INTELI

INSTITUTE OF TECHNOLOGY AND LEADERSHIP

COMPUTER ENGINEERING

Restructuring of offers and capital call screens

Course Conclusion Paper

Giovanna Rodrigues Araujo

São Paulo – São Paulo

2025

INTELI

INSTITUTE OF TECHNOLOGY AND LEADERSHIP

COMPUTER ENGINEERING

Giovanna Rodrigues Araujo

Restructuring of offers and capital call screens

Course Conclusion Paper submitted to the Institute of Technology and Leadership - Inteli as a partial requirement for obtaining the degree of Bachelor in Computer Engineering.

Supervisor: Rafael Will Macedo de Araújo - Computer Science PhD

São Paulo – São Paulo

2025

Abstract

Building upon the structural redesign proposed in the previous work—where the Shareholders by Capital Call screen was modularized into "Boletagem" and "E-mails" to support new features like email reminders—this study details the complete development lifecycle of that solution. It covers all phases involved in bringing the redesigned interface to production, including technical planning, implementation, iterative testing, and approval processes. The document outlines how design decisions were translated into code, how the integration was handled, and how automated and manual tests ensured reliability and performance. Emphasis is also placed on the validation and homologation stages, during which feedback from stakeholders shaped the final version of the system. Ultimately, this work demonstrates how structured planning and development practices contributed to a more scalable, maintainable, and user-friendly system architecture.

List of Figures

List of Tables

Contents

1

Technical and Structural Foundations

1.1 Structural Mapping

Documenting the structure of a technology project is essential to ensure clarity, organization, and alignment among all stakeholders involved in development. This documentation outlines how the project is organized — including folder architecture, data flows, division of responsibilities, and adopted standards — enabling new developers to integrate more quickly and allowing technical decisions to be made based on a solid understanding of the system. Furthermore, it reduces reliance on tacit knowledge, facilitates code maintenance and scalability, and contributes to standardization across different teams or projects. Without this documented foundation, there is a greater risk of inconsistencies, rework, and loss of productivity over time.

1.1.0.1 Tools

React

The project uses React, a JavaScript library focused on building user interfaces based on reusable components. Its declarative approach and use of the Virtual DOM provide a more efficient and high-performance development experience. Additionally, its large community, robust documentation, and integration with various libraries make React a solid choice compared to alternatives such as Vue or Angular, especially in projects that require flexibility and broad market adoption.

Webpack

Webpack is adopted as the module bundler, allowing the transformation and optimization of JavaScript, CSS, images, and other assets into production-ready bundles. It offers great flexibility through loaders and plugins, as well as features like Hot Module Replacement (HMR), which accelerate development. Compared to other tools such as Parcel or Vite, Webpack stands

out for its maturity, widespread adoption in enterprise projects, and advanced customization capabilities, making it ideal for applications with specific and complex requirements.

Axios

For API communication, the team uses Axios, a Promise-based library that simplifies the execution of HTTP requests. Its simple syntax, support for interceptors, and more robust error handling compared to the native fetch make Axios a practical and efficient choice. Compared to other libraries such as superagent or ky, Axios stands out for its popularity, active community support, and ease of configuration in more complex scenarios, such as authentication and header control.

ESLint

Finally, the project uses ESLint as a linting tool to ensure code quality and consistency. It helps identify errors, enforce best practices, and maintain a consistent coding style among developers. Compared to other tools such as JSHint or StandardJS, ESLint offers greater customization capabilities, plugin support, and integration with TypeScript, making it the most complete and adaptable option for modern projects.

1.1.0.2 Navigation and Routes

In frontend development, routes refer to the defined paths or URLs that correspond to different views or components within a web application. Navigation is the mechanism that allows users to move between these routes, typically using links, buttons, or browser controls. Together, routing and navigation form the backbone of single-page applications (SPAs), enabling dynamic content updates without full page reloads. They are essential for creating a seamless user experience, organizing the application's structure, and managing state transitions. Properly implemented routes also allow for deep linking, browser history integration, and improved accessibility and user flow.

Route	**Title**	**Description**	**Parameters**
/dashboard	Dashboard	Screen containing charts for monitoring daily transactions, facilitating error detection and task execution.	-
/capital-call	CapitalCall	Redirects to the page responsible for all screens and actions related to capital calls.	**OfferId (required)**
/simulate-cash-transfer	TED Simulator	Route available only in testing environments, used for simulating TEDs and testing transaction functionalities.	-
/offerings	Offerings List	Listing of all registered offerings, which can be filtered by fund, share type, subscription type, etc.	**FundId (optional)**: used when it is necessary to filter offerings by fund.
/offerings/edit	Offering Edition	Form screen for editing the information of an already registered offering.	**OfferId (required)**
/offerings/subscription-commitments	SubscriptionCommitments	Page responsible for all screens and actions related to subscription bulletins.	**OfferId (required)**

Table 1 – Navigation routes

1.2 Pattern Mapping

1.2.0.1 Componentization

Componentization in front-end development is an approach that involves dividing an application's interface into smaller, reusable, and independent parts called components. Each component represents a functional unit of the interface, such as a button, a form, a product card, or even entire sections of a page. This practice is widely adopted in modern libraries and frameworks such as React, which provides native structures for creating and composing components.

The main advantage of componentization is code reuse, which reduces duplication and facilitates maintenance. Additionally, it promotes project organization and readability, allowing different parts of the application to be developed, tested, and updated in isolation. This also improves system scalability, as new features can be added with minimal impact on the rest of the code. Another important benefit is ease of testing, as well-defined components tend to be more predictable and easier to validate.

Deciding what should or should not be turned into a component involves observing repetition patterns and clearly defined responsibilities. A good indicator is when an interface fragment appears in more than one place or is likely to be reused in the future. It is also common to create components for elements that encapsulate specific logic (such as form validation or local state management) or that need to be isolated to facilitate testing. In general, the more cohesive and independent a part of the interface is, the more likely it is to be a good candidate for componentization.

Some of the elements to be componentized include a NotificationFloater and an Input Error Feedback message.

NotificationFloater

The NotificationFloater is used to provide visual feedback to the user, informing them of the outcome of actions performed in the system, such as success or error messages. Its implementation is based on the component of the same name from the Orquestra library — the BTG Pactual design system — along with state management that controls the visibility, message content, and notification type to be displayed. To avoid code duplication across multiple pages where the NotificationFloater is needed, it is recommended to create a reusable component (Code block 1.1) that encapsulates its structure and logic.

Code block 1.1 – Reusable NotificationFloater component in React (TypeScript)

```
1 // NotificationFloater.tsx
2 import React from 'react';
3 import { NotificationFloater as OrquestraNotificationFloater } from
    '@orquestra';
```

```
4
5  type NotificationType = 'success' | 'error' | 'info' | 'warning';
6
7  interface NotificationProps {
8    message: string;
9    type: NotificationType;
10   isVisible: boolean;
11   onClose: () => void;
12 }
13
14 const NotificationFloater: React.FC<NotificationProps> = ({
       message, type, isVisible, onClose }) => {
15   if (!isVisible) return null;
16
17   return (
18     <OrquestraNotificationFloater
19       type={type}
20       message={message}
21       isVisible={isVisible}
22       onClose={onClose}
23       autoDismiss={5000}
24     />
25   );
26 };
27
28 export default NotificationFloater;
```

Input Error Feedback

As the design system is still recent, it does not yet cover all error scenarios in the available components; for instance, some types of input do not have native support for error properties. To ensure appropriate user feedback in error situations, while maintaining visual consistency with the bank's design standards, it was necessary to develop a custom component that follows the same visual structure as the existing elements in the design system as demonstrated in the code block 1.2.

Code block 1.2 – InputErrorFeedback component

```
1  import React from 'react';
2  import { Icon } from '@orquestra';
3  import styled from 'styled-components';
4
5  interface InputErrorFeedbackProps {
```

```
  message: string;
}

const Container = styled.div`
  display: flex;
  align-items: center;
  margin-top: 4px;
  color: var(--orquestra-color-content-secondary);
  font-size: 12px;
`;

const ErrorIcon = styled(Icon).attrs({ name: 'x-circle', size: 16 })`
  color: var(--orquestra-color-feedback-error);
  margin-right: 4px;
`;

const InputErrorFeedback: React.FC<InputErrorFeedbackProps> = ({ message }) => {
  return (
    <Container>
      <ErrorIcon />
      <span>{message}</span>
    </Container>
  );
};

export default InputErrorFeedback;
```

1.2.0.2 CSS Leakage

CSS leakage occurs when styles applied to one element inadvertently affect other interface elements that should not be impacted. This typically happens due to the use of generic or global selectors, such as `div`, `p`, or `.container`, which lack a well-defined scope. In large projects or those involving multiple developers, this kind of interference can lead to hard-to-trace side effects, such as unexpected layout changes, broken styles on specific pages, or conflicts between reusable components.

This type of issue compromises the maintainability and predictability of the code, making it more difficult to evolve the application without causing visual regressions. Moreover, it hinders component reuse, as components may behave differently depending on the context in which they

are used. In collaborative environments, CSS leakage can also lead to rework and productivity loss, since simple style changes may impact unforeseen parts of the application.

An effective way to prevent this problem is to scope styles using more specific selectors, such as unique IDs or class names per component. The use of SCSS (Sass) enhances this approach by allowing selector nesting, which helps keep styles organized and confined to the component's context. For example, as described in "Scalable and Modular Architecture for CSS" by Jonathan Snook (Snook 2012), encapsulating styles within a root ID or class using SCSS nesting, combined with SMACSS's modular naming conventions, ensures that styles are applied only to the intended elements, significantly reducing the risk of CSS leakage. Snook emphasizes that structured naming and categorization of styles into modules create predictable and reusable components, minimizing unintended side effects in complex projects.

Additionally, SCSS enables the use of variables, mixins, and functions, which contribute to visual standardization and facilitate code maintenance. With a well-defined structure, it is possible to create predictable, reusable visual components that are resistant to side effects caused by style leakage. This practice is especially important in applications with a design system or component libraries shared across projects.

1.2.0.3 State Management

State management in the front-end refers to how an application handles and shares information that changes over time — such as user data, loading statuses, form values, shopping cart items, and more. The "state" represents the current condition of the interface and application data, and managing it correctly is essential to ensure that the interface responds coherently to user actions and data changes.

The importance of state management lies in the application's predictability and consistency. When state is well-managed, the interface behaves reliably, making maintenance, scalability, and user experience easier to achieve. As highlighted by Donvir et al. (Donvir 2024), structured state management techniques, such as using libraries like Redux or native solutions like the Context API, enable different parts of the application to share information in an organized manner, minimizing bugs and enhancing code organization. This structured approach eliminates reliance on ad hoc solutions, promoting a predictable and robust data flow in modern web and mobile applications.

However, managing state can be challenging, especially in larger applications. As the number of components grows, keeping state synchronized among them becomes more complex. Common issues include "prop drilling" (when data must be passed through many component levels), difficulty tracking where and why state changes occurred, and data duplication across different parts of the application.

To address these challenges, React offers the Context API, a native tool that allows

state to be shared among components without the need to manually pass props through the entire component tree. With Context, it is possible to create a "global context" that stores data accessible by any component that needs it, resulting in cleaner code and reduced coupling between application parts. The Context API is especially useful for state that needs to be accessed by many components, such as themes, authentication, user preferences, or session data.

In the context of the current project, the application's complexity does not yet require external libraries such as Redux. The Context API, combined with good practices in state organization and scoping, has proven sufficient to meet the application's needs, offering a lightweight, native, and maintainable solution for data sharing among components.

Form State Management

In front-end applications, managing form state is a critical part of development, as it involves collecting, validating, and handling user-inputted data. Poor state management can result in unexpected behaviors, data loss, inconsistent validations, and a frustrating user experience. Additionally, forms are often sensitive areas of the application, as they handle important information such as personal data, credentials, or transactions.

Complexity increases as the form grows, with multiple fields, conditional validations, interactions between inputs, and specific business rules. Therefore, it is essential that field states are well-managed, ensuring that data remains synchronized with the interface, errors are properly handled, and application performance is not compromised.

An efficient solution for this scenario is the use of React Hook Form, a lightweight and powerful library for form management in React applications. It allows fields to be registered easily, tracks their state (such as values, errors, and validations), and integrates seamlessly with validation libraries. Unlike approaches based on `useState`, React Hook Form minimizes unnecessary re-renders, which significantly improves performance, especially in large forms.

Moreover, React Hook Form provides an intuitive API for handling synchronous and asynchronous validations, data submission, field resets, and dynamic form control. It also integrates well with both controlled and uncontrolled components, offering flexibility for different development styles. With it, developers can maintain cleaner, more modular, and easier-to-maintain code, while delivering a smoother and more reliable user experience.

1.3 Integration with the Backend

Integration with the backend in a frontend application is typically achieved through the creation of services, modular functions or classes responsible for handling HTTP requests to backend APIs. These services encapsulate the logic for interacting with endpoints, sending and receiving data, and converting responses into usable formats for the UI. By centralizing this communication logic, services promote code reuse, separation of concerns, and easier testing.

Error handling is a critical aspect of this integration; services often include mechanisms to catch and process errors, such as network failures, timeouts, or unexpected response formats. These errors are then surfaced to the user through feedback components and may trigger fallback behaviors or retries depending on the context. This structured approach ensures robust, maintainable communication between the frontend and backend layers.

To support the frontend interface, two backend projects will be utilized: the Offers Project and the Capital Call Project. Both are implemented following the BFF (Backend-for-Frontend) architectural pattern.

The BFF pattern is a software architecture approach that proposes creating a dedicated backend for each type of frontend (e.g., web, mobile). This allows each interface to have a backend tailored to its specific needs, improving efficiency, security, and decoupling between application layers. In this context, the BFFs act as intermediaries that adapt and expose only the necessary data to the frontend in the appropriate format.

1.3.1 Endpoints

Method	**Endpoint**	**Description**
GET	CapitalCall/ListAll	Returns all capital calls associated with a given offer.
PUT	InvestorEmail/Update	Updates one of the investor's registered email addresses.
POST	InvestorEmail/Add	Adds a new email address for the investor.
DELETE	InvestorEmail/Delete	Removes a registered email address from the investor.
GET	InvestorEmail/List	Retrieves all email addresses associated with a given investor.
GET	Shareholders/ByCapitalCall	Lists all investors imported in a specific capital call.
GET	EmailDownload/ByContribution	Downloads a Base64 ZIP file containing the email sent to the investor.
GET	Sender/SelectedForCapitalCall	Checks if a sender has been selected for a specific capital call.
GET	Sender/ListAll	Retrieves all email senders registered in the system.
POST	Receipt/SendToInvestors	Initiates the sending of subscription receipts to selected investors.
GET	Receipt/DownloadFiles	Downloads a Base64 ZIP containing the email receipt attachments.

Table 2 – Capital Call Project Endpoints

Method	**Endpoint**	**Description**
GET	Distributors/List	Returns all distributors registered in the system.
GET	HeadCoordinators/List	Returns all head coordinators registered in the system.
GET	Fund/InitialQuotaValue	Returns the initial quota value for the selected fund.
GET	Offer/NextNumber	Retrieves the next available offer number for a given fund.
GET	Fund/ByCgeCode	Retrieves fund details using its CGE identification code.
POST	Offer/Create	Inserts a new offer for the selected fund.

Table 3 – Offers Project Endpoints

2
Project Pipeline

An IT project pipeline refers to the structured sequence of stages that a technology project must go through from its initial conception to its final deployment and maintenance. This pipeline ensures that each phase of the project is properly planned, executed, validated, and delivered with quality and alignment to business goals.

2.1 Stages

2.1.1 Requirements Gathering and Specification

This is the initial phase where business needs are translated into technical requirements. It involves meetings with stakeholders, documentation of functional and non-functional requirements, and the definition of the project scope.

This phase, along with the system design, was developed during the first part of this project and has already been documented in detail.

2.1.2 System Design and Planning

Based on the specifications, the architecture and design of the system are defined. This includes database modeling, interface design, and the selection of technologies and tools.

As mentioned above, this stage has also been completed and detailed previously.

2.1.3 Development

This is the actual coding phase, during which developers implement the features and functionalities outlined in the specification document. This stage typically adheres to agile

methodologies, emphasizing iterative deliveries, continuous integration, and close collaboration between team members.

In this project, the development process is structured into two main interactive phases: the first includes the implementation of the email submission interface and the offer registration module; the second focuses on the billing interface.

To ensure quality and minimize rework, a mandatory requirement is established for transitioning from the development phase to the code review phase. Developers must present a document containing evidence that the system functions correctly, based on test scenarios created by the QA team. This measure is intended to prevent the handoff of code with basic errors and to avoid unnecessary back-and-forth between developers and testers.

2.1.4 Code Review

Once the system's functionality has been validated, the code is submitted for review. During this phase, the code is thoroughly analyzed and critiqued according to established best practices, including readability, maintainability, performance, and adherence to coding standards. The goal is to ensure high-quality, robust, and scalable code before it proceeds to the next stage.

2.1.5 Testing and Quality Assurance (QA)

Before the system is released, it undergoes a comprehensive testing process to identify and resolve any bugs, performance bottlenecks, or security vulnerabilities. This includes unit testing, integration testing, and system testing. The QA team executes these tests based on predefined scenarios to ensure that the system meets all functional and non-functional requirements.

2.1.6 Homologation (User Validation)

In this stage, the system is validated by the business users in an environment that simulates production. It ensures that the solution meets the business expectations and is ready for real-world use.

2.1.7 Deployment (Go-Live)

Once approved, the system is deployed to the production environment. This step may involve data migration, infrastructure configuration, and final validations.

2.1.8 Monitoring and Maintenance

After deployment, the system is monitored for performance and stability. Maintenance activities include bug fixes, updates, and continuous improvements based on user feedback.

The stages from development to deployment will be the focus of this phase of the project and will be detailed accordingly.

2.2 Importance

The IT project pipeline plays a critical role in ensuring the success and sustainability of software solutions. By following a structured and phased approach, it guarantees that each component of the system is carefully planned, developed, and validated before reaching the end user. This reduces the likelihood of critical failures in production and enhances the overall reliability of the system. Moreover, the pipeline fosters clear communication between technical and business teams, as each stage involves specific stakeholders and deliverables. It also enables better risk management by identifying issues early in the process, which helps avoid costly corrections later. Additionally, a well-defined pipeline supports accurate planning of timelines, budgets, and resources, making project execution more predictable and efficient. Finally, it promotes continuous improvement, as feedback collected throughout the stages can be used to refine both the product and the development process itself.

2.3 Development

2.3.1 Strategy of development

The SOLID principles, originally formulated within the context of object-oriented programming, have proven to be highly valuable in frontend development as well. As modern frontend applications grow in complexity, adopting architectural strategies that promote maintainability, scalability, and clarity becomes essential. SOLID—an acronym for five foundational design principles: Single Responsibility, Open/Closed, Liskov Substitution, Interface Segregation, and Dependency Inversion—offers a robust framework for achieving these goals. As noted by Gamma et al. (Gamma et al. 1994), the application of principles like Dependency Inversion and Interface Segregation, which are closely related to design patterns, enables developers to create flexible and reusable components, enhancing the scalability and clarity of frontend architectures in frameworks like React or Angular.

The Single Responsibility Principle (SRP) asserts that a module or component should have only one reason to change. In frontend development, this translates into designing components, services, or hooks that are focused on a single task. For instance, a component responsible for rendering a list (Code block 2.2) should not also handle data fetching (Code block 2.1). Instead, data retrieval can be delegated to a separate service or hook, allowing each part of the system to evolve independently and with greater clarity.

Code block 2.1 – GetHeadCoordinators method

```
//method/HeadCoordinatorsMethod.ts
import { useEffect, useState } from 'react';

export function getHeadCoordinators() {
  const [headCoordinators, setHeadCoordinators] =
     useState<string[]>([]);
  const [loading, setLoading] = useState(true);

  useEffect(() => {
    fetch('/Offer/HeadCoordinators')
      .then(res => res.json())
      .then(data => {
        setHeadCoordinators(data);
        setLoading(false);
      });
  }, []);

  return { headCoordinators, loading };
}
```

Code block 2.2 – HeadCoordinatorsSelect component

```
// components/HeadCoordinators.tsx
import { useState } from 'react';
import { getHeadCoordinators } from
   '../methods/HeadCoordinatorsMethod';
import { OrquestraSelect } from '@orquestra';

export function HeadCoordinatorsSelect() {
  const { headCoordinators, loading } = getHeadCoordinators();
  const [selected, setSelected] = useState<string | null>(null);

  const options = headCoordinators.map(bg => ({ label: bg, value:
     bg }));

  if (loading) return <p>Carregando  op   es...</p>;

  return (
    <div>
      <h3>Selecionar Fundo</h3>

      <OrquestraSelect
```

```
        options={options}
        value={selected}
        onChange={(option: { label: string; value: string }) =>
           setSelected(option.value)}
        placeholder="Escolha um fundo"
      />
    </div>
  );
}
```

The Open/Closed Principle (OCP) encourages developers to design systems that are open for extension but closed for modification. In practice, this means building components that can be extended through props, composition, or configuration rather than altering their internal logic. This approach is particularly useful in UI libraries like React or Vue, where reusable and customizable components are a cornerstone of scalable design systems.

This principle is exemplified by the implementation of the *NotificationFloater*, as discussed in the Componentization section on page 8.

The Liskov Substitution Principle (LSP) emphasizes that components or classes should be replaceable by their subtypes without affecting the correctness of the application. In frontend terms, this principle supports the use of polymorphic components or interfaces that can be extended or overridden without breaking existing functionality. For example, a generic input component (Code block 2.4) should be easily replaceable by a specialized version (Code block 2.5) without requiring changes in the consuming code.

Code block 2.3 – ActionButton interface

```
// interfaces/ActionButtonInterface.ts
export interface ActionButton {
   label: string;
   onClick: () => void;
}
```

Code block 2.4 – ActionButton component

```
// components/ActionButton.tsx
import { ActionButtonInterface } from
   '../interfaces/ActionButtonInterface';
import { OrquestraButton } from '@orquestra';

export function ActionButton({ label, onClick }:
   ActionButtonInterface) {
   return <OrquestraButton label={label} onClick={onClick}/>;
```

```
}
```

Code block 2.5 – DownloadButton component

```
// components/DownloadButton.tsx
import { ActionButton } from './ActionButton';

export function DownloadButton() {
   const handleDownload = () => {
       console.log('Baixando arquivo...');
   };

   return <ActionButton label="Download" onClick={handleDownload}
   />;
}
```

The Interface Segregation Principle (ISP) advocates for the use of small, specific interfaces rather than large, general-purpose ones. In TypeScript-based frontend projects, this principle is particularly relevant. It encourages the creation of narrowly defined interfaces that reflect the actual needs of each component, avoiding the pitfalls of bloated props or configuration objects that are difficult to maintain and understand.

Finally, the Dependency Inversion Principle (DIP) promotes the idea that high-level modules should not depend on low-level modules, but rather on abstractions. In frontend development, this can be achieved by decoupling components from direct data sources or APIs. Instead, components should rely on abstracted services or hooks that encapsulate the logic for data access. This not only improves testability but also enhances flexibility, as the underlying implementation can be changed without affecting the component's behavior.

The abstraction of the first example is demonstrated in the code block 2.6.

Code block 2.6 – HeadCoordinators Service

```
//services/HeadCoordinatorsService.ts
export const HeadCoordinatorsService() {
  async getHeadCoordinators {
    const res = await fetch('/Offer/HeadCoordinators');
    return res.json();
  }
};
```

Code block 2.7 – HeadCoordinators hook

```
//hooks/HeadCoordinatorsHook.ts
import { useEffect, useState } from 'react';
```

```
3  import { HeadCoordinatorsService } from
       '../services/HeadCoordinatorsService';
4
5  export function getHeadCoordinators() {
6      const [headCoordinators, setHeadCoordinators] =
           useState<string[]>([]);
7
8       useEffect(() => {
9             HeadCoordinatorsService
10             .getHeadCoordinators()
11             .then(setHeadCoordinators);
12      }, []);
13
14      return headCoordinators;
15 }
```

In summary, applying SOLID principles in frontend development leads to cleaner, more modular, and more maintainable codebases. These principles help teams build interfaces that are easier to test, extend, and reason about—qualities that are increasingly important in modern web applications. By embracing SOLID, frontend developers can align their practices with proven software engineering standards, resulting in more robust and sustainable solutions.

2.3.2 Environments

In software development, maintaining separate environments—such as Development (DEV), User Acceptance Testing (UAT), and Production (PROD)—is essential for ensuring quality and stability throughout the project lifecycle.

The DEV environment is used by developers to build and test new features in an isolated space, allowing for rapid iteration without affecting other stages. Once stable, the code is promoted to the UAT environment, where QA teams and stakeholders validate the system's behavior in conditions that closely resemble production. This stage is crucial for identifying bugs and confirming that the solution meets business requirements.

Finally, the PROD environment is the live system accessed by end users. It must be secure, stable, and performant, as any issues here can directly impact operations. Deployments to production are carefully managed to minimize risk.

Using multiple environments allows teams to work in parallel, reduce errors, and ensure that only thoroughly tested and approved code reaches end users.

2.3.2.1 Test Cenarios

Id	Type	Test Scenario Title	Test Scenario Description
TS0001	Capital Call	Selected capital call must always be the most recent	Ensure that the system always selects and displays the most recent capital call for the user.
TS0002	Capital Call	Download capital call model	Verify that the user can successfully download the capital call model from the system.
TS0003	Capital Call	Edit investor email	Test the functionality that allows the user to edit the email address of an investor.
TS0004	Capital Call	Send receipt	Check that the system can send a receipt to the investor's email address.
TS0005	Capital Call	Delete email	Ensure that the user can delete an investor's email address from the system.
TS0006	Capital Call	Add email	Verify that the user can add a new email address for an investor.
TS0007	Capital Call	Test content suite with General Options, Ticketing, and Email	Validate the content suite functionality with general options, ticketing, and email features.
TS0008	Capital Call	Validate email and receipt options via investor screen	Ensure that the email and receipt options are correctly displayed and functional on the investor screen.
TS0009	Capital Call	Validate pagination on new screen	Test the pagination functionality on the newly implemented screen to ensure it works correctly.
TS0010	Capital Call	Validate pagination on investor screen by capital call	Check the pagination functionality on the investor screen specifically for capital calls.
TS0011	Capital Call	Validate receipt resend	Verify that the system can resend a receipt to the investor's email address.
TS0012	Capital Call	Validate receipt send status: Not sent, Send error, Sent	Ensure that the system correctly displays the status of receipt sending as Not sent, Send error, or Sent.
TS0013	Capital Call	Verify invalid email validation	Test the system's ability to validate and handle invalid email addresses.
TS0014	Capital Call	Receipt view and download	Ensure that the user can view and download receipts from the system.
TS0015	Capital Call	Return via offer listing with selected fund - Capital Call	Verify that the user can return to the offer listing with the selected fund for capital calls.
TS0016	Capital Call	Return via offer listing with selected fund - Commitments	Ensure that the user can return to the offer listing with the selected fund for commitments.

Table 4 – QA Test Scenarios

2.4 Code Review

In collaborative software development projects, the practice of code review plays a central role in building a more reliable, sustainable, and standardized codebase. When multiple developers contribute to the same project, it is essential to ensure that all contributions adhere to consistent quality standards and align with the team's technical and business objectives. In this context, code review serves not only as a technical validation step but also as a vital communication tool among team members.

Code review is typically carried out through a Pull Request (PR) — a request for changes made in a specific branch to be reviewed before being merged into the project's main codebase. Another developer, who is not the author of the changes, takes on the role of reviewer. The reviewer assesses whether the code is correct, readable, efficient, secure, and in accordance with the team's established standards and best practices. Additionally, the reviewer may suggest improvements, highlight inconsistencies, or raise questions that help refine the proposed solution.

This process strengthens collaboration within the team by fostering an environment

where knowledge is naturally shared. By reviewing each other's code, developers gain a broader understanding of various parts of the system, reducing reliance on specific individuals and increasing the team's overall autonomy. It also promotes a stronger sense of collective ownership and responsibility for the quality of the software. When everyone participates in the review process, everyone shares accountability for the final outcome.

At the same time, code review helps keep the codebase clean, consistent, and maintainable — which is crucial in medium- and long-term projects where many contributors are involved and the system is under continuous evolution.

In summary, code review goes far beyond merely identifying errors — it is a process that enhances communication, technical quality, and team cohesion. When properly implemented, it becomes one of the cornerstones of a healthy and collaborative engineering culture.

2.5 Testing

Conducting tests in a User Acceptance Testing (UAT) environment that closely replicates the production environment is a critical step in the success of any technology project. This type of environment is carefully configured to simulate the same technical and operational conditions as the real environment used by clients, including infrastructure, integrations, data, and usage flows. The goal is to ensure that the developed functionality is validated in a context as close to reality as possible, allowing for an accurate assessment of its behavior and performance.

During this phase, tests are carried out based on predefined scenarios designed to cover all possible user interactions with the functionality. These scenarios are comprehensively crafted to include both the most common paths and exceptional situations, ensuring that the system responds appropriately to different usage patterns. This approach allows for the identification of bugs, inconsistencies, or unexpected behaviors before the system is released to production, significantly reducing the risk of negative impacts on end users.

Moreover, testing in an environment that mirrors production provides a more realistic and reliable validation. This increases the confidence of development, quality assurance, and business teams in the stability of the solution, making it easier to decide on its release. It also contributes to improving the user experience, as usability or performance adjustments can be made based on concrete observations gathered during testing.

Another important aspect is the early detection of issues related to integration with external systems, such as APIs, third-party services, or shared databases. In a well-structured UAT environment, these integrations can be tested under real conditions, preventing surprises after go-live. As a result, the acceptance process becomes more robust, efficient, and aligned with business expectations.

In summary, performing tests in a UAT environment that simulates production is an

essential practice to ensure the quality, reliability, and alignment of technological solutions with user needs. This strategy not only anticipates and resolves issues before final delivery but also strengthens confidence in the product and contributes to the overall success of the project.

Some examples of the issues mentioned in the PR were:

- The `downloadZip` function lacks error handling, which may lead to silent failures if problems occur during the download process.
- The `catch` blocks (around lines ~78 and ~173) do not capture the specific error object and fail to log detailed error information for debugging purposes.
- The `defineCallsList` function does not validate whether the `totalCalls` array is empty, which could result in unexpected behavior.

The solutions applied were:

- Adding a `try/catch` block to capture and handle potential errors during the creation and manipulation of the download link, displaying an appropriate message to the user.
- Modifying the `catch` blocks to capture the error object and log it to facilitate debugging.
- Adding a check for the existence and length of the array at the beginning of the function.

2.6 Quality Assurance

Quality Assurance (QA) is a strategic practice in technology projects, aimed at ensuring that the final product is reliable, functional, and aligned with both business expectations and user needs. Rather than being an isolated phase, QA is a continuous process that spans the entire development lifecycle — from planning to delivery — with a focus on preventing defects and promoting sound engineering practices.

Within this process, automated frontend testing plays a critical role. The frontend is the layer of the system that users interact with directly, and it must be validated not only visually but also in terms of behavior, logic, and integration with other parts of the application. By automating the verification of interactions, behaviors, and navigation flows, these tests help maintain a consistent user experience, even as the codebase evolves.

The presence of these tests in the development pipeline enables rapid detection of regressions, efficient validation of new features, and a significant reduction in the risk of production issues. They also foster greater confidence in each release, ensuring that the application continues to function as expected after every change. By covering different levels

of the application — from isolated units to complete user flows — automated frontend tests contribute to a broader and more reliable view of system quality.

Incorporating automated frontend testing into the QA process not only improves team efficiency but also reinforces a culture of quality throughout the project. It allows teams to focus on evolving the product with confidence, knowing that a solid foundation of continuous validation supports every delivery.

The most significant issue identified during testing was related to the recent update of the design system. One of the changes documented in the changelog had not been properly implemented, which resulted in the data grid failing to load all values on the page. To resolve this, it was necessary to consult the changelog and incorporate a specific tag that facilitates table scrolling, thereby restoring the expected functionality.

2.6.1 Component tests

Within the automated testing strategy, different types of tests are employed, each serving a specific purpose: unit tests, component tests, and end-to-end (E2E) tests.

This diversity of testing approaches allows for broad coverage of the application. Unit tests validate the behavior of isolated functions or methods, ensuring that small units of logic operate correctly. E2E tests simulate user behavior across complete application flows, validating the integration between different layers of the system. Component tests occupy an intermediate position: they verify whether interface components — such as buttons, forms, modals, and lists — function correctly in isolation, while still reflecting a user-centric perspective.

In the context of this project, component tests play a particularly important role. They offer a balance between granularity and realism, allowing for the validation of internal component logic, state interactions, and responses to various inputs. Additionally, they are faster and more stable than E2E tests, while providing a more realistic view of the user interface than pure unit tests.

There are different approaches to writing component tests, with the most common being the inside-out and outside-in strategies. The outside-in approach focuses on the observed behavior of the component, treating it as a “black box” without concern for its internal implementation. The inside-out approach, which was adopted in this project, is based on a detailed understanding of the component’s internal structure and behavior. This choice enables precise validation of aspects such as state management, internal function calls, conditional rendering, and event reactivity.

Choosing the inside-out approach brings significant benefits: it allows for greater control over the tests, facilitates the identification of specific failures, and provides a solid foundation for safe refactoring. Furthermore, it aligns well with component-based architectures, where

presentation logic and behavior are strongly encapsulated.

2.6.2 Cypress

Cypress is a test automation tool primarily aimed at web applications. Originally designed for end-to-end (E2E) testing, it has evolved to also support unit testing and, more recently, component testing. This versatility makes it a powerful choice for teams seeking to ensure the quality of their interfaces in an integrated and efficient manner.

In the context of component testing, the tool's distinguishing feature is its ability to execute these tests directly in the browser, within a real environment, providing an accurate visualization of the component's behavior. This is particularly useful for identifying rendering issues, styling problems, or user interactions that might go unnoticed in purely logical tests.

Moreover, Cypress offers an interactive experience during test development. It allows developers to see, in real time, what is being tested through a graphical interface (Figure 1) that displays the DOM, the executed commands, and the expected results. This greatly facilitates debugging and understanding the component's behavior under different conditions.

Another strong point of Cypress is its ability to simulate user interactions—such as clicks, typing, and focus events—in a manner that closely resembles real application usage. This enables validation not only of the component's appearance but also of its dynamic behavior. Combined with the ability to use browser inspection tools, Cypress becomes a comprehensive solution for ensuring that each interface component functions correctly in isolation, even before being integrated into the broader application. The code block 2.8 represents how to produce a test interacting with a button.

Code block 2.8 – Cypress spec

```
1  // cypress/component/Button.cy.jsx
2  import React from 'react';
3  import Button from '../../src/components/Button';
4
5  describe('Button component', () => {
6    it('should render with initial text and color', () => {
7      cy.mount(<Button />);
8      cy.get('button')
9        .should('have.text', 'Click me')
10       .and('have.css', 'background-color', 'rgb(128, 128, 128)');
           // gray
11   });
12
13   it('should change text and color when clicked', () => {
14     cy.mount(<Button />);
```

```
15      cy.get('button').click();
16      cy.get('button')
17        .should('have.text', 'Clicked!')
18        .and('have.css', 'background-color', 'rgb(0, 128, 0)'); //
            green
19    });
20  });
```

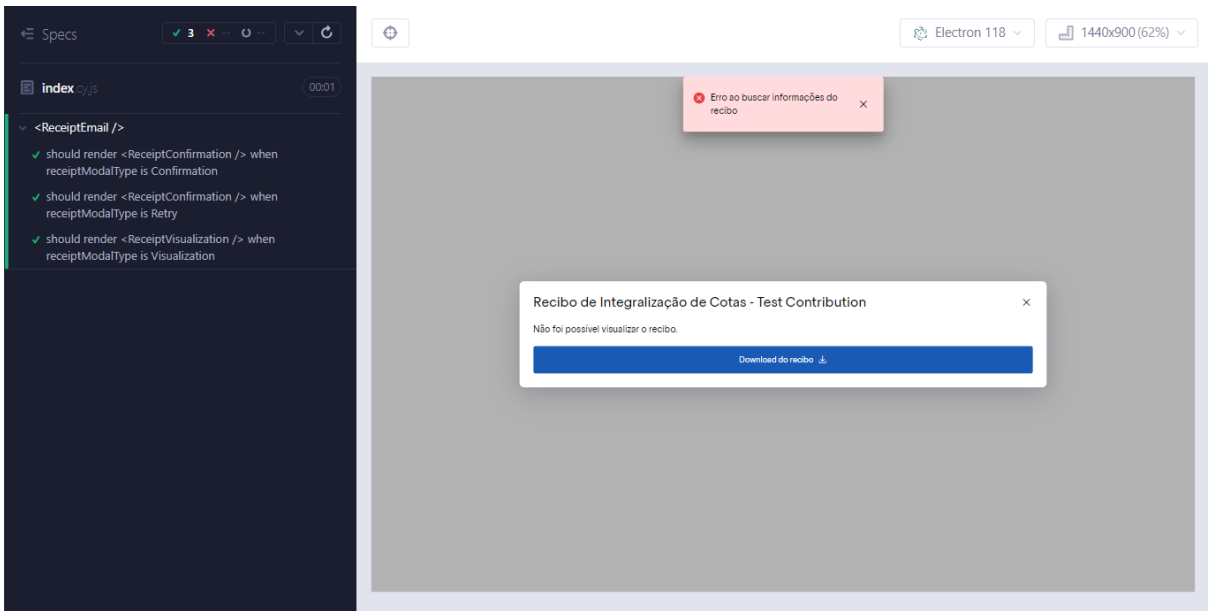

Figure 1 – Cypress interface

2.7 Homologation

Homologation represents the final stage in the software development lifecycle, during which the implemented solution is validated by key users or business representatives prior to its official release to the production environment. This process is essential to ensure that the system fully meets the functional requirements and practical needs of end users. Unlike technical testing performed by development or quality assurance teams, homologation focuses on the user experience, assessing whether the solution is functional, intuitive, and aligned with the organization's real-world processes.

The importance of homologation lies in its role as a final quality gate before a feature is made publicly available. Through this validation, it is possible to identify unexpected behaviors, inconsistencies, or failures that may have gone undetected during automated or integration testing. Furthermore, the process helps align business expectations with the delivered solution, increasing user confidence and promoting smoother adoption in operational workflows.

To ensure an effective homologation process, it is crucial to conduct it based on a structured testing script. The purpose of a homologation script is to guide users through the test execution, ensuring that all relevant scenarios are systematically and thoroughly evaluated. Without such planning, validation may become incomplete or overly subjective, compromising the reliability of the process.

Typically, a homologation script includes a description of each test scenario, the steps to be followed, required input data, the expected result, and a section to record the actual outcome. It often also contains pass/fail indicators for each test item, as well as a space for additional comments. This standardized format not only facilitates the execution of the homologation but also provides organized documentation of any issues encountered, thereby supporting future corrections and continuous improvement.

Due to scheduling constraints and the unavailability of key users during the development of this work, the homologation results could not be documented at this time. As a result, it was not possible to conduct a comprehensive evaluation of the system's behavior in a real usage scenario within the timeframe of this study. In the following semester, a continuation of this work will address the main findings from the homologation phase, detailing the issues encountered, the proposed resolutions, and the strategies adopted to monitor the solution in a production environment.

Bibliography

DONVIR, A. Application state management (asm) in the modern web and mobile applications: A comprehensive review. *arXiv*, abs/2407.19318, 2024. Citado na página 11.

GAMMA, E. et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1994. ISBN 0201633612. Citado na página 16.

SNOOK, J. *Scalable and Modular Architecture for CSS*. Vancouver: Snook, 2012. Citado na página 11.