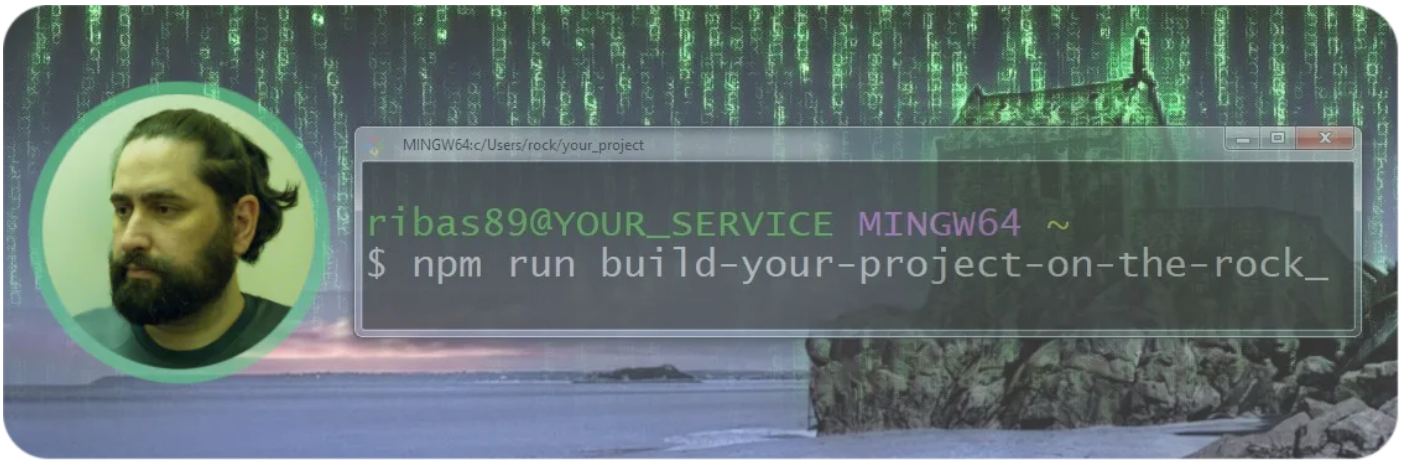


build your project on the rock

Summary

📌 Professional Summary

A journey of a thousand miles begins with a single step

My PC from **2007**

With this PC, I took my first steps building sites with **Adobe Flash and PHP**, a combo long gone, but one that set me on **the developer path** I plan to walk for **as long as I can**.

This is where this path has brought me so far. For **more details**, check the 🔬 Project Case Studies.

Sector	Company	Stack / Role	Duration
Startup	Neurogram	React Web	Nov 2023 - Nov 2024
Startup	XTeam	React Native	Sep 2022 - Jul 2023
Retail	Riachuelo	React Native	Aug 2020 - Aug 2022
Banking	Safra	AngularJS	Jun 2019 - Aug 2020
Banking	Itaú	Angular 2+	Mar 2018 - Jun 2019
Academia	UEPG	Java	Mar 2015 - Aug 2017
Insurance	Virtual	Delphi / MEAN	May 2013 - Mar 2015
Self-Employed	Freelance	Software and hardware technician	2004 - 2008

🎯 Career Objective

I’m seeking a role in an environment that **embraces transparency and communication**, where **clear processes** and **collaboration** can bring out **the best** in **my professional values and skills**.
My goal is to **join or help a team become high-performing**, deliver an **unforgettable** developer and **user experience**, and **improve the quality of life** for both the **team** and the **end users**.

⚖️ Professional Values

Through my previous roles, I’ve learned **values that go beyond the code**, values that **support the team** from concept to release. These are the values I bring to every team I join.

🤝 Transparency and communication

This is my **top priority.** I make sure the **team knows when** a task will be done, **how we plan** to develop it, and **why we approach** it that way.

If we can’t deliver on time, the next step is to **discuss openly what can be done** within the time we have.
Delaying bad news, keeping a task **“almost done”** for weeks, or **omitting the release date** are just **ways of avoiding accountability**. That is why we **commit to transparency**, because **when we share** issues early, **we gain** room to react and **adapt together**.

⏳ Process

Every development team has a process, even if it is **‘Go Horse’**, it is still a process.
We seek to **understand** how the **process works**, **document it** clearly, and **improve it** step by step.
This is the core of **Agile practice**: building **predictability** and achieving **sustainable delivery over time**.

🤩 User impact first

Frontend codebases **“die”** on average **in 5 years** through rewrites, framework shifts, or redesigns.
Backend codebases last on average **10 years** before major replacement or replatforming.

But a **lost user** is **lost forever**.

New features, analytics, redesigns, and refactorings **mean nothing** if the **user is gone**.
That is why we **prioritize user impact** above all, making sure every decision serves **what is best** for the **final user**.

🏠 Ownership mindset

I **treat the software** as if I am the **owner.**
That means **caring** about quality, stability, and **user experience**, **not** just moving **tasks to “done.”**

After years of dealing with **bad code**, I feel responsible for **long-term maintainability** and for always **leaving the codebase better than I found it**.

🍎 Healthy code

If you don’t take time to maintain the code, **the code will take the time** for you.

Maintenance is **always** required.

To save maintenance **costs**, I write **clean**, structured **code** from the **start**.
When that is **not possible**, I **refactor bad patterns**, improve readability, and **simplify structures**.
Sometimes a codebase needs **significant maintenance**, and the **only safe way** is for the **team** to acknowledge it and **schedule time** for it.

🏰 Resilient code

Healthy code is **not always resilient.**
A codebase can be **clean and organized**, yet still **collapse at runtime** under **unexpected cases or heavy load**.

A **mindful developer considers** the code, the hardware, the environment, the data flow, and **system behavior under pressure**.

That is why I **code** for the **worst case**: catching exceptions, validating input data, **creating fallbacks** instead of assuming the happy path, and **logging external processes**.

🖼️ Personal Interests

My interest in high quality code goes beyond work and deep into my personal interests. But I also have others, here is a list:

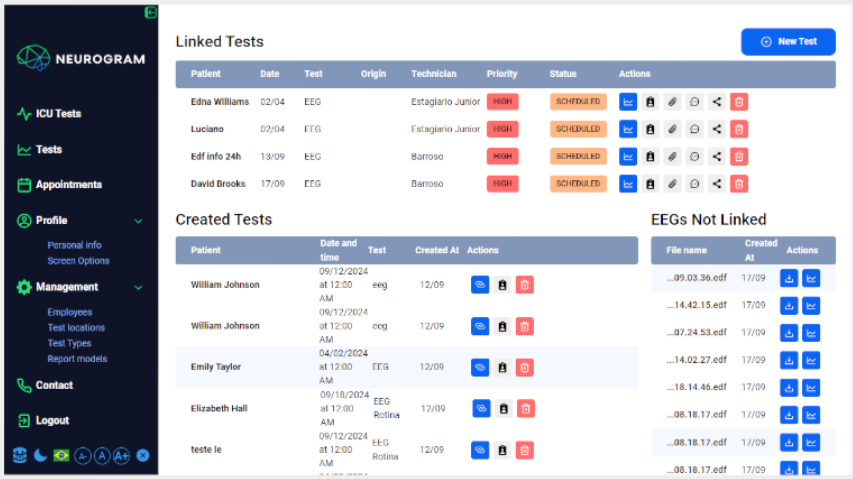
- **Blogging:** This is an old blog from 2013, now archived. I made several blogs over the years and plan to consolidate all of them soon.
- **Photography:** A personal hobby I keep improving over time.
- **Music:** I’m no musician, but I try to play some instruments. My favorite genres are hard rock and 2000s techno.
- **Game modding:** I love writing code for game mods and recently started publishing them. Here’s my collection for Project Zomboid check it out.
- **Nature:** The only reasons I ever leave home are to buy groceries, or stay in parks and campgrounds. I feel at peace being in nature, listening to the symphony of bugs and birds.

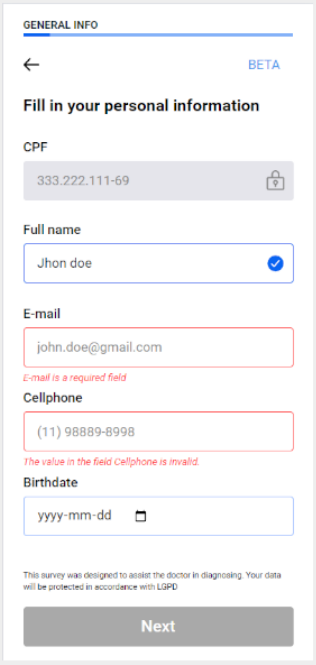
🔬 Project Case Studies

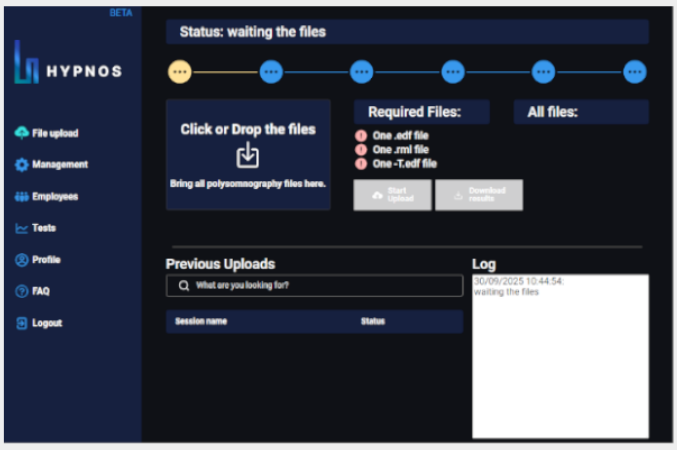
My career objective has been shaped by every project I’ve worked on.
Here is how I contributed to each of them, presented using the **STAR** (Situation, Task, Action, Result) approach.

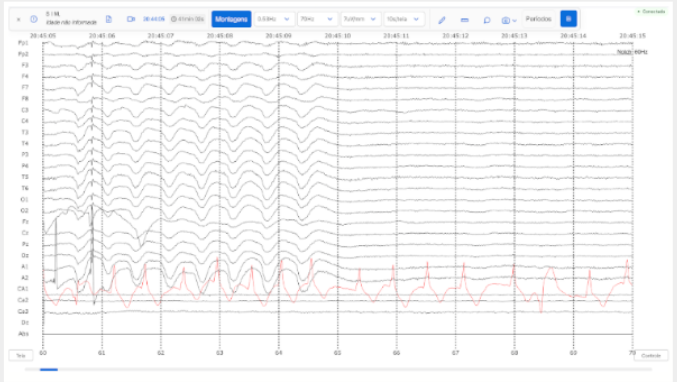
🧠 Neurogram

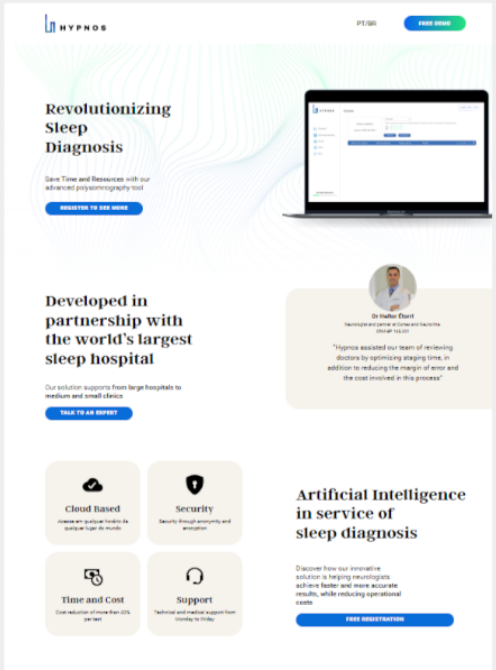
Senior Frontend Engineer - React.js 📅 Nov 2023 - Nov 2024











Neurogram Header

🌲 Project Treemap

Neurogram Treemap

🧱 Tech Stack

- **Backend and Cloud:** Firebase
- **Build Configuration:** Install, NPM, Vite, Vite Plugin JavaScript Obfuscator, Vite Plugin SVGR, Vite TSConfig Paths
- **Charts and Visualization:** Plotly JS, Plotly JS Basic Dist, Plotly JS Dist, React Minimal Pie Chart, React Plotly JS, Victory Native

- **Component Library and UI:** Bootstrap, Framer Motion, React Draggable, React Zoom Pan Pinch, React Device Detect, React Helmet Async, React Intersection Observer
- **Crypto:** Crypto ES, JSEncrypt
- **Data Fetching and Networking:** Axios, Axios Retry, React Use WebSocket, Retry
- **Date and Time:** Day JS, Moment
- **Documents and PDFs:** React PDF Renderer, Canvas2Image, HTML2Canvas, JSPDF, PDFJS Dist, Quill To PDF, React PDF
- **Encryption and Security:** Crypto ES, JSEncrypt
- **Forms and Validation:** Hookform Resolvers, React Hook Form, Yup, Yup Locales
- **Internationalization:** Brazilian Utils, Get User Locale, I18Next, React I18Next, Yup Locales
- **JavaScript Framework:** React, React DOM
- **Media and Players:** React Player, Video React
- **Mocking and Testing:** Mirage JS, MSW, Jest, ESLint, ESLint Plugin React Hooks, ESLint Plugin React Refresh, TypeScript ESLint Plugin, TypeScript ESLint Parser
- **Routing:** React Router DOM
- **State Management:** React Query
- **Storage:** LocalForage, LocalForage Session Storage Wrapper
- **Styling and Normalization:** Modern Normalize
- **Text and Editors:** Suneditor, Suneditor React
- **Utilities:** Buffer, Filt, Get User Locale, Lodash, Match Sorter, Randomatic, ShortID, Sort By, Stream Browserify, UUID
- **Visualization Enhancements:** React Zoom Pan Pinch, React Draggable, Plotly JS, React Plotly JS
- **Web Platform:** React DOM, React Native Web

🌟 STAR Cases

STAR Case - Fragmented codebase

Situation

The **previous codebase** combined Rails, React, Tailwind, GraphQL, and Docker across **multiple repositories** with **duplicated components** and **scattered configs**. This fragmentation caused **long onboarding** times, inconsistent standards, and clear signs of **vendor lock-in**.

Task

Considering the in-house team's **low seniority** and the **complexity** of the **project**, I set out to **reshape the project's stack**, replacing **dependency hell** with an architecture that was **simple**, maintainable, and sustainable, pursuing the **following objectives**:

1. **Standardize tooling** into a single, reliable workflow.
2. **Accelerate onboarding** so developers could focus on building instead of setup.
3. **Align stack with team skills** rather than forcing over-engineered solutions.
4. **Prevent future rewrites** by choosing technologies compatible with the existing backend.

Actions

My **first step** was to **assess** what could be **salvaged** from the frontend. After **several attempts**, I confirmed that **refactoring** would be **slower than starting fresh**. Between **Next.js** and **Vite**, the latter was chosen for its **faster builds**, **simpler configuration**, and better **alignment with the backend stack** based on **Firebase** and **Google Cloud Platform**.

Instead of leaving **developers** to **wrestle with** separate **Babel**, **PostCSS**, **Tailwind**, and **Docker** setups, I **collapsed dependencies** into one consistent **Vite configuration**. This **dramatically improved setup time** and enabled features like **obfuscation** and **vendor chunking** by default.

Below is the build configuration file I introduced:

```
import react from "@vitejs/plugin-react";
import tsconfigPaths from "vite-tsconfig-paths";
import obfuscatorPlugin from "vite-plugin-javascript-obfuscator";

type buildConfigOptions = {
  manualChunks?: boolean;
  vendors?: string[];
};

export const buildConfig =
  ({ vendors, manualChunks }: buildConfigOptions = {}) =>
  ({ mode, command }) => {
    const isProdBuild = ...;
    const noMinify = ...;
    const minify = ...;

    const vendorPath = [
      "jsencrypt",
      "i18n",
```

```
    "yup",
    "lodash",
    "dayjs",
    ...
    "lib-framework/src/fonts",
    "lib-framework/src/icons",
    "lib-framework/src/assets",
    "lib-framework/src/hooks",
    "lib-framework/src/tokens",
    "lib-framework/src/components",
    ...(vendors || []),
  ];

  const config = {
    define: {
      "process.env": {},
    },
    plugins: [
      react(),
      tsconfigPaths(),
      obfuscatorPlugin({
        apply: () => isProdBuild,
        ...
      }),
    ],
    build: {
      minify,
      rollupOptions: {
        treeshake: true,
        output: {
          manualChunks(id: any) {
            if (!manualChunks) {
              ...
            }

            for (const vendor of vendorPath) {
              ...;
            }

            return ...;
          },
        },
```

```
        },
      },
      ...

    return config;
  };
```

Results

1. **Build security improved**: JavaScript obfuscation protected intellectual property and reduced reverse-engineering risks.
2. **Bundle performance optimized**: **vendor chunking** and **tree-shaking** in the Vite configuration reduced payload size and improved runtime efficiency.
3. **Maintenance costs lowered**: **collapsing** scattered **configs** into a **single pipeline** simplified upkeep and **reduced time wasted** troubleshooting environment inconsistencies.
4. **New projects bootstrapped quickly**: standardized Vite setup enabled starting **fresh projects in minutes** with all necessary configurations **ready to use**.
5. **Onboarding accelerated**: environment setup time dropped from **5 days to 5 minutes**, making it straightforward for developers of **any seniority** level to start **coding immediately**.
6. **Team focus regained**: developers shifted attention **back to delivering features** instead of **resolving** fragmented build and config **issues**.

STAR Case - Unreliable Deployments

Situation

The deployment pipeline was entirely **controlled by the consultancy**, including the **production and staging environments**. Deployments were **triggered automatically** with every change, **but the maturity** of the software and the team **was not ready** for such automation. As a result, **bugs were** introduced directly **into production**, **breaking the user experience** and creating unnecessary **troubleshooting overhead** for the team.

Task

Shift the **ownership** of the pipeline and environments **back to the company**, simplify the deployment, and give **full control** to the **in-house team**. The objective was to design a **fast, easy, and manual deployment flow** that would only run when a developer explicitly triggered it, **reducing accidental** breakages in **production**.

Actions

I **redesigned** the deployment **process** using the **minimum** resources and complexity **possible**. Since the company was part of the **Google for Startups** program, the infrastructure of choice was **Firebase**. To make deployments **simple and predictable**, I created a **GitHub Actions workflow** that consolidated all frontend projects **one codebase**. This ensured the process was manual, quick, and transparent, while **eliminating** the **hidden consultancy-owned** pipelines.

Below is the GitHub Action I authored to handle all frontend projects in one place:

```yaml
name: manual deploy

on:
  workflow_dispatch:
    inputs:
      project-name:
        type: choice
        ...
      build-type:
        type: choice
        ...
env:
  ...

run-name: ${{ inputs.build-type }} TO ${{ inputs.project-name }} AT $
        {{ github.event.repository.pushed_at }} WITH ${{ github.sha }}
jobs:
  deploy_firebase:
    runs-on: ubuntu-latest

    defaults:
      run:
        working-directory: ${{ github.workspace }}/proj-${{inputs.project-
        name}}/

    steps:
      ...
      - name: Deploy
        run: |
          curl -sL https://firebase.tools | bash
          firebase deploy --only hosting:target-${{inputs.project-name}} --
        token ... --project=project-${{inputs.project-name}}-DEV --config="../
        lib-framework/firebase.json"
```

Results

1. **Auditable and predictable deployments**: GitHub Actions provided **structured logs** and **version traceability**, ensuring every release could be **tracked** and verified.
2. **Full control of environments**: **staging** and **production ownership** returned to the in-house team, preventing external bottlenecks and **restoring confidence** in releases.
3. **Independence from external vendors**: embedding the deployment pipeline internally removed reliance on **opaque consultancy infrastructure** and secured long-term ownership.
4. **Production incidents virtually eliminated**: replacing **consultancy-controlled auto-deploys** with **manual GitHub Actions workflows** reduced the risk of pushing unstable code directly into production.
5. **Simplified release governance**: a **single reusable workflow** handled all frontend projects, reducing coordination overhead and increasing **team-wide transparency**.
6. **Streamlined deployment process**: releases became a **two-click manual action**, intentionally designed to be **easy and reliable for any seniority level** on the team.

STAR Case - Code duplication

Situation

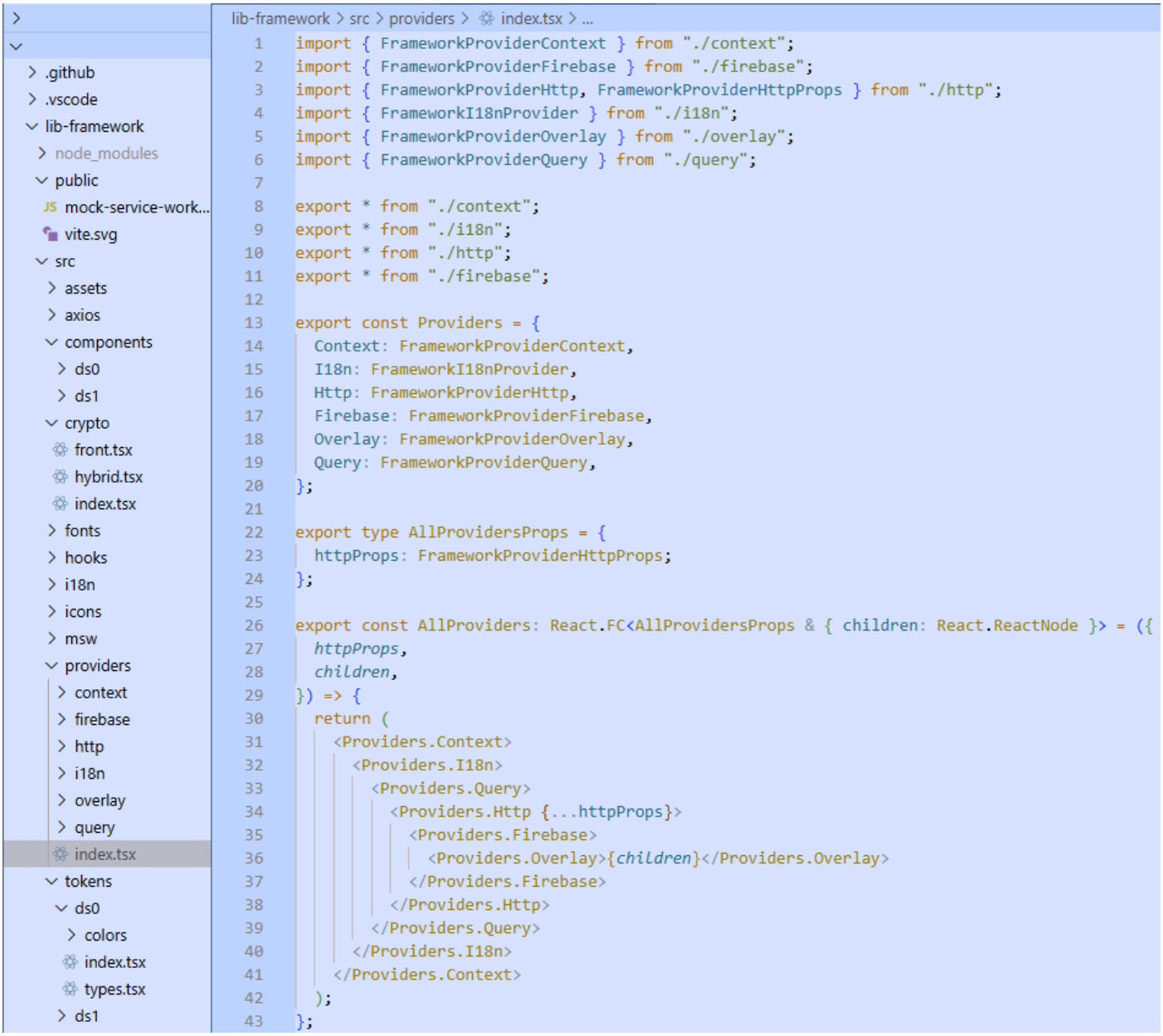
Multiple projects implemented the **same logic** for API calls, encryption, i18n, and UI components. This led to **frequent code duplication**, inconsistencies between projects, and bugs caused by drift in how core features were handled.

Task

Eliminate duplicated logic by creating a **single framework** that standardized core features and could be **reused across all projects**. The goal was to ensure consistency, reduce maintenance overhead, and accelerate new project setups.

Actions

I authored a centralized internal library-framework that included:
1. A custom **axios layer** with interceptors and typed adapters.
2. A unified **crypto module** using a hybrid combination of **RSA** and **AES**.
3. **Mock Service Worker** patterns and mock data for consistent testing and development.
4. Preconfigured **project templates and setup files** to enable fast project creation. 5. Providers for **Firebase, context, i18n, overlay, and query handling**.
6. Reusable **UI components** and design tokens.



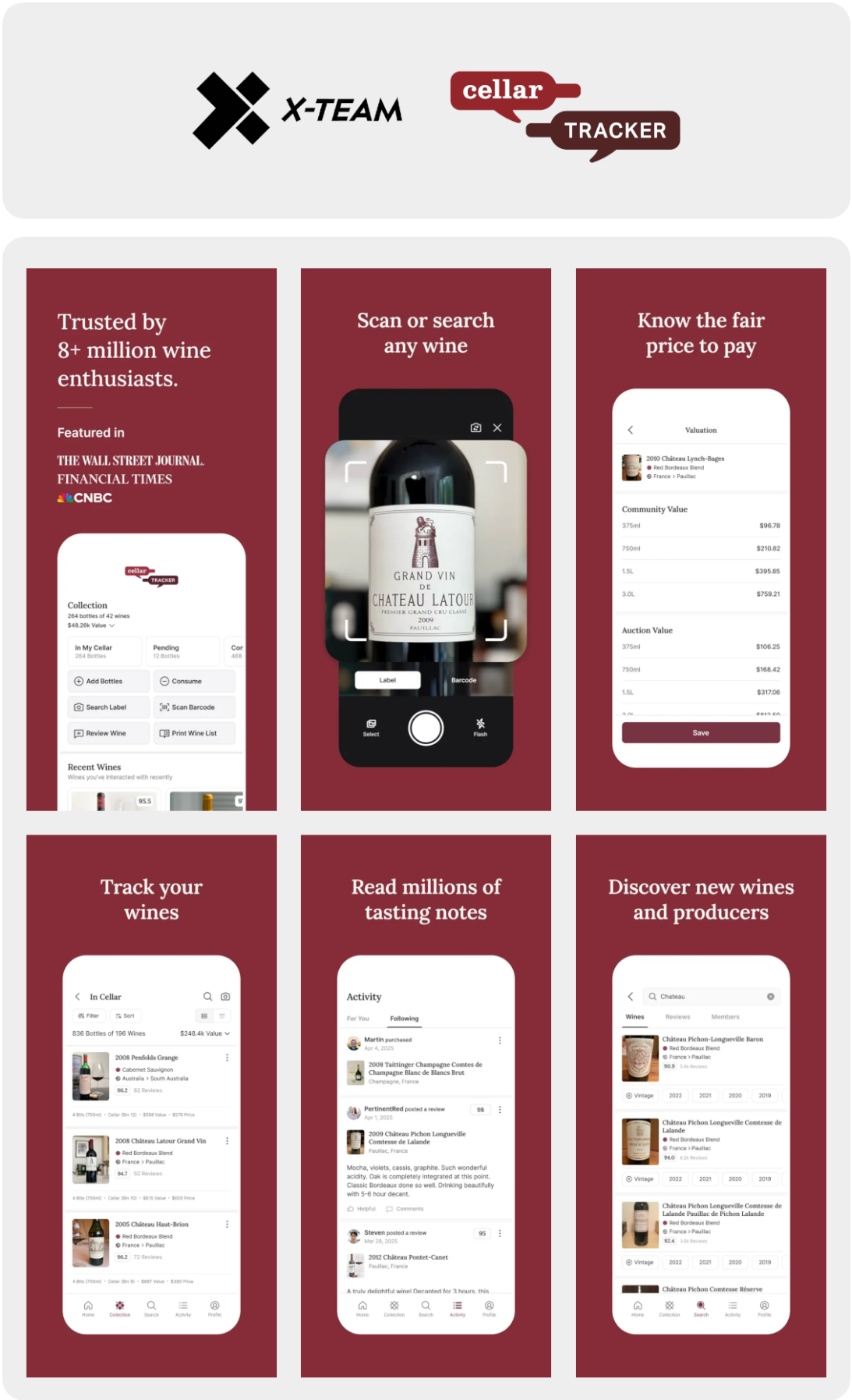
Lib Framework Providers

Results

1. **Bug rates reduced**: redundant and inconsistent **duplicate logic** was **eliminated**, **reducing** maintenance **issues**.
2. **Core features by default**: i18n, crypto, HTTP, and mocks were **included** by default in **every project**.
3. **New projects scaffolded in minutes**: a complete baseline setup was **instantly available** for new development.
4. **Single source of truth**: centralized frontend architecture **improved maintainability** and **increased team productivity**.

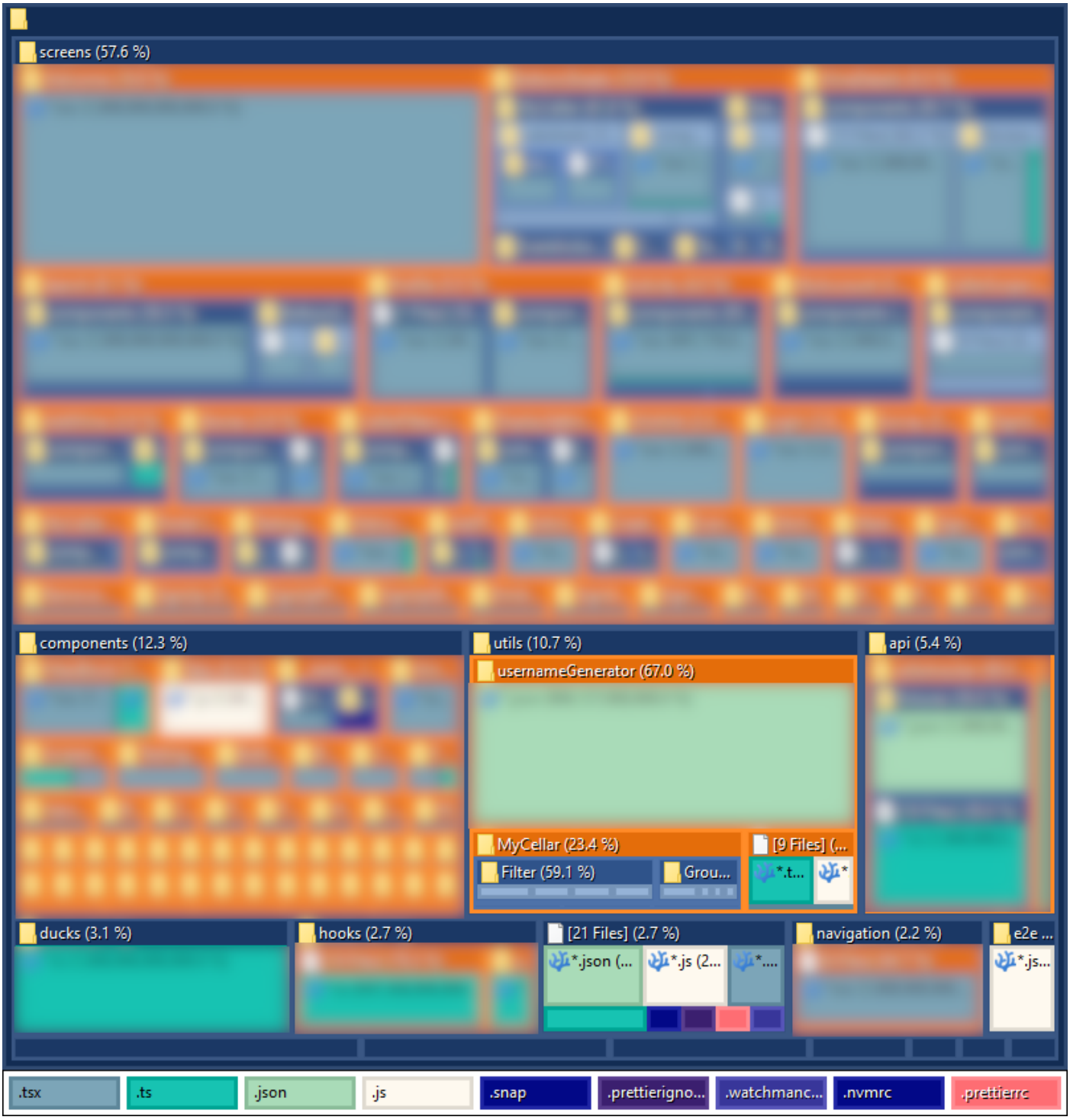
X-Team

Senior Frontend Engineer - React Native 📅 Sep 2022 - Jul 2023



CellarTracker Header

🌲 Project Treemap



CellarTracker Treemap

🧱 Tech Stack

- **Analytics:** Mixpanel React Native
- **Animations:** Moti, Motify Interactions, React Native Reanimated
- **Assets Files:** Expo Asset, React Native Dynamic App Icon
- **Auth:** Expo Apple Authentication, Expo Auth Session, Expo Local Authentication

- **Build Configuration:** Copy Files From To, Expo Build Properties, Expo Dev Client, Patch Package, Postinstall Postinstall
- **Camera:** Expo Camera
- **Charts:** React Native Chart Kit, Victory Native
- **Clipboard:** React Native Clipboard
- **Config Plugins:** Config Plugins Detox, Config Plugins Dynamic App Icon
- **Crypto:** Expo Crypto
- **Data:** Apisauce, Axios Auth Refresh, Deepmerge, QS
- **Dates:** Date Fns
- **Deployment:** Expo Updates
- **Device:** Expo Application, Expo Constants, Expo Device, Expo Linear Gradient, Expo Linking, Expo Location, Expo Mail Composer, Expo Notifications, Expo Random, Expo Splash Screen, Expo Status Bar, Expo System UI, Expo Web Browser
- **Feature Flags:** Flagged
- **Fonts Icons:** Expo Fonts, Expo Google Fonts Arbutus Slab, Expo Google Fonts Inter, Expo Vector Icons, React Native Vector Icons
- **Forms Validation:** Formik, Hookform Resolvers, React Hook Form, React Native Formik, Yup
- **Images:** Expo Image, Expo Image Manipulator, Expo Image Picker
- **Internationalization:** Expo Localization, FormatJS Intl DateTimeFormat, FormatJS Intl GetCanonicalLocales, FormatJS Intl Locale, FormatJS Intl NumberFormat, FormatJS Intl PluralRules, Intl, React Intl
- **JavaScript Framework:** Expo, React, React Native
- **Markup Parsing:** BBob Plugin Helper, BBob Preset, BBob React, React Native Render HTML
- **Media:** Expo AV, Expo Barcode Scanner, Expo Media Library
- **Monitoring:** Reactotron React Native, Reactotron Redux, Redux Logger
- **Navigation:** React Navigation Bottom Sheet, React Navigation Bottom Tabs, React Navigation Drawer, React Navigation Elements, React Navigation Material Top Tabs, React Navigation Native, React Navigation Native Stack, React Navigation Stack
- **Performance:** Shopify FlashList
- **Search:** Diacritics, Fuse JS
- **State Management:** React Redux, Redux Persist, Redux Toolkit
- **Storage:** React Native Async Storage
- **Styling:** Gorhom Bottom Sheet, React Native Calendars, React Native Keyboard Aware Scroll View, React Native Masked View, React Native Multi Slider, React Native Pager View, React

Native Paper, React Native Paper Dropdown, React Native Safe Area Context, React Native Screens, React Native Segmented Control, React Native Tab View

- **Testing:** Detox, Detox Recorder, Jest, Jest Expo, React Native Testing Library, React Test Renderer, Testing Library Jest DOM, Testing Library Jest Native, Testing Library React, Testing Library React Native, TS Jest
- **TypeScript Linting:** ESLint, ESLint Config Prettier, ESLint Plugin FormatJS, ESLint Plugin Prettier, TypeScript
- **Web:** React DOM, React Native Web

🌟 STAR Cases

STAR Case - Performance Bottleneck

Situation

Before my involvement, the app was built only with a **dozen hard-coded entries**. This limited dataset masked scalability issues in the fetching and rendering logic. After I integrated the backend, the app received **thousands of real bottle registries**, and the existing implementation could not handle this scale, causing the app to **freeze during feching, filtering, grouping, and searching**.

Task

Re-architect how the app **handled large-scale data** to:
1. Build a **modular and maintainable architecture** for data and UI.
2. Enhance **user experience with smooth navigation and search**.
3. Ensure **accurate results across all features**.
4. Support **thousands of records without freezing**.

Actions

First I needed to **guarantee** the **data** was **loaded quickly and accurately**. For that **I reviewed** how **the app** was fetching and storing information **and found deeply nested loops, duplicated logic and recalculations on every action**. To **reorganize the data** fetching and storage flow, I **created consistent patterns** for how records were requested, saved and displayed. I also **introduced a preloading logic** that ensured data was available **before the UI** rendered. This **eliminated** unnecessary **reload cycles** and **gave users** a **faster and smoother** experience **from cold boot**.

After that I **addressed the UI data rendering code**. Many filtering and sorting elements **were duplicated and inconsistent** so I **refactored** them into **reusable components** which made the interface **easier to maintain and extend**. To improve the experience of browsing large

inventories I **introduced sectioned lists and infinite scroll** which reduced rendering cost and **gave users** a smooth and **responsive navigation**.

Below is one of the optimizations I introduced to the selectors.

```
- export const selectCustomFilters = (...) => {
-   ...
-   return {
-     locations: [
-       ...new Set(
-         cellar
-           .map(i => {
-             if (i.Holdings) {
-               return i.Holdings.map(holding => {
-                 if (holding.Locations) {
-                   return holding?.Locations.map(k => {
-                     return k.Location;
-                   });
-                 } else {
-                   return;
-                 }
-               }).flat();
-             }
-             return;
-           })
-           .flat()
-           .filter(i => typeof i === 'string'),
-       ),
-     ],
-     ...


+ const FilterPendingDataArray = (inCellarWines: InCellarWinePending[]) => {
+   const resultData = {} as FilterObject<PendingFilters>;
+
+   inCellarWines?.forEach?.(wine => {
+     const hasBottles = wine?.Purchases?.some?.(p => !!p?.Quantity);
+     if (!hasBottles) return;
+
+     addNewFilterDataItem(resultData, 'appellation', wine?.Appellation);
+     ...
+
```

```
+     wine?.Purchases?.forEach(p => {
+       if (!p?.Quantity) return;
+       addNewFilterDataItem(resultData, 'bottleSize', p?.Size);
+     });
+   });
+
+   const { appellation, bottleSize, country, masterVarietal, region,
        subRegion, type } = resultData;
+   [appellation, bottleSize, country, masterVarietal, region, subRegion,
        type].forEach(a => a?.sort?.(sortStringAsc));
+
+   const { vintage } = resultData;
+   [vintage].forEach(a => a?.sort?.(sortNumberAsc));
+
+   return resultData;
+ };
```

Results

1. **Better user experience**: responsive infinite scrolling and grouped lists improved navigation.
2. **Fast search and filtering**: thousands of entries could be queried without performance loss.
3. **Higher developer productivity**: reusable, modular UI components reduced duplication.
4. **Scalable architecture**: a foundation that supported new features and long-term growth.
5. **Smooth performance**: large datasets became responsive and near-instant to operate on.

STAR Case - Unstructured Backend Responses

Situation

After we managed to consistently **recover thousands of entries** across the app, the next challenge was **the lack of a proper model** to organize, sort, and feed data **into the UI**. The **core issue** was the **same as before**: the **data model** was **partially incorrect and hardcoded**. In addition, the **backend did not provide** a reliable way to **validate its payloads**, and in many cases **critical fields were missing**.

Task

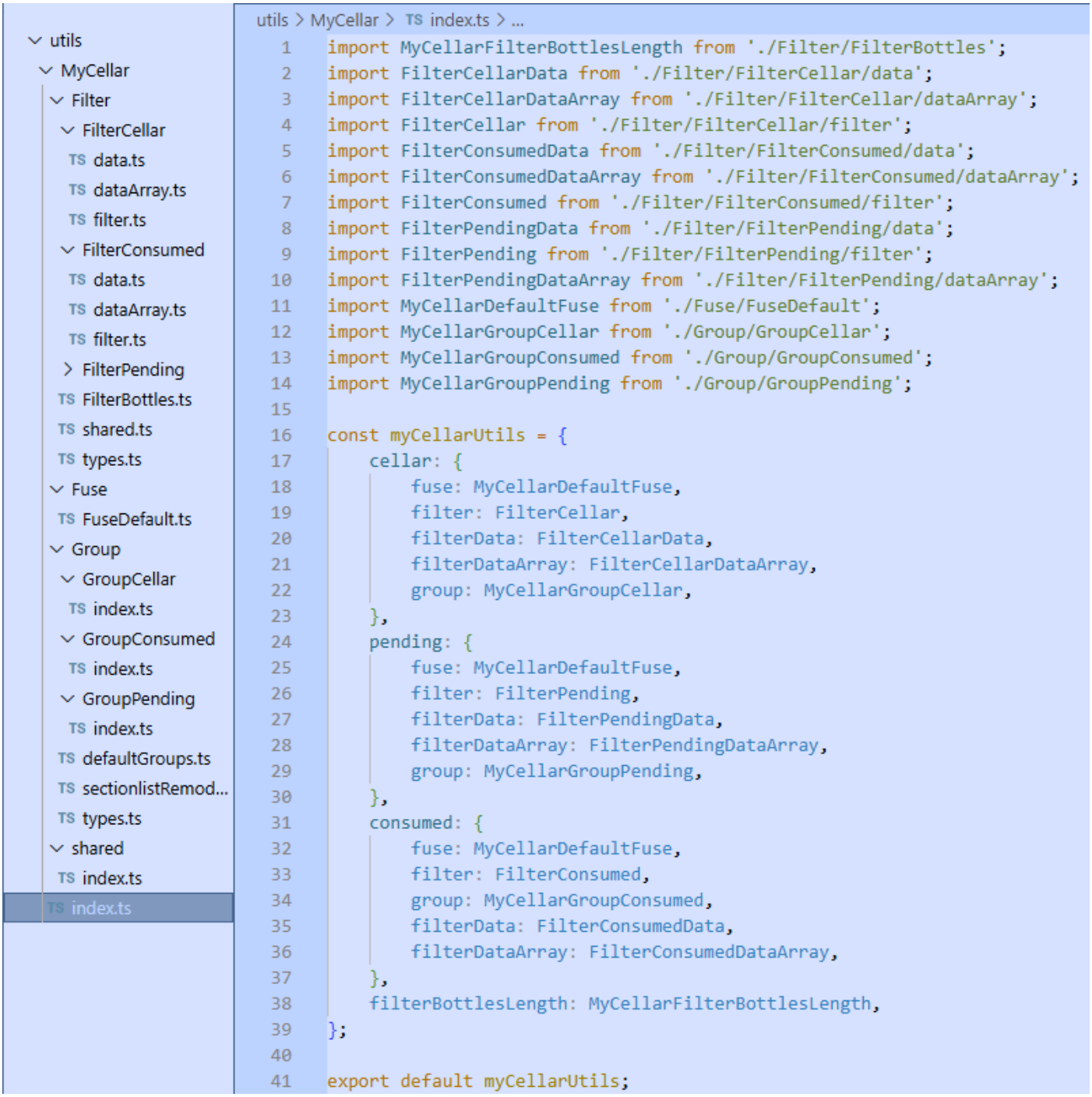
Create a **reliable way** to handle the app’s deeply nested and inconsistent **backend data** in order to:
1. Enable users to **quickly find and browse bottles** with accurate results.
2. Ensure the model could **scale to thousands of entries** without breaking.

3. Establish a **consistent foundation** for filtering and search.
4. **Resolve** issues caused by **missing and unreliable fields**.

Actions

To solve this I **restructured** the data handling **into a graph-structured** traversal model where each level of **information** (bottles, holdings, locations, bins) was treated as a **connected node**. This approach created a **navigable structure** where starting from a node like a **location** you could **immediately find** the **bottles** stored there and **from those bottles** trace back to **their vintage** or other attributes. This replaced **scattered nested loops** with a **clear and predictable flow**, making the model **easier to maintain, extend, and scale**.

This graph approach gave three major advantages:
1. **Consistency**: the same traversal logic powered cellar, pending, and consumed states, removing duplication and errors.
2. **Extensibility**: adding a new filter meant only extending traversal rules for a node, not rewriting entire loops.
3. **Traversal clarity**: instead of nested loops, each level of the data (wine → holding → location → bin) contributed in an organized way.



Transversal graph model

Results

1. **Extensible filters**: new filter types were added without breaking existing functionality.
2. **Faster and accurate search**: browsing and filtering across thousands of entries became smooth and responsive.
3. **Maintainable data model**: a graph-inspired approach organized unstructured responses into a predictable system.
4. **Reliable filtering**: users could apply filters consistently even when backend data was incomplete.

STAR Case - Missing Internationalization

Situation

The app serves a **global audience** of wine collectors who expect **multiple language support**. While there were some **early attempts at internationalization**, the application was **inconsistent and incomplete**. Many components still relied on **hardcoded English strings** for filters, chips, dropdowns, and error messages. This incomplete approach made the UI feel **disjointed and awkward**, blocking **full localization** and **limiting** the app's ability to deliver a **scalable, accessible international experience**.

Task

Establish a consistent internationalization **pattern** to:
1. Create a **scalable i18n foundation** that developers could apply uniformly across the app.
2. Ensure UI elements like filters, chips, and dialogs were **fully translation-ready**.
3. Fix prior inconsistencies and enable a **seamless multilingual experience** for end users.
4. **Replace** remaining **hardcoded strings** with localized messages.

Actions

I refactored the application to use a **consistent internationalization pattern**, replacing static strings with translated messages across components. To simplify adoption, I created hooks to make it easy to pull translations into any new component.

Below is the hook I introduced, which **encapsulated the logic** for pulling translation messages, formatting them with **react-intl**, and wiring them into **navigation flows**. This removed duplication and made internationalization extensible across filters, chips, bottom sheets, and dropdowns:

```
import {useNavigation, useRoute} from '@react-navigation/native';
import {useIntl} from 'react-intl';
import {useEffect, useRef} from 'react';

const useEventActionSheet = ({messages, title, params, onSelect}) => {
  const route = useRoute();
  const {formatMessage} = useIntl();
  const navigation = useNavigation();
  ...

  useEffect(() => {...}, [route?.params]);

  const openSheet = () => {
    const options = Object.keys(messages).map(key => {
```

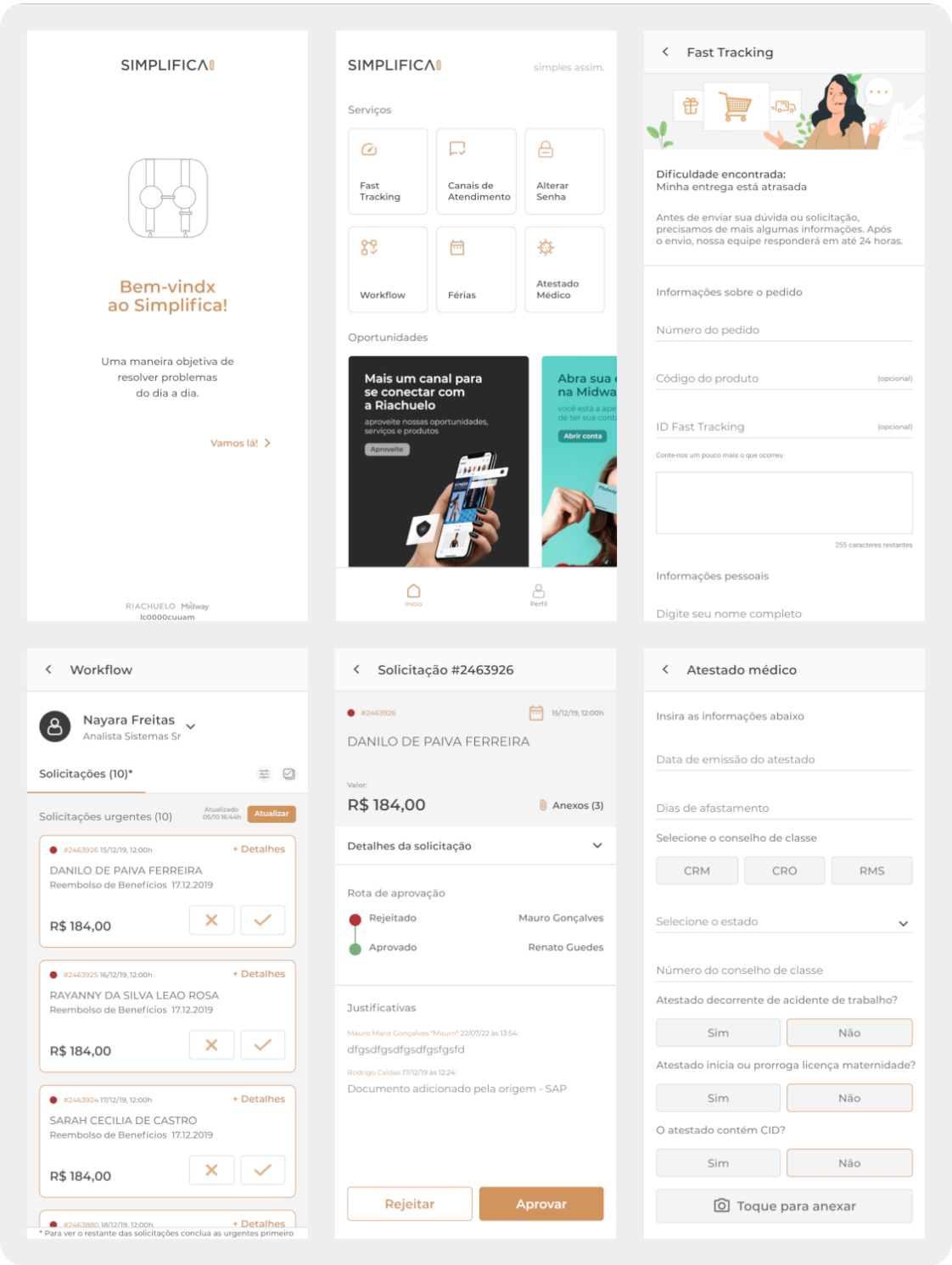
```
      const message = messages[key];
      return {
        label: formatMessage(message),
        value: message.value,
      };
    });

    navigation.navigate('EventActionSheet', {...});
  };

  return [openSheet];
};

export default useEventActionSheet;
```

Results

1. **Broader international reach**: multilingual support prepared the app to serve a global user base effectively.
2. **Consistent user interface**: standardized messaging eliminated awkward or disjointed UI patterns.
3. **Faster development workflows**: developers could add new components with built-in i18n support and less duplication.
4. **Scalable i18n foundation**: reusable hooks centralized logic and ensured consistent adoption across the app.
5. **Translation-ready UI**: all hardcoded strings were removed and replaced with localized messages.

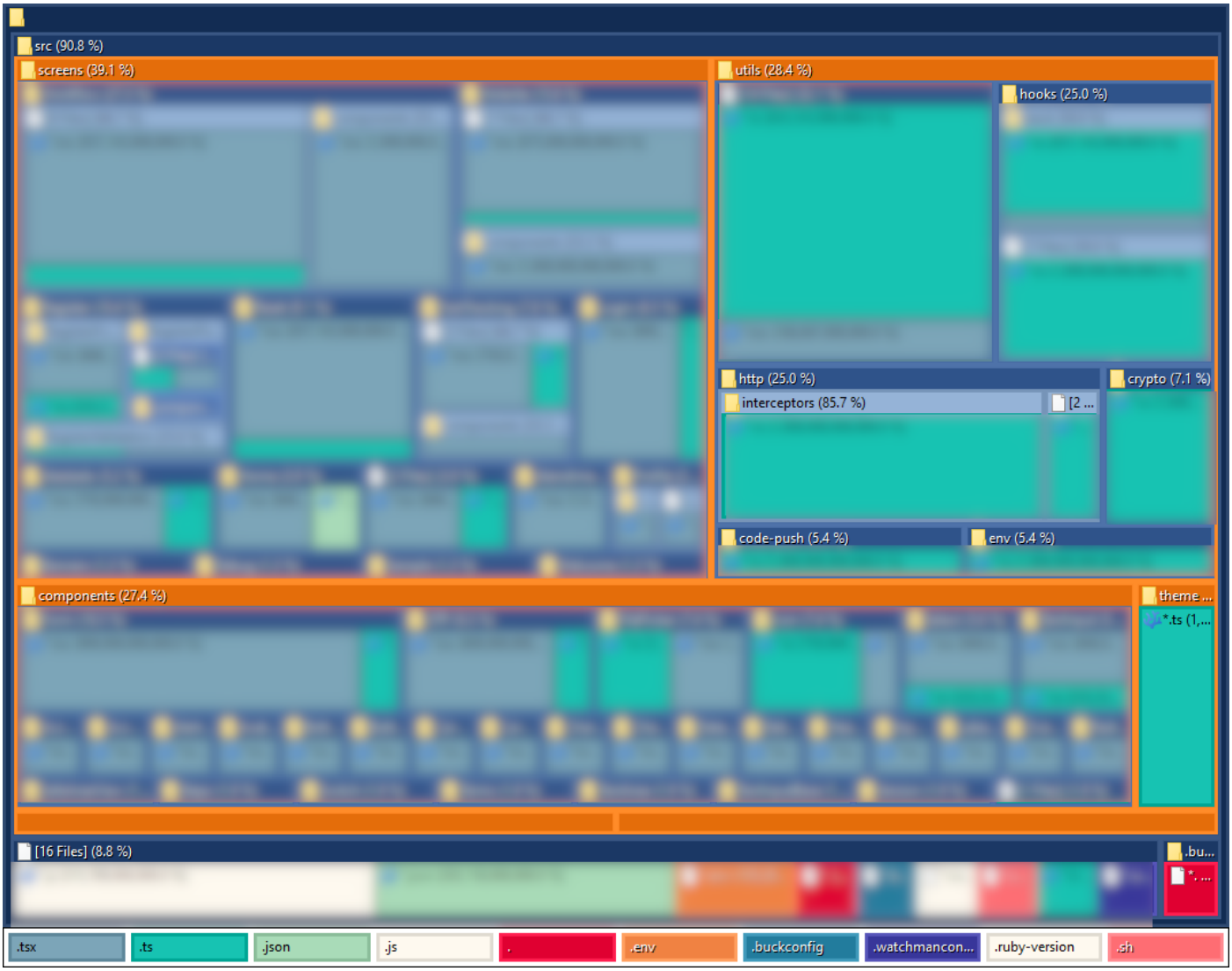
Riachuelo

Senior Frontend Engineer - React Native 📅 Aug 2020 - Aug 2022



Simplifica Header

🌲 Project Treemap



Simplifica Treemap

🧱 Tech Stack

- **Analytics:** React Native Firebase Analytics
- **Animations:** Lottie React Native, React Native Collapsible, React Native Modal, React Native Modal Datetime Picker, React Native Switch Toggle
- **Authentication:** React Native OTP Framework, RN Local Authentication, RN SMS Retriever
- **Build Configuration:** Babel Plugin Root Import, Metro React Native Babel Preset, React Native Config
- **Camera:** React Native Camera
- **Charts:** React Native SVG Charts
- **Crypto:** Crypto ES, JSEncrypt
- **Data:** Axios, Axios Cache Adapter, Axios Retry, Cheerio
- **Dates:** Day JS, React Native DateTimePicker

- **Deployment:** App Center Code Push, React Native Code Push
- **Device:** React Native Device Info, React Native Version Number
- **Documents and Files:** React Native Blob Util, React Native Document Picker, React Native File Viewer, React Native PDF, React Native Share
- **Encryption:** React Native Sensitive Info
- **Forms Validation:** Hookform Resolvers, React Hook Form, Yup, Yup Locale PT
- **Internationalization:** Brazilian Utils
- **JavaScript Framework:** React, React Native
- **Media:** React Native Image Pan Zoom, React Native Image Picker, React Native Image Slider Box, React Native SVG, React Native SVG Transformer
- **Monitoring:** Reactotron React Native, Reactotron Redux
- **Navigation:** React Navigation Native, React Navigation Native Stack
- **Permissions:** React Native Permissions
- **Pickers UI:** React Native Picker, React Native Picker Module
- **State Management:** Redux Thunk, Redux Toolkit
- **Storage:** React Native Async Storage
- **Styling:** React Native Keyboard Aware Scroll View, React Native Lottie Splash Screen, React Native Ratings, React Native Safe Area Context, React Native Screens, React Native String Style
- **Testing:** React Test Renderer
- **TypeScript Linting:** ESLint, React Native ESLint Config, TypeScript, TypeScript ESLint Parser, TypeScript ESLint Plugin
- **Utilities:** Day JS, Lodash, Randomatic

🌟 STAR Cases

STAR Case - Insecure Client-Server Communication

Situation

The company launched a **security initiative** requiring **protection** of confidential **data** on rooted devices, emulators, and man in the middle attacks. It required to **strengthen security beyond HTTPS** and ensure **compliance without** adding **latency** or **disrupting** the **user experience**.

Task

Implement an additional **security mechanism** to **protect data** exchanged **between React Native and Java endpoints**. **Ensure** key management, payload **encryption**, and **compatibility with existing systems**.

Actions

Instead of relying solely on HTTPS, I implemented an **RSA + AES model**, where each request was **encrypted and decrypted** on both **React Native client and Java Server**.

On the **frontend**, I created a **TypeScript module** that generated AES keys per session, encrypted them with the public RSA key, and transparently handled encryption and decryption through **Axios interceptors**.

On the **backend**, I developed a **Java utility library** to mirror this logic. It managed **key exchange**, **payload decryption**, and **response re-encryption**, ensuring perfect alignment between platforms.

Below is **Axios interceptor** to encrypt payloads, creating a transparent and reusable security layer across the entire app.

```
import { AxiosRequestConfig, AxiosResponse } from 'axios';
import { ... } from '~/utils/crypto';
import { ... } from '~/utils/env';

import { Interceptor } from './types';

const onRequest = async (
  config: Promise<AxiosRequestConfig>,
): Promise<AxiosRequestConfig> => {
  const oldConfig = await config;

  const { data: value } = oldConfig;

  const publicKey = await ...();

  const newConfig = {
    ...oldConfig,
    data: {
      data: encrypt({
        publicKey,
        value,
      }),
```

```
    },
  };

  return newConfig;
};

export const hybridEncryptInt: Interceptor = {
  onRequest,
};

const onResponse = async (
  response: Promise<AxiosResponse>,
): Promise<AxiosResponse> => {
  const _response = await response;

  try {
    const privateKey = await ...();
    const data = decrypt({
      privateKey,
      value: _response.data.data,
    });
    return { ..._response, data };
  } catch (err) {
    console.error(`[HYBRID DECRYPT ERROR]`, err);
    return _response;
  }
};

export const hybridDecryptInt: Interceptor = {
  onResponse,
};
```

Below is the **Java counterpart** that handled the **server-side decryption and encryption**.

```
package hybridCrypto;

public class RSACrypt implements Serializable {

    public static void generateKeys() {
        try {
            KeyPairGenerator keyGen = KeyPairGenerator.getInstance("RSA");
            ...
```

```
        } catch (Exception e) {
            ...
            e.printStackTrace();
        }
    }

    private static PublicKey getPublicKey(String base64PublicKey) throws
        Exception {
        try {
            X509EncodedKeySpec keySpec = new ...;
            return KeyFactory.getInstance("RSA").generatePublic(keySpec);
        } catch (Exception e) {
            throw new Exception("Erro na chave pública", e);
        }
    }

    private static PrivateKey getPrivateKey(String base64PrivateKey) throws
        Exception {
        try {
            PKCS8EncodedKeySpec keySpec = new ...;
            return KeyFactory.getInstance("RSA").generatePrivate(keySpec);
        } catch (Exception e) {
            throw new Exception("Erro na chave privada", e);
        }
    }

    private static byte[] encrypt(String plainText, PublicKey publicKey)
        throws Exception {
        Cipher cipher = getCipher();
        cipher.init(...);
        return cipher.doFinal(...);
    }

    private static byte[] decrypt(byte[] cipherText, PrivateKey privateKey)
        throws Exception {
        Cipher cipher = getCipher();
        cipher.init(...);
        return cipher.doFinal(...);
    }

    private static Cipher getCipher() throws Exception {
        return Cipher.getInstance(...);
    }
```

```
public static String aesEncrypt(String plainText, String base64PublicKey)
    throws Exception {
    PublicKey publicKey = ...;
    byte[] cipherText = ...;
    return toBase64(cipherText);
}

public static String hybridEncrypt(String plainText, String
    base64PublicKey) throws Exception {
    AESCrypt aescrypt = new AESCrypt();
    String aesEncryptedData = aescrypt.encrypt(...);

    String encryptedText = ...;
    return encryptedText;
}

public static String decrypt(String cipherText, String base64PrivateKey)
    throws Exception {
    byte[] cipherBytes;
    cipherBytes = fromBase64(cipherText);
    try {
        PrivateKey privateKey = ...;
        String decryptedText = ...;
        return decryptedText;
    } catch (Exception e) {
        throw new Exception("Erro na decriptacao", e);
    }
}

public static String hybridDecrypt(String encryptedText, String
    base64PrivateKey) throws Exception {
    AESCrypt aescrypt = new AESCrypt();
    String decriptedData = ...;
    return decriptedData;
}

private static byte[] fromBase64(String str) {
    return DatatypeConverter.parseBase64Binary(str);
}

private static String toBase64(byte[] ba) {
    return DatatypeConverter.printBase64Binary(ba);
```

```
    }
}
```

Results

1. **Auditable compliance**: the hybrid layer **met internal security standards** and **passed the company audit** for handling executive-sensitive data.
2. **Developer transparency**: Axios interceptors and mirrored Java utilities **made encryption invisible** to business logic and developer workflows.
3. **Minimal latency**: encryption **added no noticeable delay** to user actions and **did not degrade UX**.
4. **Reduced incident surface**: mitigated risks from **rooted devices**, **emulators**, and **man-in-the-middle attacks**.

STAR Case - Design System Drift

Situation

Initially built with **React Native Paper** as the **design choice**, the project **later underwent UI redesigns** and custom management requests that pushed the **library beyond its intended scope**. The team had **stretched component customizations to their limit**, creating complex **overrides and inconsistent layouts**. This led to **design drift** across the app and made the codebase **increasingly difficult to maintain** and scale.

Task

Replace the **overextended UI library** with a **flexible styling system** that could **keep pace with frequent design changes**. The new solution needed to let developers **build and style components in one place**, reduce the need for overrides, and **speed up page creation** without sacrificing consistency or readability.

Actions

I designed and built React Native String Style, an **inline styling tool** inspired by **Tailwind CSS**, to **replace React Native Paper** and **eliminate dependency** on rigid components. **It enabled developers to write** utility-based class strings **directly in JSX**, merging **structure and styling** in a single file.

I **refactored screens and components** to adopt this syntax, simplifying layout creation and **removing the need** for complex **styled component files**. I also introduced **design tokens** for colors, spacing, and typography, enabling **quick global updates** whenever the design system changed.

Below is an example showing the two ways to use the styling tool: converting objects to styles with **objToRNStyle()** and applying inline utility classes with **sstyle**.

```
export const RadioButton: React.FC<RadioButtonProps> = (...}) => {
  const [selectedItem, setSelectedItem] = useState(value);

  const handleOnPress = (radioItem: RadioItem) => {
    setSelectedItem(radioItem);
    if (onPress) onPress(radioItem);
  };

  const buttonStyle = objToRNStyle({
    position: 'jcc aic fg',
    height: 'min-height-36',
    border: 'w-100% bd-ra-4 bg-radioButton.bg bd-width-1 bd-style-solid',
    active: 'bd-color-radioButton.bd.active',
    inactive: 'bd-color-radioButton.bd.inactive',
  });

  return (
    <>
      {!!title && (
        <View sstyle={`pd-b-16${hp ? ' pd-l-16' : ''}`}>
          <Text sstyle="fs-13 lh-16 ff-me c-text.title">{title || ''}</Text>
        </View>
      )}
      <View sstyle={`fdr${hp ? ' pd-h-' + hp : ''}`}>
        {radioItems.map((radioItem, index) => {
          const active = selectedItem?.value === radioItem.value;
          const lastItem = index === (radioItems?.length || 1) - 1;
          return (
            <View sstyle={lastItem ? 'fg' : 'fg pd-r-8'} key={index}>
              <TouchableOpacity onPress={() => handleOnPress(radioItem)}>
                <View
                  style={[
                    _.values(buttonStyle),
                    active ? buttonStyle.active : buttonStyle.inactive,
                  ]}>
                  <Text sstyle="fs-14 lh-20 ff-me">{radioItem.label}</Text>
                </View>
              </TouchableOpacity>
            </View>
```

```
          );
        })}
      </View>
    </>
  );
};
```

Results

1. **Aligned consistency**: one styling language **kept visuals uniform** across every screen.
2. **Clearer readability**: concise syntax **made components easier to understand** and **simplified maintenance**.
3. **Easier maintenance**: **removed dependency** on heavy UI libraries and **complex overrides**.
4. **Faster onboarding**: new developers **adapted quickly** thanks to the familiar Tailwind-style syntax.
5. **Greater flexibility**: design updates **were applied quickly without breaking** existing layouts.

STAR Case - Inconsistent State Propagation

Situation

Overuse of Redux for managing **simple UI and navigation states** caused bloated reducers, tight coupling, and **limited scalability**.

Task

Solve the **limitations** of a **Redux-only architecture** by establishing a **more flexible state management model**. It needed to support **different state scopes** without losing **consistency**, **clarity**, or **performance** across the app.

Actions

To **separate concerns** between **global, local, and transient state**, I introduced **React Context** for **feature-specific** flows and **navigation parameters** for **transient data transfers**. To keep the team aligned, I **documented the new conventions**, built **typed navigation helpers**, and refactored existing modules to follow the new layered structure.

Below is an example showing how a screen **combines Redux**, **Context**, and **navigation parameters** to manage all states in an organized way.

```
import { useRoute } from '@react-navigation/native';
import React, { useContext, useState } from 'react';
import { useSelector } from 'react-redux';

export const ProfileScreen = () => {
  const route = useRoute();
  const { biometricsEnabledRoute } = (route?.params || {}) as any;

  const [biometricEnabled, setBiometricEnabled] = useState(!!
biometricsEnabledRoute);
  const context = useContext(...);

  const profile = useseselector(...)

  const handleOnPressBiometrics = async () => {
    const newValue = !biometricEnabled;

    setBiometricEnabled(newValue);
    biometrics.toggle(newValue);

    if (!newValue) return;

    modal.warning({
      context,
      ...
      description: `Hi ${profile.name}...`,
      buttons: [
        ...
      ],
    });
  };

  return (
    ...
  );
};
```

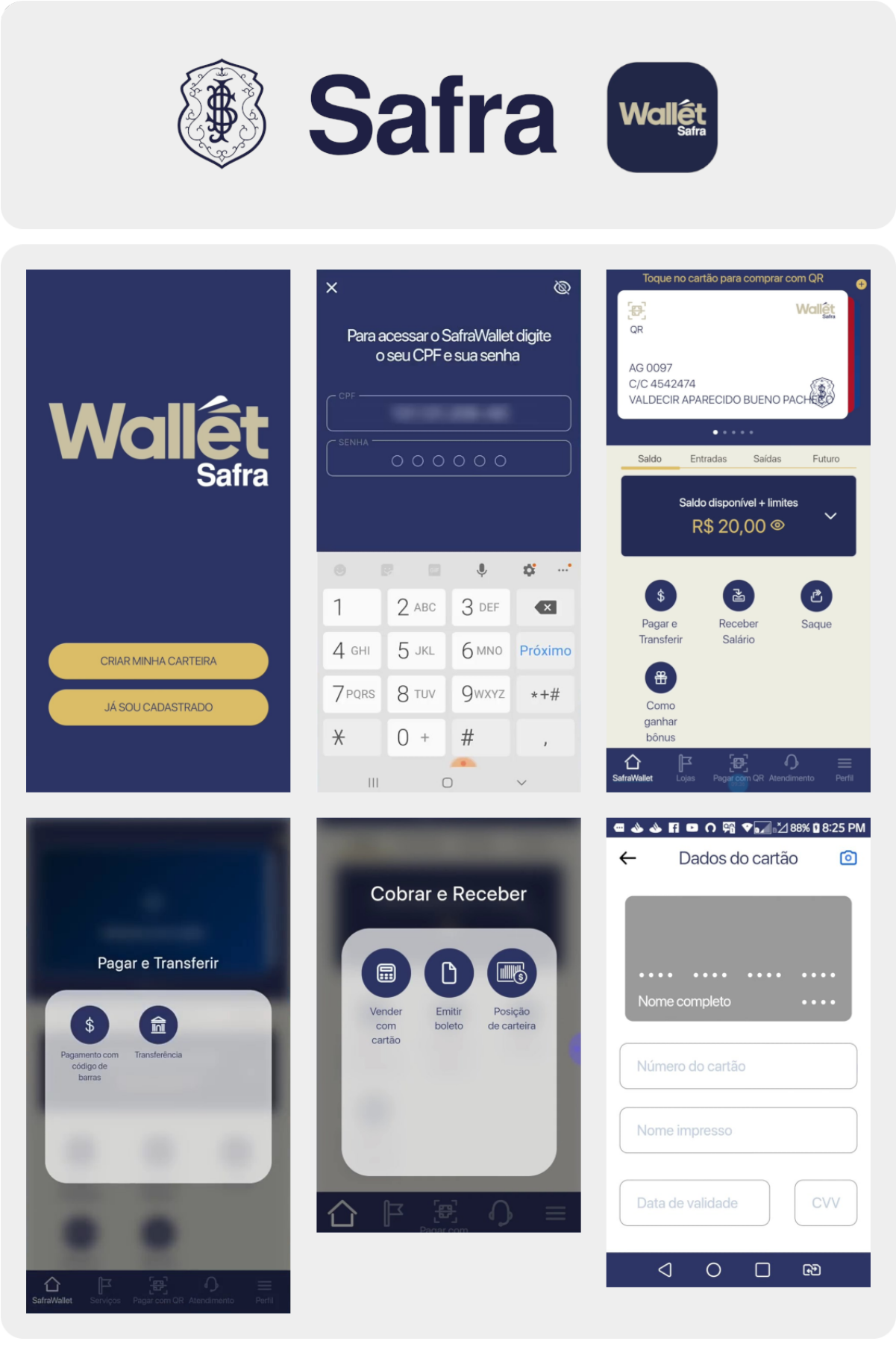
Results

1. **Cleaner architecture**: unified rules **removed conflicting** data-handling patterns.
2. **Faster debugging**: predictable data flow simplified **issue tracing** across screens.

3. **Improved performance**: isolating state scope reduced **unnecessary re-renders**.
4. **Lower maintenance cost**: fewer duplicated states **simplified refactors**.
5. **Stronger reliability**: consistent data handling **prevented broken flows** during navigation.
6. **Team alignment**: shared conventions **improved onboarding** and collaboration between developers.

💵 Safra

Staff Frontend Engineer - AngularJS 📅 Jun 2019 – Aug 2020



Safra Header

🌲 Project Treemap

Unavailable due to development constrained on the company internal environment.

🧱 Tech Stack

- **Frameworks:** AngularJS, Angular 9
- **Languages:** TypeScript, JavaScript, HTML5, CSS3, SASS
- **Build Tools:** Webpack, npm, Node.js
- **CI/CD:** Jenkins, Git, Custom Git Workflow
- **Testing:** Jasmine, Karma
- **UI Tools:** Sass, KSS
- **Version Control:** Git (internal repository)
- **Project Management:** Jira, Confluence

🌟 STAR Cases

STAR Case - Multiple Teams Conflicts Under Tight Deadline

Situation

The project faced an aggressive **6-month deadline** with **30+ frontend engineers** working across **multiple projects** (cards, ATM, onboarding, payments, profile). Each team maintained a **separate codebase**, yet all merged into a **single release pipeline**. The compressed schedule and overlapping workstreams created a **chaotic environment** of frequent overwrites, lost commits, **unstable builds**, and ongoing issues during **quality assurance**.

Task

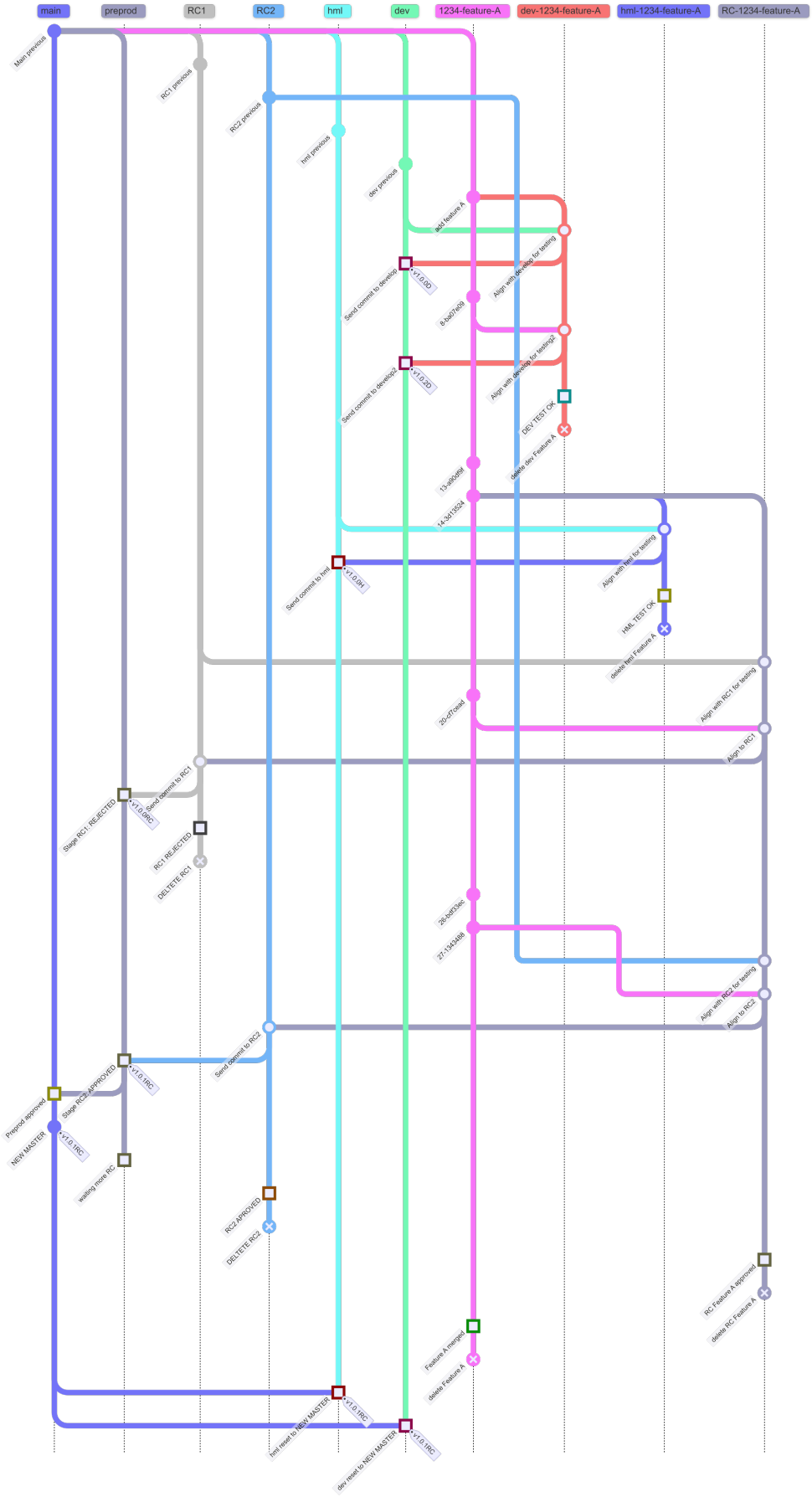
Define and implement a reliable integration and **release model** to restore **environment stability**. The goal was to **reduce merge conflicts**, prevent code loss, and ensure **predictable releases** while meeting the **delivery deadline**.

Actions

I started by **breaking down** the entire release process **from the top**. To understand why the builds kept failing, **I reverse engineered the pipeline**, tracing it **from the App Store approval** steps all the way back **to the developer commits**.
Once I mapped the flow, **I updated the Jenkins** configuration to build **only from predefined tags**. Since there was **no versioning strategy** in place, I introduced a manual semantic release process that triggered deployments **only when a new tag** was created.

With the pipeline under control, I **defined a new branching model** inspired by **GitHub Flow and GitFlow**, extended to **handle multiple environments**.
After **defining the model**, I aligned **with the team leads** about the new process, detailing how developers should create branches, tag releases, and **merge safely** into the shared **pipeline**.

Below is a simplified diagram of the branching and release model that illustrates how the pipeline operated after those changes.



Results

1. **Controlled rollbacks**: release tags and environment branches **allowed instant restoration** of any previous version.
2. **Consistent environments**: cascading rebases kept develop, homologation, and preproduction branches **synchronized with master** after each release.
3. **Cross-team stability**: the new model **reduced merge conflicts from** around **15** per release down to **2** on average.
4. **Developer safety**: isolated branches **protected individual work** and the shared pipeline.
5. **Eliminated QA instability**: fixed tags and controlled release candidates **made every change traceable**, ensuring consistent validation across environments.
6. **Predictable releases**: flexible release candidates included **only main commits and approved features**, ensuring **stable builds** for users and quality assurance.

STAR Case - Inconsistent UI Across Teams

Situation

Under **tight deadlines**, each squad had its **own designer** and developers building features with **no alignment across teams**. Without a shared design source or centralized library, teams **recreated the same components** in different ways, leading to **duplicated code**, inconsistent visuals, and a fragmented user experience **across the app**.

Task

Bring visual and structural **consistency back to the product** by creating a **single source of truth** for UI. I needed to **align all squads** around one shared component library that could live inside the **company restricted environment** and be easy for every team to **adopt without slowing delivery**.

Action

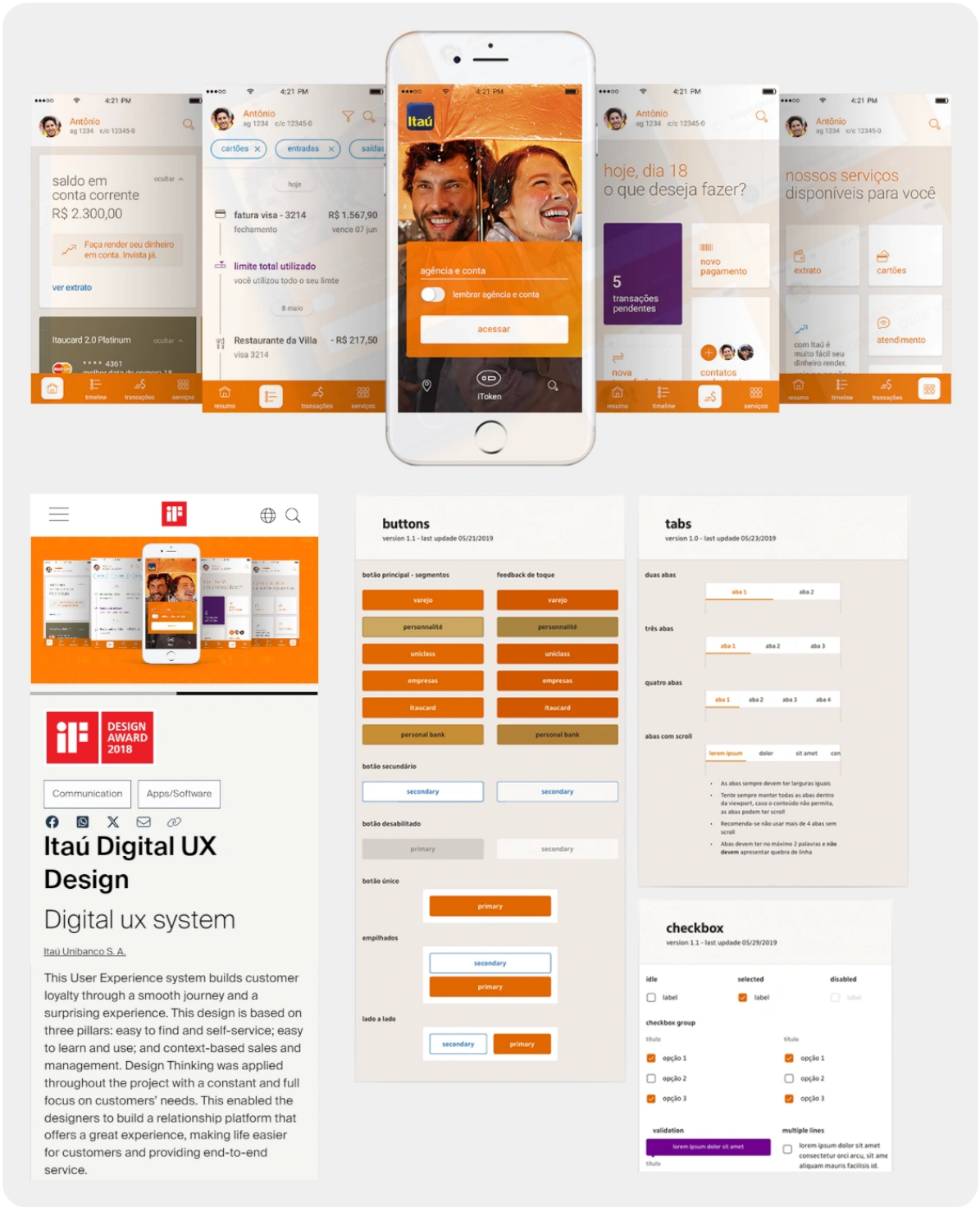
I **joined the design squad** to understand the centralized work they were creating and **took responsibility** for bridging communication **between designers and developers**. After assessing each team’s workflow, I determined that the **most effective way** to standardize the UI was by creating a **framework-agnostic, ready-to-use CSS** component library instead of AngularJS components, allowing squads to **use it in any context**. **I built** the library with **Sass** for consistent styling and **used KSS to document** it with clear visual component references. To **distribute** it within the **restricted network**, I set up a **bare Git repository** on a shared internal resource, enabling **all squads to pull updates directly** into their projects.

Results

1. **Wide adoption achieved**: the system’s simplicity and open contribution model **enabled all squads to adopt it within weeks**.
2. **Design drift eliminated**: shared components and standardized styling **unified behaviors and visuals** across every module.
3. **Library stability proven**: the architecture **endured three facelifts and two complete redesigns** without major refactoring.
4. **Documentation standardized**: the KSS guide **became the single reference** for all squads and **defined future documentation practices**.

💵 Itaú

Frontend Engineer - Angular 2+ | 📅 Mar 2018 - Jun 2019



Itau Header

🌲 Project Treemap

Unavailable due to development constrained on the company internal environment.

🧱 Tech Stack

- **Accessibility:** Axe, Screen Reader APIs, WCAG AA
- **Build Tools:** Node.js, npm, semantic-release, Webpack
- **CI/CD:** Git, Husky, JFrog Artifactory
- **Collaboration:** Cross-squad integration, Shared Knowledge Base
- **Documentation:** Confluence, Internal Design System Docs
- **Frameworks:** Angular 7, AngularJS
- **Languages:** CSS3, HTML5, JavaScript, TypeScript, SASS
- **Project Management:** Confluence, Jira
- **Testing:** Headless Chrome, Jasmine, Jest, Karma
- **UI Tools:** Design Tokens, KSS, Mixins, SASS Utilities, Storybook
- **Version Control:** Git (multi-repo architecture, company repository)

🌟 STAR Cases

STAR Case - No compliance with WCAG Standards

Situation

With more than **90 million** users and about **24%** of the population living with **some form of disability**, a significant number of customers **faced barriers** using the app. The company had begun **enforcing WCAG** accessibility standards across **all digital products**.

Task

As part of a **cross-squad design system** team, I was responsible for **ensuring** the new Angular 7 **components complied with WCAG 2AA** accessibility standards. The team was tasked to:
1. **Integrate accessibility requirements** into the design system’s development workflow.
2. **Validate accessibility with real users**, supported by a dedicated QA subteam composed of people with disabilities.
3. **Guide external consultancy** to accelerate implementation and ensure technical alignment with accessibility best practices.

Action

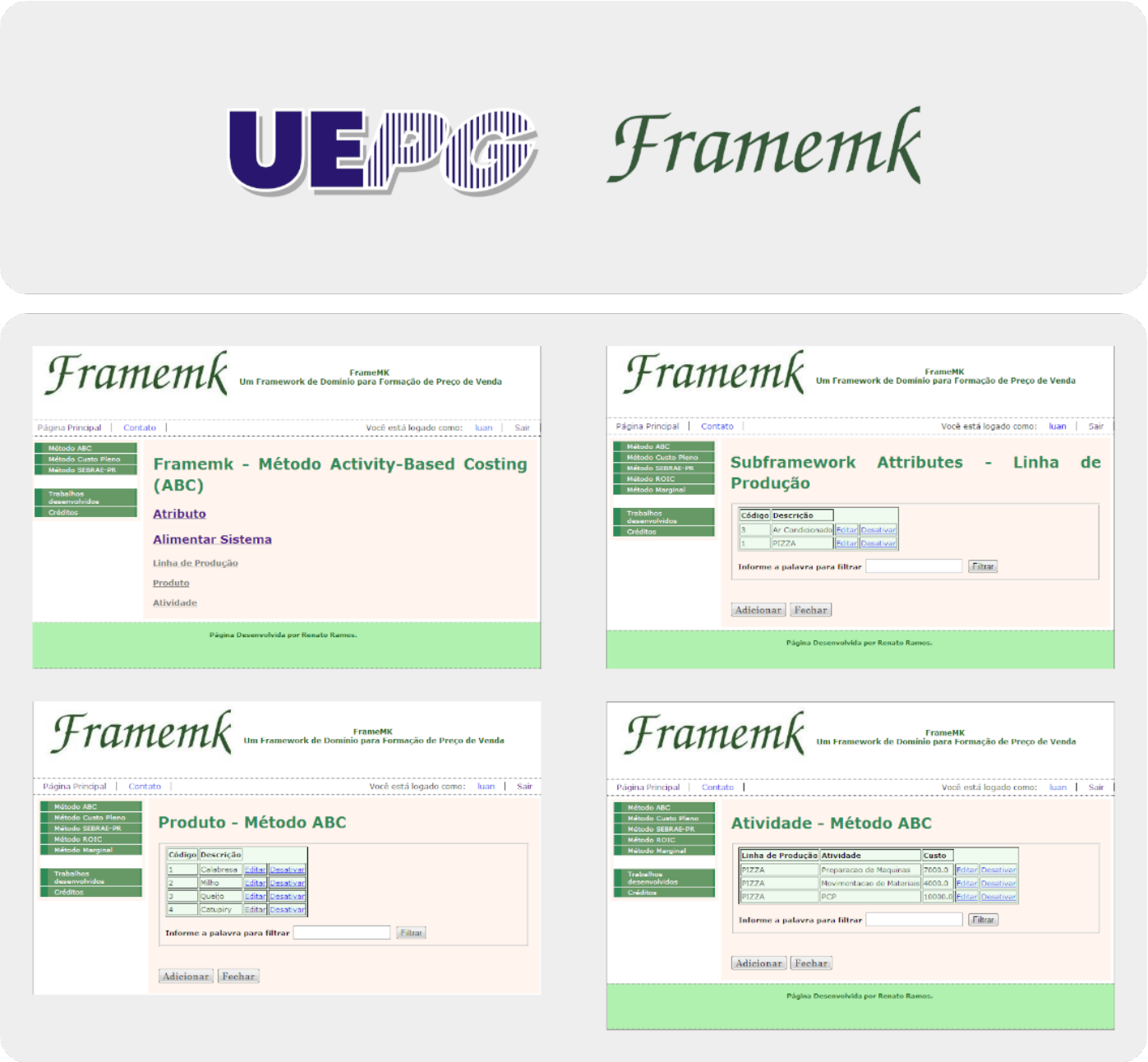
The effort began by **contracting and onboarding a 7-person consultancy team**, integrating them into our workflow and **setting up their environments** to match the internal CI process. With the **team established**, collaboration expanded to the **QA group of testers with disabilities**, whose feedback guided accessibility refinements in **real use cases**. As the work evolved, our squad became the **bridge between design and engineering**, revisiting UI patterns and adjusting layouts, color palettes, and interaction models **to meet WCAG 2AA standards**. The **development phase** introduced **ARIA roles**, **keyboard navigation**, and **color-contrast adjustments**, ensuring full **compatibility with NVDA**, **VoiceOver**, and **TalkBack**. This transformed accessibility from a patch into a **core design system feature**. To close the cycle, the new practices were **documented and distributed** so future squads could maintain the same accessibility standards.

Results

1. **Achieved WCAG 2AA compliance**: all components were validated through **real-user testing** with **NVDA**, **VoiceOver**, and **TalkBack**.
2. **Enabled inclusive access**: more than **90 million users with** gained full accessibility across the company digital products.
3. **Standardized accessibility by design**: accessibility rules became **part of the design system architecture**, not a post-release addition.
4. **Ensured long-term scalability**: documented practices and reusable patterns **preserved accessibility compliance** in all future products.
5. **Unified multidisciplinary collaboration**: design, engineering, QA, and consultancy teams **worked under a shared accessibility effort**.

🎓 UEPG

Full stack Researcher - Java 📅 Mar 2015 to Aug 2017



UEPG Header

🌲 Project Treemap

Unavailable due to research environment constraints.

🧱 Tech Stack

- **Architecture:** Service-Oriented Architecture (SOA)
- **Build Tools:** Ant, Eclipse IDE
- **Database:** Firebird, MySQL

- **Documentation:** Internal Technical Guides, UML Models
- **Environment:** Windows,Debian Linux, VirtualBox
- **Frameworks:** Apache Struts, FrameMK, JUnit
- **Integration:** REST, SOAP, WSDL
- **Languages:** CSS, HTML, Java, JSP, SQL, XML
- **Modeling:** UML (Use Case, Class, Package, Activity Diagrams)
- **Testing:** JUnit (Unit and Persistence Layer Tests)
- **Version Control:** Manual dependency management, SVN
- **Web Services:** REST APIs, SOAP/WSDL for ERP Integration

🌟 STAR Cases

STAR Case - Fragmented Research Environment

Situation

The **framework** ran on a **legacy Java Struts architecture** with **complex dependencies**, **Firebird databases**, and **distributed tools**. Each contributor **manually configured environments by himself**, resulting in **version drift**, **failed builds**, and **long onboarding times**.

Task

Design a **reproducible environment** that **unified all dependencies**, databases, tools, JUnit test automation, and SOA integration workflows **across all machines**.

Action

Packaged the full research stack **into a virtual machine image** containing Java SDK, Apache Struts, Ant, JUnit, Firebird, and SVN integration. Embedded **startup scripts** to initialize the database, **compile the framework, and deploy local web services**. Consolidated all UML diagrams, documents, and guides inside the VM for self-contained reproducibility.

Results

1. **Reduced setup time**: full onboarding decreased from **weeks to one hour**.
2. **Standardized research execution**: all contributors operated within a **consistent and controlled environment**.
3. **Unified dependencies and tooling**: **all components** integrated into a **single virtual machine image**.

4. **Preserved research continuity**: versioned VM snapshots and SVN references maintained **long-term environment** parity.

☂️ Virtual

Full-stack Engineer - Delphi / MEAN 📅 May 2013 – Mar 2015

Virtual Header

🌲 Project Treemap

Unavailable due to development constrained on the company internal environment.

🧱 Tech Stack

- **Languages:** Delphi, JavaScript, TypeScript, SQL, HTML
- **Frontend:** AngularJS
- **Backend:** Node.js

- **Databases:** MongoDB, Microsoft SQL Server
- **Reporting:** QuickReport, Dephi
- **Project Management:** Scrum

🌟 STAR Cases

STAR Case - Code Ossification

Situation

The insurance management **system relied** on a Delphi-based **PDF parser** that converted **files into plain text** and navigated fields using **company-made** custom **parse functions**.
For more than a decade, the team maintained this **fragile approach**, a clear case of **code ossification** that prevented **simpler and more maintainable** solutions like regular expressions from **being adopted**.

Task

Enable **reliable automated** extraction of data **without breaking** existing parsers.

Action

After a **few months** adjusting the parser through the **company's custom functions**, I **proposed introducing regular expressions** to simplify data extraction. The idea faced **initial resistance** due to years of **code ossification** and comfort with the old cursor logic.
After **repeated attempts and personal insistence**, I finally **got approval to add** the regex function to the core parser. Once integrated, **it quickly proved its value**, successfully parsing complex document sequences that previously required extensive manual position tracking.
With those results, I was **asked to help the team** on writing and maintaining **regex-based** extraction **rules** for other document schemas, **formalizing pattern usage as the new standard** for PDF parsing **within the system**.

Results

1. **Stabilized PDF extraction logic**: improved fragile custom parsing with **regex-driven matching**, eliminating data loss from manual offsets.
2. **Reduced maintenance overhead**: legacy functions were **gradually deprecated**, simplifying debugging and onboarding for new developers.
3. **Improved document coverage**: the new regex parser **handled multiple schema variations** that were previously **unsupported by** cursor logic.
4. **Enabled team-wide adoption**: after internal training, regex usage **became a standard practice** for **all document** parsing routines.

5. **Transformed legacy mindset**: the function **broke a decade of code ossification**, proving modernization could happen without risking stability.