THOMAS MORE

UI Builder

Realisation document

Oleksii Pidnebesnyi
Student Bachelor Applied Computer Science

2024 - 2025

Table of Contents

1. INTRODUCTION 5

1.1. Personal Role and Responsibilities 5

1.2. Project Background and Objective 5

1.3. Document Structure 6

2. ANALYSIS 7

2.1. Project Analysis 7

2.2. Possible Approaches 8

2.2.1. Manual Page Creation 8

2.2.2. Dynamic UI Configuration Approach 9

2.2.3. Comparison of Approaches 10

2.2.4. Summary and Chosen Direction 10

2.3. Limitations 11

2.4. Tools and Technologies 12

2.4.1. UI/UX and Diagram Design Tools 12

2.4.2. Programming Language Choice 13

2.4.3. Frontend Framework 13

2.4.4. UI Framework 14

2.4.5. Forms and Validation 14

2.4.6. API Integration and Data Handling. 15

2.4.7. State and Context Management 16

2.4.8. Shared Component Library 16

2.4.9. Backend and Infrastructure (brief overview) 16

2.5. Conclusion of Analysis 17

3. ARCHITECTURE OVERVIEW 18

3.1. System Architecture 18

3.2. Folder Structure and Code Organization 18

3.2.1. High-Level Project Layout 19

3.2.2. Pages Structure 19

3.3. Client–Server Communication 20

3.4. Authentication and Authorization 20

3.5. Shared Libraries and Component Reuse 21

3.6. Deployment Environment & CI/CD 21

3.7. Architecture – Summary 21

4. UI BUILDER: LAYERED ARCHITECTURE 22

4.1. Configuration Layer – the authoring studio 22

4.1.1. Data-Model Overview 23

4.1.2. Field Metadata 24

4.1.3. View 24

4.1.4. Relation-Item Definitions 24

4.1.5. Actions 24

4.1.6. Conclusion 24
4.2. Builder Engine Layer 25
4.2.1. Lifecycle of a dynamic page 25
4.2.2. Anatomy of the Generated Screens 26
4.2.3. How a Page Behaves at Runtime 26
4.2.4. Proof-of-Concept Outcomes 26
4.2.5. Summary 26
4.3. Integration Layer 27
4.3.1. Two Families of Calls 27
4.3.2. How a requests are built 28
4.3.3. Create / Update Flow 28
4.3.4. Environment Awareness 29
4.3.5. Conclusion 29
5. UI BUILDER (CONFIGURATION LAYER) 29
5.1. Managing Modules 29
5.2. Module Configuration 30
5.3. Page Builder Form 30
5.4. Retrieving Metadata 31
5.5. Page Configuration 32
5.6. Fields 32
5.7. Views 33
5.7.1. The View Builder 33
5.7.2. Why Multiple Views Matter 34
5.8. Relation Items 34
5.9. Actions 35
5.10. Conclusion 35
6. BUILDER ENGINE - RUNTIME LAYER OF DYNAMIC PAGES 36
6.1. Plug-and-Play Integration 36
6.2. Renderer Tree 37
6.3. Auto-Expanding Navigation 37
6.4. Pages Overview 37
6.5. Views Overview 38
6.5.1. Table View 39
6.5.2. Kanban View 39
6.5.3. Item-Fields Modal / Off-canvas 40
6.6. Rendering Page Actions 40
6.7. Search Page (Read Many) 41
6.8. Create Item 42
6.9. Item-Details Page (Manage one) 43
6.9.1. Updating the Record 43
6.9.2. Browsing relations and embedded views 44
6.10. Conclusion 45
7. DATA FLOW & INTEGRATION LAYER 46

7.1. Where the Endpoint Comes From 46
7.2. A Small, Opinionated Client 46
7.3. React-Query Hooks 47
7.4. From View to SearchRequestDTO 47
7.4.1. Algorithm Flow 47
7.4.2. Single vs. Multi-Row Search 47
7.5. Create / Update: the *dynamic* endpoint 48
7.6. Data Flow 48
7.7. Key touch-points in code 49
7.8. Integration-Layer Recap 49
8. CONCLUSION 50
8.1. Recap of the Project 50
8.2. Key Learnings and Outcomes 51
8.2.1. Major achievements 51
8.2.2. Against the project plan 51
8.2.3. Technical & Professional Skills Gained 52
8.3. Reflection on Challenges and Solutions 52
8.4. Recommendations and Future Work 53
8.5. Demo 53
8.6. Closing remarks 53
REFERENCE LIST 55
ATTACHEMENTS 56

1. Introduction

This realisation document explains **how** the ICT Manager module of the Fenics ERP platform was rebuilt as a dynamic, configuration-driven web application during my 2025 internship at **Imas NV**. It elaborates on the objectives, milestones and deliverables that were defined in the Internship Project Planning *(Figure 20: Internship Project Plan*) and traces each phase of work from analysis to deployment.

Company context – Van Genechten Packaging (VGP) is a leading European folding-carton producer with operations in multiple countries and a strong focus on sustainable packaging. Its IT subsidiary, **Imas NV**, sustains that footprint with roughly 30 specialists in Business Intelligence, DevOps, Cyber-Security, IoT and software engineering. Imas runs two on-premises data centres and a four-stage deployment pipeline: Research and Development (RAD), System Integration Test (SIT), User Acceptance Test (UAT) and Production (PRD), ensuring both reliability and controlled releases. Within that ecosystem, Fenics is a long-standing ERP suite now being migrated from a Java-Swing desktop to a modular React/Quarkus stack.

1.1. Personal Role and Responsibilities

During the internship I acted as a **frontend engineer** and technical initiator for a more scalable rebuilding strategy. My key responsibilities included:

- **Designing and implementing** reusable React components and Syncfusion-based pages.
- **Analysing** the legacy Java client to catalogue patterns and pain points.
- **Proposing** a metadata-driven UI Builder to reduce repetitive work.
- **Co-ordinating** closely with backend engineers on contract design and integration.
- **Splitting** tasks into sprint-sized user stories and steering code reviews.
- **Participating** in iterative testing and continuous improvement loops.

Although not in a formal leadership role, I drove key architectural choices that shaped the module's direction.

1.2. Project Background and Objective

The legacy system consists of hundreds of interconnected pages, many of which share similar structures and logic. Rebuilding these pages one by one - each with its own dedicated API and interface - quickly proved to be an inefficient and time-consuming approach. As a result, the project goal shifted toward creating a more dynamic and configurable system for generating pages, aiming to streamline development and improve long-term maintainability.

The ICT Manager module, which manages data such as hardware, software, users, and responsibilities, was chosen as the starting point. A closer look revealed that most of its pages follow repeating patterns—such as search views, item management pages (create/edit), and relations between tables (e.g. one-to-many or many-to-many). These patterns made it an ideal candidate for testing a system that could generate interfaces and logic from configuration rather than hardcoding each screen individually.

The ICT Manager module demonstrates how this dynamic approach could later be extended to other modules within the Fenics ERP system. This shift represents an important step toward aligning the company's IT infrastructure with modern development practices.

1.3. Document Structure

This realisation document is divided into eight coherent chapters that guide you from context to conclusions:

Chapter	Focus	Purpose
1 Introduction	Personal role, project background and goals	Explains *why* the assignment matters and the author's responsibilities.
2 Analysis	Legacy-system study, approach comparison, limitations, tool selection	Provides the fact base and reasoning that shaped later design decisions.
3 Architecture Overview	High-level system & code organisation	Shows how the solution fits into the company's micro-frontend strategy.
4 UI Builder: Layered Architecture	Config, Engine and Integration layers	Details the inner workings of the dynamic UI platform.
5 UI Builder (Config Layer)	Authoring studio for pages, fields, views, relations and actions	Demonstrates how non-developers configure new screens.
6 Builder Engine – Runtime Layer	Route injection, rendering flow, view catalogue	Explains how metadata is turned into live React pages at runtime.
7 Data Flow & Integration Layer	DTO builders, REST contract, environment handling	Clarifies how the UI Builder communicates with any compliant backend.
8 Conclusion	Outcomes, learnings, reflection, future work	Evaluates results against objectives and sketches next steps.

Table 1: Document structure overview

Together, these chapters let both technical and business stakeholders trace the project's logic end-to-end—from the initial problem statement through architectural choices to measured results.

2. Analysis

The purpose of this chapter is to explain the reasoning behind the dynamic UI-Builder solution chosen for the ICT Manager migration. It documents what was learned from the legacy application, evaluates alternative strategies, and justifies the tools adopted in the realisation phase.

2.1. Project Analysis

The company maintains its own enterprise resource planning system called **Fenics**, which has been developed and expanded over the years to meet various internal business needs. One of the modules in this system is the **ICT Manager**, which handles key IT-related data such as users, devices, software, network access, responsibilities, and internal ICT workflows.

To stay in line with modern development standards and improve long-term maintainability, the company made a strategic decision to migrate Fenics from a legacy desktop environment to a modern web-based platform. This transition is intended to make the system more accessible, scalable, and easier to maintain across different departments.

The old ICT Manager system was built using **Java Swing**, a desktop-oriented UI framework. It includes hundreds of tightly coupled screens—each representing specific tables or functionalities. While the system was robust and mature, much of the interface and logic was hardcoded per screen. For every new table or process, code had to be written manually, even in cases where the structure or behavior was very similar. Although reusable components existed, they were not designed for high configurability, which made it difficult to generalize common functionality.

This lack of abstraction resulted in:

- High development and maintenance effort,
- Difficulty introducing changes across the system consistently,
- Limited scalability for adding new features or modules.

During the initial analysis of the ICT Manager module, it became clear that most pages followed a predictable structure. Each object in the system was typically represented by a database table and included:

- A **search page** with configurable filters;
- A **table view** displaying relevant items;
- A **create/edit form** for managing single entries;
- A **relation view** for connecting data across tables (e.g., assigning software to a user or linking a device to a location).

(see Figure 21: Repeating page patterns in the ICT Manager module*)*

These recurring patterns highlighted the potential for a more efficient approach. Although such patterns were implemented manually in the old system, they were not abstracted or unified in a way that allowed fast reuse or page generation.
This realization laid the groundwork for considering a new approach to interface building—one based on reusability and configuration—described in the following sections.

2.2. Possible Approaches

Based on the analysis of the legacy system and its recurring structures, two main approaches were considered for rebuilding the ICT Manager module. Both aimed to achieve the same end result—bringing the system to the web—but differed significantly in how development effort, maintainability, and scalability would be handled.

The first option focused on manually recreating each page and its functionality, following the structure of the original Java-based implementation. The second option explored a more dynamic solution based on configuration, aiming to reduce repetition and centralize logic for easier reuse.

The following sections compare these two approaches, highlighting their strengths, limitations, and the rationale behind the final choice.

2.2.1. Manual Page Creation

The first possible approach for rebuilding the ICT Manager module involves recreating each screen manually, closely following the structure and workflow of the original Java Swing implementation. In this model, each entity (e.g., applications, devices, software, users) is treated as a separate case with its own user interface, data model, and API integration.

Even though this approach may leverage reusable UI components—such as tables, input fields, and layout containers—it still requires dedicated configuration and logic for each new object. Each page needs to be developed individually, which includes defining how data is fetched, how forms are validated, how actions are triggered, and how relationships are managed.

Key elements of this approach include:

- **Creating reusable components** for similar page types (e.g., table views, search bars, form layouts), while still writing a separate instance of those components per object.
- **Developing dedicated API endpoints** for each object or dataset, including standard CRUD operations (create, read, update, delete) and additional business logic specific to that object.
- **Manually defining labels, field names, and layouts** in the frontend for each page, often duplicating patterns across multiple modules.
- **Managing routing and navigation** for every new view, requiring updates to the frontend routing logic for each new entity.
- **Handling relation logic separately** for each table connection (e.g., mapping a user to software or devices), which increases code complexity.
- **Testing and validation** for every individual object/page, with limited opportunity for reuse of testing logic or data validation rules.

Although this approach offers flexibility and complete control over how each page behaves, it also results in high development overhead, increased maintenance burden, and a slower onboarding process for new developers. Furthermore, introducing a change to a commonly used component or interaction pattern often requires updates across many manually-built pages.

In systems with a small number of objects or modules, this method can work effectively. However, in a large-scale ERP system like Fenics, with potentially hundreds of similar pages and thousands of fields, this approach becomes unsustainable. As the system grows, development becomes slower and more error-prone, and consistency across pages becomes harder to maintain.

2.2.2. Dynamic UI Configuration Approach

An alternative to manual page development is the creation of a **Dynamic UI Configuration System**, often referred to as a **UI Builder**. Rather than writing a custom interface and logic for each object in the system, this approach treats each page as a configurable unit—generated from metadata that defines how data should be displayed, edited, and related.

The central idea is simple: instead of building a new page for every table, the system takes a snapshot of that table's structure and allows an **administrator** to define how it should look and behave.

One key benefit of this approach is the ability to reuse logic. Actions like "assign user," "generate QR code," or "archive item" can be defined once and attached to multiple objects without duplicating code. Similarly, relations between tables—like linking software to users or devices to locations—can be described in configuration, automatically generating the necessary UI components.

As a result, new objects can be integrated into the system much faster. For example, adding support for a new table like network equipment would not require new routes, forms, or components to be coded manually. Instead, the object could be created through the UI Builder, with its views, fields, and actions defined in a matter of minutes.

This approach reduces development time, simplifies maintenance, and creates a more unified user experience. While it requires an initial investment in building the configuration system, it becomes increasingly valuable as the application grows—especially in a system like Fenics, where many modules share similar patterns and functionality.

2.2.3. Comparison of Approaches

After analyzing both the manual page creation method and the dynamic UI configuration approach, it becomes clear that each comes with its own strengths, trade-offs, and ideal use cases. While the manual method offers fine-grained control over every detail of a page, it comes at the cost of scalability and maintenance effort. On the other hand, the dynamic approach requires more initial setup and abstraction but significantly reduces repetitive work in the long term and makes the system easier to evolve.

The table below summarizes the key differences between the two strategies:

Aspect	Manual Page Creation	Dynamic UI Configuration
Development Speed	Slow; each page built individually	Fast after setup; pages generated from config
Scalability	Limited; effort grows linearly with number of pages	High; new pages require little to no development
Maintainability	Difficult; changes must be repeated across pages	Easier; updates apply system-wide via config
Reusability	Partial; shared components may be reused manually	High; actions and components reused automatically
Flexibility in UI Design	Full control per page	Flexible within defined patterns (views, layouts, widgets)
Complex Logic Integration	Easy to customize per object	Requires abstraction and generalization of logic
Initial Setup Effort	Low initial setup; starts fast	High initial setup; tool development and data modeling needed
Admin Involvement	None; all changes require developer work	High; non-developers can define or adjust objects via UI Builder
Consistency Across Pages	Can vary if not strictly enforced	High; structure and styling are centralized
Best For	Small projects or unique, one-off views	Large systems with many similar modules

Table 2: Comparison Manual Page Creation vs Dynamic UI Configuration

2.2.4. Summary and Chosen Direction

After comparing both approaches, the decision was made to proceed with the dynamic UI configuration approach. While it required more planning and technical setup in the beginning, it offered a clear advantage in terms of scalability, consistency, and speed of development.

Given the scope of the ICT Manager module and the time constraints of the internship period, it became evident that manually recreating each page and endpoint would be inefficient and unfeasible. The dynamic approach provided a realistic path forward by reducing repetitive development and allowing new pages to be configured instead of built from scratch.

2.3. Limitations

While exploring and planning the transition to a dynamic, configuration-driven system, several technical and structural limitations of the existing application—and the broader system environment—had to be taken into account. These challenges highlighted the complexity of fully automating interface generation and informed many design decisions made during development.

Below is an overview of key bottlenecks identified during the analysis phase:

1. Inconsistent and Missing Labels

- Field labels were not stored centrally in the old system.
- In many cases, labels were hardcoded in the Java Swing application.
- This meant that for every table introduced into the new system, all field names and their human-readable labels had to be manually defined in the configuration.

2. Contextual Page Behavior

- Pages in the legacy system sometimes behaved differently depending on how they were opened.
- For example, a page used as a **root search view** might show full records, while the same table embedded in a **relation context** (e.g., user → devices) only displayed partial data or limited actions.
- This required the configuration system to account for different *view modes* and permissions per context.

3. Custom Actions and Behaviors

- Many screens included **custom buttons or workflows** that triggered object-specific logic (e.g., autogenerate field).
- These actions were often deeply embedded in the page logic and not standardized.
- Creating a reusable action system required abstracting these behaviors into configurable patterns, while still allowing for page-specific customization.

4. Evolving Database Schema

- Since the database is still maintained and extended, field names, types, and relations can change during development.
- This raised the challenge of keeping the configuration **in sync with schema changes**, and required implementing update mechanisms or consistency checks.

5. Field Size and UI Layout Variations

- The original system used varying component sizes and alignments for different fields (e.g., longer text fields, compact checkboxes, wide dropdowns).
- The new system had to support **customizable layout configurations**, otherwise the UI would feel inconsistent or hard to use.

6. Filters Not Bound to a Single Field

- Some search filters were not directly tied to a single database field.
- For example, filtering users by status might involve multiple conditions or mapped values.
- The filter system needed to allow **custom logic and mappings**, instead of assuming a one-to-one field binding.

7. Non-standard Field Types and Behaviors

- Some fields required **special formatting or business logic**, such as calculated values, tags, or conditional dropdowns.

- These exceptions had to be considered in the component design, especially in edit forms and table displays.

8. Permission and Visibility Rules

- In some cases, access to fields or actions depended on the user's role or object state (e.g., read-only for archived entries).
- A future-proof system needed to allow role-based visibility rules at the field and action level.

9. Performance Considerations

- Generating large views dynamically with many relations and filters could impact performance.
- The system needed to optimize queries and rendering, especially when handling deeply nested relations or thousands of rows.

These challenges shaped the development of the dynamic UI system and emphasized the need for a flexible, extensible configuration model. While full automation was not possible in every case, identifying these limitations early helped ensure that the system could support real-world use cases and adapt to legacy complexities.

2.4. Tools and Technologies

The technology stack for the ICT Manager migration was constrained by Van Genechten Packaging's existing infrastructure yet still allowed room to select the most productive, industry-proven options for a form-heavy, component-based front end. This section records every significant tool decision and cites publicly available comparisons that support the choice.

2.4.1. UI/UX and Diagram Design Tools

Design played an important role in structuring the interface and aligning the work with both frontend and backend teams. Throughout the project, I used Penpot and Figma for wireframing and brainstorming concepts, and diagrams.net for creating system and database diagrams.

Wireframing and Brainstorming Tools

Penpot became the primary wire-framing application because it is open-source, self-hostable and offers real-time collaboration similar to Figma's FigJam while avoiding vendor lock-in. (Alves, 2024)

Figma was retained only for ad-hoc brainstorming sessions where its extensive community templates accelerated ideation. (Solomakha, 2024)

Tool	Open Source	Collaboration	Self-Hosting	Community Templates	Use Case
Penpot	Yes	Yes	Yes	Limited	Primary wireframing tool
Figma	No	Yes	No	Extensive	Brainstorming via FigJam only

Table 3: Comparison of wireframing tools

Penpot was selected as the primary wireframing tool because:

- It is user-friendly and easy to adopt
- It offers real-time collaboration
- It is open source and can be self-hosted
- The company had already deployed it internally

Diagram and ERD Tools

For ERDs and flowcharts the team selected diagrams.net (formerly Draw.io). Its browser and desktop modes provide offline work, while SVG exports integrate smoothly with the documentation pipeline. StarUML, though feature-rich, was ruled out because its single-user licence model conflicted with the internship budget and lacked real-time collaboration. (draw.io, 2024)

Tool	Open Source	Collaboration	Offline Mode	Complexity	Use Case
diagrams.net	Yes	Yes	Yes	Low	Used for ERD and flowcharts
StarUML	No	No	Yes	Medium	Not used in this project

Table 4: Comparison Diagram Tools

Diagrams.net was chosen because:

- It is simple and intuitive
- It allows real-time collaboration
- It works both online and offline
- It is well-suited for quick ERD creation and sharing with team members

2.4.2. Programming Language Choice

TypeScript was chosen over plain JavaScript to add static typing, safer refactoring and IDE-level autocompletion—capabilities that are essential in a codebase expected to outlive the internship. Industry comparisons note that TypeScript's compile-time checks reduce runtime defects and ease long-term maintenance. (Upadhyay, 2025)

Language	Type Safety	Scalability	Tooling Support
JavaScript	No	Moderate	Extensive
TypeScript	Yes	High	Strong

Table 5: Comparison Programming Language

Why TypeScript was chosen:

- Enables static typing and reduces runtime errors.
- Improves developer productivity with better tooling support.
- Provides better maintainability for large-scale applications.
- Fully compatible with React and modern build systems

2.4.3. Frontend Framework

React is already the corporate standard and remains the most widely adopted web framework according to the 2024 Stack Overflow survey and recent market analyses (Stack Overflow, 2024). Vue and Angular were explored but would either duplicate effort (learning curve) or conflict with existing component libraries

Framework	Community Support	Flexibility	Learning Curve	Final Choice
React	Very High	High	Moderate	Used
Vue	Moderate	High	Low	Not used
Angular	High	Medium	High	Used

Table 6: Comparison Frontend Frameworks

Why React was chosen:

- Standardized across the company
- Large ecosystem and strong community support
- Compatible with various third-party libraries
- Matches well with TypeScript and component-driven architecture

2.4.4. UI Framework

A two-layer strategy balances speed and richness:

- **Bootstrap** supplies the base grid and typography, familiar to all internal teams.
- **Syncfusion React UI** adds 65+ enterprise-grade widgets (tables, Gantt, Kanban) that would be time-prohibitive to build in-house. External benchmarks highlight its broader component catalogue compared with generic libraries such as Material-UI or React-Bootstrap. (Arunodi, 2024)

Framework	Styling Foundation	Advanced Components	Customizability	Final Use
Tailwind CSS	Utility-based	No	High	Not used
Bootstrap	Component-based	Basic	Moderate	Used
Material UI	Component-based	Moderate	Moderate	Not used
Syncfusion	Component-based	Extensive (Gantt, Kanban, etc.)	Moderate–High	Used

Table 7: Comparison UI Framework

Why Bootstrap was used:

- Provides a well-structured base for layout and components.
- Familiar across teams and easy to extend.
- Serves as a foundation for the company's internal UI standards.

Why Syncfusion was used:

- Offers a wide range of ready-to-use, enterprise-grade components.
- Enabled fast implementation of complex views like tables, Kanban boards, and Gantt charts.
- Reduces the need for developing and maintaining custom UI logic for advanced elements.

The combination allowed for rapid UI development while maintaining design consistency across the application.

2.4.5. Forms and Validation

Dynamic forms drive the UI-Builder, so performance and declarative validation were critical. The combination of React Hook Form (RHF) and Zod met both needs: RHF minimises re-renders in large forms, and Zod supplies TypeScript-native schema validation reusable across pages. Independent tests and community discussions consistently rate RHF faster and more ergonomic than Formik for large, dynamic forms. (Ihnatovich, 2025). Zod's schema-first approach likewise draws praise for reducing runtime errors. (Anshul, 2024)

After evaluating several options, the combination of React Hook Form and Zod was selected to handle form logic and validation.

Tool	Performance	Integration with React	Schema Validation	Final Use
Bare State Handling	Low	Manual	No	Not used
Formik	Moderate	Good	External libraries	Not used
React Hook Form	High	Excellent	Yes (Zod-compatible)	Used
Zod (Validation)	Lightweight	N/A	Yes	Used

Table 8: Comparison Form and Validation Tools

Why React Hook Form was chosen:

- Excellent performance, especially for large and dynamic forms.
- Simplifies integration with controlled and uncontrolled components.
- Provides clean separation of logic and UI.

Why Zod was chosen for validation:

- Allows schema-based validation outside of the component.
- Offers TypeScript-first design, which fits the project's strongly typed structure.
- Enables reuse of validation logic across views.

2.4.6. API Integration and Data Handling.

TanStack React Query abstracts server-state caching, background refetching and optimistic updates while remaining agnostic of the underlying fetch client. Comparative articles show it covers concerns—caching, invalidation, mutation state—that Axios alone does not address. (Patel, 2024)

Tool	Caching	Mutation Support	Boilerplate	Final Use
Axios	No	Manual	High	Not used
RTK Query	Yes	Yes	Moderate	Not used
React Query	Yes	Yes	Low	Used

Table 9: Comparison API integration tools

Why React Query was chosen:

- Lightweight and easy to integrate with existing React components.
- Built-in caching, invalidation, and loading state management.
- Minimal boilerplate compared to alternatives.

2.4.7. State and Context Management

State management needs in the project were limited, mostly involving global UI states and user session context. Given the relatively simple requirements, the built-in **React Context API** was selected.

Tool	Complexity	Boilerplate	External Dependency	Final Use
Redux Toolkit	High	High	Yes	Not used
Zustand	Low	Low	Yes	Not used
React Context API	Moderate	Low	No	Used

Table 10: State Mangement tools

Why React Context API was chosen:

- Sufficient for the scale of global state needed.
- Integrated into React with no extra dependencies.
- Easy to maintain and extend when required.

2.4.8. Shared Component Library

To guarantee a uniform look-and-feel across every Fenics web module, Imas NV curates **@imas/react-core**, a mono-repo published to the internal GitLab NPM registry. Each consuming application pins a semantic version (e.g., ^6.3.0), so breaking changes never reach production unexpectedly and can be promoted in a controlled release train.

The library includes:

- Reusable UI components (buttons, inputs, layout containers)
- Domain-specific components tailored to internal workflows
- Utility functions for common tasks
- Custom hooks for form handling, modals, and state
- Context providers (e.g., theme, permissions, layout)

Using this library allowed for faster development, reduced boilerplate, and ensured alignment with the company's design and behavior standards. It also simplified integration across projects by offering prebuilt components that follow consistent structure and logic.

2.4.9. Backend and Infrastructure (brief overview)

Although my role was focused on frontend development, integration with the backend and infrastructure was essential throughout the project. The application is built on a backend powered by **Java with Quarkus**, chosen for its high performance and suitability for microservice-based architectures.

The main technologies in the backend and infrastructure include:

- **Java Quarkus** – Lightweight Java framework used to build REST APIs consumed by the frontend.
- **PostgreSQL** – The primary relational database used across all modules.
- **Keycloak** – Identity and access management system used for user authentication and authorization.
- **GitLab (self-hosted)** – Used for version control, code review, CI/CD pipelines, and issue tracking.

This stack allows horizontal scaling, zero-downtime updates, and a clear separation of concerns between UI and data services.

2.5. Conclusion of Analysis

The analysis phase provided a comprehensive understanding of the legacy ICT Manager module and the challenges associated with migrating it to a modern web-based system. Through a detailed evaluation of existing patterns, system limitations, and project constraints, two potential implementation strategies were identified and compared. The decision to develop a dynamic UI configuration system was guided by both practical limitations—such as time and maintainability—and a long-term vision for scalability across the ERP platform.

In parallel, the choice of tools and technologies was either predefined by the existing company infrastructure or selected to meet the specific needs of a form-heavy, component-based frontend system. The selected stack, centered around React, TypeScript, React Hook Form, and Syncfusion, proved to be well-suited for the goals of this project.

Altogether, the analysis laid a clear foundation for the realization phase and justified the chosen direction both technically and strategically.

3. Architecture Overview

This chapter explains how the ICT Manager module fits into the wider Fenics web ecosystem and how its codebase is organised to maximise reuse, independence and continuous delivery. The narrative moves from high-level system topology (*System Architecture*) down to deployment pipelines *(Deployment Environment & CI/CD*) so you can trace every design choice from browser to Kubernetes cluster.

3.1. System Architecture

The Fenics ERP is moving from a monolithic, Java-Swing desktop to a modular web platform. Core tooling—build, authentication, and deployment—has already been standardised by the company's platform team, giving the ICT Manager module a stable foundation.

Within that framework I defined the internal structure of my own apps: code organisation, state & routing strategy, and the config-driven rendering model.

To meet Fenics' goals for scalability and independent releases, we adopted Webpack Module Federation:

(see *Figure 22: Fenics module federation overview*)

Role	Description
Shell	Hosts global layout & navigation, exposes top-level routes
Remote micro-frontends	Independently built apps (e.g. *UI Builder*, *ICT Manager*)—each ships its own remoteEntry.js and a single Page.tsx
Shared libraries	React, router, and utility packages are marked *singleton & shared* to prevent duplication

Table 11: Module Federation Architecture

At runtime the shell lazily imports each remote: localhost ports during development and a CDN URL in production. Because UI-Builder itself is a remote, its renderer can generate pages on-the-fly while the studio lets analysts edit JSON schemas without redeploying code. This pattern supports dozens of future modules without sacrificing a seamless user experience.

3.2. Folder Structure and Code Organization

To promote clarity, maintainability, and scalability, the codebase follows a layered, feature-oriented architecture. At the highest level, the repository separates static assets, application entry points, build configuration, and source code. Within the src/ directory, concerns are strictly partitioned into thematic layers - each with a well-defined responsibility - so that developers can quickly locate, reason about, and modify code without unintended side effects.

3.2.1. High-Level Project Layout

The repository follows a **layered-feature** approach that limits coupling and speeds up onboarding. *Figure 1: Src folder structure* illustrates the directory tree under src/ and *Table 12: High level folder structure.* Explains the purpose of each layer

Layer	Folder	Purpose (single responsibility)
Presentation	components/	Stateless UI primitives; no data fetching.
State	contexts/	React Context providers for tokens, theme, permissions.
Logic	hooks/	Declarative side-effects, fetch logic, complex UI flows.
Pages	pages/	Route entry points; each folder owns its own Page.tsx.
Data	service/	Fetch clients, DTO mappers, feature-agnostic business logic.
Domain	types/	TypeScript interfaces, Zod schemas, validation rules.
Utility	utils/	Generic helpers (date formatting, deep-clone, rule builders).

Table 12: High level folder structure.

```
src/
├─ components/   ← _Presentation Layer_
├─ contexts/     ← _State Layer_
├─ hooks/        ← _Logic Layer_
├─ pages/        ← _Pages Layer_
├─ service/      ← _Data Layer_
├─ types/        ← _Domain Layer_
├─ utils/        ← _Utility Layer_
├─ routes.tsx    ← Main application routing (React Router DOM)
└─ renderer.tsx  ← Builder-engine hydration and route renderer
```

Figure 1: Src folder structure

3.2.2. Pages Structure

Each page or feature under src/pages/ follows an identical, self-contained layout. This feature module pattern ensures that all code related to a given page (UI, state, logic, subpages) coexists in one directory:

components/
Groups UI elements unique to this page—grids for tables, off-canvas panels, modal dialogs, navigation bars, and any form or tab components.
hooks/
Contains specialized hooks that encapsulate data loading, mutation, or side effects specific to this feature
state/
Defines a React Context (and optional reducer) for feature-scoped state, such as form data or UI flags.
functions/
Includes pure helper that operate on domain data without React or side effects.
subpages/
Enables multi-step or nested routing within the same feature, each submodule mirroring the same folder layout and always culminating in a Page.tsx.

[page].tsx
Orchestrates rendering: it imports its own components, wraps children in the feature's state provider, invokes any hooks, and nests subpages routes via React Router.

3.3. Client–Server Communication

All traffic flows through a RESTful API surface exposed by Quarkus services. A thin wrapper around fetch automatically attaches JSON Web Tokens (JWTs), applies standard time-outs and funnels error objects to a central toast system. TanStack Query provides:

- Automatically injects the authentication token into request headers
- Centralises error handling and retry logic
- Applies consistent timeouts and response parsing

As a result, identical requests never reach the server twice and stale data is refreshed without user intervention.

3.4. Authentication and Authorization

Login is delegated to **Keycloak** via the OpenID Connect implicit flow (Figure 2: Authentication Flow Diagram).

After redirect login:

1. Keycloak issues a short-lived ID Token and longer-lived Refresh Token.
2. A KeycloakProvider persists these in memory and refreshes silently.
3. The shell queries the Identity micro-service for role–permission maps.
4. Routes are guarded by <PermissionGate> components from @imas/react-core; feature hooks perform granular checks (e.g., field-level read-only).

This model removes hard-coded ACL logic from pages and keeps access rules declarative.

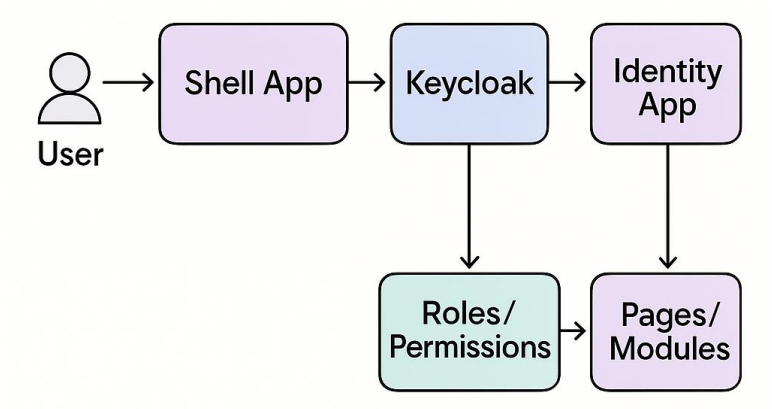

Figure 2: Authentication Flow Diagram

3.5. Shared Libraries and Component Reuse

@imas/react-core acts as the single source of truth for design tokens, atomic components and cross-cutting hooks. Publishing happens via semantic-versioned tags; every release runs visual-regression tests in Chromatic before promotion to latest. Centralising common assets delivers three concrete benefits:

- Consistency – colour palette, spacing and motion curves are enforced automatically.
- Productivity – new modules start with battle-tested primitives, avoiding boilerplate.
- Maintainability – bug fixes or accessibility upgrades propagate through the dependency graph without manual cherry-picks.

3.6. Deployment Environment & CI/CD

Deployments run through the company's custom CI/CD platform, which integrates with GitLab and other internal services. Four environments are used in sequence:

Stage	Purpose	Key gates
RAD	**Developer previews on isolated namespaces**	**ESLint, unit tests**
SIT	**Integration with live services**	**Contract tests, SCA scan**
UAT	**Business validation**	**Manual QC sign-off**
PRD	**Production**	**Automated blue-green switch, observability alerts**

Table 13: Environments

GitLab CI builds a Docker image, pushes it to the internal registry and executes Helm charts per environment. Rollbacks are one-click by redeploying the previous image tag; secrets are injected via Vault so no credentials ever enter the repository.

3.7. Architecture – Summary

The combination of module federation, a layered feature structure, declarative Query/Context state, and a hardened CI/CD pipeline equips Fenics for incremental rollout, independent team velocity and long-term maintainability—all prerequisites for the systemic modernisation described in Chapter 2.

4. UI Builder: Layered Architecture

This chapter explains how configuration files turn into live pages at runtime. To make configuration-driven development possible, the UI Builder is split into three distinct layers:

1. **Configuration Layer**
   A dedicated studio where business users define and preview page schemas—no code required.
2. **Builder Engine Layer**
   A runtime interpreter that takes those schemas, combines them with context (module, table, user role, environment), and instantiates the actual React components and routes.
3. **Integration Layer**
   A lightweight gateway that handles data fetching, environment-specific endpoints, and telemetry—keeping all external service calls out of the engine itself.

This clear separation lets non-developers craft or tweak screens on the fly while ensuring the runtime remains predictable, testable, and easy to maintain—and all API and security concerns stay neatly encapsulated in one place.

4.1. Configuration Layer – the authoring studio

The **Configuration Layer** is the authoring studio of the UI Builder. It stores every piece of information required to generate a fully working screen—modules, pages, fields, views, relations, and actions—in a database-backed schema. Business users work exclusively in this layer, using a web interface to design and instantly preview applications while writing **zero** code.

Purpose & Key Responsibility

- Turn business intent into a declarative model that the Builder Engine can interpret at runtime.
- Guarantee that every definition (label, rule, permission) lives in one place, eliminating code duplication.
- Provide live preview (via build engine) so authors see exactly how a change will look before it reaches production.
- Maintain full auditability & versioning for compliance and rollback.

4.1.1. Data-Model Overview

Figure 3: Configuration flow from creating to runtime shows the high-level flow: authors work in the Config Layer → save a **Dynamic App Setup** record → the Builder Engine consumes that record to render Nodes, Pages, Views and Actions inside the running ERP.

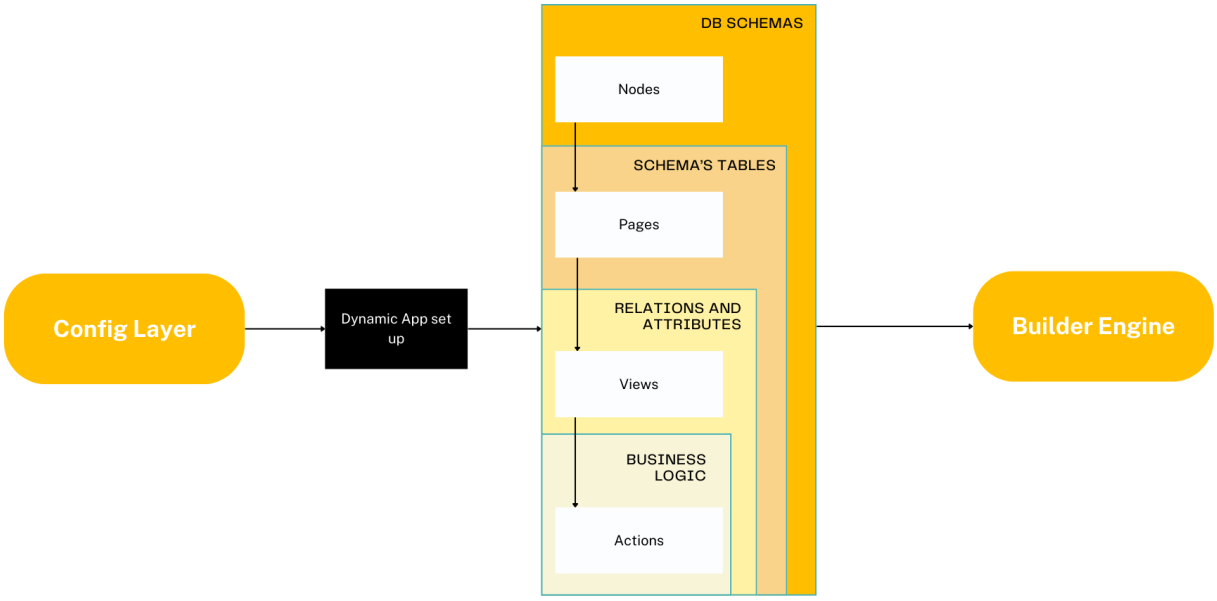

Figure 3: Configuration flow from creating to runtime

The entity relational model (*see Figure 23: ER diagram of the Configuration schema)* captures this hierarchy:

Entity	Key Fields	Why It Exists
Module	code, schema_name	Top-level container in the ERP navigation (Dynamic application)
Node	name, parent_id	Folders / sub-menus that group pages
Page	table_name, pk_field	One table-backed CRUD screen
Field	field, component, options	Visual and behavioural metadata for each column
View	type, code	Layout blueprint: Search / Create / Edit
RelationItem	referenced_table	One-to-many child collections
Action	action_code, field, referenced_table	UI buttons that trigger predefined logic

Table 14: Hierarchy of tables

4.1.2. Field Metadata

- **Display Settings** – component type, label, width, visibility.
- **Conditional Rules** – show / hide / disable based on another field's value.
- **Auto-Generation** – timestamp, UUID, or sequence injection.
- **Value Styling** – colour indicators for statuses (e.g., closed = red).
- **Validation** – required flags, regex patterns, min/max, cross-field checks.

These attributes are persisted per-field so the Builder Engine can render the proper widget and enforce validation uniformly across all views.

4.1.3. View

Every page stores multiple **View** records, each containing:

- **Type** (search | create | edit | nav)
- **Code** (table | kanban | item_form | modal ...)
- **Tabs – Section - Selected Fields** – which columns appear, how to display and in what order.
- **Criteria** – preset filters for quick search tabs.

4.1.4. Relation-Item Definitions

One-to-many relationships are expressed in RelationItem rows:

- **Display Label** – how the child collection appears in the UI.
- **Display Location** – navigation drawer, detail tab, or inline grid.
- **Child View** – which View is reused to render the set.

When the Page loads, the engine detects these definitions and automatically injects tabs or grids so end-users can browse, add, or delete child records without extra code.

4.1.5. Actions

Actions are lightweight descriptors that attach behaviour to buttons:

- **Navigate** – open item details or external URL.
- **Quick-Create** – spawn a pre-filled child record.
- **Future** – planned generic REST call with payload mapping.

Each record stores the **trigger**, **target** and optional **payload**, allowing new business flows to be rolled out by config only.

4.1.6. Conclusion

The Configuration Layer serves as the single source of truth for every UI element—from modules and pages down to fields, views, relations, and actions—while empowering business users to design rich screens without writing code. By centralizing these declarative definitions, we eliminate duplication, enforce consistency, and support full versioning and audit trails. This solid foundation enables the Builder Engine to reliably render pages at runtime and ensures that every change is traceable and reversible—fulfilling the promise of true configuration-driven development.

4.2. Builder Engine Layer

The Engine turns static metadata into interactive React pages and wires them to TanStack-Query data hooks. The Builder Engine is delivered as a self-contained micro-frontend. Once a host module toggles Dynamic Pages = ON, the shell downloads the engine's remoteEntry.js, mounts its exported routes, and the rest happens automatically.

4.2.1. Lifecycle of a dynamic page

1. Route match /ict/device/123
2. Fetch config for Page=DEVICE, plus its child Field, View, RelationItem, Action.
3. Normalise & cache with Zod; invalid configs fail fast with a toast.
4. Render tree – lookup component registry; lazy-load heavy widgets (Gantt, Kanban) via React-lazy.
5. Wire data – generate query keys (device-123) and mutations.
6. Handle actions – delegate to router or integration client.

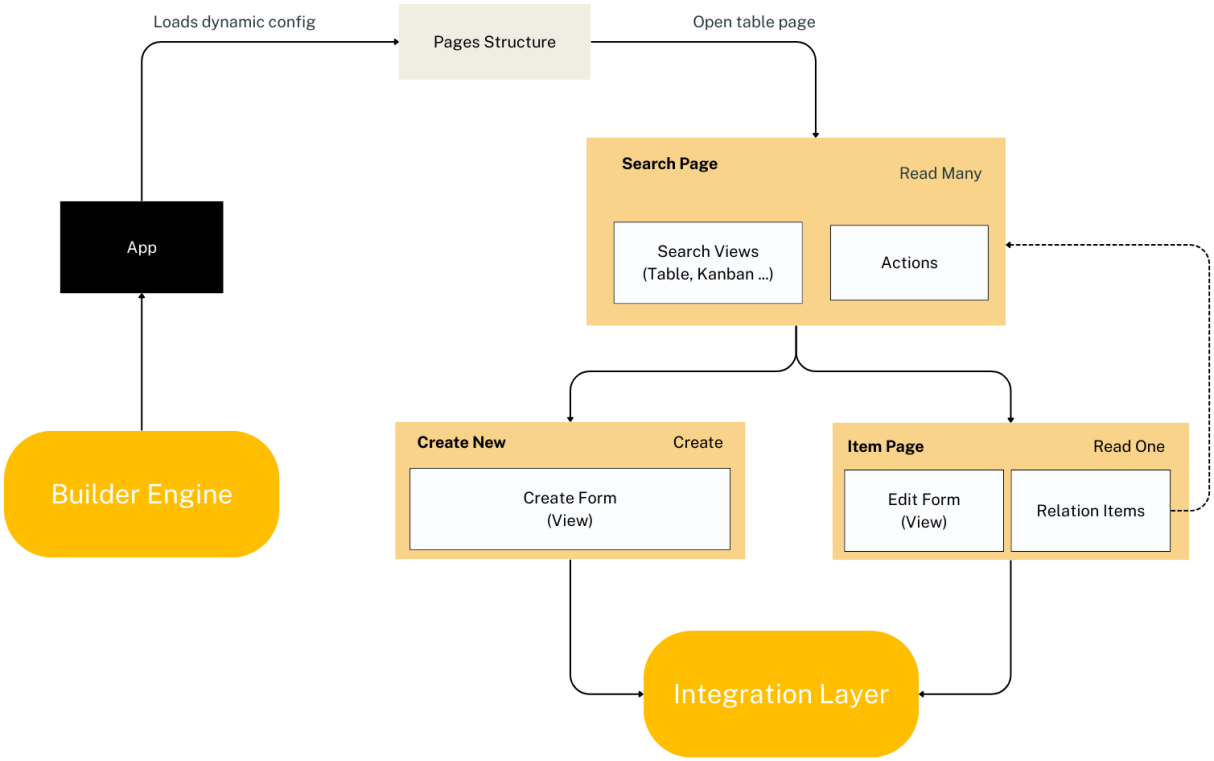

Figure 4: End-to-end page lifecycle

4.2.2. Anatomy of the Generated Screens

Page Type	Core Purpose	Auto-generated Elements
Search Page	Read-many; filtering & bulk actions	Search-view selector, result grid, “New” button
Create Page	Create-one	Managed form with validation; pre-fill via URL params
Item Page	Read/Update-one	Header with dynamic title, edit toggle, relation tabs

Table 15: Generated Screens Overview

All three share the same skeleton loaders, permission checks, and header scaffolding, so new tables gain a consistent UX by default.

4.2.3. How a Page Behaves at Runtime

When a user opens a dynamic table, three familiar screens come to life:

- **Search page**
  - Loads the list layout defined in the Config Layer.
  - If more than one layout exists, a tab bar lets users flip between them.
- **Create page**
  - Shows a blank form based on the chosen “create” view—or falls back to a simple field list.
  - Can arrive pre-filled if launched from a parent record (“quick-create”).
- **Item page**
  - Displays full details of a single record and, if allowed, an Edit button.
  - Relation tabs appear automatically so users can browse child items without extra clicks.

All three screens share the same loading skeletons, error panels, and “optimistic” save behaviour, so every table feels identical from the user’s point of view.

4.2.4. Proof-of-Concept Outcomes

- **Pages render from pure config** – The same JavaScript bundle serves any table as long as a schema exists.
- **Uniform UX** – Because grids, forms, headers, and modals are shared, visual drift is impossible.
- **Zero redeploy for text tweaks** – Change a label in the Config Layer, refresh the browser, and the new text appears.
- **Incremental rollout** – Legacy, hand-coded screens live side-by-side with dynamic ones; modules migrate one table at a time.

4.2.5. Summary

The Builder Engine converts static metadata into living React screens and wires them to live data via the Integration Layer. Packaged as a lightweight micro-frontend, it can be mounted inside any ERP module—instantly turning database tables into fully-featured CRUD pages without further deployments.

4.3. Integration Layer

Integration Layer purpose is to decouple the Engine from every module's private REST controllers.
The **Integration Layer** is the narrow bridge between the UI Builder world (schemas, views, actions) and each business module's REST API. Its job is simple in principle

Task	Why it matters
Read	Pull configuration and data rows the UI needs to show.
Translate	Turn view or action definitions into concrete SQL-like payloads.
Validate	Ensure requests obey page-level permissions and data types.
Write	Forward create, update, and delete calls to the module's own API.

Table 16: Integration layer job

Because the layer sits outside the page renderer, a module can adopt dynamic pages without touching its controllers; all it needs is the small, versioned *utils* endpoint set.

4.3.1. Two Families of Calls

The Integration Layer exposes two primary endpoints—one for read-only operations and one **for data-write** actions—which the Builder Engine invokes to populate screens and persist changes (see Figure 24). *Table 17: Calls explained* summarises their roles and payload contracts.

Call family	Endpoint	Used by	Typical payload
Read-only	utils/combined	Search & Item views	SearchRequestDTO (selected columns, joins, criteria)
Data-write	utils/dynamic	Create / Update / Delete actions	JSON object keyed by column names

Table 17: Calls explained

Read-only calls return an **array of { column_name: string; value: any }** pairs, enabling the Builder Engine to hydrate any field type dynamically without schema-specific code. Data-write calls accept a JSON object with column-name keys; the Integration Layer then:

1. **Injects audit fields** (e.g. created_by, updated_at).
2. **Validates** data types and user permissions.
3. **Forwards** the request to the module's REST controller.

Both endpoints return a standardised response envelope: an HTTP status code plus, on error, an error object containing a code, message and optional field-level details. This uniform contract ensures that filtering, mutations and error handling behave predictably across every dynamic page.
(see Figure 24: Sequence Diagram: Filter and Mutate requests)

4.3.2. How a requests are built

When a user opens a search page, the Builder Engine constructs and dispatches a SearchRequestDTO in four clear stages. This process guarantees that dynamic filters, nested relations and performance optimisations are applied consistently across every table.

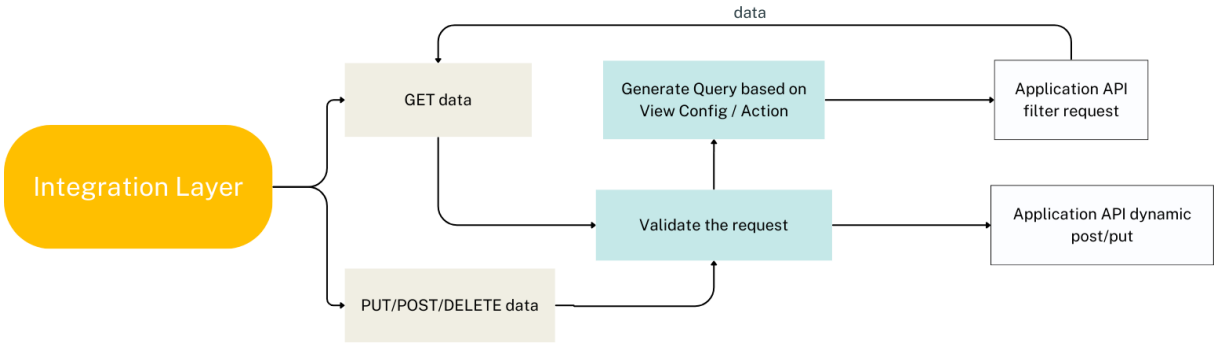

Figure 5: Search Request Generator

The engine applies these steps in sequence:

1. Inspect configuration – Traverse the selected-fields tree on the View record, including any nested RelationItem definitions, to determine which columns and joins are required.

2. Assemble DTO – A dedicated builder function allocates unique table aliases, formulates join clauses, and maps each field's filter criteria into the DTO structure.

3. Validate and optimise – Zod schemas validate types and reject malformed criteria; empty filters are removed and duplicate joins are merged to streamline the query.

4. Execute request – POST the final DTO to /utils/combined. The Integration Layer forwards it to the module's API, which returns a standard array of { column_name, value } pairs.

Because the same DTO-building logic is invoked for relation-item clicks and quick-create actions, parent-record keys are automatically injected and aliased correctly. By isolating alias allocation, validation and join deduplication, the system ensures both performance (minimal SQL complexity) and security (parameter-bound queries), regardless of how deeply nested the view definitions become.

4.3.3. Create / Update Flow

When a user submits a form on a Create or Edit page, the Integration Layer handles it in four steps:

1. **CreateItem** and **EditItem** screens collect form data.
2. A tiny helper trims blank fields and adds audit columns (created_by, updated_at).
3. The payload is sent to utils/dynamic with HTTP verb **POST** (insert) or **PUT** (update).
4. On success, the engine clears cached form state and refreshes the Search or Item page.

All write calls move through the same pipe, making it trivial to decorate them with global auth tokens, audit headers, or environment-specific base URLs.

4.3.4. Environment Awareness

The same utils endpoints support all four stages of the release cycle—RAD, SIT, UAT and PRD—by reading the appropriate base URL from the configuration stored in the Config Layer. The Builder Engine remains entirely agnostic: moving from test to production is as simple as selecting a different environment in the studio.

4.3.5. Conclusion

By exposing only two endpoints and reusing the same DTO builders and validation logic, the Integration Layer achieves loose coupling between the UI Builder and each module's own controllers. This ensures zero duplication in filtering, creation and updates, and makes environment switching trivial. In the following chapters you will see each layer explored in depth

5. UI Builder (Configuration Layer)

This chapter describes the **authoring studio** of the UI Builder—the web app where business analysts define every element of a dynamic module without writing code. It covers how modules are managed, how pages and their navigation nodes are structured, the Page Builder form and metadata retrieval, and the four key editors for Fields, Views, Relation Items and Actions. Finally, we summarise how versioned metadata drives the runtime renderer
After reading this chapter, you will understand how business users craft modules end-to-end—turning tables into search screens, detail forms, relation tabs and contextual buttons—by merely filling in forms and saving metadata. This establishes the single source of truth that the Builder Engine and Integration Layer consume at runtime.

5.1. Managing Modules

The Modules screen provides a central catalogue of all dynamic applications registered in the UI Builder and enables administrators to perform lifecycle operations without touching code or the database directly. Upon loading the screen, each module is presented in a row showing:

- **Module code** (the unique identifier used in URLs and API routes)
- **API endpoints** for the four environments (RAD, SIT, UAT, PRD)
- **Status indicators** for pending changes or version mismatches

From this view you can:

1. **Open configuration** – Navigate into the module's tree of Nodes and Pages (see next section).
2. **Rename** – Change the human-friendly label of the module without affecting its code or endpoints.
3. **Delete** – Remove the module and all associated metadata (audit logs remain for compliance).

Clicking **Create module** launches a two-step wizard:
1. **Enter module code** – Specify a unique identifier (e.g. ict-manager).
2. **Configure endpoints** – Provide the base URLs for RAD, SIT, UAT and PRD.

Once the wizard completes, the new module is atomically registered in the identity service, allowing page-level permissions to be granted immediately. The user is then redirected to the fresh configuration tree, ready to add Nodes and Pages.

5.2. Module Configuration

Opening a module reveals its hierarchical tree of **Nodes** and **Pages**.

- **Nodes** represent folders in the left-hand ERP navigation. They are added or edited in a lightweight modal that captures name, parent node. Nodes can be nested indefinitely; some pages live directly under the module root and therefore have no parent node at all.
- **Pages** are the heart of the configuration. Clicking “New Page” launches a full-screen form powered by the Zod schema. The form fetches live metadata—schemas, tables, columns—via the Integration Layer so drop-downs stay in sync with the actual database.

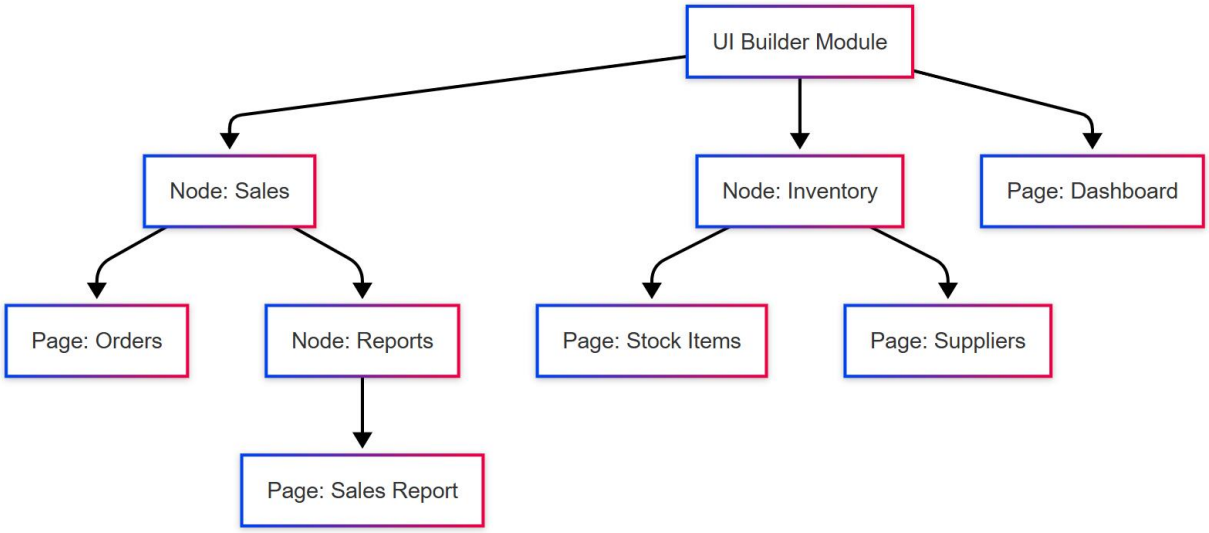

Figure 6: Example of the module tree

5.3. Page Builder Form

Table selection

The user chooses a table from the module’s schema. Once selected, a helper populates the **label** (human-friendly title), infers the **primary key**, and offers a list of candidate columns for the **default display field**. (see **Error! Reference source not found.**)

Navigation & grouping

An optional **node** drop-down decides where the page will appear in the ERP menu. A toggle labelled “Show in navigation” lets the author hide the page from the sidebar while still exposing it through relation links.

Fields and relations

Behind the scenes, once the table is confirmed the form generates two default collections:

- fields — every physical column becomes a field object with sensible defaults (component type, visibility, validation flags).
- items — any one-to-many relations discovered by the metadata query are added as relation-item stubs.

The user does not edit these lists here; they are refined later in the dedicated **Field Editor** and **Relation Manager** screens.

Save pipeline

On first save the UI Builder:

1. Persists the Page row together with its auto-generated field and relation skeletons.
2. Creates three baseline **Views**—*Search*, *Create*, and *Item*—using helper functions that map the table structure to the default grid or form layout.
3. Navigates directly to the new dynamic page so the author can verify the result in the running module.

An edit path follows the same form but skips the auto-generation steps; modifications are written back via an update mutation.

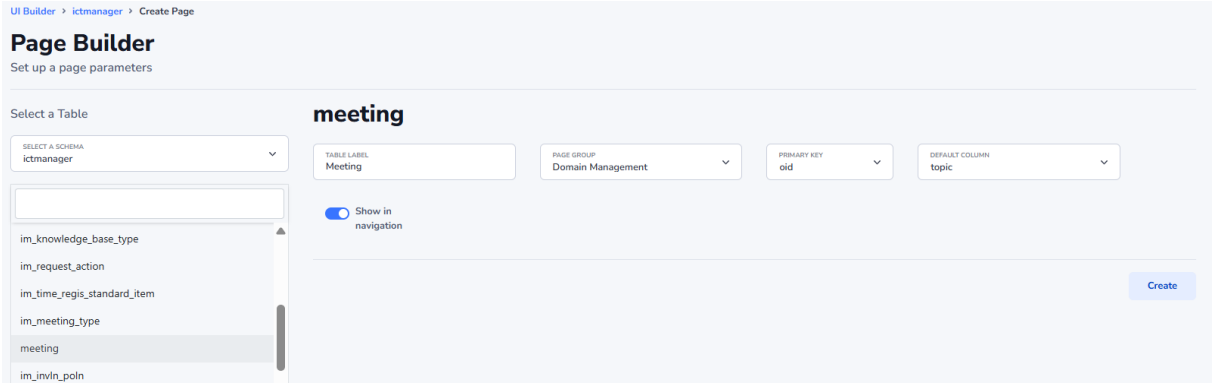

Figure 7: Page Builder form

5.4. Retrieving Metadata

Before any page or field can be configured, the UI Builder must know what tables, columns and relationships actually exist in the target schema. To this end, every drop-down or auto-completion widget in the authoring forms is backed by four read-only endpoints exposed via the Integration Layer:

- **GET /tables/{schema}**
  Returns an array of table names for the given schema. Used to populate the “Select table” list in the Page Builder form.
- **GET /columns/{table}**
  Returns column metadata (name, data type, primary-key flag) for the specified table. Drives both Field defaults and Zod schema generation.
- **GET /relations/{table}**
  Lists foreign-key constraints where the given table is the parent—used to seed Relation Items for one-to-many children.
- **GET /relations_other_tables/{table}**
  Lists inverse relations (where the given table is the child), enabling bi-directional navigation and automatic join definitions.

5.5. Page Configuration

Once metadata is available, authors use the Table Builder screen to refine every aspect of a page. A persistent four-tab ribbon lets them jump directly to the editor they need:

1. Fields
   The default view on open. Presents a sortable grid of all detected columns (from /columns/{table}), displaying label, component type, required flag and inclusion in the Item form. Drag-and-drop reordering persists immediately, ensuring the runtime layout matches the author's intent.

2. Views
   Lists all layout blueprints (search grids, item forms, Kanban boards) defined for this page. Authors can add, clone or delete views, then use the View Builder to pick columns, grouping fields and default filters.

3. Relation Items
   Shows one-to-many links discovered from /relations/{table}. Authors choose where each child collection appears (inline grid vs. detail tab vs. navigation link) and can toggle it on or off.

4. Actions
   Manages contextual buttons—such as "New record", "Export" or "Navigate to details." Each action metadata row captures its type, target endpoint or route, and any parameter mappings. Ordering here dictates toolbar sequence at runtime.

5.6. Fields

The Fields tab is a sortable grid that mirrors every column the engine discovered in the underlying database table. Each row shows the field's label, its input component, whether it is required, and whether it appears in the item-details form. Authors can drag-and-drop rows to change display order; the sequence is persisted immediately so the end-user layout always matches the list.

Selecting a row opens the **Field Builder** in a full-page overlay. The editor is organised in stacked panels that can be collapsed or expanded:

- **Field Info** (read-only) confirms the physical column, data-type, and—if the field is a foreign key—the referenced table.
- **Display Settings** let the author pick an input component (text box, date picker, dropdown, multi-select, etc.), supply a friendly label, choose grid width, and mark the field editable or read-only.
- **Live Preview** renders the chosen component in real time with the current settings, so the author sees exactly what the end user will see.
- **Rules** provide conditional behaviour: disable, hide, or colour a field when another field meets a condition.
- **Autogeneration** defines how IDs, codes, or timestamps are produced automatically, including source columns if the value is constructed from other fields.
- **Value Customisation** lets the author map specific values to visual indicators—typically colours or badges—to highlight statuses in a grid.

Saving the form writes a new version of the field record and returns to the list; no page reload or redeploy is needed. The Builder Engine will pick up the updated metadata the next time the dynamic page is rendered.

5.7. Views

Views control the layout and look of your data—whether it's a table, Kanban board, Gantt chart, or list—without touching the underlying table. Under the Views tab, each view appears as a card (icon, name, layout type) with quick actions.

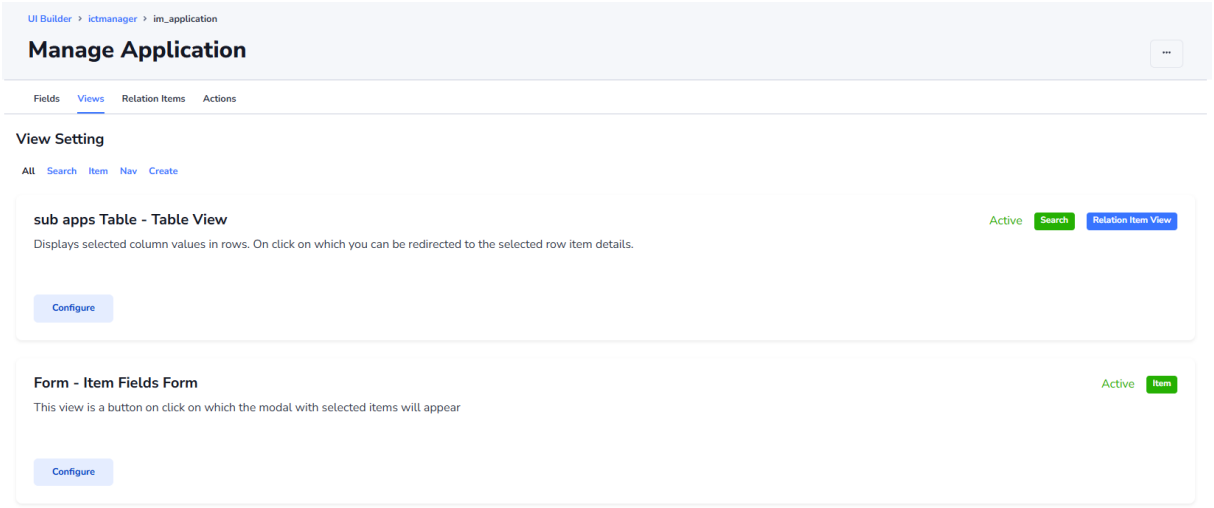

Figure 8: Views Section

Clicking "Add View" opens a simple wizard: choose a view type (search, item, nav, create), pick a layout, name it, and optionally clone an existing view. When you finish, a placeholder record is saved and you're dropped straight into the View Builder to configure fields, filters, and grouping.

5.7.1. The View Builder

The editor starts with a fixed header containing Save and Delete buttons, then flows through a series of collapsible sections that activate or hide themselves according to the chosen layout.

- **View Settings** collects the basics: label, layout code, active flag, and for grouped layouts (Kanban, Group Table) the field that drives grouping.
- **Selected Fields** is where the author decides which columns appear and in what order. A dual-pane interface lets them drag fields from *Available* to *Selected*. For advanced layouts, the pane adapts—Kanban asks for "Title," "Subtitle," and "Avatar Source," while a Gantt view needs "Start," "End," and "Duration."
- **Tabs and Sections** appear only when the author toggles them on. Tabs let a single *Item* view split into logical subsections (for example *General*, *Pricing*, *Audit*), while Sections provide columnar groupings inside a form. Both elements can be reordered by drag-and-drop.
- **Criteria** defines saved filters and default sorts. Each criterion specifies a field, an operation (=, >, <, like), and a value. When a view is marked as *search-enabled*, these criteria become the quick-filter chips end-users see at the top of the grid or board.
- **Preview** shows a live, data-driven rendering of the view against real records. The button stays disabled until every mandatory field is configured, keeping previews meaningful.

Saving commits the complete view definition, versioned under the page record. The Builder Engine reads the new metadata on the next refresh, so users see the grid or board instantly.

5.7.2. Why Multiple Views Matter

A single table often serves different audiences. Operations staff may want a dense grid for bulk editing, project managers prefer a Kanban or Gantt board, and executives just need a list of headline metrics. By letting each audience choose its own lens, the UI Builder prevents one screen from becoming overloaded while still guaranteeing that every view remains in sync with a single, central data source. The separation also future-proofs the page: new visualisations—pivot tables, calendars, or dashboards—can be added later without touching field definitions or business logic.

5.8. Relation Items

Relation Items define the one-to-many links between your page's table and other tables—think "Order - OrderLines" or "Project - Tasks." When you switch to the **Relations** tab, you see every detected foreign-key relation rendered as a list: each row shows the relation's label, the target table, its display location (tab or menu), and whether it's active.

Clicking any row opens the **Relation Item Builder** (**Error! Reference source not found.**). The header confirms the referenced table, and the form fields let you:

- Edit the human-friendly **Label**
- Choose **Display Location** (inline tab vs navigation link)
- Toggle **Active** on or off

Below the form, an embedded **View Section** shows which Search or Item views will render this child collection at runtime. Saving commits your changes immediately—no redeploy—and the Builder Engine picks up the new relation metadata on the very next page load.

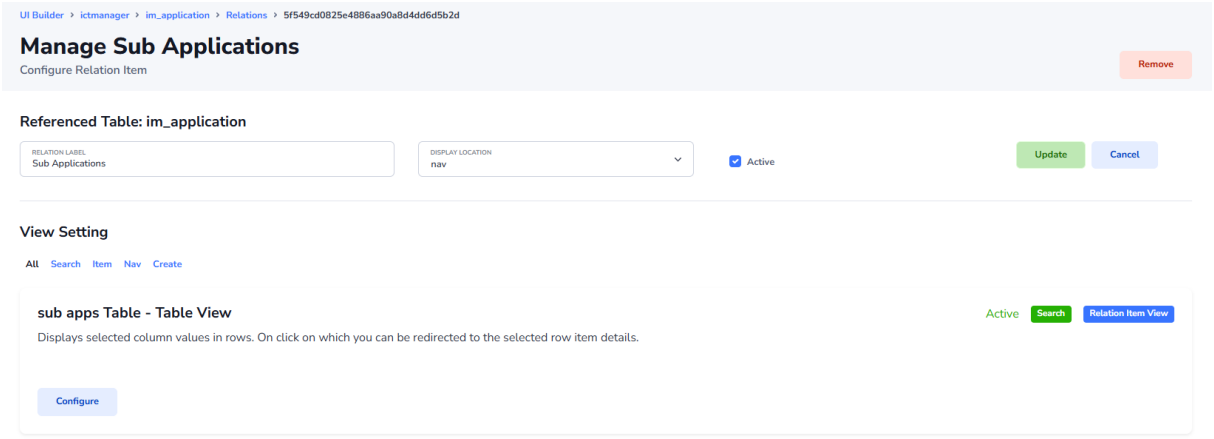

Figure 9: Relation Item Builder

5.9. Actions

The **Actions** tab lets you attach interactive buttons to your dynamic pages—everything from “Go to Details” and “Remove Item” to opening a modal of related records or kicking off a child-record creation flow. Each action is stored as a simple metadata record (its action_code, a label, and any required parameters like a field or related table), and users see them in a draggable list so you can control the order in which buttons appear.

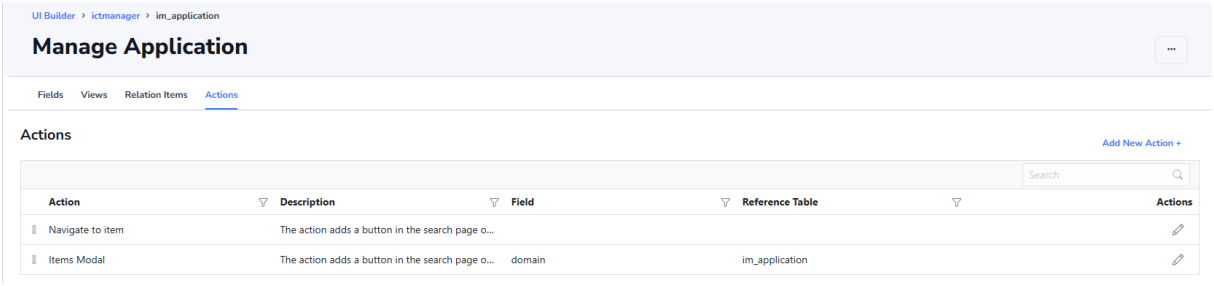

Figure 10: Manage Actions Section

At runtime, the engine maps each action_code to a React component and passes in the stored parameters. When an end-user clicks the button, the component runs its built-in logic—navigating, opening a modal, or creating a related record—without any additional wiring.

5.10. Conclusion

This chapter has demonstrated how the Configuration Layer provides a single source of truth for every aspect of a dynamic module—from initial API registration through Nodes and Pages, down to individual Fields, Views, Relation Items and Actions. By persisting all definitions as versioned metadata in the UI Builder studio, business analysts can introduce new screens, tweak layouts or adjust permissions simply by editing forms and clicking Save—no compilation, redeploy or database migration required.

Because the Builder Engine (Chapter 6) consumes this metadata at runtime, end users experience instant updates—new fields, views or buttons appear immediately upon page refresh—while developers benefit from a stable contract that cleanly separates authoring, rendering and integration. This configuration-driven approach not only speeds up delivery and ensures UI consistency across all Fenics modules but also empowers business teams to own their data presentation and workflow logic.

The next chapter explores the Builder Engine in depth, revealing how the metadata described here is transformed into live React components, how performance optimisations are applied at render time, and how client-side state and routing are orchestrated to deliver a seamless experience.

6. Builder Engine - Runtime Layer of Dynamic Pages

This chapter details the Builder Engine—the micro-frontend that consumes configuration metadata at runtime to mount fully functional CRUD pages without any per-table code. It explains how the shell injects routes, the structure of the renderer tree, and how navigation is augmented on the fly. It then reviews the three core page types and their view components, how actions are rendered, and walks through the user journeys for Search, Create and Item pages.

By the end of this chapter, you will understand exactly how configuration authored in Chapter 5 is transformed into interactive screens and how the Builder Engine remains decoupled from both authoring and data-access logic.

6.1. Plug-and-Play Integration

The shell keeps a **site-map** object. Any entry that sets dynamicRouter: true signals that the application wants dynamic pages. When React starts, a small effect in the shell:

1. Reads the current path (/ictmanager/…).
2. Downloads that remote's routes bundle.
3. If dynamicRouter is enabled, lazily pulls **uibuilder/renderer** and appends its route tree as a child of the remote's base route.
4. Hydrates React-Router with the merged route object.

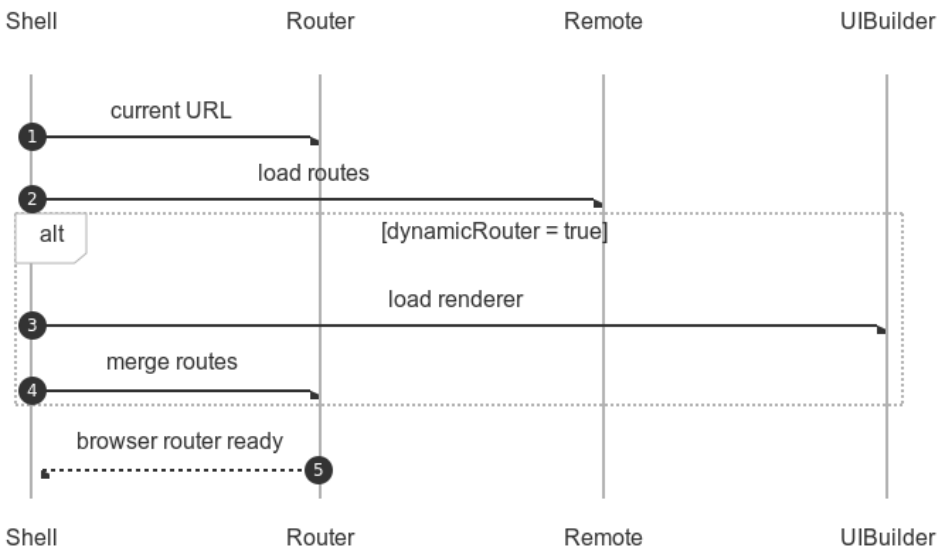

Figure 11: Route-injection sequence

6.2. Renderer Tree

renderer.tsx exports a self-contained subtree:

```
└── <HeadersProvider>
    └── <PageLayout>
        ├── /:table       → <SearchPage>
        ├── /:table/create → <CreateItem>
        └── /:table/:id   → <PageItem>
```

The HeadersProvider injects auth tokens; PageLayout supplies the common toolbar and breadcrumb; each leaf component calls the Integration Layer for config and data. Because the tree lives in one place, upgrading the layout or adding a new sub-route requires touching a single file.

6.3. Auto-Expanding Navigation

A custom hook (useDynamicRoutes) scans the site map at runtime:

- For every dynamicRouter app it fetches that module's **Nodes** and **Pages**.
- Permission context filters out items the user cannot read.
- Routes are cloned, then enriched with new **folder nodes** and **root-level pages**.
- The sidebar re-renders, revealing the new links without reloading the remote app.

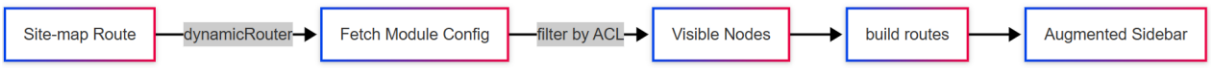

Figure 12: Navigation augmentation pipeline

6.4. Pages Overview

The renderer exposes exactly three routable page-components, each mapped to a concrete CRUD concern:

Route	Purpose	What it renders	When it is shown
/:tableName	Read-many	**SearchPage**	Always—the landing view for a table
/:tableName/create	Create	**CreateItem**	Only if the page's metadata marks create_enabled =true

Route	Purpose	What it renders	When it is shown
/:tableName/:itemId	Read / Update	**PageItem**	When a user drills into a single record

Table 18: Rendered routes overview

All three pages receive their full behaviour—fields, views, relation items, actions—through the same Page record pulled from the Integration Layer, so adding a new table never requires new code or routes.

6.5. Views Overview

A *view* is a ready-made React component registered in viewTypes.
At runtime the renderer inspects view.code, looks it up in that registry, and mounts the corresponding component, injecting:

- View: full metadata (selected fields, criteria, grouping)
- Actions: page-level button definitions
- relationItem, value, parentTable when the view is used inside a child list
- mode (preview | select | read) so the same component can power the Builder preview, a “select record” modal, or the normal page

Below is the catalogue bundled today:

code	Layout	Primary use	Notable options
table	Grid	Search	inline create / delete, column reorder
group_table	Tabbed grid	Search	groupSettings picks the grouping field
kanban	Drag-drop board	Search	title / subtitle / avatar / colour indicator slots
gantt	Timeline	Search	start, end, duration, progress, parent
list	Media list	Search	title + description + date
item_fields	Embedded form	Item	renders inside PageItem body
item_fields_modal	Modal form	Navigation	opens from a header button
item_fields_offcanvas	Off-canvas form	Navigation	same as above but side-panel

Table 19: List of views

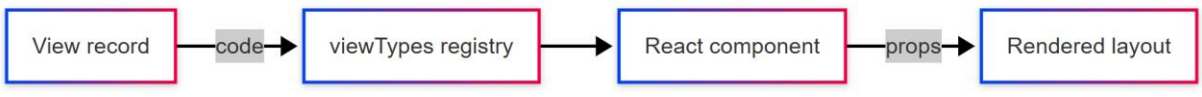

Figure 13: Resolving a view at runtime

6.5.1. Table View

The Table View renders your selected fields as a familiar data grid. Column definitions—labels, types, grid widths, and even colour-coding rules you set in the Field metadata—drive every cell's appearance. Above the grid, any page-level Actions (New, Remove, custom buttons) are injected into the header bar, giving end users one-click access to common workflows. Selecting a row highlights its primary key and streams that oid into child components (for relation-driven "Add Item" flows) or into any Action that needs to know which record is active.

Key props

- view – full metadata (columns, criteria)
- actions – array of buttons to render in the header
- onSelect(oid) – callback when a row is clicked
- relationItem? / defaultSc? – pre-filters for embedded child lists

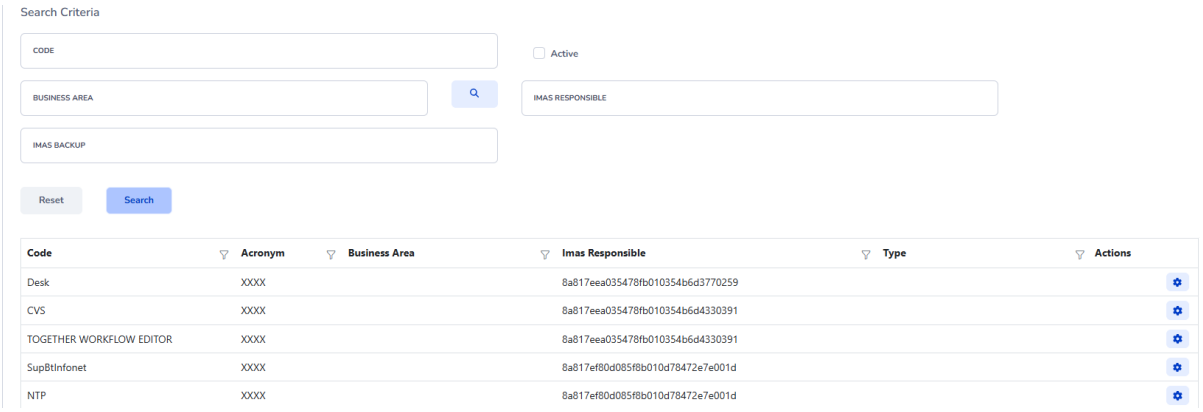

Figure 14: Table View

6.5.2. Kanban View

The Kanban View turns tabular data into draggable cards grouped by a single "group" field of your choice. Each column in the board corresponds to one value of that field. When a card is dragged to a new column, onDragEnd fires: the engine builds an updateItem DTO that patches only the changed group-column, then calls the API. Double-clicking any card cancels the built-in Syncfusion dialog and instead opens your metadata-driven **EditCardModal**, reusing the same form logic you already defined for CRUD screens.

Key props

- columns[] – the list of grouping values & labels
- groupField – database column used for grouping
- headerKey / contentKey – which selected_items map to card header and subtitle

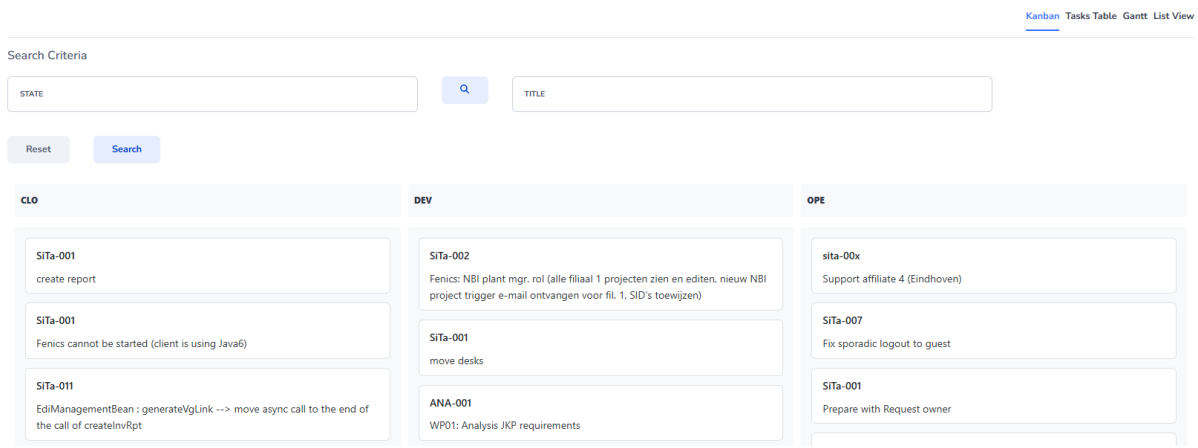

Figure 15: Kanban Board

6.5.3. Item-Fields Modal / Off-canvas

When you need a full-page form in a popup, these views wrap the core <ManageItemForm> inside either a Bootstrap Modal or an off-canvas side panel. A single wrapper component:

1. Renders the trigger (button or nav-tab)
2. Supplies container chrome (title, close button)
3. Manages visibility with useToggle

Because the form fields and validation rules come entirely from your View metadata, the same wrapper handles both **create** and **edit** scenarios without extra code.

Key props

- useToggle() – hook returning { show, setShow }
- Trigger – optional custom trigger component
- Container – optional custom wrapper (modal vs offcanvas)
- All standard ViewProps (view, value, onSubmit, etc.)

6.6. Rendering Page Actions

Every page in the Builder Engine can host an arbitrary set of action-buttons, driven entirely by the module's **Action** records and rendered through the shared pageActions registry. When the Search or Item page mounts, it fetches the list of configured actions for that page and—for each one—looks up its action_code in the registry to find the corresponding React component and required props.

Under the hood, the registry is a simple array of { action_code, Component, display_type, props } entries. At render time, the page filters that array to pick only actions whose display_type matches the current layout (for instance "top" buttons above a grid, or "item" buttons inside each row). It then instantiates each Component by passing it:

- the action record itself (so it knows its oid, field, or referenced_table parameters),
- the current parentTable name,
- any selection or context values (like the oid of the highlighted row).

Because each component lives inside the shared Builder Engine, they all share the same hooks for headers, queries, and modals—but each one encapsulates a distinct piece of logic:

Action code	UI component (file)	Rendered in*	What it does	Key runtime props
navigate_to_item	NavigateToItem.tsx	Item-row tool-bar	Opens the details page of the selected row	value (selected oid)
remove_item	RemoveItem.tsx	Item-row tool-bar	Soft-deletes or inactivates the row	value, parentTable
navigate_to_relation_item	NavigateToRelationItemDetails.tsx	Grid header	Jumps straight to a *related* record (FK column)	value, field, referenced_table
open_items_modal	ItemsModal.tsx	Grid header	Pops up a modal that lists child items from a junction table	value, referenced_table
create_relation_item	AddRelationItem.tsx	Grid header	Opens a modal with a *ManageItemForm* pre-filled so the user can add a new child record	value, referenced_table

Table 20: List of implemented actions

6.7. Search Page (Read Many)

When the user navigates to /module/{table} the SearchPage component wakes up, pulls the page record and every *search* view for that table, then decides—based on the permission provider—whether the visitor is allowed to continue. A denied visitor only sees a red alert; an authorised one is given a fully-featured screen assembled from metadata.

If the page owns more than one search view, SearchPage adds an underline-style tab bar. Each tab uses the label stored in configuration; switching tabs simply swaps the activeView object in local state.

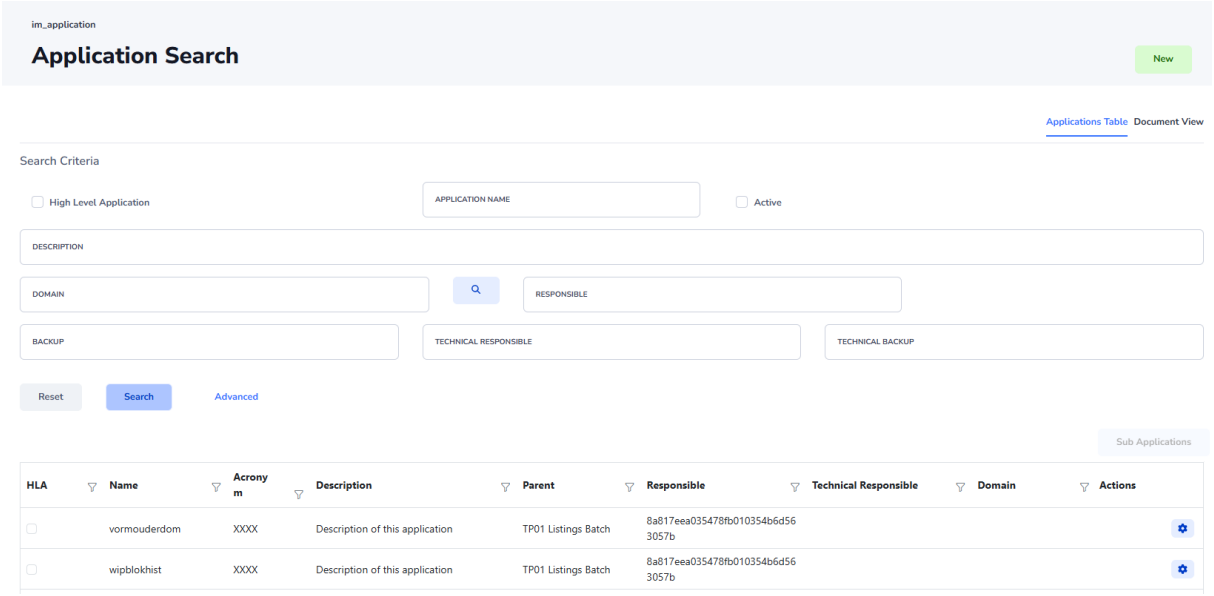

Figure 16: Example: Configured Application Search Page

Below that tab bar the real work happens. SearchPage looks up the React component associated with the chosen view code (table, kanban, gantt, etc.) in the shared registry and mounts it:

```
<viewConfig.component view={activeView}
                      actions={page.actions}/>
```

Nothing else in SearchPage needs to know whether the user ends up with a grid, a Kanban wall or a timeline. The view component understands its own props, fetches rows through useTableViewSearch, streams the selected primary-key back to action buttons, and honours any colour rules or grouping metadata defined earlier in the UI Builder.
Edge cases are folded in at the bottom:

- If the table has no configuration the user gets a blue **Create configuration** link back to the builder.
- Any server error bubbles up as a red ErrorAlert.
- Deleting the entire page is one click away through a confirmation modal wired to useRemovePageMutation.

The net result is a repeatable, five-step recipe—metadata → permissions → header → optional tabs → pluggable view—that can render every search screen in the ERP without writing a bespoke component per table.

(see *Figure 25: High-level flow Search Page*)

6.8. Create Item

When the route / :tableName /create is hit the **CreateItem** component assembles a form in three quick passes.

Permissions & Metadata
It first pulls the *Page* record (to confirm the user can create) plus the raw column list for the table and any *create* views that might exist.
If no create right = red alert; otherwise continue.

Pick a layout

- If the page already owns one or more **create views** the code simply resolves viewTypes.find(v => v.code === view.code) and renders the chosen component.
  Most modules just point at item_fields which wraps a ManageItemForm inside a modal or off-canvas but teams can swap in something richer (wizard, stepper, etc.).
- If no view is defined the engine falls back to a plain ManageItemForm built from every editable column returned by useTableFieldsQuery.

Query-string helpers (?column_name=…&value=…) pre-fill a foreign-key field when the Create page was opened by an “Add child item” button.

Save & return

On **Save** the form calls useCreateItemMutation; when it resolves the page:

- clears any draft JSON cached in localStorage,
- navigates back to the previous URL,
- lets React-Query broadcast its cache invalidation so the parent Search grid refreshes.

Because both success and error paths surface as toast pop-ups, the user never loses context.
Under the hood the heavy lifting is still done by the generic **ManageItemForm**: it reads the same Field metadata used everywhere else, applies the correct input component, validation rule and auto-generation logic, then serialises its payload as a typed CreateItemRequest. All CreateItem really does is decide which *container* to hand that form to—full page, modal or off-canvas—based on whether a bespoke create view exists.

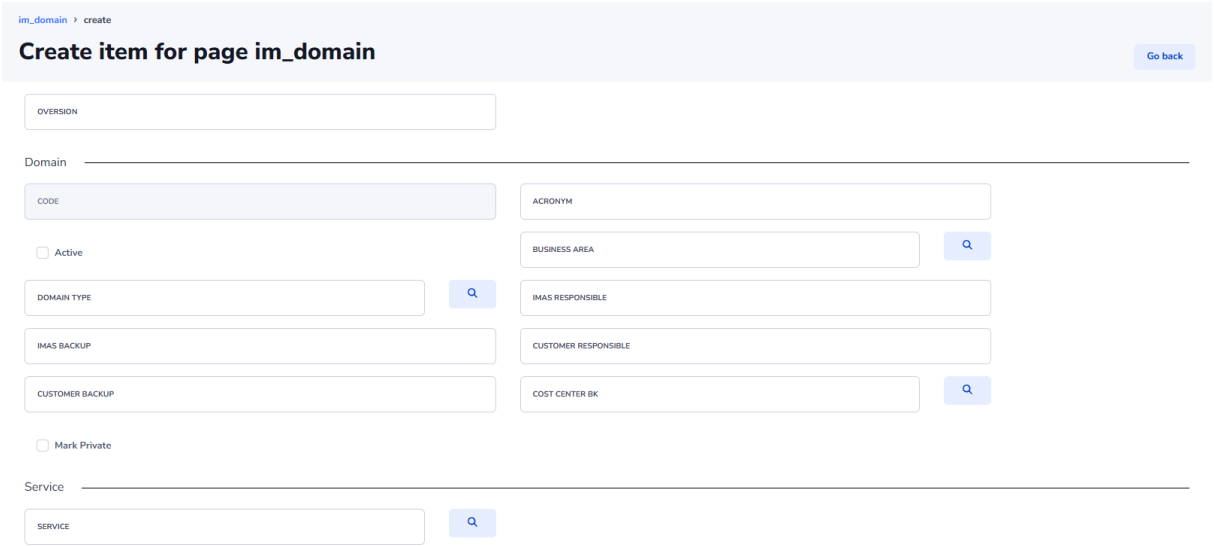

Figure 17: Example: Create item Form in im_domain table

6.9. Item-Details Page (Manage one)

6.9.1. Updating the Record

When / :tableName /:itemId is opened the page loads three datasets in parallel:

- the **Page** record (permissions, default label, relation list),
- the column list,
- the item row itself.

The page always looks for *item*-type views first:

useGetFullTableViewsQuery(headers, table, 'page', 'item')

If at least one view is returned, each view's React component (viewTypes.find(…)) is rendered.

- Typical choice is **item_fields** which embeds a <ManageItemForm> in a modal/off-canvas and provides its own *Save* button.
- More specialised layouts (charts, read-only cards, etc.) work the same way because they are declared in metadata.

Only when *no* item views exist does the page fall back to an inline **default form**.

Most modules never see the fallback—authors define an item_fields view and the form lives inside that component. The header-level *Edit* button exists only to service legacy tables that haven't gained a custom view yet, keeping the behaviour consistent while the catalogue of views grows.

(For detailed flow, look at *Figure 26: Sequence Diagram Update Item****Error! Reference source not found.***)

6.9.2. Browsing relations and embedded views

Below the header a small underline tab bar is built from the page definition:

- **Item Fields** — the default tab shows either
  - a stack of "item" views (if the author created them) or
  - the raw form described above,
    plus any relation items that are flagged **display location = item**.
- *A tab per relation* — every 1-to-many link flagged **display location = nav** is rendered as a tab; clicking it swaps in a <RelationItem> component that:
  - chooses the correct *search* view for the related table,
  - injects a pre-built filter so only child rows linked to the parent are fetched,
  - reuses the same grid / kanban / list components already registered for normal pages.
- *Optional "nav" views* — if the page owns extra navigation views (e.g. summary charts) they are injected as additional tabs the same way.

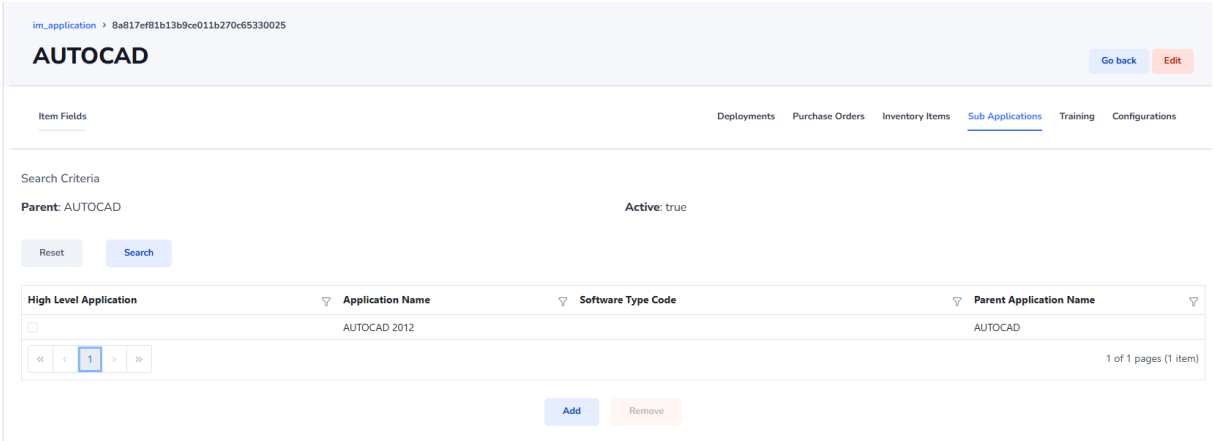

Figure 18: Relation Item Tab

Because both the form and every relation view are driven by exactly the same Field and View metadata as the rest of the system, the Item-Details page needs no table-specific code: toggling edit, saving, revealing relations, even colour rules or drag-and-drop columns all materialise automatically from configuration.

6.10. Conclusion

The Builder Engine reconciles **configuration** with **runtime** by:

- **Injecting routes** dynamically based on which modules opt in.
- **Rendering a small subtree** that handles headers, layout, permission checks and metadata fetches.
- **Augmenting navigation** so dynamic pages appear seamlessly in the ERP menu.
- **Mounting view components** (tables, Kanban boards, modals) by looking up view.code in a registry.
- **Instantiating actions** purely from metadata, without additional code.
- **Executing CRUD lifecycles** through uniform data flows for search, create and item-details.

Because every page component, form and data hook derives its behaviour from the same metadata registry, the host application's codebase never needs to change when new tables are introduced. This runtime layer closes the loop on configuration-driven development, delivering a true plug-and-play experience for Fenics modules.

The next chapter—7 Data Flow & Integration Layer—dives into how every API request from these dynamic pages is constructed, dispatched and handled, ensuring secure, performant communication with backend services.

7. Data Flow & Integration Layer

This chapter describes how the UI Builder's runtime communicates with backend services to fetch configuration and persist data. It details how the module's API endpoints are resolved, how an opinionated client is constructed, and how React Query hooks surface typed queries and mutations. It then explains the transformation of View metadata into SearchRequestDTO objects, the structure of the dynamic Create/Update payloads, and the end-to-end data flow. Finally, it highlights the key code components that glue these pieces together before summarising the Integration Layer's role in our architecture.

By the end of this chapter, you will have a clear understanding of how metadata-driven pages turn into concrete HTTP requests—and how this thin membrane of logic keeps the Builder Engine decoupled from any module's internal controllers.

7.1. Where the Endpoint Comes From

Every deployable ERP module carries its own REST facade.
During configuration the author supplies the four root URLs once—one for each delivery lane (RAD / SIT / UAT / PRD):

```
{
  "code": "ictmanager",
  "api_url_rad": "https://rad-api.erp.local/ict",
  "api_url_sit": "https://sit-api.erp.local/ict",
  "api_url_uat": "https://uat-api.erp.local/ict",
  "api_url_prd": "https://api.erp.com/ict"
}
```

At start-up the shell inspects process.env.NODE_ENV, picks the matching entry, and plants it in **ApiBaseUrlContext**.
From that moment every child component can grab the active host in one line:

```
const { baseUrl } = useApiCtx();
```

No hard-coding, no if-else ladders—just a single source of truth.

7.2. A Small, Opinionated Client

To minimise surface area and enforce consistency, the client exposes only the two endpoints the Builder Engine requires. Table 21: Endpoints overview summarises these routes and their intended usage.

Endpoint	HTTP	Purpose	Called from
utils/combined	**POST**	**Filter + Join search** — returns rows shaped by SearchRequestDTO	All Views, Relation-items
utils/dynamic	**POST**	Create item (body =CreateItemRequest)	ManageItemForm / Kanban "Add"
utils/dynamic	**PUT**	Update item (body =EditItemRequest)	Inline drag-drop, Edit forms

Table 21: Endpoints overview

7.3. React-Query Hooks

The opinionated client is surfaced through a small set of typed hooks. These hooks automatically inject the base URL and headers via useApiCtx() so they work unchanged in every module:

```
const { data: rows } = useFilterSearchQuery(dto)        // POST utils/combined
const createItem      = useCreateItemMutation()        // POST utils/dynamic
const updateItem      = useUpdateItemMutation()         // PUT  utils/dynamic
```

useApiCtx() injects the **baseUrl** and **headers** so the same hook set can be reused by every module.

7.4. From View to SearchRequestDTO

A Search view never writes SQL. Instead it calls getSearchRequestForView(view) which walks the view's **Selected Items** tree and emits a pure-data DTO:

- selectedFields[] the columns (with aliases) that should be returned
- foreignKeys[] ordinary 1-to-N joins
- junctionTables[] self-joins for recursive relations
- searchCriteria[] user filters + default filters
- schema_name / table / identifier routing info

Below is the algorithm in plain language.

7.4.1. Algorithm Flow

Rather than hand-crafting SQL, the engine transforms a View's metadata into a SearchRequestDTO in four clear passes. *Table 22: Steps of the VIEW -> DTO algorithm* outlines each step in this transformation.

Step	What it does	Key objects it touches
1. Initialise	Grab root table / schema / PK from the page and seed dto.selectedFields with the PK.	rootTable, pkCol, selectedFields[]
2. Walk the "selected items" tree	*Depth-first* recursion builds two things:• **SELECT list** — the actual columns, with **self-join aliases** when a table references itself.• **Join metadata** — foreignKeys[] for simple FK joins, junctionTables[] for self-joins / many-to-many.	handleItem() recursion
3. Apply criteria	Merge view-level filters with any **default** criteria supplied by the caller (e.g. "show only items belonging to this parent").	searchCriteria[]
4. Prune & deduplicate	Strip empty criteria, remove unused joins, ensure FK list is unique.	getSearchRequest()

Table 22: Steps of the VIEW -> DTO algorithm

7.4.2. Single vs. Multi-Row Search

Depending on context, the engine adds filters according to one of two modes:

- **Single-item look-ups** (e.g. *open item details*) simply add one more criteria entry: table = rootTable, column = pk, value = selectedOid.

- **Bulk grids / Kanban** omit that clause; they return the entire result set, paged by the module's API.

Everything else—columns, joins, even self-referencing aliases—comes straight from the view's tree, so the UI author never worries about SQL shape.

With the algorithm in place, the **/utils/combined** endpoint receives a predictable JSON contract no matter how complex the joins.

7.5. Create / Update: the *dynamic* endpoint

All Create and Update operations POST or PUT a strict DTO to /utils/dynamic. *Table 23: Manage item DTO fields* shows the fields included in that payload.

field	**purpose**
table_name / schema_name	where the row lives
type_identifier	primary-key column on that table
identifier *(edit only)*	value of the PK for the row being patched
properties[][property]	column name
properties[][value]	typed value (or FK oid)
properties[][column_value]	human-readable label when the column is a relation
properties[][data_type]	string | number | boolean – keeps the server honest

Table 23: Manage item DTO fields

Why both value and column_value?

- value is what the database needs (e.g. FK oid)
- column_value is what the user sees (e.g. "Project Alpha") – it can be re-used in the response without an extra join.

7.6. Data Flow

To illustrate the end-to-end journey of a Create or Update request, *Figure 19: Post / Put Data Flow* shows the sequence from form submission to cache invalidation.

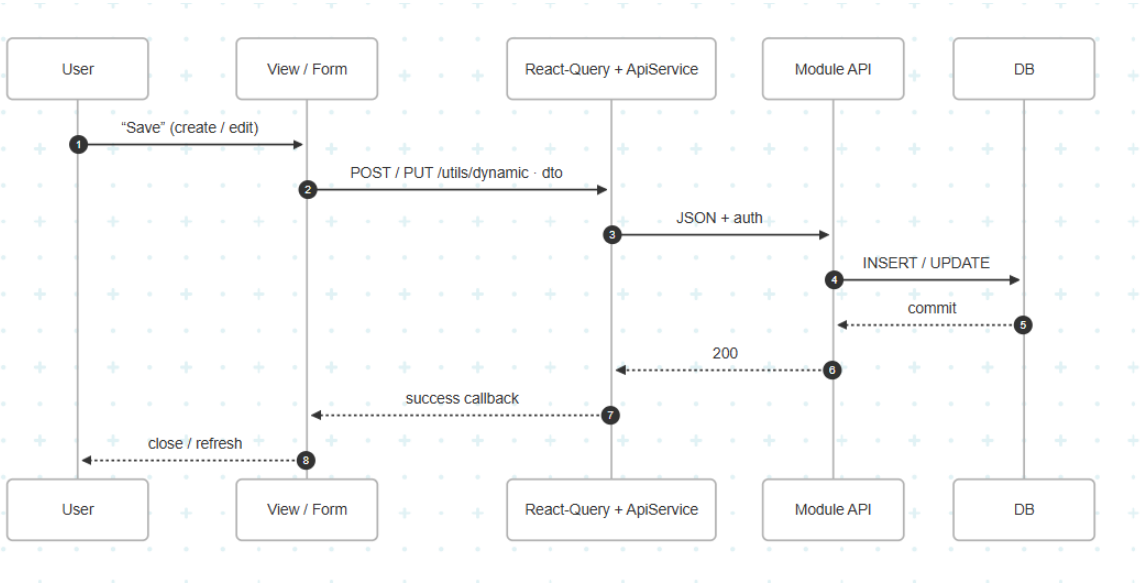

Figure 19: Post / Put Data Flow

7.7. Key touch-points in code

The following table highlights the main components and hooks that make the Integration Layer work.

layer	highlight
ManageItemViewWrapper	decides *create* vs *edit* and where to mount the form (modal / off-canvas / full page).
ManageItemForm	builds *properties* array, runs zod schema generated from Field metadata, flags unsaved changes, fires onSubmit.
hooks/module-queries.ts	useCreateItemMutation & useUpdateItemMutation wrap the ApiService calls and re-broadcast success to React-Query cache.
ApiService	adds Authorization, retries, JSON parse, and returns a clean object.

Table 24: Integration Layer key components and functions

No server-side knowledge leaks upward; the Builder Engine only "knows" about utils/dynamic.

7.8. Integration-Layer Recap

The Integration Layer is the thin membrane that turns author-friendly metadata into stable HTTP calls and back again.

1. **Environment switch** – the correct api_url_* is injected into ApiBaseUrlContext at start-up; the rest of the code never hard-codes a host.
2. **Opinionated client** – createModuleApi(baseUrl) presents only six endpoints; anything adhering to that mini-contract becomes "UI-Builder-ready" instantly.

3. **DTO builders & validators** – getSearchRequestForView() (reads) and Zod-generated item DTOs (writes) guarantee that malformed traffic never leaves the browser.
4. **React-Query wrapper** – hooks add retries, cache keys, optimistic UI and header injection with zero boiler-plate.
5. **Error funnel** – all non-2xx responses surface through a single toast / alert path, so modules behave consistently without extra code.

Together these pieces let the Builder Engine concentrate on *what* to fetch or update while the Integration Layer quietly handles *how* to speak to any compliant module.

8. Conclusion

8.1. Recap of the Project

The assignment began with a deceptively simple brief: **“Rebuild the ageing ICT Manager module of Fenics ERP as a modern web application.”**

What made it daunting was scale—**200 + pages, hundreds of tables and relations, and two decades of ad-hoc features**. Re-coding everything screen-by-screen would have taken months and locked the new system into the same rigidity as the old one.

During the first two weeks I dissected the legacy Java client, catalogued page patterns, and mapped every table, relation and CRUD flow. That analysis led to a pivotal insight:

Most pages differ only by metadata *(table, columns, relations, permissions).*
If we could externalise that metadata, we could generate the UI instead of hard-coding it.

From that idea the **UI Builder** platform was born. It introduced:

- **Config Layer** – page, field, view, relation and action descriptors stored in PostgreSQL via a low-code React editor.
- **Integration Layer** – a thin REST contract (/utils/schema/data, /utils/combined, /utils/dynamic) that every module API implements so the same React Renderer can talk to any backend.
- **Builder Engine / Renderer** – dynamic routes, view components (Table, Kanban, Gantt, List, Item-Form) and a small opinionated client that turns config into live pages at runtime.

With these three layers the project achieved its core aim:

- **ICT Manager was migrated without manual page rewrites**—new tables are now onboarded in minutes.
- The same mechanism is already being reused for other Fenics modules (e.g. Purchasing) with minimal overhead.

8.2. Key Learnings and Outcomes

8.2.1. Major achievements

- **About 90 % functional coverage** of the legacy ICT Manager was reached without hand-coding screens.
- **One-click onboarding** – a new table can be exposed to users by filling out the Page wizard and pressing *Create*; views and default field settings are generated automatically.
- **Module-agnostic contract** – the /utils/* endpoint set has already been adopted by Purchasing and Assets, proving the pattern portable.
- **Developer velocity** – what would previously take a fortnight of React work is now a morning of metadata tweaks.

8.2.2. Against the project plan

Milestone	Planned	Delivered	Comment
Analysis & page inventory	W-2	W-2	on time
POC dynamic grid	W 4	W 4	met
Config Layer MVP	W 6	W 7	+1 week – extra time for Zod validation
Renderer v1 (Table & Form)	W 8	W 8	on time
Kanban / Gantt views	W 10	W 10	
Security	-	W11	Not really planned, but delivered

Milestone	Planned	Delivered	Comment
Roll-out to 80 % pages	W 12	W 12	met
Close-out & docs	W 13	W 13	met

Table 25: Milestones: Planned vs Delivered

8.2.3. Technical & Professional Skills Gained

1. **Deep Dive into Syncfusion UI Library**
   Worked extensively with Syncfusion to build highly interactive and customizable UI components, improving my ability to deliver polished, enterprise-grade interfaces.
2. **Mastering Multi-Environment Workflows**
   Learned to structure applications for seamless deployment and operation across multiple environments (RAD, SIT, UAT, PRD), enhancing the project's reliability and flexibility.
3. **Embracing Module Federation Architecture**
   Gained hands-on experience integrating Module Federation, enabling independent, plug-and-play modules across the ERP system.
4. **Frontend Engineering**
   Significantly improved my proficiency with modern React patterns, hooks, and advanced component design, especially in dynamic and metadata-driven contexts.
5. **Analytical Problem-Solving**
   The project demanded deep system analysis and creative thinking to translate existing manual processes into configurable, automated solutions.
6. **Collaboration & Communication**
   Strengthened my skills in working within a multidisciplinary team, collecting requirements, demoing progress, and aligning with both technical and business stakeholders.

8.3. Reflection on Challenges and Solutions

The Translation Maze
The hardest nut to crack was turning a human-friendly View into the single JSON payload that the /utils/combined endpoint understands. Every field, join and criterion had to be flattened without losing hierarchy or aliases. The breakthrough came with a depth-first "walker" that writes the query tree as it goes, inventing safe aliases for self-joins and attaching default or advanced criteria only at the very end. Once this engine was stable, every other feature cascaded into place.

Taming Self-Joins
Kanban boards, bills-of-materials and organisational charts all point back to the same table. The legacy API could join a table to itself only once, which broke these scenarios. We extended the Postgres wrapper to accept an alias column and let the Builder request LEFT JOIN table AS t2, t3, t4… on demand. This single change unlocked the entire family of recursive screens.

One Widget, Everywhere
Early prototypes treated "forms" and "search views" as two species, making it impossible to reuse a colour badge or range slider across layouts. The solution was brutal and simple: abolish the distinction. Everything—table grids, forms, modals, off-canvas panels—is now just a View rendered by the same Selected-Items grammar. Any component can appear anywhere, which slashes future maintenance.

Guard-rails Against Injection
Because search criteria are user-supplied, a stray quote could sneak into a LIKE condition. A lightweight sanitation layer was added in the Node middleware: it refuses multi-statement hints, unbalanced quotes or comment markers before they ever reach Postgres.

8.4. Recommendations and Future Work

1 — Loosen the Backend Contract
Today a dynamic page is tied to the exact table/column names exposed by /utils/combined. To unlock truly bespoke screens, let authors override the endpoint and HTTP verb per View, then map field names to payload keys in the UI Builder itself. With this indirection we could point a Kanban lane to a stored procedure, or trigger a Lambda without touching backend code.

2 — Event-Driven Triggers
Introduce a "Triggers" palette (onSave, onDelete, onDrop, onRowSelect…) where authors pick a client-side event and wire it to an endpoint or JavaScript snippet. This would cover approval workflows, notifications and cross-module synchronisation without extra deployments.

3 — Schema-Less Fields
Allow "virtual" fields that do not exist in the database—calculated formulas, external look-ups, or UI-only flags. The View engine would resolve them after the main search call and merge them into the result set.

4 — Progressive Typing & Validation
Move field validators from Zod into the metadata so that non-developers can declare regex, min/max or enumeration lists directly in the Builder. The runtime would pick the right validator automatically.

5 — Automated Regression Harness
Once the endpoint override and triggers land, the surface area grows. A headless Playwright suite that replays the inventory of dynamic pages nightly would prevent silent breakage—especially important when multiple modules share a single Builder runtime.

If these improvements are adopted, the UI Builder will cover needs of most applications in the whole ERP system of the company and become a genuine low-code engine for any future module.

8.5. Demo

A full demonstration of the UI Builder in action—including dynamic page setup, view creation, search, edit, and relation navigation—can be found in the following video:

Watch the Demo Video

This demo highlights both standard and advanced scenarios, showing how even complex ERP screens can be assembled and maintained with minimal code.

8.6. Closing remarks

This project set out to transform the way business applications—starting with ICT Manager, but extendable to any compatible module—handle dynamic UI generation. The resulting UI Builder now enables rapid, metadata-driven screen assembly in any application that implements the integration contract.

While there are still edge cases and room for more advanced customization, the core solution is robust and highly adaptable. By reducing manual development and making UI configuration a dynamic, low-code process, this approach paves the way for greater scalability and maintainability—not just for one ERP module, but for the entire platform moving forward.

Reference list

Alves, C. (2024). *7 reasons Penpot is more than just a Figma alternative*. Retrieved from penpot.app: https://penpot.app/blog/7-reasons-penpot-is-more-than-just-a-figma-alternative/

Anshul, A. (2024). *Schema validation in TypeScript with Zod*. Retrieved from log.rocket: https://blog.logrocket.com/schema-validation-typescript-zod/

Arunodi, N. (2024). *Top 10 React Component Libraries Every Developer Should Know*. Retrieved from Syncfusion: https://www.syncfusion.com/blogs/post/top-react-component-libraries

draw.io. (2024). *What makes a good UML diagram tool?* Retrieved from draw.io: https://www.drawio.com/blog/uml-diagram-tools

Ihnatovich, D. (2025). *React Hook Form vs Formik*. Retrieved from medium: https://medium.com/%40ignatovich.dm/react-hook-form-vs-formik-44144e6a01d8

Patel, D. (2024). *RTK Query Vs. React Query: Breaking Down the Technicalities*. Retrieved from dhiwise: https://www.dhiwise.com/post/rtk-query-vs-react-query-breaking-down-the-technicalities

Solomakha, V. (2024). *Penpot vs Figma Comparison: Full Review For UI And UX Design*. Retrieved from banani.co: https://www.banani.co/blog/penpot-vs-figma-review

Stack Overflow. (2024). *Web Frameworks and technologies*. Retrieved from Stack Overflow: https://survey.stackoverflow.co/2024/technology#1-web-frameworks-and-technologies

Upadhyay, Y. (2025). *TypeScript takes over: Why JavaScript developers are switching in 2025.* Retrieved from Medium: https://medium.com/@upadhyayyuvi/typescript-takes-over-why-javascript-developers-are-switching-in-2025-51d4f4225f48

ATTACHEMENTS

Internship-Project-Planning.docx

Figure 20: Internship Project Plan

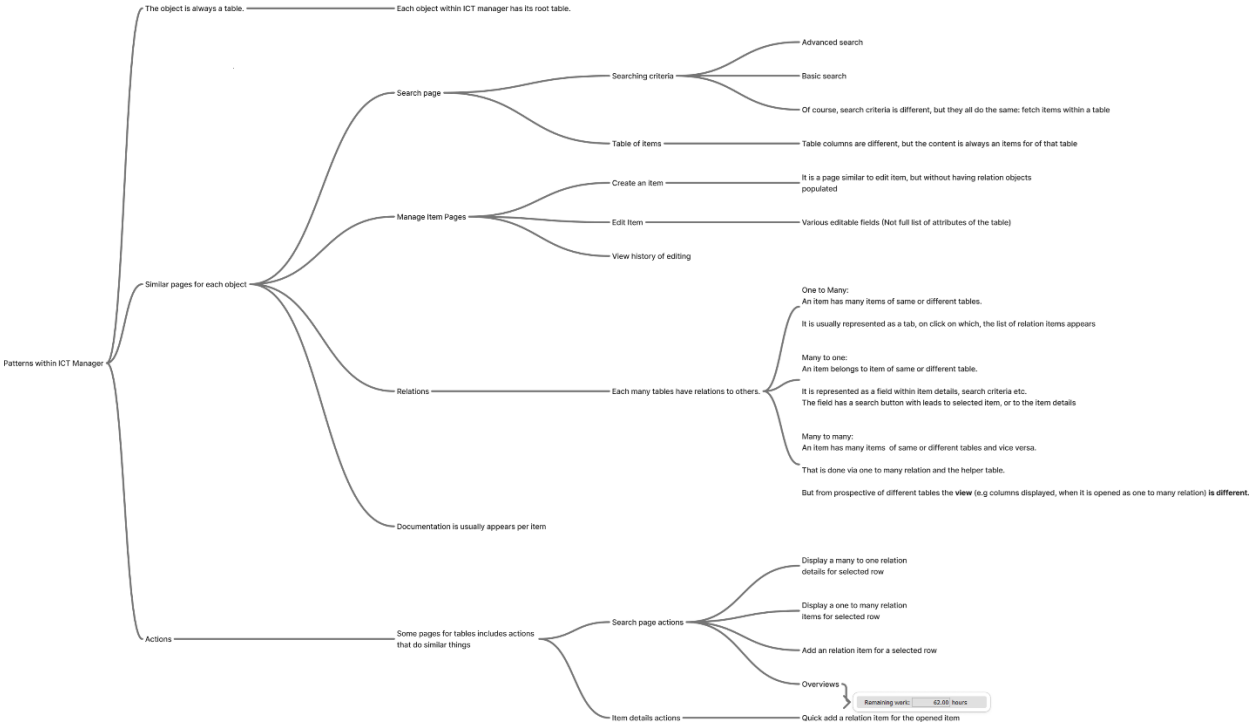

Figure 21: Repeating page patterns in the ICT Manager module

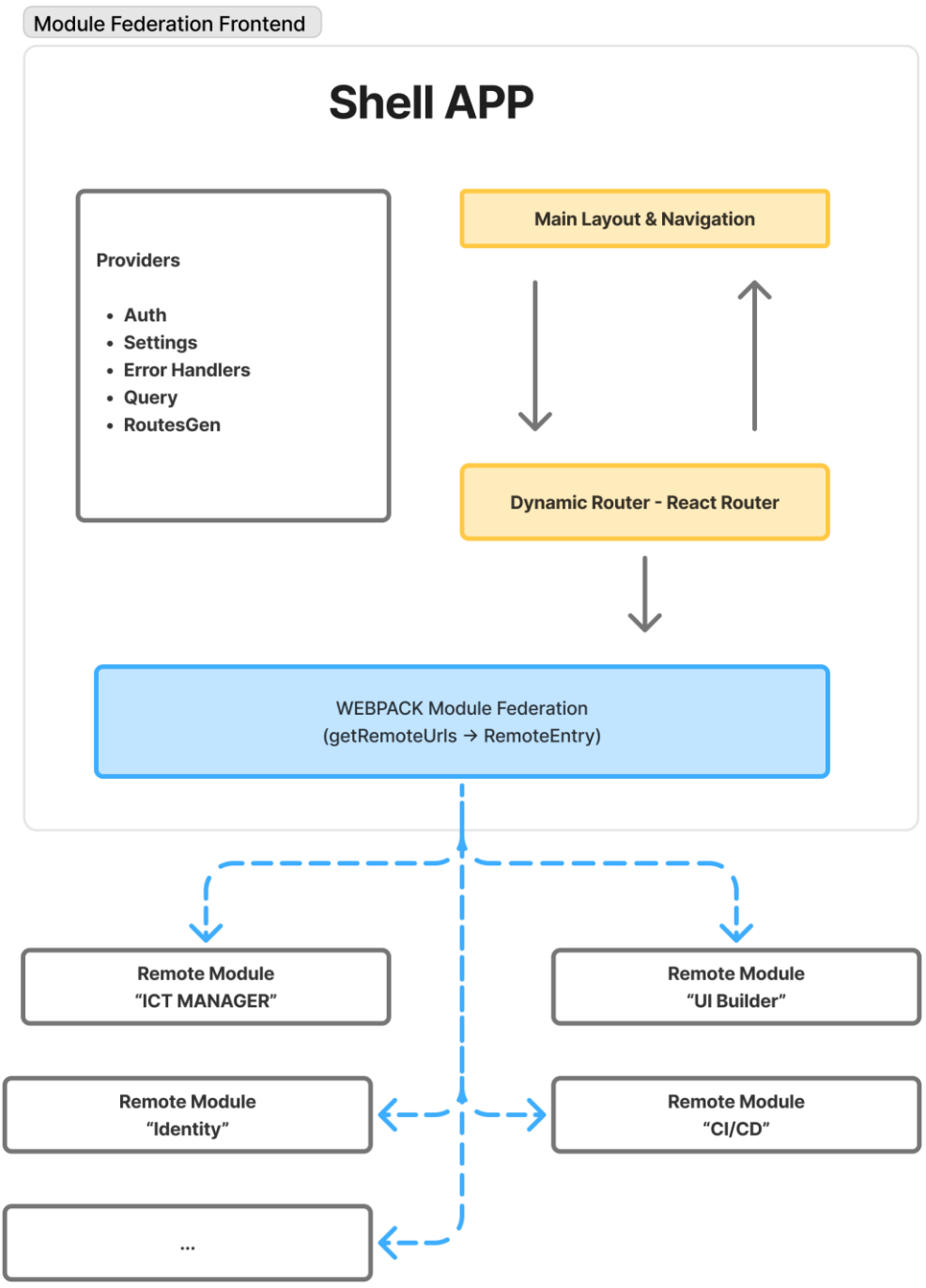

Figure 22: Fenics module federation overview

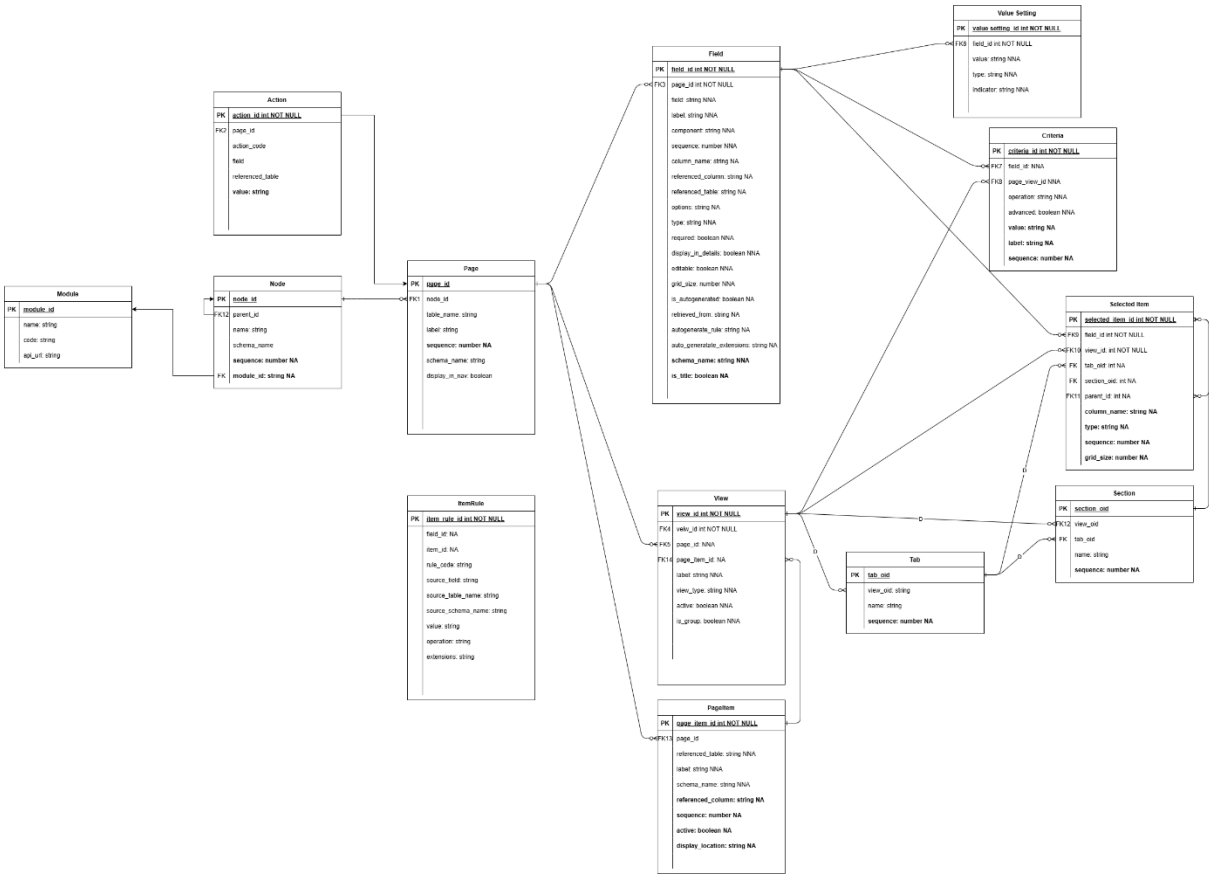

Figure 23: ER diagram of the Configuration schema

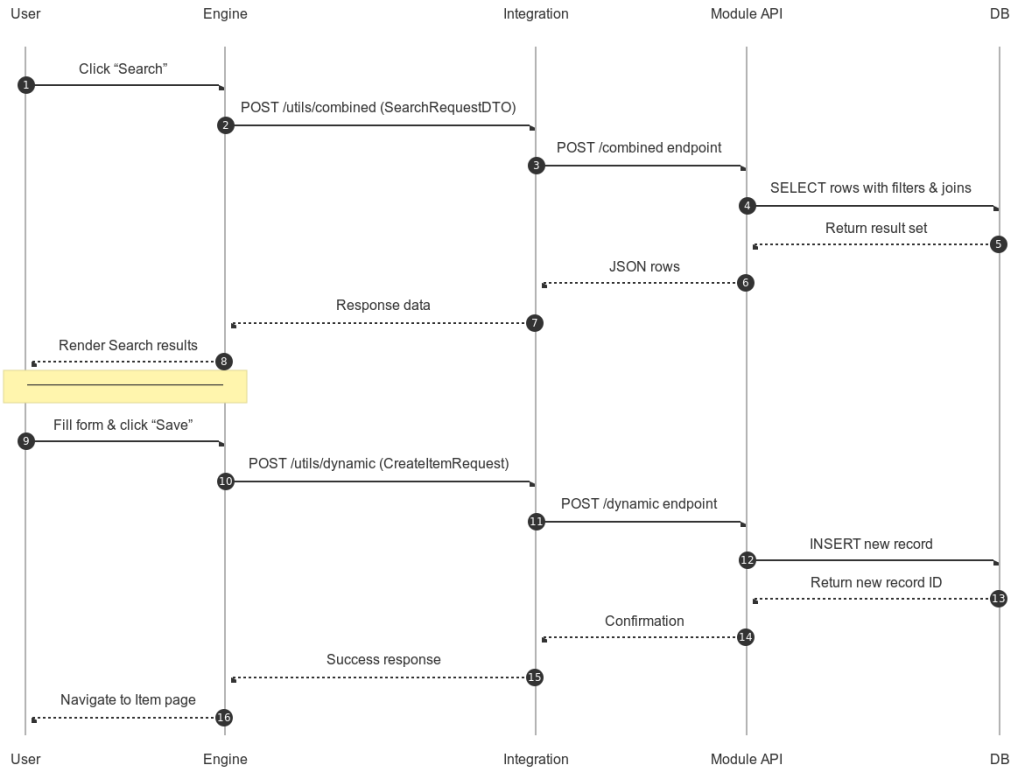

Figure 24: Sequence Diagram: Filter and Mutate requests

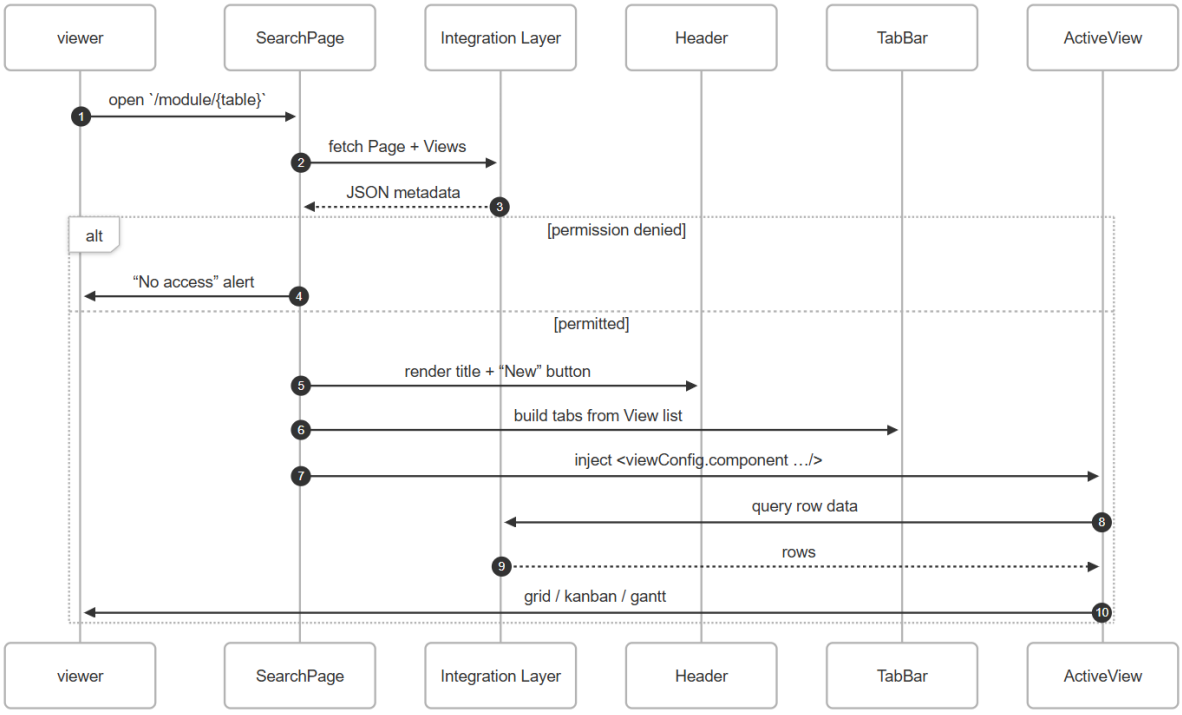

Figure 25: High-level flow Search Page

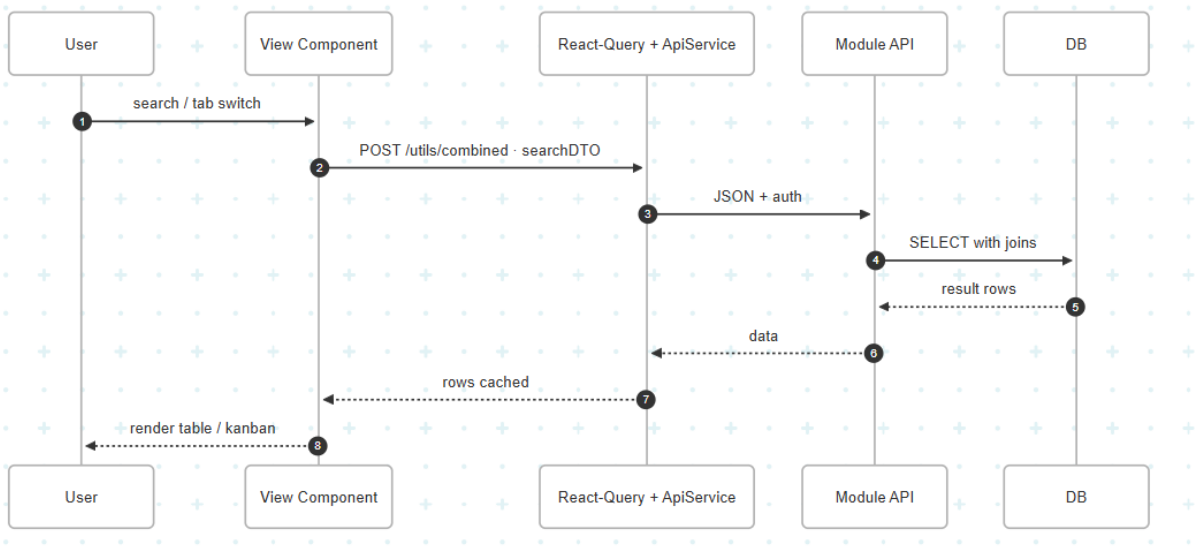

Figure 26: Sequence Diagram Update Item