        expression={component.expression || ""}
        componentKey={component.key}
        path={path}
      />
    );

  default:
    console.warn(`Unsupported component type: ${type}`);
    return null;
}
```

Each component is created dynamically through a rendering function that maps the JSON attributes (such as field type, label, validation, or data source) to their corresponding React elements. The renderer also handles conditional visibility rules, dynamic data retrieval from APIs, and nested structures such as groups, tables, and lists.

```
<Box id={id} borderWidth={component.showOutline ? "1px" : "0"} p={4}>
  {component.components?.map(it => (
    <CamundaFormElement key={it.id} component={it} path={groupPath} />
  ))}
</Box>
```

The overall structure is built using `react-hook-form`, which provides form state management, validation, and submission logic. When a form is submitted, the data is sent to the backend through the `completeTask` mutation, which finalizes the current Camunda task.

```
const onSubmit = (data: any) => {
  completeTask({
    id: taskId!,
    formName: task.formId,
    formData: data,
  }).unwrap().then(() => {
    toaster.create({ title: "Attività completata con successo", type: "success"
    ↪ });
```

```
    navigate(-1);
  }).catch(err => {
    toaster.create({ title: "Errore durante il completamento", type: "error"
    ↪  });
  });
};
```

Below, there is an example of the JSON structure generated by Camunda, used as the first form of the Purchase phase:

```
{
  "executionPlatform": "Camunda Cloud",
  "executionPlatformVersion": "8.6.0",
  "exporter": {
    "name": "Camunda Web Modeler",
    "version": "70aaa0c"
  },
  "schemaVersion": 18,
  "id": "form_pianificazione_acquisto",
  "components": [
    {
      "text": "# Pianificazione acquisto\nInserire i dati di pianificazione
      ↪  considerando i **tempi di evasione** del fornitore",
      "type": "text",
      "layout": {
        "row": "Row_1vjl5t2",
        "columns": null
      },
      "id": "Field_1dlrbeh"
    },
    {
      "label": "Nuova pianificazione",
      "components": [
        {
          "label": "Settimana",
          "type": "number",
          "layout": {
            "row": "Row_11ttwb1",
            "columns": null
          },
          "id": "Field_1d8zy9t",
          "key": "settimana",
          "validate": {
            "required": true,
            "min": "1",
            "max": "52"
          },
          "serializeToString": false
        },
        {
```

```
      "subtype": "date",
      "dateLabel": "Data",
      "type": "datetime",
      "layout": {
        "row": "Row_11ttwb1",
        "columns": null
      },
      "id": "Field_0ybne0s",
      "key": "data",
      "validate": {
        "required": true
      }
    },
    {
      "label": "Razza",
      "type": "select",
      "layout": {
        "row": "Row_11ttwb1",
        "columns": null
      },
      "id": "Field_0bv42xr",
      "key": "razza",
      "validate": {
        "required": true
      },
      "valuesKey": "razze"
    },
    {
      "label": "Allevamento maturazione",
      "type": "select",
      "layout": {
        "row": "Row_17qb4br",
        "columns": null
      },
      "id": "Field_0wmk01c",
      "key": "allevamento_maturazione",
      "searchable": true,
      "validate": {
        "required": true
      },
      "valuesKey": "allevamenti_maturazione"
    },
    {
      "label": "Unità",
      "type": "number",
      "layout": {
        "row": "Row_17qb4br",
        "columns": null
      },
      "id": "Field_1ascrfp",
```

```
        "key": "unita",
        "validate": {
          "required": true,
          "min": 1000,
          "max": 1000000
        }
      },
      {
        "label": "Destinazioni",
        "components": [
          {
            "label": "Allevamento riproduttori",
            "type": "select",
            "layout": {
              "row": "Row_16fw1u8",
              "columns": null
            },
            "id": "Field_0lqb0fc",
            "key": "allevamento_riproduttori",
            "searchable": true,
            "validate": {
              "required": true
            },
            "valuesKey": "allevamenti_riproduttori"
          }
        ],
        "showOutline": true,
        "isRepeating": true,
        "allowAddRemove": true,
        "defaultRepetitions": 1,
        "type": "dynamiclist",
        "layout": {
          "row": "Row_1xfez2q",
          "columns": null
        },
        "id": "Field_0ebq2m2",
        "path": "destinazioni",
        "disableCollapse": true,
        "nonCollapsedItems": 2
      }
    ],
    "showOutline": true,
    "type": "group",
    "layout": {
      "row": "Row_1g5c6rc",
      "columns": null
    },
    "id": "Field_0kt7s7m",
    "path": "pianificazione"
  }
```

```
  ],
  "type": "default",
  "versionTag": "0.1"
}
```

In summary, this approach brings several advantages:

- Any change in the BPMN model or in the form definition within Camunda is immediately reflected in the frontend, without requiring manual updates to the codebase.
- The time required to implement new processes and forms is significantly reduced, since the renderer automatically adapts to different schemas.
- Maintenance costs are minimized, as the form logic and layout are centralized in a single, reusable component.
- Iterative changes introduced by the workflow provider or client during development can be incorporated instantly, without disrupting the application's overall structure.

4.1.6 Form Management with Validation and Conditional Fields

Validation rules defined in Camunda—such as required, min, max, or type constraints—are automatically mapped to React validation through the convertValidation utility. This avoids duplicating validation logic in the frontend and ensures that the constraints defined in the workflow model are applied exactly also in the UI. Some fields appear only when specific conditions are met. Conditional visibility is handled through the `conditional.hide` expression in the schema, which is evaluated at runtime using the Feelin expression engine. If the expression evaluates to true, the field is omitted from the rendered form. Default values and pre-existing process data are merged to initialize the form state. A recursive helper extracts default values from the schema, which are then combined with the input data provided by the task. This guarantees that the form reflects the current state of the workflow.
The renderer also supports dynamic content. Select fields can load their options from the backend via RTK Query hooks or through schema-defined API requests. In addition, Camunda allows lists of options to be defined through `valuesExpression`, which enables the options of a field to be computed dynamically based on other form inputs or process data. The expression is evaluated at runtime, producing a filtered or transformed collection that adapts automatically as the user interacts with the form. This mechanism makes it possible to implement dependent dropdowns or

context-aware selections without any hard-coded logic in the client.
Finally, upon submission, the form data is validated and then sent to the backend via the workflow API to complete the corresponding Camunda task. Feedback is provided through toast notifications, ensuring users receive immediate confirmation or error messages.

4.2 PWA

As it has been explained in previous sections, the most important part of this project is the PWA. In this section, the Implementation of different essential components that are needed for the PWA is explained.

4.2.1 Manifest

The `manifest.json` file defines how the Progressive Web Application is presented to the user and how it behaves when installed on a device. It provides the metadata required by browsers to recognize the web application as installable and to display it with native-like characteristics. The manifest contributes to the app's identity, defining its name, icons, color scheme, and launch configuration. In the manifest, the following main properties are specified:

- **name** and **short_name** — Define the application's full and abbreviated titles that appear respectively in the installation prompt and on the device's home screen.
- **start_url** — Indicates the entry point of the application when it is launched, ensuring a consistent user experience regardless of where it was last accessed.
- **display** — Set to `standalone`, allowing the PWA to open in its own window without the standard browser interface, providing an experience similar to a native mobile application.
- **background_color** and **theme_color** — Define the splash screen and browser UI colors, ensuring visual consistency with the application's overall design.
- **icons** — Provide the image resources used for different platforms. In this project, both an SVG logo and an Apple touch icon are included to ensure proper rendering across devices. It is also important to note that the inclusion of the `apple-touch-icon.png` is essential for ensuring proper installation on iOS devices. Unlike Android and most modern browsers, iOS does not yet fully support the `Web App Manifest` standard. As a result, Safari relies on

the presence of the `apple-touch-icon` link to display the application icon correctly when the PWA is added to the home screen. Without this icon, the installed app may appear with a generic placeholder image or fail to show the intended branding.

The manifest file used in the project is shown below:

```
{
  "name": "MartiniFlow BPM",
  "short_name": "MartiniFlow",
  "start_url": "/",
  "display": "standalone",
  "background_color": "#fff",
  "theme_color": "#3f51b5",
  "icons": [
    {
      "src": "logo.svg",
      "sizes": "any",
      "type": "image/svg+xml"
    },
    {
      "src": "apple-touch-icon.png",
      "sizes": "any",
      "type": "image/png"
    }
  ]
}
```

The manifest, together with the service worker, enables the application to be installed on mobile and desktop devices.

4.2.2 Service Worker

The service worker operates as an independent script running in the background of the Progressive Web Application. Its primary responsibilities include intercepting network requests, managing cached resources, handling version updates, and supporting background message delivery through Firebase Cloud Messaging (FCM). In this project, the service worker was implemented using the `vite-plugin-pwa` library, which integrates **Workbox**—a framework designed to simplify service worker configuration and asset caching.

Integration with Vite and Workbox The integration is based on the `injectManifest` strategy provided by the plugin. During the build process, Vite automatically generates a manifest of hashed static assets (JavaScript, CSS, icons) and injects this information into the custom service worker. This allows the application to benefit

from automatic pre-caching while preserving full control over runtime behavior. The following configuration excerpt shows the relevant section of the Vite setup:

```
// vite.config.ts (excerpt)
import { VitePWA } from 'vite-plugin-pwa';

VitePWA({
  registerType: 'autoUpdate',
  strategies: 'injectManifest',
  srcDir: 'src',
  filename: 'custom-sw.js',
  injectManifest: {
    globPatterns: ['**/*.{js,css,html,svg,png}'],
  },
});
```

At runtime, Workbox manages the pre-caching and versioning of static resources through the Cache Storage API. Outdated entries are automatically removed when a new service worker version becomes active. The `autoUpdate` registration mode ensures that the latest version of the service worker is downloaded in the background and activated as soon as all older instances are closed.

Custom logic and caching strategies The custom service worker defines explicit logic for installation, activation, and network request interception. Static build assets are pre-cached automatically, while runtime caching policies are defined for specific resource types. For example, HTML files follow a *NetworkFirst* strategy to ensure the most recent content is displayed, while static images are handled with a *StaleWhileRevalidate* strategy to improve loading performance. The following excerpt illustrates the main structure of the service worker:

```
// custom-sw.js

// 1) Precache build assets (injected manifest)
import { precacheAndRoute } from 'workbox-precaching';
precacheAndRoute(self.__WB_MANIFEST || []);

// 2) Lifecycle management
self.addEventListener('install', () => self.skipWaiting());
self.addEventListener('activate', (event) =>
↪  event.waitUntil(clients.claim()));

// 3) Runtime routing examples
import { registerRoute, setDefaultHandler, NavigationRoute } from
↪  'workbox-routing';
import { NetworkFirst, StaleWhileRevalidate } from 'workbox-strategies';
```

```
registerRoute(new NavigationRoute(new NetworkFirst({ cacheName: 'html-nav'
↪  })));

registerRoute(
  ({ request }) => request.destination === 'image',
  new StaleWhileRevalidate({ cacheName: 'images' })
);

setDefaultHandler(({ request }) => fetch(request));
```

In addition to resource caching, the service worker also includes the event listeners necessary for background message handling. This integration allows FCM notifications to be received and displayed even when the application is closed or running in the background.

Rationale for the chosen configuration Workbox offers two main setup strategies: `generateSW` and `injectManifest`. The first automatically generates a generic service worker with predefined caching behavior, which is suitable for static sites but limited in flexibility. The `injectManifest` approach was selected because it allows to maintain full control over lifecycle events, caching logic, and integration with third-party services such as Firebase.

Update and lifecycle management The update process follows the standard service worker lifecycle, summarized in Figure 4.1. When a new build is deployed, the browser downloads the updated service worker and prepares it in the background. Once the previous version is no longer in use, the new one becomes active, replaces outdated caches, and claims all open clients. This approach ensures that users always access the latest version of the application without any need to download or refresh manually.

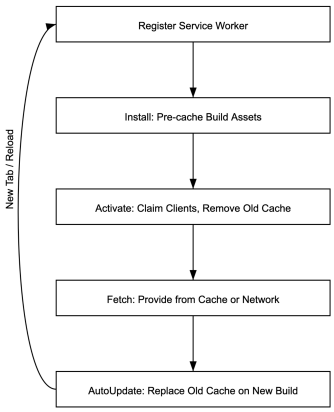

Figure 4.1: Service Worker Lifecycle and Update Flow.

4.2.3 Notifications

Firebase Setup

In order to set up the system with Firebase Cloud Messaging service, a new project was created through the Firebase console (`https://console.firebase.google.com/`). After signing in, it is necessary to create a new Firebase project. The new project can be optionally linked to a Google Analytics account for monitoring purposes. From the project overview page, its possible to access the settings and the project configurations, then its needed to create a new "web application project" from this page, that will contain the necessary credentials required to be initialize the Firebase SDK inside the client application, as shown in the figure below a snippet of the configuration is generated that needs to be added to the React application.

```
{
  const firebaseConfig = {
  apiKey: "gHF7JOxA2vqY6cFwDFdalSgHF7JOxA2vqY6cFwDFdalSg",
  authDomain: "martini-dev.fa.com",
  projectId: "martini-dev",
  storageBucket: "martini-dev.fa.app",
  messagingSenderId: "111177771114",
  appId: "1:111177771114:web:dc49f70bdc49f70bdc49f",
  measurementId: "G-0000V00000"
  };
}
```

In addition to this configuration, a key should be generated to be used in the backend; it's possible to generate it from the "service accounts" tab in the project settings. By selecting the preferred programming language and pressing "generate a new private key", a JSON file will be downloaded containing all the necessary credentials. This file should be stored securely in the backend server and used to initialize the Firebase Admin SDK. It includes the information such as the project ID, client email, private key,auth URI,auth provider x509 cert URL,client x509 cert URL, and other details required for server-side authentication and authorization when sending messages to client applications. The next step is to enable push notifications in the application; a new key pair should be generated for web push certificates, this can be done from the "cloud messaging" tab in the project settings. This key pair (VAPID keys) is used to allow Firebase Cloud Messaging (FCM) to authenticate push notifications sent from the backend to client browsers. [13]

Device registration and token management Each client instance must register with FCM by obtaining a unique device token. This token identifies the browser instance and is used by the backend to target specific users when sending

notifications. In the implementation, the `getToken()` method retrieves the token using the project's VAPID key and the registered service worker. Once obtained, the token is sent to the backend via an authenticated API call to associate it with the user account.

```
export const requestFirebaseNotificationPermission = async ():
↪  Promise<boolean> => {
  try {
    const token = await getToken(messaging, {
      vapidKey: "BHTAf07PvofiGUUnxWG04",
      serviceWorkerRegistration: await navigator.serviceWorker.ready,
    });

    if (token) {
      const response = await fetch("/api/notification/addDevice", {
        method: "POST",
        headers: { "Content-Type": "application/json" },
        credentials: "include",
        body: JSON.stringify({ key: token })
      });
      return response.ok;
    }
    return false;
  } catch (err) {
    console.error("Error retrieving or sending FCM token:", err);
    return false;
  }
};
```

This mechanism ensures that each authenticated user device is registered in the backend and can later receive notifications related to assigned tasks or process updates in that device.

Server-side configuration On the backend, the downloaded *service account* JSON file is used to initialize the Firebase Admin SDK. It contains the credentials required for server-side authentication (project ID, client email, and private key). This configuration allows the backend to send messages through the Firebase Cloud Messaging API to registered devices. The generated device tokens are stored securely and used for targeted or broadcast notifications.

In addition, a pair of **VAPID keys** (Voluntary Application Server Identification) must be generated from the "Cloud Messaging" tab in the Firebase console. These keys enable Firebase to authenticate and authorize push requests between the backend and client browsers, ensuring secure message delivery.[13]

Foreground and background message handling The client application distinguishes between messages received while it is open (foreground) and those received

when it is inactive (background). Foreground messages are handled directly in the React client using the `onMessage()` listener, which triggers a callback to display in-app notifications. Background messages are processed by the same service worker described in the previous section, which allows push notifications to be displayed even when the PWA is closed.

```
export const onForegroundMessage = (callback: (payload: MessagePayload) =>
↪  void) => {
  return onMessage(messaging, (payload) => {
    console.log("Foreground message received:", payload);
    callback(payload);
  });
};
```

By combining the service worker context with Firebase's messaging API, the application ensures consistent delivery of push notifications in all operational states. This design enables real-time updates for users in the field—such as technicians and veterinarians—who rely on timely notifications to perform their assigned activities efficiently.

Permission Handling Across Devices

Before notifications can be displayed, explicit user consent is required. Modern browsers implement strict permission models to prevent unsolicited notifications. When the application first attempts to register for messaging, the browser prompts the user to grant or deny notification access through the 'Notification.requestPermission()' API. Only when the permission status is set to `granted`, Firebase Cloud Messaging can issue and display notifications on that device. If the user denies permission, no further requests can be made until the permission is manually changed in the browser settings.

Each Browser applies slightly different policies regarding notification prompts and background delivery. For instance, Chromium-based browsers (such as Google Chrome, Edge, and Opera) support FCM through standard Web Push APIs and allow background notifications via the registered service worker, provided that the user has interacted with the site. Mozilla Firefox follows a similar model but may automatically block repeated permission prompts if users ignore them multiple times. Apple's Safari browser supports web push on macOS and iOS only for installed web applications and requires that notifications be triggered through the native Apple Push Notification service (APNs) bridge managed by Firebase. These differences make explicit permission handling and user feedback mechanisms critical for a reliable cross-platform notification experience.

4.3 Backend and Integrations

The backend of the system was developed with the **Kotlin** programming language and using the **Spring Boot** framework. It serves as a centralized integration layer connecting the Frontend with the **Camunda 8** process engine, the **Firebase Cloud Messaging** service, and the **Keycloak** identity management system. The backend runs inside a **Kubernetes** cluster and exposes a REST API tailored for the frontend through a **Backend-for-Frontend (BFF)** architecture, ensuring secure, optimized, and domain-specific communication.

4.3.1 BFF Endpoints

As mentioned before, BFF acts as a dedicated gateway for the client application, simplifying the communication with multiple internal services such as Camunda, Zeebe workers, and Keycloak. It aggregates and validates data before forwarding it to the process engine, applies access control based on user roles, and manages the lifecycle of tasks and process instances. It reduces latency and isolates the client from the internal complexity of the system by exposing a single, unified REST API.

4.3.2 Data Formatting for the UI

Before being transmitted to the frontend, the backend performs a structured data transformation process. The information retrieved from **Camunda** process instances and other microservices often contains nested, verbose, or workflow-specific structures that are not directly suitable for rendering within the React interface. To improve performance and maintain a clear separation between business logic and presentation, the Backend-for-Frontend layer converts these responses into concise, standardized JSON objects specifically designed for the UI components.

Each transformation step includes filtering out non-relevant fields, flattening hierarchical data, and aggregating attributes that are frequently accessed together by the frontend. For example, instead of sending complete process variables or internal BPMN identifiers, the backend returns compact summaries containing only the essential attributes—such as task name, status, assigned role, and related timestamps. This ensures that the client receives data in a predictable structure, minimizing the need for additional parsing or formatting operations on the client side.

Furthermore, the BFF performs **data validation** and **type normalization** to align backend responses with the expected data models defined in the frontend application. This includes converting date and time fields to ISO-8601 format, mapping enumerations to consistent labels, and ensuring numerical values adhere to fixed precision. Through these transformations, the backend guarantees that

the data displayed in tables, summaries, and forms remains consistent with the definitions of the BPMN process models, regardless of their internal implementation or source system.

4.3.3 Camunda Usage During Development

During the development phase, the **Camunda 8** platform played a central role not only as the process orchestration engine but also as a reference environment for validating workflows and verifying the correctness of the frontend integration. Because the workflow design was handled by another company, the development team used Camunda's built-in interfaces such as Modeler, Tasklist, and Operate as shared tools to ensure alignment with the process definitions and to validate task behavior.

Tasklist Interface The **Tasklist** was used as a practical reference for inspecting how each task appeared once deployed on the Camunda engine.

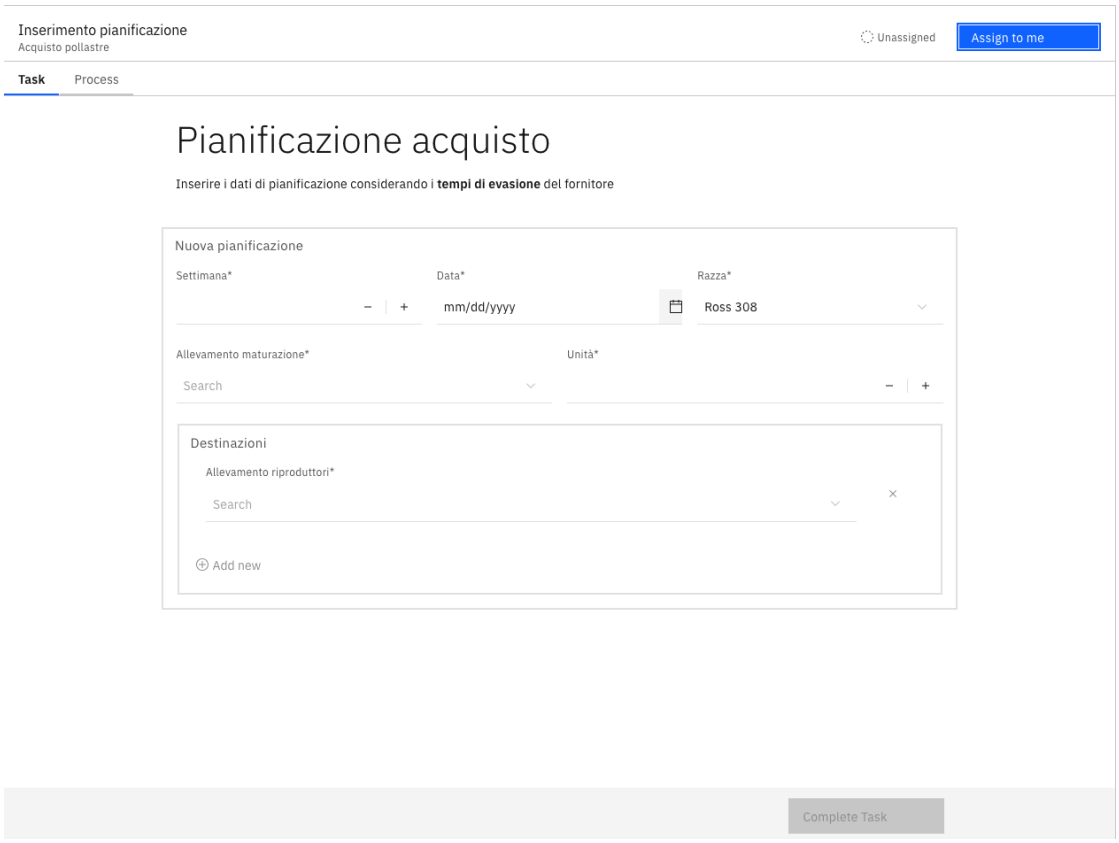

Figure 4.2: Camunda Tasklist interface.

By opening a task instance, developers could review the automatically generated form, confirm mandatory fields, and examine variable names and data types. This ensured that the dynamic form renderer implemented in the frontend produced an equivalent layout and logic, fully consistent with the BPMN schema. This step was especially useful during form iteration, when process definitions changed frequently.

Operate Interface The Operate dashboard was mainly used to monitor the status of process instances started from the frontend. It provided an immediate visual confirmation of the active workflow step and the values exchanged between the application and the engine. Through the variable panel (Figure 4.3), it was possible to verify that the data sent through the BFF was correctly propagated within the process and that each BPMN path executed as expected. This proved valuable for debugging synchronization issues and validating process transitions without direct API inspection.

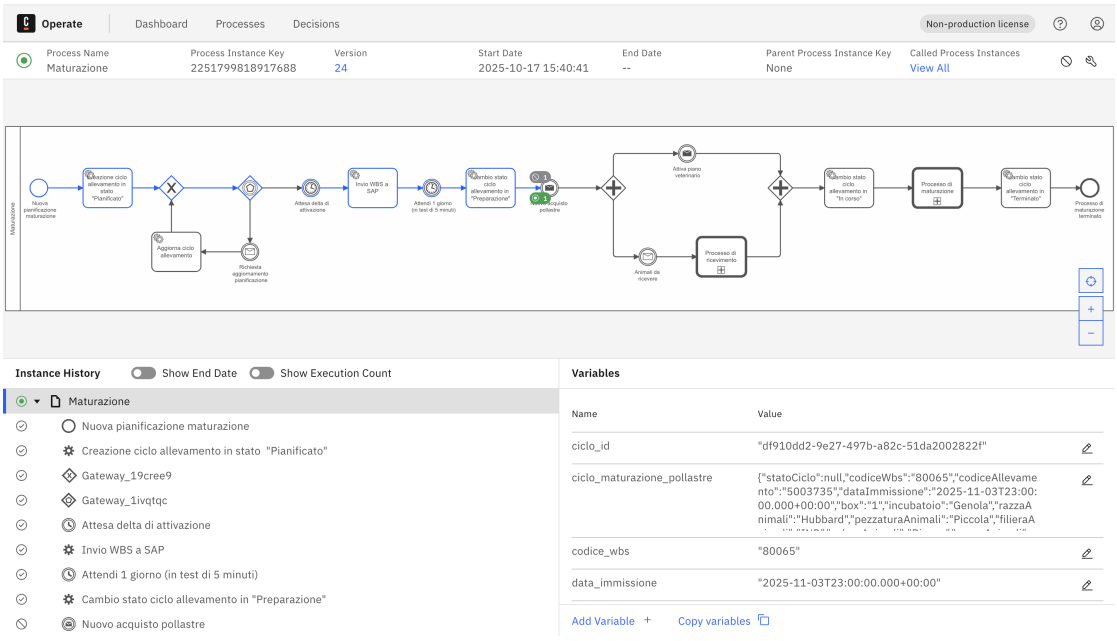

Figure 4.3: Camunda Operate interface.

Integration via BFF All interactions with Camunda occurred through the BFF layer, which mediated authentication and API communication. The frontend never directly accessed Camunda's REST endpoints; instead, the BFF retrieved task definitions, delivered JSON schemas for form rendering, and handled task completion requests. This ensured that both security and role-based access control remained centralized, while developers could still cross-check behavior using the

Tasklist and Operate tools.

4.3.4 Authentication and Authorization

User authentication and authorization in the system are based on the **Keycloak** identity management platform, which implements the OAuth2 and OpenID Connect protocols. The backend, developed with **Spring Boot**, acts as an OAuth2 resource server and validates access tokens issued by Keycloak. Each authenticated session is represented by a **JSON Web Token (JWT)** that encodes user information such as username, email, and assigned roles. These roles correspond to the organizational structure of the company and include *tecnico*, *veterinario*, *pianificatore*, *manager*, and *admin*. Through this model, access to backend resources is granted only to users with valid tokens and the appropriate roles, ensuring a secure and traceable authentication process.

Authentication Flow Overview The authentication process follows the OpenID Connect (OIDC) authorization code flow mediated by the **APISIX** gateway. When the user accesses the PWA and the application calls the endpoint `/api/me`, the gateway checks for an existing session cookie. If no valid session is found, the request is forwarded to the **BFF**, which returns an HTTP `403 Unauthorized` response. At this point, the frontend detects that the user is not authenticated and redirects to a protected path (`/secure`), triggering the gateway's OIDC plugin.

The gateway then redirects the browser to **Keycloak**, which displays the login page. After the user submits their credentials, Keycloak verifies them and redirects back to APISIX with an authorization code. APISIX exchanges this code for a JSON Web Token (JWT) and sets a session cookie in the user's browser. This cookie is encrypted and used to maintain the authenticated state across subsequent requests.

When the page reloads, the frontend again calls `/api/me`. This time, APISIX retrieves the JWT from the session cookie, validates it, and forwards the request to the BFF. The BFF confirms the token's validity and responds with the user's information, which allows the frontend to recognize the authenticated session.

From this point onward, all API requests automatically include the JWT in the authorization header, ensuring continuous authentication until the token expires or the session is cleared.

This approach ensures that credentials are handled exclusively by Keycloak, while the APISIX gateway manages secure session handling and token exchange on behalf of the frontend.

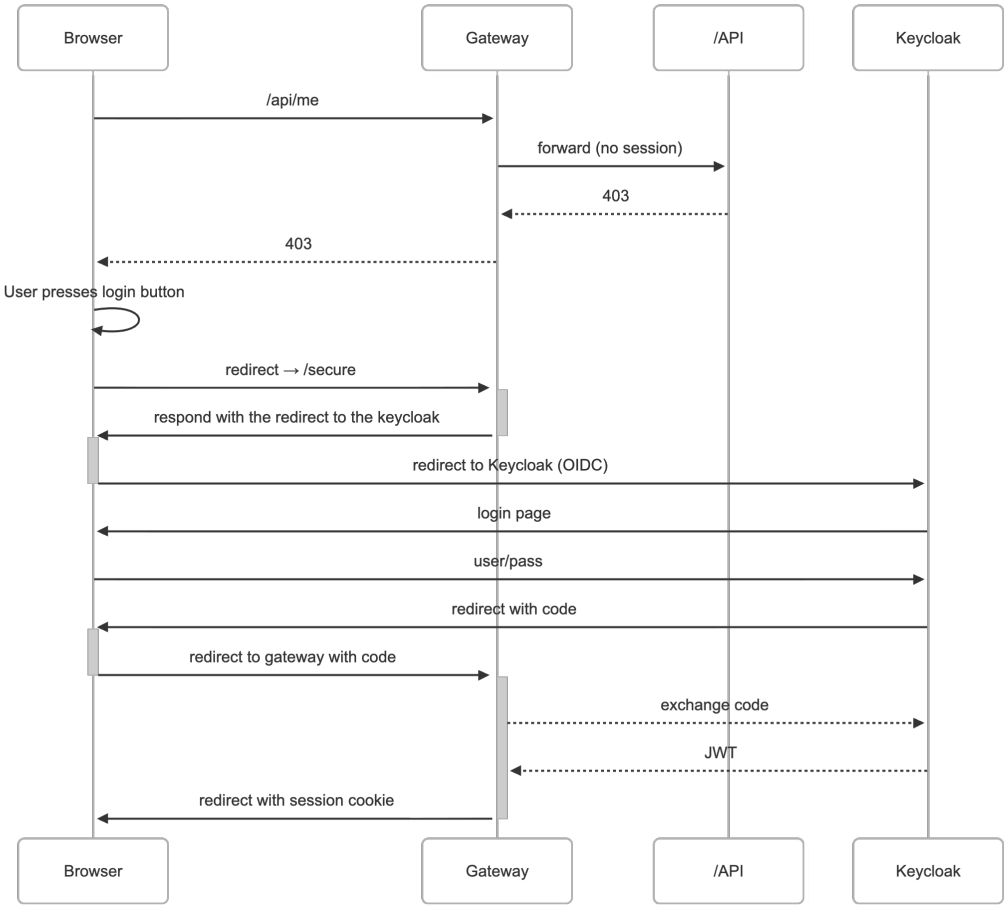

Figure 4.4: Authentication flow.

As a result, sensitive authentication data never passes directly through the client application, preserving security.

Frontend integration and role management On the frontend, authentication is handled using the official `keycloak-js` library. When the application starts, the Keycloak client is initialized with the realm, client ID, and server URL parameters, prompting the user to log in via the Keycloak interface. Upon successful login, the client receives an access token and a refresh token. The access token is then attached automatically to every API request made by the application, while the refresh token allows the session to be renewed transparently before expiration.

To keep track of authentication and apply role-based permissions, the application relies on **Redux Toolkit**. User details and assigned roles are saved in the `UserSlice` of the global store, making them available across all components.

This shared structure supports **Role-Based Access Control (RBAC)** directly in the frontend: interface elements such as pages, buttons, or sections appear only when the user's role allows it. For instance, features like task management or process monitoring are visible only to specific user groups. Access is verified through a `RequireRole` component, which checks the roles in the current state before rendering the page. If a user does not have the necessary permissions, an error message is shown instead.

```
// Example of route protection with RequireRole
export const RequireRole = ({ allowedRoles }: RequireRoleProps) => {
  const roles = useSelector((state: RootState) => state.user.roles);
  const isAuthorized = roles.some(role => allowedRoles.includes(role));

  if (!isAuthorized)
    return <ErrorScreen error="Non sei autorizzato a visualizzare questa
    ↪  pagina"/>;

  return <Outlet/>;
};
```

This mechanism ensures that authorization rules are applied consistently both in the backend and in the user interface.

4.4 Deployment

The deployment process was designed to make the system reliable, modular, and easy to update across different environments. To achieve this, both the frontend and backend were containerized and deployed through a Kubernetes cluster, using Helm charts to manage configuration and releases. This approach provides consistency between environments, simplifies updates, and allows the application to scale or recover automatically when required.

4.4.1 Containerization

The frontend application was packaged as a Docker image using a multi-stage build. During the first stage, a Node.js environment compiles the React source code into static assets. In the second stage, these assets are copied into a lightweight Nginx container, which serves them on port 80. This separation between build and runtime environments results in a smaller, more secure image that contains only the files needed in production.

```
# Stage 1 - Build
FROM --platform=$BUILDPLATFORM node:22 AS builder
```

```
WORKDIR /app
COPY package*.json ./
RUN npm install
COPY . .
RUN npm run build

# Stage 2 - Serve
FROM nginx:alpine
RUN rm -rf /usr/share/nginx/html/*
COPY nginx.conf /etc/nginx/conf.d/default.conf
COPY --from=builder /app/dist /usr/share/nginx/html
EXPOSE 80
CMD ["nginx", "-g", "daemon off;"]
```

Once built, the image is pushed to a private registry and later retrieved by the Kubernetes cluster using registry credentials stored in the `regcred` secret. The backend service follows a similar approach, packaged as a standalone container that exposes REST endpoints consumed by the frontend.

4.4.2 Kubernetes Deployment

The deployment on Kubernetes is managed through Helm charts, which define the manifests for each component and allow dynamic configuration through parameterized values. Each service—the frontend, backend, and related infrastructure—is deployed within the same namespace, simplifying networking and ensuring internal communication between pods.
An example of the deployment configuration is shown below:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: frontend
  namespace: {{ .Values.namespace }}
spec:
  replicas: 1
  selector:
    matchLabels:
      app: frontend-app
  template:
    metadata:
      labels:
        app: frontend-app
    spec:
      imagePullSecrets:
        - name: ...
      containers:
        - name: notification-frontend
```

```
          image: {{ .Values.images.frontend.name }}:{{
          ↪  .Values.images.frontend.tag }}
          ports:
            - containerPort: 80
```

Apache APISIX manages incoming HTTP traffic as the ingress controller and adds routing and authentication features. In this configuration, APISIX integrates directly with **Keycloak** using an OpenID Connect (OIDC) plugin. This allows the gateway to handle authentication at the ingress level, ensuring that only authenticated users can access protected routes before requests reach the frontend service.

```
apiVersion: apisix.apache.org/v2
kind: ApisixPluginConfig
metadata:
  name: oidc-plugin
  namespace: {{ .Values.namespace }}
spec:
  plugins:
    - name: openid-connect
      enable: true
      config:
        client_id: {{ .Values.iam.client_id }}
        discovery: https://{{ .Values.iam.url }}/realms/{{ .Values.iam.realm
        ↪  }}/.well-known/openid-configuration
        realm: "{{ .Values.iam.realm }}"
        scope: "openid email profile"
---
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  annotations:
    k8s.apisix.apache.org/plugin-config-name: oidc-plugin
  name: ingress-ingress-secure
  namespace: {{ .Values.namespace }}
spec:
  ingressClassName: {{ .Values.ingress.className }}
  rules:
    - host: {{ .Values.ingress.externalUrl }}
      http:
        paths:
          - path: /secure
            pathType: Prefix
            backend:
              service:
                name: notification-frontend-svc
                port:
                  number: 80
```

With this setup, secure routes under the `/secure` path are automatically protected by the OIDC authentication flow, which delegates user verification to Keycloak. Kubernetes handles pod scheduling, scaling, and rolling updates, while APISIX ensures authenticated access and HTTPS routing. This combination provides a reliable and flexible production environment where new releases can be deployed seamlessly and securely.

4.5 Application Interface and Main Pages

In this part, the main pages of the application are explained, and the interface is presented through a series of screenshots to show the final product.

4.5.1 Main pages and Functionalities

The interface of the application was designed to provide an intuitive and uniform experience across devices, prioritizing clarity, accessibility, and responsiveness. Each page reflects a specific stage of the production workflow and allows users to access or record information according to their assigned role in the system. The following subsections summarize the main pages and their corresponding functionalities.

Navigation Layout Upon login, users are directed to the home page. The main layout in desktop view includes a top navigation bar that provides direct access to all important modules, including Cycles, Forms, and Notifications. This persistent structure ensures that essential controls remain visible while navigating between pages. The interface is fully responsive, adapting the layout to smaller screens. For small screens such as mobile, this navbar turns into a mobile drawer menu.

Tasks The Tasks section is where users carry out activities created by Camunda workflows. It serves as the main area for getting work done in the application. Each active task can be opened to display the corresponding form provided by the BPMN process. This section is primarily used by technicians and veterinarians to record measurements, health data, or inspections. The page has two tabs: Assigned Tasks, which lists tasks currently assigned to the user, and Unassigned Tasks, which shows activities that are available to be assigned.

Plannings The *Planning* page supports the configuration and monitoring of upcoming production cycles. Planners can view the list of plans (with search and filtering by date, farm, or status), open a plan to inspect its details, and trigger actions such as creating or updating purchase proposals for chicks. Each plan shows its associated destinations, quantities, and scheduling constraints, together

with the current approval status. From a plan, users can navigate to the related workflow tasks (e.g., health communications or transport authorizations) and, where required, submit data via the dynamic forms described earlier. Data are retrieved and synchronized via RTK Query to ensure that changes made by planners or approvers are reflected immediately across the interface. Access to approval actions is restricted to users with the appropriate role (e.g., planner or supervisor/approver), aligning the UI with the process responsibilities defined in the workflow. It is also possible to filter, sort, or search plannings based on various criteria, such as date, status, or farm location.

Cycles The *Cycles* page provides access to all active and historical production cycles. For each cycle, users can view detailed information about the farm, the animals in the cycle, and the current production phase (Maturation, Reproduction, or Fattening). From this view, planners and managers can monitor the progress, access related summaries, or navigate to task lists for further actions. It is also possible to sort or search cycles based on various criteria, such as cycle code, date, or farm location.

Summary Dashboards Each production cycle includes dedicated summary views—*General Summary*, *Feed Summary*, *In/Out Summary*, and *Medicine Summary*. These dashboards aggregate process data collected throughout the workflow and present it in structured tables or charts. The summaries support managerial decision-making by allowing planners and supervisors to analyze performance indicators such as feed consumption, mortality rates, or treatment frequency. All views are accessible through nested routes and are updated automatically when new data is registered by operators.

Notifications The notification panel provides real-time feedback about newly assigned tasks, completed activities, and workflow events triggered by Camunda. Users can filter notifications or open them directly to view related forms or process details. It's also possible to filter, delete, or mark the notifications as read.

In summary, the above pages are the main functional areas of the application, each designed to support specific user roles and process stages. Each of them contains the data and information or functionalities that are designed based on the requirements of the stakeholders and are shown in the screenshots.

4.5.2 Interface Layout and Visual Representation

The following figures illustrate the main sections of the developed Progressive Web Application. They include both desktop and mobile views, as well as examples in

dark mode, to demonstrate the responsive and theme-adaptive design implemented through React and Chakra UI. Each page corresponds to one of the functional areas described in the system architecture and routing structure.

Installation As mentioned in Chapter 2, the application can be installed on the user's device. On iPhones and iPads, when the user opens the application in Safari and taps the *Share* icon, the option *"Add to Home Screen"* becomes available. Selecting it installs the application as a standalone icon on the home screen like normal applications and remains there unless the user decides to uninstall it. Once launched, the PWA runs in full-screen mode without the browser interface and behaves similarly to a native application. On Android, browsers such as Chrome, Edge, and Firefox natively support PWA installation. When the user visits the application, it's possible to access the browser menu by tapping the three dots in the upper-right corner and selecting the option *"Add to Home screen"*. Depending on the browser and device, an installation banner or prompt may also appear automatically, inviting the user to install the app. Confirming this installs the PWA as an independent application entry in the system's launcher, with its own icon, settings, and notification permissions. Both platforms are shown in the following figures.

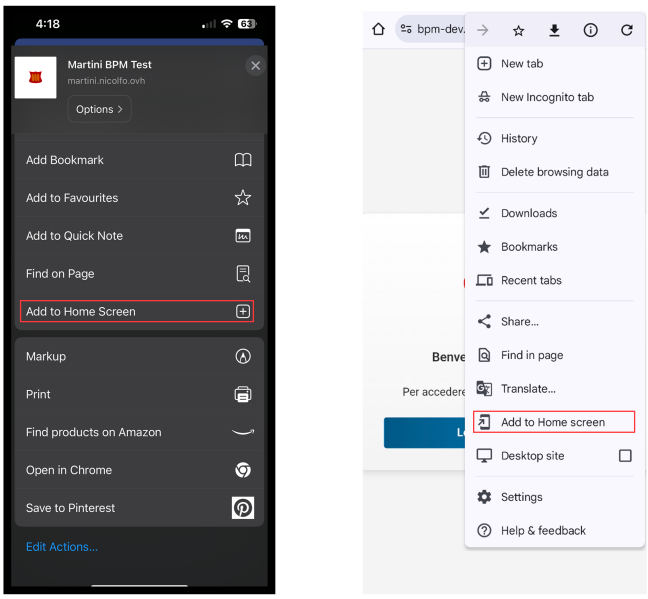

Figure 4.5: Installation on iOS and Android devices.

Notification In Figure 4.6, the notification interfaces are shown. The first screen shows a system-level push notification received on the device lock screen.

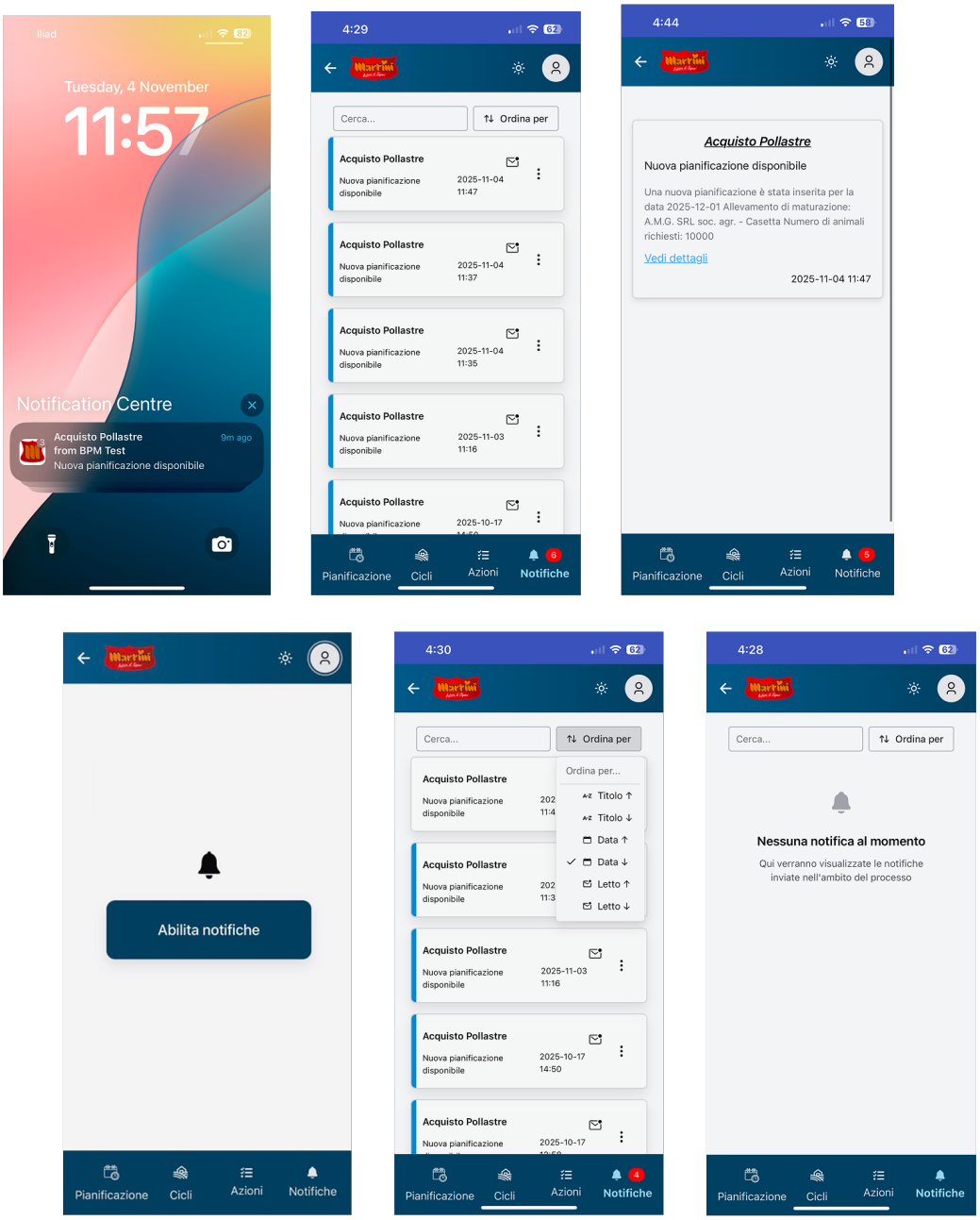

Figure 4.6: Notification interfaces.

Inside the application, there is a red badge near the notification icon in the navigation bar that shows the number of unread messages. Inside the notification page, it's possible to see the list of all notifications, read or unread, that can be opened to show more details about the process or task. Users can also delete messages or mark them as read directly from this page.

In the settings page, there is a button *"Abilita notifiche"* (Enable Notifications), to enable the permission to receive notifications. The reason for putting this button is to support browser-specific permission policies. It is particularly useful for devices where the automatic permission banner is not displayed, ensuring that users can still activate push notifications explicitly. An empty-state message notifies the user that there are no ongoing updates at this time.

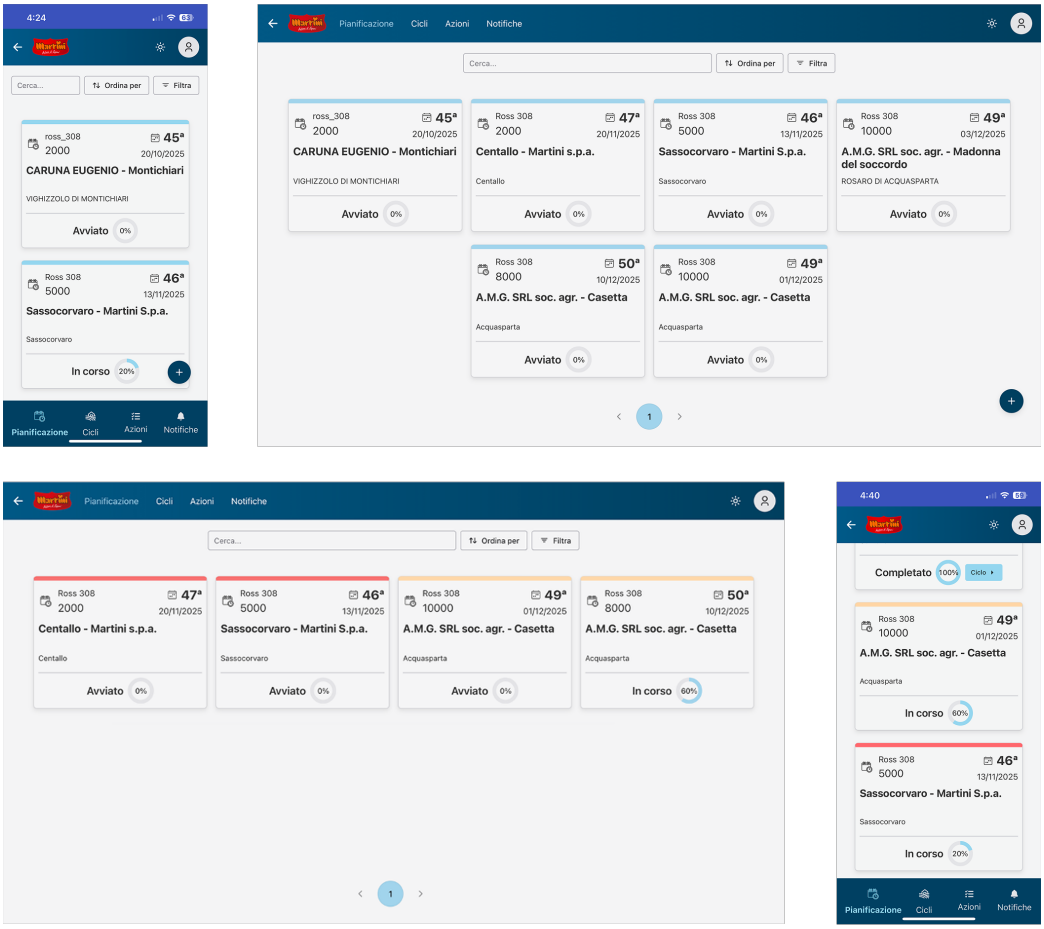

Figure 4.7: Planning List page in mobile and desktop.

Planning The screenshots in Figures 4.7 and 4.8 represent the Planning module, one of the central components of the application, both in mobile and desktop environments.

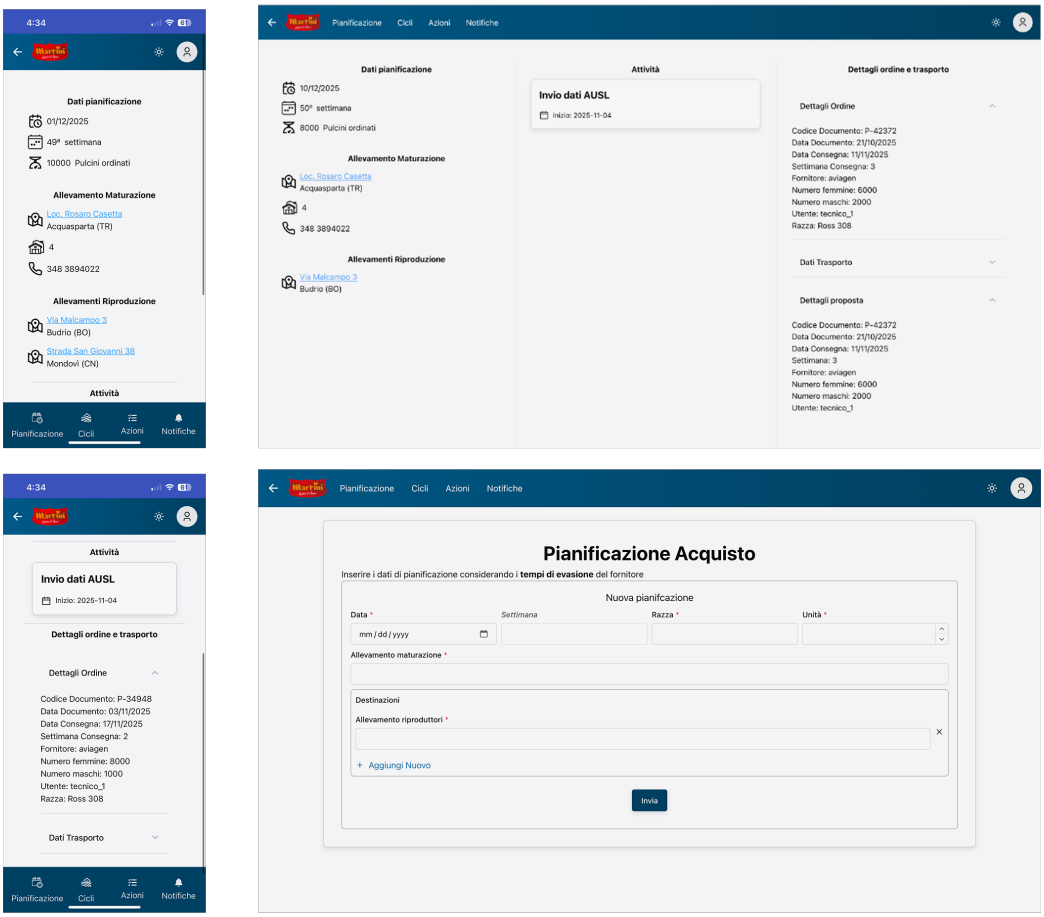

Figure 4.8: Planning Details page and Purchase Planning Form.

Figure 4.7 shows the Planning List page in mobile and desktop versions. For planners, this view displays all existing plans and includes a floating "+" button that allows them to create new planning entries. Each card summarizes essential data such as delivery date, breed (razza), number of chicks, and destination farms. Technicians, on the other hand, see the same list but with color-coded headers that highlight the status of related tasks — for instance, red or orange headers are used to indicate urgent or pending actions, while neutral tones mark completed or non-critical ones. This visual hierarchy allows users to quickly identify priorities without needing to open each plan individually.

Figure 4.8 displays two complementary views: On top, the Planning Details

page presents all information associated with a specific plan, including supplier data, transport details, and related tasks to that specific plan. Below, the Purchase Planning Form (Pianificazione Acquisto) is shown, which is only visible to planners, and it's used to create new planning entries.

Cycles The screenshots in Figures 4.9 and 4.10 represent the Cycles pages, the part designed to track and monitor the status of cycles.

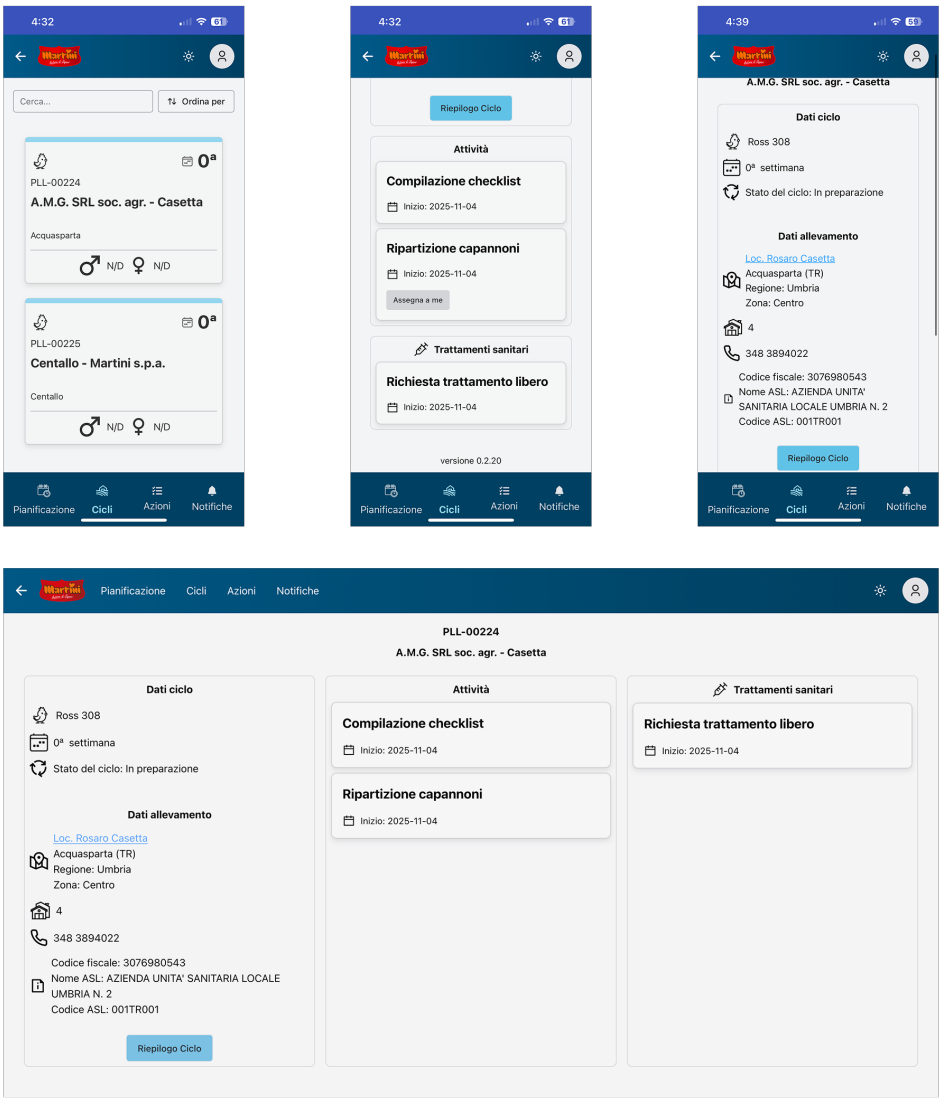

Figure 4.9: Cycles list and Cycles details pages.

Figure 4.9 shows the Cycle List page, which provides an overview of all active and completed cycles. Each card contains essential information such as the cycle identifier, farm location, breed, and number of animals. Figure 4.10 illustrates the Cycle Detail and Summary (Riepilogo) pages, accessible from the cycle card via the Riepilogo Ciclo button. This section provides an in-depth view of the production process, including farm information, animal distribution, and activity lists such as checklists or sanitary treatments. The summary interface shows combined statistics such as mortality rate, animal count, and average performance indicators. It also includes visual charts and tables.

These views allow users to evaluate production results and identify trends directly from the interface. Tabs such as Generale, Trasferimenti, Medicinali, and Mangimi provide access to additional process data collected during the cycle, such as the medicine used or transportation information.

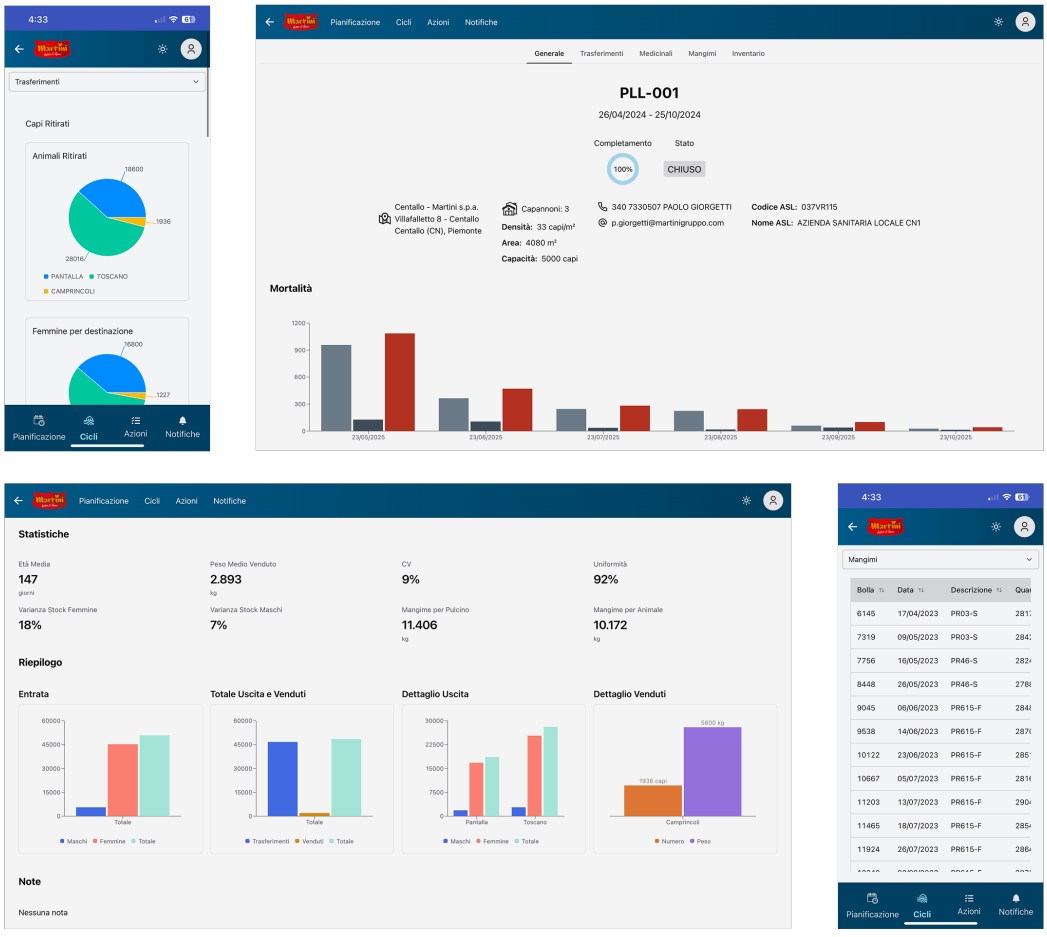

Figure 4.10: Cycle summary pages

Tasks Figure 4.11 presents the Task Overview page, divided into two main tabs: Assigned Tasks and Unassigned Tasks. Assigned tasks represent activities that are already claimed by the current user, while unassigned ones can be taken over by pressing the Assign to me button. This mechanism ensures clear task ownership and prevents duplicate work among users. Each task card includes its title, start and due dates, and, when relevant, an Urgent status badge to highlight priority tasks. The interface adapts responsively between desktop and mobile layouts.

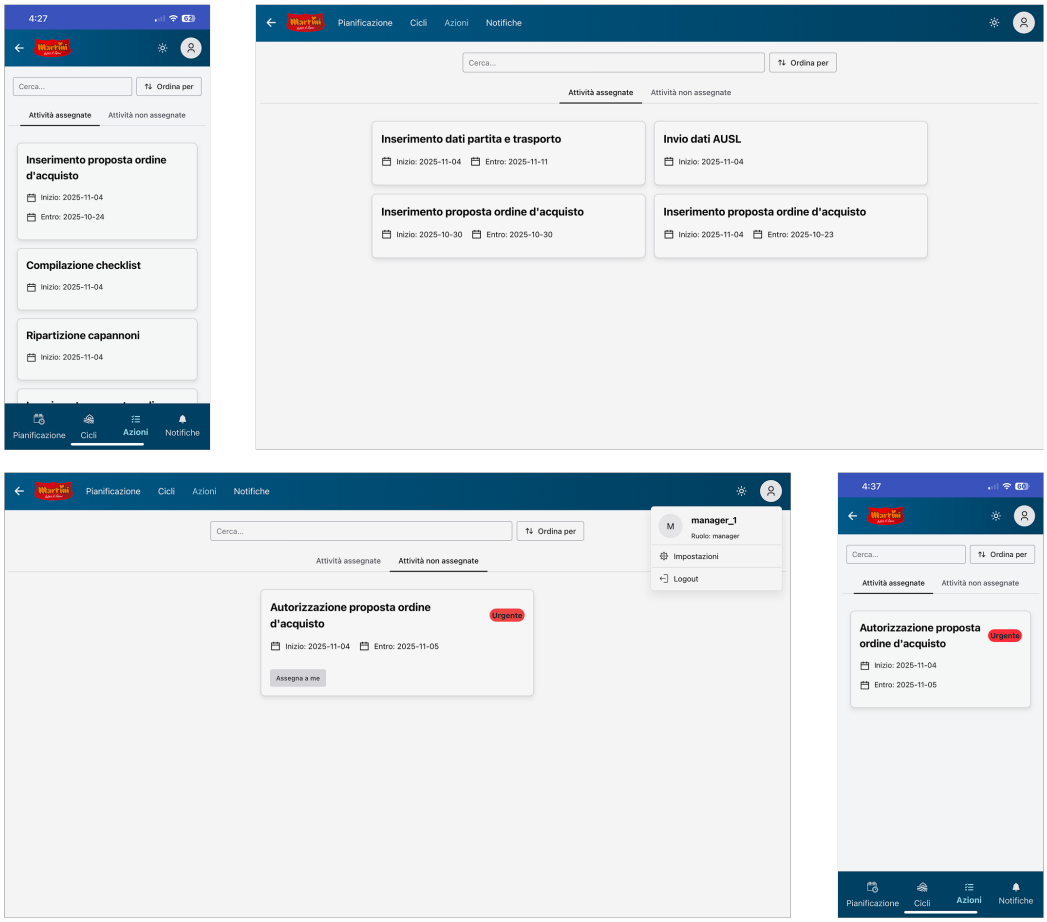

Figure 4.11: Task list view with assigned and unassigned activities

Figure 4.12 illustrates the form view displayed when a user opens a specific task. Each form represents a BPMN process step. Depending on the activity, the form may have various input components, such as text fields, selectors, checkboxes, numeric inputs, and file upload sections. In more complex forms, such as data entry or transport reporting, structured tables are used to summarize related live

data.

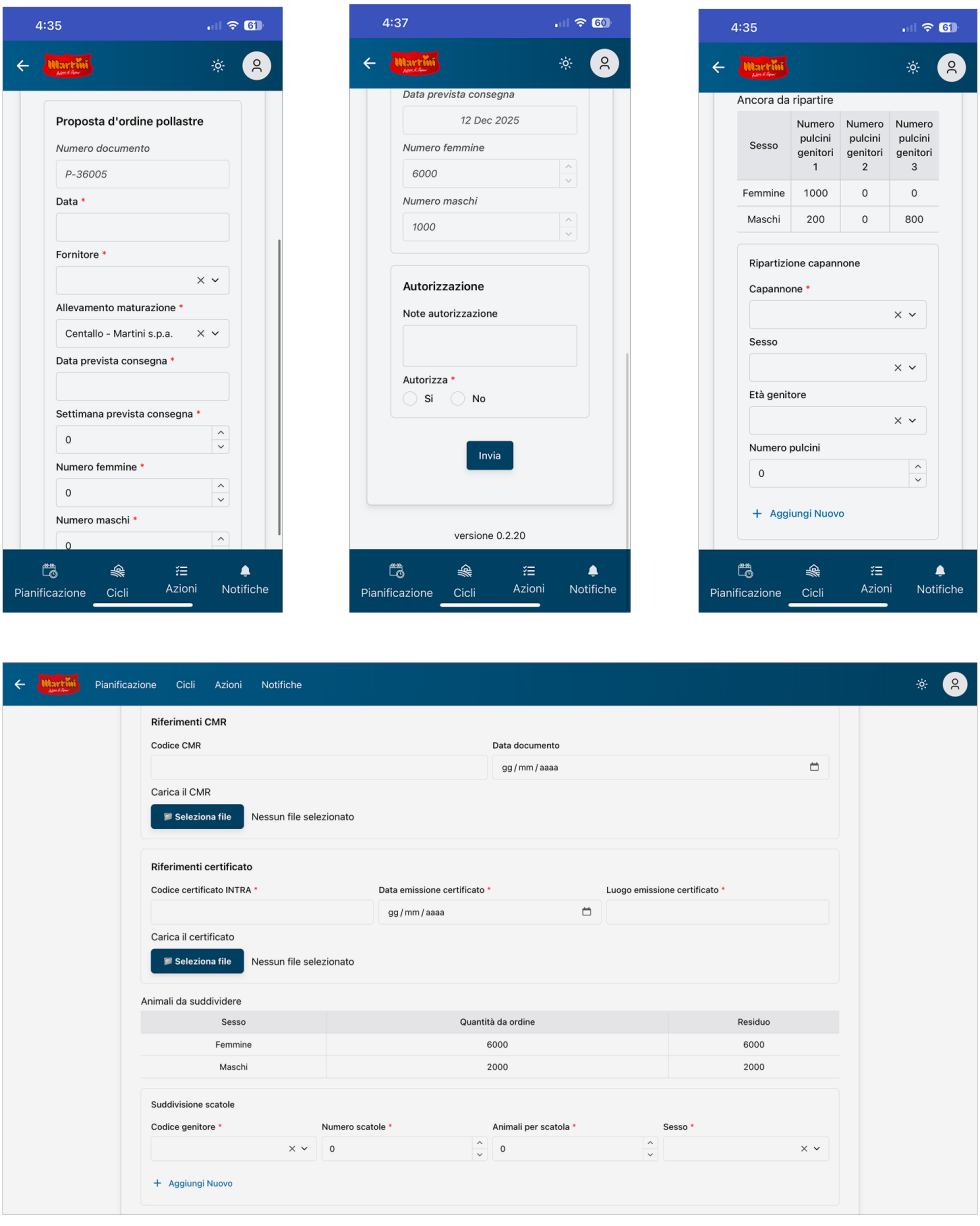

Figure 4.12: Example of different task forms interfaces

Dark Mode The application also supports a manual dark mode toggle, allowing users to switch themes using the dedicated sun/moon button located in the navigation bar. When activated, the interface switches to darker tones optimized for low-light environments, while preserving readability and consistent visual contrast across all components. Figure 4.13 shows an example of the dark mode applied to the notifications and tasks pages, in mobile and desktop views.

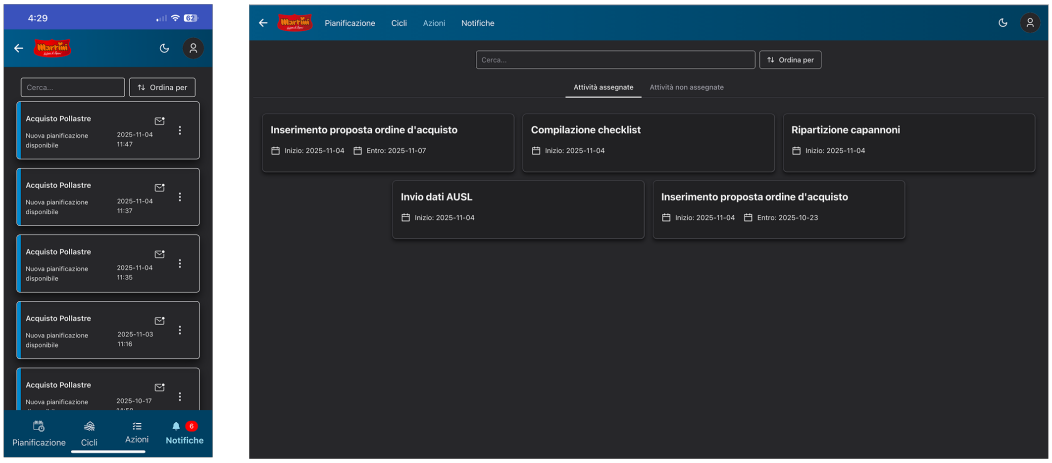

Figure 4.13: Application interface in dark mode

Chapter 5

Conclusions and Future Work

5.1 Conclusion

The development of the proposed system represents a concrete implementation of a process-oriented digital platform that replaces manual, spreadsheet-based workflows with a structured and automated solution. The objective of creating a scalable and user-friendly application that could support the company's daily operations has been fully achieved through the combination of modern web technologies and a workflow automation engine.

The use of a Progressive Web Application has proven to be an effective choice, providing accessibility across devices and enabling rapid deployment without the need for native applications. The integration of Camunda 8 allowed the formalization of all operational processes through BPMN models, ensuring consistency and traceability of every step within the production chain. The chosen architecture, which uses a Backend-for-Frontend layer, API Gateway, and containerized microservices, has proven flexible and easy to maintain. Each module can evolve on its own, and debugging, versioning, and deployment are simpler. Moreover, the adoption of Kubernetes for deployment made it possible to define a clear, modular, and reproducible infrastructure, guaranteeing scalability and resilience under different operational loads. Together, these features create a strong digital ecosystem that meets industry standards.

Beyond the technical achievements, the project also demonstrated the importance of collaboration between software engineering and domain experts. The iterative validation of the application with company stakeholders ensured that the implemented workflows reflected actual business needs, reducing ambiguity and improving adoption by end users. This cooperation also highlighted the value of visual

process modeling as a communication tool between developers and non-technical participants.

In conclusion, this thesis has shown that the integration of business process management with modern web technologies can provide an efficient, maintainable, and user-centred solution for industrial digitalization. The results confirm that combining process automation, modular web architecture, and cloud-native deployment can effectively enhance both operational efficiency and long-term sustainability in a real industrial context.

5.2 Future Work

The current implementation represents a complete and functional system for workflow management and process monitoring. However, several possible directions have been identified to extend its capabilities and improve its performance in future developments.

Performance and comparative testing A first line of improvement concerns the evaluation of the system's performance under realistic operational conditions. Future work should include structured load tests to measure the application's behaviour with multiple concurrent users and under different network constraints. It would also be relevant to perform a comparative study between the developed Progressive Web Application and equivalent native solutions, in order to evaluate not only development and maintenance costs, but also runtime efficiency, and scalability in long-term usage scenarios.

Extended monitoring and analytical panels Another improvement could be adding more visualization and control panels. While the current version focuses on operational workflows, future iterations could include more detailed dashboards to support management and historical analysis. This could include showing multi-year trends, historical reports, or combined indicators for production, health monitoring, and veterinary activities. These additions would make the system useful not just for daily operations but also as a tool to support decision-making.

iChain platform integration Among the potential future improvements, the company has shown interest in evaluating the adoption of **iChain**, a platform developed by Wiseside for supply-chain management and traceability in the agri-food sector. The system is designed to provide real-time visibility over the entire production process, ensuring transparency, control, and compliance with industry standards. Future work may involve studying how iChain could be integrated into the existing architecture to extend traceability beyond the current internal

workflows. This would include analyzing how production events and process data collected by the PWA could be synchronized with the iChain platform, and assessing the technical implications in terms of interoperability, data exchange, and system overhead. Such integration could enhance the company's ability to certify and document every stage of the production cycle in a secure and auditable manner.

Evaluation of alternative workflow engines Camunda 8 has been a strong and reliable tool, but it comes with ongoing maintenance and licensing costs. A possible future activity is to assess alternative open-source solutions, such as Camunda 7, Flowable, or JBPM, in order to identify a platform that provides similar functionality with lower operational costs. This analysis should consider compatibility with BPMN standards, migration complexity, API integration, and community support.

Bibliography

[1] *Martini Alimentare — Company Overview*. Accessed November 2025. 2025. URL: https://www.martinialimentare.com/en/company/ (cit. on p. 2).

[2] *Martini Alimentare — Supply Chain*. Accessed November 2025. 2025. URL: https://www.martinialimentare.com/en/supply-chain/ (cit. on p. 2).

[3] *Progressive Web Apps — Tutorials — MDN Web Docs*. Accessed November 2025. 2024. URL: https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps/Tutorials/js13kGames/App_structure (cit. on pp. 7–9).

[4] *Progressive Web Apps with React — Ionic Documentation*. Accessed November 2025. 2025. URL: https://ionicframework.com/docs/react/pwa (cit. on pp. 7, 8).

[5] *Service Worker API — MDN Web Docs*. Accessed November 2025. 2024. URL: https://developer.mozilla.org/en-US/docs/Web/API/Service_Worker_API (cit. on pp. 8, 9).

[6] *Vite Plugin PWA — Guide*. Accessed November 2025. 2025. URL: https://vite-pwa-org.netlify.app/guide/#service-worker (cit. on pp. 8, 9).

[7] *Turn Your React Vite App into a PWA*. Accessed November 2025. 2025. URL: https://dev.to/bhendi/turn-your-react-vite-app-into-a-pwa-3lpg (cit. on p. 9).

[8] *Notifications API — MDN Web Docs*. Accessed November 2025. 2024. URL: https://developer.mozilla.org/en-US/docs/Web/API/Notifications_API (cit. on p. 9).

[9] *Push Notifications on the Open Web — Google Developers*. Accessed November 2025. 2024. URL: https://developer.chrome.com/docs/web-platform/push-notifications (cit. on p. 9).

[10] *Requesting App Permissions — Android Developers*. Accessed November 2025. 2024. URL: https://developer.android.com/training/permissions/requesting (cit. on pp. 9, 10).

[11] *User Notifications — Apple Developer Documentation*. Accessed November 2025. 2024. URL: https://developer.apple.com/documentation/usernotifications (cit. on pp. 9, 10).

[12] Yang Cui, Fei Zhao, and Lei Tang. «Exploring User's Experience of Push Notifications: a Grounded Theory Approach». In: *Proceedings of the International Conference on Human–Computer Interaction* (2023) (cit. on p. 10).

[13] Firebase Documentation. *Firebase Cloud Messaging*. Accessed: 2025-11-03. 2025. URL: https://firebase.google.com/docs/cloud-messaging (cit. on pp. 10, 48, 49).

[14] *React – A JavaScript library for building user interfaces*. Accessed November 2025. 2024. URL: https://legacy.reactjs.org/ (cit. on p. 11).

[15] *React – Official Documentation*. Accessed November 2025. 2024. URL: https://react.dev/ (cit. on p. 12).

[16] *TypeScript Documentation*. Accessed November 2025. 2024. URL: https://www.typescriptlang.org/ (cit. on p. 12).

[17] *TypeScript Introduction — W3Schools*. Accessed November 2025. 2024. URL: https://www.w3schools.com/typescript/typescript_intro.php (cit. on p. 12).

[18] *Using TypeScript with React — React Documentation*. Accessed November 2025. 2024. URL: https://react.dev/learn/typescript (cit. on p. 12).

[19] Vite Documentation. *Vite: Next Generation Frontend Tooling*. Accessed: 2025-11-03. 2025. URL: https://vite.dev/ (cit. on p. 12).

[20] Redux Documentation. *Getting Started with Redux Toolkit*. Accessed: 2025-11-03. 2025. URL: https://redux-toolkit.js.org/introduction/getting-started (cit. on pp. 13, 14).

[21] Redux Documentation. *Redux Essentials Tutorial*. Accessed: 2025-11-03. 2025. URL: https://redux.js.org/tutorials/essentials/part-1-overview-concepts (cit. on pp. 14, 36).

[22] Redux Documentation. *Why RTK is Redux Today*. Accessed: 2025-11-03. 2025. URL: https://redux.js.org/introduction/why-rtk-is-redux-today (cit. on p. 14).

[23] Redux Documentation. *RTK Query Overview*. Accessed: 2025-11-03. 2025. URL: https://redux-toolkit.js.org/rtk-query/overview (cit. on p. 15).

[24] Redux Documentation. *RTK Query Usage - Queries*. Accessed: 2025-11-03. 2025. URL: https://redux-toolkit.js.org/rtk-query/usage/queries (cit. on p. 15).

[25] Redux Documentation. *RTK Query Usage - Mutations*. Accessed: 2025-11-03. 2025. URL: https://redux-toolkit.js.org/rtk-query/usage/mutations (cit. on p. 16).

[26] Mozilla Developer Network. *CSS: Cascading Style Sheets*. Accessed: 2025-11-03. 2025. URL: https://developer.mozilla.org/en-US/docs/Web/CSS (cit. on p. 17).

[27] Mozilla Developer Network. *CSS Cascade*. Accessed: 2025-11-03. 2025. URL: https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_cascade (cit. on p. 17).

[28] Tailwind CSS Documentation. *Styling with Utility Classes*. Accessed: 2025-11-03. 2025. URL: https://tailwindcss.com/docs/styling-with-utility-classes (cit. on p. 17).

[29] Chakra UI Documentation. *Chakra UI: A simple, modular and accessible component library for React applications*. Accessed: 2025-11-03. 2025. URL: https://chakra-ui.com/ (cit. on p. 18).

[30] Microsoft Documentation. *Backends for Frontends*. Accessed: 2025-11-03. 2025. URL: https://learn.microsoft.com/en-us/azure/architecture/patterns/backends-for-frontends (cit. on p. 18).

[31] *What is Workflow Management?* Accessed November 2025. 2024. URL: https://www.atlassian.com/agile/project-management/workflow-management (cit. on p. 19).

[32] *Camunda Platform — Overview and Architecture*. Accessed November 2025. 2025. URL: https://docs.camunda.io/docs/ (cit. on p. 19).

[33] *Business Process Model and Notation (BPMN) Specification*. Accessed November 2025. 2023. URL: https://www.omg.org/spec/BPMN (cit. on p. 19).

[34] *Camunda Platform — Overview*. Accessed November 2025. 2025. URL: https://docs.camunda.io/docs/ (cit. on p. 19).

[35] *Camunda Platform — GitHub Repository*. Accessed November 2025. 2025. URL: https://github.com/camunda/camunda-platform (cit. on p. 19).

[36] Camunda Documentation. *Introduction to Tasklist*. Accessed: 2025-11-03. 2025. URL: https://docs.camunda.io/docs/components/tasklist/introduction-to-tasklist/ (cit. on p. 20).

[37] Camunda Documentation. *Introduction to Operate*. Accessed: 2025-11-03. 2025. URL: https://docs.camunda.io/docs/components/operate/operate-introduction/ (cit. on p. 20).

[38] Camunda Documentation. *Introduction to Operate*. Accessed: 2025-11-03. 2025. URL: https://docs.camunda.io/docs/components/modeler/about-modeler/ (cit. on p. 20).

[39] IBM Documentation. *Authentication vs. Authorization: What's the difference?* Accessed: 2025-11-03. 2025. URL: https://www.ibm.com/think/topics/authentication-vs-authorization (cit. on p. 20).

[40] NIST Documentation. *Identity and Access Management (IAM).* Accessed: 2025-11-03. 2025. URL: https://www.nist.gov/identity-access-management (cit. on p. 20).

[41] Auth0 Documentation. *Authentication and Authorization.* Accessed: 2025-11-03. 2025. URL: https://auth0.com/docs/get-started/identity-fundamentals/authentication-and-authorization (cit. on p. 20).

[42] Keycloak Documentation. *Keycloak: Open Source Identity and Access Management.* Accessed: 2025-11-03. 2025. URL: https://www.keycloak.org/ (cit. on p. 21).

[43] IBM. *What is Containerization?* Accessed: November 2025. 2024. URL: https://www.ibm.com/think/topics/containerization (cit. on p. 21).

[44] IBM. *What is Container Orchestration?* Accessed: November 2025. 2024. URL: https://www.ibm.com/think/topics/container-orchestration (cit. on pp. 21, 22).

[45] Docker Inc. *What is a Container?* Accessed: November 2025. 2024. URL: https://www.docker.com/resources/what-container/ (cit. on p. 22).

[46] IBM. *The Benefits of Containerization and What It Means for You.* Accessed: November 2025. 2024. URL: https://www.ibm.com/think/insights/the-benefits-of-containerization-and-what-it-means-for-you (cit. on p. 22).

[47] X. Zhang, Q. Zhao, and S. Li. *A Survey on Kubernetes: Architecture, Challenges, and Future Directions.* Accessed: November 2025. 2023. URL: https://arxiv.org/abs/2303.04080 (cit. on p. 22).