Slide 1: Title Slide

Good morning, everyone. My name is Eduardo Ezponda, and today I will present my Final Degree Project. It focuses on adapting a Django backend to support the migration of a frontend to React.

This project was developed at Splorotech, a company specializing in innovation consultancy. The main goal was to improve Kronis, a platform used for managing funded technology projects, by making it more scalable, maintainable, and efficient. In this presentation, I will discuss the challenges faced, the solutions applied, the methodologies used, and the results obtained.

Slide 2: About Splorotech

Splorotech is a consultancy that helps to manage innovation projects with public funding. It provides tools and expertise for planning, executing, and tracking technology projects.

The company also offers software solutions to simplify project management. One of its main products is Kronis, the platform we will focus on in this project.

Slide 3: About Kronis

Kronis is an advanced platform developed by Splorotech to manage funded innovation projects efficiently. It provides users with a centralized system to track all phases of a project, from initial planning to final reporting.

Slide 4: About Kronis

The platform helps different teams work together easily, including project managers, financial analysts, and technical staff. It provides tools for budgeting, managing documents, tracking performance.

Slide 5: Problems in Kronis

While Kronis is a robust and functional platform, it faced several key challenges that needed to be addressed to ensure its long-term sustainability and efficiency.

One of the main issues was **scalability**. The original system was built in Django, including both backend and frontend. This approach made it difficult to introduce new features and required developers to have full expertise in Django, which limited recruitment options.

To solve these problems, we decided to migrate the frontend to React and Next.js while keeping the backend in Django, adapting it to function as an API for the new frontend.

Slide 6: The Solution

To solve Kronis' problems, the frontend (what user sees) was separated from the backend (the server logic). Instead of using Django templates, we switched to React and Next.js. This made the website faster and more interactive.

The backend, which used to handle both logic and page rendering, was changed to work only as an API. This means it now provides data to the frontend through RESTful endpoints.

Slide 7: Adaptation of the Backend

Migrating the frontend to React required significant modifications in the Django backend to ensure smooth communication between both layers.

The main adaptation was the **creation of a REST API** using **Django Rest Framework (DRF)**. This involved:

- Defining new API endpoints to expose data in JSON format.
- Implementing authentication mechanisms to secure API access.

Slide 8: Objectives

The main goals of this project were:

1. **Optimization:** Make the platform faster and easier to maintain.
2. **Scalability:** Let the frontend and backend teams work separately.
3. **User Experience:** Make the website more interactive and user-friendly.
4. **Efficiency:** Simplify code maintenance and speed up development.

By switching to a modular design, we made sure future updates could be added smoothly without breaking anything.

Slide 9: Methodology

To keep the development organized and efficient, we used Scrum, an agile method. Scrum works with short cycles called sprints, allowing continuous feedback and improvements.

The project was divided into sprints, each covering specific tasks. This helped us to:
• Focus on important features in small steps.
• Adapt to new needs and challenges easily.

Slide 10: GitHub

Additionally, we used **GitHub** as a version control platform, ensuring proper organization and collaboration among team members. Each feature or modification was implemented in separate branches, reviewed, and merged into the main branch.

Slide 11: Backend Development

The backend of Kronis was developed using **Django**, a high-level web framework written in Python.

Slide 12: Django

Django follows the **Model-View-Template (MVT)** pattern to separate different concerns:

- **Model:** Defines the structure of the database and the business logic.
- **View:** Manages user interactions and processes requests. Previously, views in Kronis also rendered templates, but after the migration, they serve JSON responses through the API.
- **Template:** Used to present data to users. In the old system, templates were integrated with Django, but with React, the frontend is now responsible for rendering the UI.

Django's **ORM** allows using **Python objects** instead of SQL queries, making database tasks easier.

During the **migration**, we converted views into **REST API endpoints**, separating frontend and backend.

Slide 13: Django Rest Framework (DRF)

To transform the backend into an API-driven system, we used Django Rest Framework (DRF), which simplifies API development by providing:

- Serializer classes to convert Django models into JSON data.
- Routers to handle API requests efficiently.
- Authentication and permission handling for securing API endpoints.

Slide 14: Frontend Development

For the new frontend, we chose Next.js, a framework based on React. It has useful features for modern web apps.

React uses a component-based structure, making it easier to create reusable UI elements. Next.js improves React by adding better rendering options, helping with speed and SEO.

Slide 15: Next.js - Hybrid Rendering

One of the key benefits of Next.js is its flexibility in rendering methods, which include:

• **Server-Side Rendering (SSR):** Pages load faster because they are created on the server before reaching the user.
• **Static Site Generation (SSG):** Pages are built in advance, making them quicker and reducing server work.
• **Client-Side Rendering (CSR):** Some parts of the page update in the browser without needing a full reload.

Slide 16: Kronis Architecture

We use **Docker** for a consistent development environment. It isolates and manages services like:

- **Django (Backend):** Handles logic and API.
- **Next.js (Frontend):** Provides the user interface.
- **PostgreSQL:** Stores structured data.

- **Redis:** Speeds up performance with caching.

For hosting and scaling, **AWS** provides:

- **Frontend & Backend Servers** (Next.js & Django).
- **Load Balancers** to distribute traffic.
- **Auto-scaling** to adjust based on demand.

Slide 17: Silk – Database Query Optimization

To ensure efficient database queries, we integrated **Silk**, a Django profiling tool that helps:

- **Analyze database queries** and identify performance bottlenecks.
- **Reduce redundant queries** to improve API response times.
- **Optimize SQL execution**.

Slide 18: Migration – From Bootstrap to Tailwind CSS

As part of the frontend modernization, we migrated from **Bootstrap**, which was used in the original Django templates, to **Tailwind CSS** in the Next.js frontend.

Before (Django + Bootstrap)

- Used predefined styles with limited flexibility.

After (Next.js + Tailwind CSS)

- Uses a **utility-first approach**, enabling greater customization.
- Improves performance by eliminating unnecessary styles.

Slide 19: Frontend Project

In our frontend project, we follow **SOLID principles** to build a clean and maintainable codebase. The solid principles are:

- **Single Responsibility:** Each component does one thing well.
- **Open/Closed:** Components are open for extension but closed for modification.
- **Liskov Substitution:** Components can be replaced without breaking the app.

- **Interface Segregation:** We use small, specific interfaces instead of large ones.
- **Dependency Inversion:** Components depend on abstractions, not concrete implementations.

Slide 20: Project Modifications – CRUD Endpoints & Travel Actions

To support the new frontend, we developed multiple **CRUD (Create, Read, Update, Delete) endpoints** for different models in the Timesheets application.

Slide 21: Timesheets - CRUD Endpoints

These endpoints allow:

• Fast data retrieval and management using API calls.
• Smooth interaction between the frontend and backend.

Slide 22: Travel Actions - Endpoints

Additionally, we implemented **travel management actions**, including:

- Closing a travel request.
- Requesting a reopening.
- Cancelling a reopen request.
- Approving or rejecting reopen requests.

Slide 23: Development of a Web Interface for API Testing

To facilitate API interaction and testing, we developed a **dedicated web interface**. This interface serves as a valuable tool for developers and testers, making it easier to debug and validate API functionality.

Slide 24: Interface features

It allows:

- Sending **GET, POST, PUT, and DELETE requests** to test API endpoints.
- Dynamically inserting data and retrieving responses.

- Visualizing API interactions in a structured manner.

Slide 25: Authentication and Security

Security is a critical aspect of Kronis, and we implemented authentication mechanisms, including:

- **User login system** with secure credential handling.
- **Session-based authentication** for managing API requests.

These security measures protect user data and prevent unauthorized access, maintaining the integrity of the system.

Slide 26: Local Implementation of Kronis

For the local implementation of Kronis, we removed sensitive data (usernames, passwords, .env info) and used anonymized datasets for testing. This ensured safety while maintaining full functionality for development.

Slide 27: Example of an API Response

Here is an example of a JSON response from one of our API endpoints. This structured format ensures that the frontend can easily process and display data dynamically.

The adoption of RESTful APIs has made it easier to integrate new features and maintain a clean and consistent data flow.

Slide 28: Unit Testing – Ensuring Code Reliability

To guarantee system stability, we implemented **unit testing**, which allows us to:

- **Detect errors before deployment**, reducing potential bugs in production.

Slide 29: Unit Testing – Ensuring Code Reliability

- **Verify that individual components function correctly** without external dependencies.
- **Simulate API interactions** using mocks for more reliable testing.

Unit testing played a key role in ensuring that our migration process did not introduce unexpected issues.

Slide 30: Unit Testing Results – 186 Tests Executed Successfully ✅

As part of our quality assurance process, I executed **186 unit tests**, all of which passed successfully. This comprehensive testing ensured that the new system maintained its functionality and stability after the migration.

Slide 31: Future Plans for Kronis

Looking ahead, Kronis will continue to evolve to meet the needs of European Union funded innovation projects. Future developments will include:

- **Automating more processes** to improve efficiency.
- **Expanding integrations** with other software platforms.

Splorotech aims to position Kronis as a **leading solution for innovation project management.**

Slide 32: Closing Slide

This concludes my presentation on the **Django backend adaptation for frontend migration to React**. Thank you for your attention!

I would be happy to answer any questions you may have.