E.T.S. de Ingeniería Industrial,Informática y de

Telecomunicación

Django Backend Adaptation for Frontend Migration to React

Programa Internacional del Grado en Ingeniería Informática

Trabajo Fin de Grado

Autor: Eduardo Ezponda Igea

Director: Jesús Villadangos Alonso

Fecha de la defensa: Pamplona, 24/03/2025

upna

Universidad Pública de Navarra
Nafarroako Unibertsitate Publikoa

Agradecimientos:

A mis padres, María Teresa Igea Larrayoz y Juan José Ezponda Iradier, por su apoyo incondicional y por ser mi mayor inspiración. A mi familia, por estar siempre a mi lado. Y a mi pareja, Isabel Quílez de la Torre, por su compañía en este camino.

Summary

This project focuses on the development of functionalities for Splorotech's innovation project management system, specifically on the creation and validation of CRUD endpoints for the Kronis platform. During the internship, unit tests have been implemented in Django to ensure the quality of the code and guarantee its correct functioning. In addition, a web interface has been designed with forms and drop-downs that allow interaction with the endpoints, facilitating the execution of requests and displaying the results obtained.

The project also includes the execution of test cases that verify the functionality of the endpoints, integrating modern tools and best practices, such as automated testing and the development of intuitive interfaces. This work combines backend logic in Django with frontend design, offering an integral vision of the development and validation of functionalities in web applications.

List of keywords

Splorotech, Kronis, Amazon Web Services, GitHub, GitHub Copilot, Scrum, Software, Django, Python, Framework, REST, Next.js, Backend, Frontend, API, Crud, Silk, Excel, Pycharm

Index

Index of figures

1. Introduction and objectives

1.1 Introduction

In a constantly evolving world, organisations must adapt to new challenges to optimise processes and improve efficiency. Splorotech, a consultancy firm specialising in innovation and software services, positions itself as a key player in the digitisation of processes related to European project management and other business innovation initiatives.

Its flagship product, Kronis, is a category-leading software solution designed to support organisations in the management and justification of EU-funded research and development (R&D) grants. Kronis enables companies to efficiently report, analyse and plan projects. Its functionalities include automatic generation of timesheets, tracking of expenses, and a real-time analysis dashboard that facilitates strategic decision-making. In addition, its ability to integrate with tools such as Excel and Airtable, along with automations such as new project verification and analysis of agreements from PDF files, make it an indispensable tool for managing programmes such as Horizon Europe, EIT KICs Projects, and Interreg.

Kronis aims to optimise business productivity by simplifying administrative processes and ensuring compliance with European regulations, all on a scalable, secure and continuously updated platform.

As part of this evolution, Splorotech is facing the challenge of modernising the architecture of its platform. One of the main objectives has been to migrate the frontend of the Kronis platform, previously based on Django templates, towards a more modern and scalable solution using Next.js. This transition responds to the need to improve the user experience, increase development efficiency and lay the foundations for a decoupled frontend that allows better integration with the Django backend.

This thesis addresses precisely this context, focusing on the implementation and validation of endpoints for the Kronis platform and the development of complementary tools, such as a web interface for testing and visualising test results. The migration to a Next.js frontend, together with the design of an efficient backend, reflects the importance of adopting modern technologies to respond to the changing needs of the market.

1.2 State of the Art

1.2.1 Lack of specific literature on Django to Next.js migrations

There are not many books or research papers that explain how to migrate a web application from Django templates to Next.js while keeping Django as the backend. Most available information focuses on modernizing frontend applications in general or on full-stack migrations. Django is a strong backend framework, but its built-in templating system is often replaced by modern frontend frameworks like React, Angular, or Vue.js to improve flexibility and performance.

Because there is no clear guide on how to do this migration, I had to rely on software testing to check if the migration was successful. The main focus was on verifying that the backend API worked correctly with the new frontend instead of following a predefined migration process.

1.2.2 Software testing approach to validate the migration

Since my work focused on adapting the Django backend for a Next.js frontend, testing was important to ensure that everything worked correctly. The main type of testing I used was unit testing.

Unit testing helped me verify that individual parts of the backend, like API endpoints and data processing functions, worked correctly. This was necessary because Django templates were removed, and the frontend was now separated from the backend. Instead of testing the interface directly, I focused on making sure that the backend returned the correct responses for the frontend to display.

Other types of testing, such as integration testing or end-to-end testing, could have been useful, but they were not part of my work. The main goal was to ensure that the backend was ready to support the new frontend, so unit testing was the most relevant approach.

1.2.3 Alternatives to Next.js and comparison

For this migration, Next.js was chosen as the new frontend framework. However, there were other possible options:

1. React (client-side rendering only): This is a simple and flexible option, but it does not have built-in support for server-side rendering (SSR) or static site generation (SSG). These features require external tools like Gatsby or Remix.
2. Angular: This framework includes many built-in features and enforces a strong structure. However, it is more complex and has a steeper learning curve.
3. Vue.js: Vue is a lightweight and reactive framework, but its ecosystem is smaller compared to React.

Next.js was the best choice because it provides server-side rendering (SSR) and static site generation (SSG), which improve page speed and SEO. This was important for the application because it needed dynamic content while keeping fast performance.

By reviewing testing methods, looking at different frontend options, and recognizing the lack of detailed information on Django-to-Next.js migration, this section explains the state of the art and the decisions made in this project.

1.3 Objectives

1.3.1 Optimisation of R&D grant management

This project aims to improve the management and justification of European research and development grants by simplifying tasks such as reporting, expenditure tracking and project planning.

1.3.2 Facilitating technology migration

The aim is to transition the Kronis frontend, based on Django, to Next.js, in order to improve the architecture of the platform, offer a better user experience and increase the scalability of the system.

1.3.3 Process centralisation and automation

The project aims to centralise all operations related to the management of European projects on a single platform, automating tasks such as the generation of timesheets, cost analysis and verification of new projects, thus reducing the administrative burden.

1.3.4 Improving user experience

The redesign of the frontend aims to provide modern, intuitive interfaces adapted to the specific needs of users, improving their interaction with the platform.

1.3.5 Ensuring compatibility and regulatory compliance

The aim is to keep the platform compliant with European regulations, such as GDPR, ensuring data security and compatibility with programmes such as Horizon Europe, Interreg and the Digital Europe Programme.

1.3.6 Increasing development efficiency

With the adoption of Next.js, the aim is to optimise the frontend development process, facilitating integration with the Django backend and improving the modularity and maintainability of the code.

1.3.7 Promoting digital transformation

The project contributes to the digital transformation of organisations, promoting the use of advanced technological tools that simplify work and improve productivity.

2. Project management

2.1 Scrum methodology

During my time at Splorotech, I actively participated in the implementation and improvement of the SCRUM methodology applied to the development of the Kronis product. This agile methodology, known for its flexibility and focus on collaboration, allowed me to organise the work of the development team in an iterative and efficient way.

The team used the GitHub project dashboard as the main tool for managing tasks. This dashboard was divided into columns such as ‘To do’, ‘Current sprint’ and ‘In progress’, which made it easy to track the status of each task in real time. Daily meetings (dailys) were held early in the morning, lasting approximately three minutes per person. In these meetings, each developer reported on their progress, identified blockages and planned next actions.

Several key meetings were held on Fridays:

Sprint Review: We evaluated the deliverables of the sprint, checking whether the set objectives had been achieved.

Sprint Retrospective: We reflected on the positive aspects and areas for improvement in the development process.

Client Meeting: We met with Miguel García (CEO) and Alberto Sierra (COO) to present progress, adjust expectations and define next week's objectives. These meetings were preceded and followed by internal planning and analysis meetings.

My role included implementing backend functionalities, coordinating with Alex (frontend developer) to properly integrate data between frontend and backend, and presenting results during reviews. In addition, these SCRUM dynamics allowed me to develop skills such as accurate task estimation, agile problem solving and continuous improvement of teamwork.

The SCRUM experience at Splorotech not only resulted in increased productivity, but also reinforced core values such as communication, organisation and adaptability, which will be fundamental in my future professional development.

2.1.1 Total hours per process

Process	Completed
Dailys	4h
Development tasks	240h
Sprint reviews	8h
Sprint retrospectives	8h
Total	**260 h**

Figure 1 Total hours per process

In this case, by far the most time-consuming part was the development tasks. During this stage, I worked on implementing CRUD endpoints, developing new features, creating unit tests, and handling pull requests in Git. Each pull request went through a detailed review process, both by myself and Iván Pajares, who gave valuable feedback to improve the code and ensure its quality. This development cycle included identifying and resolving issues, as well as adjusting functionality to align with the project's goals.

On the other hand, daily 5-minute dailies allowed synchronising the work of the whole team and resolving blockages quickly. On Fridays, sprint reviews and retrospectives, each lasting about 45 minutes, helped us evaluate the progress of the sprint, identify areas for improvement, and plan goals for the following week. This balance of development, meetings, and reviews contributed significantly to the final product quality and learning during my stay.

3 System requirements

3.1 Non-functional requirements

3.1.1 Language

Kronis is completely developed in English, both in its programming code and in its user interface, as it is a platform for the management of European grants, which must be accessible to users from different countries and organisations within the European Union. The use of English as the base language in the development of the system responds to the need to create a solution that is universal and easily understood by all users, regardless of their geographical location. As it is an international platform, English is established as the common language to guarantee the interoperability, maintenance and expansion of the platform. This ensures that all technical elements, from functions and variables to business logic, are aligned with global programming standards and can be used by teams of different languages without barriers.

3.1.2 Excel and Airtable

Kronis uses integrations and automations with tools such as Excel and Airtable to optimize workflow and reduce time spent on repetitive tasks. Through these integrations, Kronis is able to automatically verify new projects and process awarded agreements from PDF files. In addition, the use of Excel enables large volumes of data to be managed and rapid analysis to be performed, while Airtable facilitates collaborative project organisation and tracking. These automations not only save time, but also improve task accuracy and free up resources to focus on delivering impactful research and development projects.

3.1.3 Sending emails and notifications

At Kronis, email notifications are a fundamental part of communication within the platform. For example, when an employee is sent an invitation to join an organisation, they receive an email with the details and a link to accept the invitation. Similarly, when a worker is accepted into the organisation, he or she is notified by mail, informing him or her of his or her joining. In addition, notifications are also used to keep users informed of key actions in the system, such as the validation of timesheets or the completion of a timesheet, ensuring that users are always aware of important events and changes related to their work.

3.1.4 Usability

Kronis is designed to be intuitive and easy to use, focusing on enhancing the user experience through a clear and accessible interface. The platform simplifies complex tasks such as R&D project management and document validation, allowing users to navigate seamlessly between different functionalities. The layout of elements is consistent, with clearly labelled buttons and options, facilitating interaction and reducing the learning curve. In addition, Kronis provides timely notifications to keep users informed about the status of their actions, improving efficiency and productivity without sacrificing clarity.

3.2 Functional requirements

The first use case diagram represents the creation of different entities within the system. The main actor in this diagram is the User, who serves as the base role from which other roles inherit additional permissions. There are two key extensions of the User role: the Project Manager and the Organisation Manager. Each of these roles has different levels of responsibility and permissions.

The Project Manager is responsible for managing projects and their related components. They can create a Project, and this action allows the possibility of creating a Work Package as an extension. Since a Work Package is always associated with a Project, the system offers this as an optional step immediately after creating a project. Furthermore, when a Work Package is created, the Project Manager also has the option to create a Reporting Period, as a Reporting Period is inherently linked to a Work Package.

In addition to project-related functionalities, the User role also includes the ability to create various entities within the system, such as Travel, Travel Documents, and Expense Sheets. The Create Expense Item action is an extension of the Create Expense Sheet action, meaning that an Expense Item must always belong to an Expense Sheet.

The Organisation Manager, at the top of the hierarchy, is the most powerful role within the system. This role extends from Project Manager, which means that the Organisation Manager inherits all the capabilities of both the Project Manager and the User. In addition to being able to create Projects, Work Packages, and Reporting Periods, the Organisation Manager has exclusive privileges for managing human resources and organizational structures. This includes the ability to create Workers, Worker Periods, Calendars, Locations, and Worker Groups. These actions ensure that the Organisation Manager can oversee all aspects of the system, from project management to employee and resource administration.

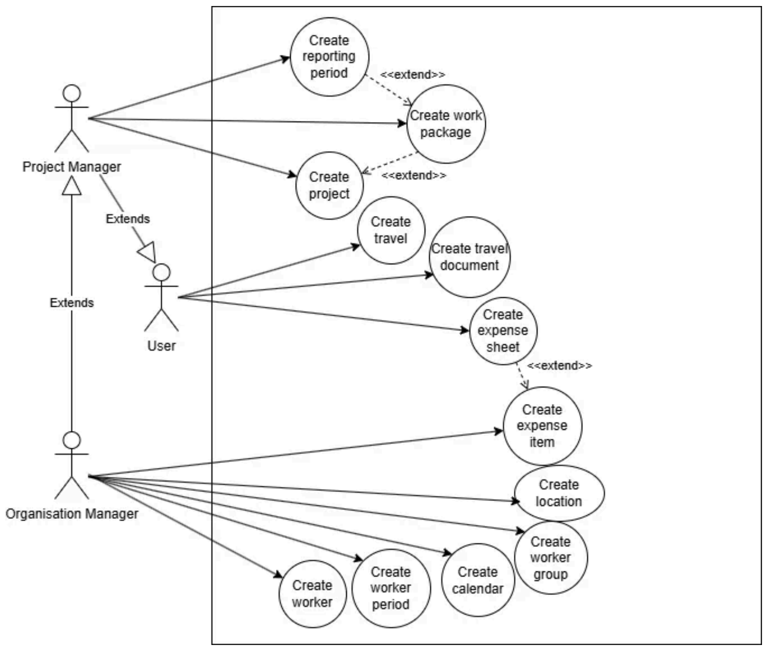


Figure 2 Create Use Diagram

The second use case diagram outlines the process for reopening and managing the status of a travel within the system. It establishes a structured workflow where different user roles interact with the reopening mechanism, ensuring that changes go through a controlled approval process. The User initiates the process, while the Organisation Manager holds greater authority, making key decisions regarding reopening and closing travels.

A User can start by submitting a Request Reopen action, which serves as the first step in requesting that a closed travel be reopened. Before the request is reviewed by the Organisation Manager, the User has the option to Cancel Reopen Request, allowing them to withdraw their request if reopening is no longer needed. This helps streamline the workflow by avoiding unnecessary processing.
Once a Request Reopen is submitted, the Organisation Manager has the power to either Accept Reopen or Reject Reopen. If the request is approved, the Reopen action becomes available, granting the User the ability to proceed with reopening the travel.

In addition to managing reopening requests, the Organisation Manager has full control over the travel's status. They have exclusive authority over the Close action, meaning they can close a travel at any point, even after it has been reopened. This ensures that the final decision on the travel's status remains with the Organisation Manager, maintaining an organized and regulated approval process.

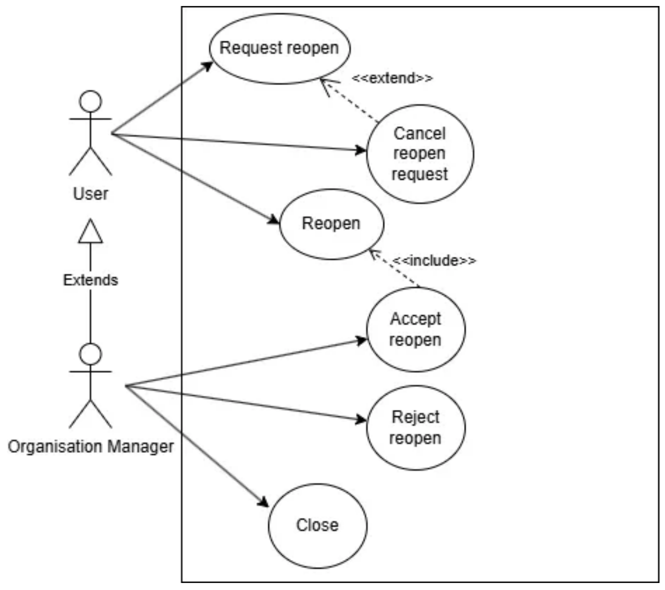


Figure 3 Travel Actions Use Diagram

4 Software requirements

4.1 Technologies

Kronis leverages a number of advanced technologies to ensure performance and efficiency. The backend is developed with Django and Python, providing a flexible and robust framework for managing data and application logic. For the frontend, Next.js, a React-based framework that offers a fast and dynamic user experience, is used. Integration with Excel and Airtable facilitates data management and automation, allowing users to manage information efficiently. In addition, Silk is used to analyse the efficiency of database queries, helping to improve execution time and reducing the number of unnecessary queries, which optimises the overall performance of the platform. These technologies, working together, ensure a smooth user experience and a highly efficient system.

4.1.1 Django

Django is a high-level web development framework based on Python, designed to facilitate the creation of robust and scalable applications. One of the fundamental principles of Django is the use of the Model-View-Template (MVT) pattern, which organises development by separating business logic, data presentation and user interaction. The models allow the structure of the data and its interaction with the database to be defined in a simple and powerful way, while the views manage the application logic and the templates manage the visual representation.

Django includes support for developing traditional web applications, but also allows the creation of APIs using the Django REST Framework (DRF). DRF makes it easy to build interfaces that expose data in JSON format, making it an ideal tool for modern projects where the frontend and backend are separate. With DRF, it is possible to define serialisers that transform model data into JSON structures and handle endpoints that respond to HTTP requests. This makes it possible to integrate Django with standalone frontend applications, such as those built with React or Next.js.

Although the MVT pattern includes the use of templates, Django is also highly flexible and can be used exclusively as a backend, serving as a data provider for frontends built on modern frameworks. In this approach, Django manages the business logic and interaction with the database, but does not generate the HTML views directly. Instead, it exposes the data through a REST API or GraphQL, allowing the frontend, developed with technologies such as React, Next.js or Vue.js, to handle the user interface. This approach is ideal for progressive web application architectures (PWA) or SPA (Single Page Applications).

Django offers several advantages that make it an excellent choice for web development. In terms of scalability, its modular design and ability to integrate with various tools make it ideal for projects that need to grow and evolve over time. It is also a highly secure framework, as it includes by default functions to protect against common threats such as SQL injection, cross-site scripting (XSS) and cross-site request forgery (CSRF). [1]

4.1.2 Next.js

Next.js is a React framework that stands out for its ability to create modern, high-performance applications. One of its main features is the ability to render both on the client and on the server, which allows optimising initial load times through Server-Side Rendering (SSR) and generating static content with Static Site Generation (SSG) to improve speed and performance in production.

It combines seamlessly with technologies such as TypeScript and Tailwind CSS, further enhancing its functionality. TypeScript adds static typing to JavaScript development, which reduces errors during coding and facilitates project maintenance, especially in large-scale applications. Its integration with Next.js is seamless, allowing developers to enjoy the benefits of clearer, safer and easier to debug code.

Tailwind CSS, on the other hand, allows you to design modern and attractive user interfaces using a utility-based approach. It eliminates the need to write custom CSS for each element by offering a wide range of predefined classes that are applied directly to components. When combined with Next.js, it significantly speeds up interface development while keeping the design consistent and easy to adjust.

In addition, Next.js includes features such as file-based routing, optimised image loading with next/image, and native API support, allowing it to handle both front and back ends in a single project. Its flexible ecosystem also encourages integration with additional libraries and tools, making it an ideal solution for scalable, modern projects. [2]

4.1.3 Excel

Excel is an essential tool for data management and analysis, especially in projects that require automation and advanced functionality. With its ability to handle large volumes of data, Excel allows you to perform calculations, organise information and generate reports with ease.

Excel is also a powerful tool for integration with other systems. In projects such as Kronis, it is used to generate automatic reports from data stored in external databases or platforms. Thanks to its flexibility, spreadsheets can be configured to receive, process and display data in real time, facilitating decision-making based on up-to-date information.

Another key advantage is its compatibility with external plug-ins and tools. This makes it possible to extend its functionality and integrate it with modern platforms, such as Airtable or project management tools, optimising workflows. Excel remains a reliable and versatile solution for managing data and automating processes in virtually any professional environment. [7]

4.1.4 Silk

Silk is a tool designed to analyse and optimise the efficiency of database queries, identifying problems such as unnecessary or repetitive queries. Its use improves the execution time of applications and ensures optimal management of server resources, which is essential in projects where performance is key, such as Kronis.

In a Django environment, integration with Silk can be done to monitor the SQL queries generated by the ORM (Object-Relational Mapping). Django Rest Framework and the Django ORM, while very powerful, can generate excessive queries if not configured properly, especially in views with complex relationships. Silk helps identify these bottlenecks and provides detailed reports so developers can optimize their code.

To integrate Silk with Django, a middleware can be configured to capture the queries made during requests. Through this integration, it is possible to analyse metrics such as the number of queries made, the average execution time of each query, and the overall impact on performance. This allows you to make informed decisions, such as implementing select related or prefetch_related, reducing the use of inefficient loops or even reorganising view logic.

With Silk, the backend is guaranteed to be more efficient, delivering a better user experience and ensuring that the application can scale without compromising performance. [3]

4.1.5 Docker

Docker is a lightweight virtualisation tool that allows you to package applications and their dependencies in isolated containers. These containers ensure that applications run consistently in any environment, regardless of the underlying operating system configurations. Docker is especially useful for ensuring portability, scalability and simplicity in managing development and production environments.

In Kronis, Docker is used to manage critical backend services, specifically through two containers: one for PostgreSQL, the main database, and one for Redis, used for tasks such as caching or handling asynchronous work queues. [8]

Configuration in the Kronis project

The Docker configuration for Kronis is centralized in environment files such as *global.env* and *postgresql.env*, where key variables for container operation are defined.

1. PostgreSQL container
   This container acts as the database server. Its configuration includes variables such as database name, user and password, all defined in the *postgresql.env* file. This ensures that credentials and settings are kept separate from code, promoting good security practices.

2. Redis container
   Redis, being configured in its own container, is responsible for handling caching and asynchronous tasks, speeding up operations and improving the overall efficiency of the system. The Redis configuration is set in the *global.env* file, which also includes other variables relevant to the overall application environment.

Advantages of using Docker in Kronis

Isolation: Each service operates in its own container, eliminating conflicts between dependencies.

Portability: Developers can easily replicate the production environment on their local machines.

Scalability: It is easy to add more containers or replicate services as needed.

Simplified maintenance: Centralised configuration in *.env* files makes it easy to update and customize environments.

With this configuration, Docker not only simplifies the management of services on Kronis, but also ensures a robust environment that is ready to scale with the demands of the project.

4.1.6 Amazon Web Services

Amazon Web Services (AWS) is one of the most widely used cloud computing platforms in the world. It offers a comprehensive set of services that enable applications to be built, deployed and scaled securely and efficiently. AWS is known for its flexibility, reliability and ability to adapt to the needs of projects of any size, from startups to global enterprises.

Figure 4 Amazon Web Services

In the case of Kronis, AWS hosts the entire platform infrastructure, providing an optimised and scalable environment for its operation. The infrastructure includes dedicated frontend and backend servers, as well as load balancers to ensure even distribution of traffic. [9]

Kronis architecture

1. Frontend server (Next.js)
   The Kronis frontend, developed in Next.js, is hosted on a dedicated server within AWS. This server handles user requests, providing a fast and efficient experience when rendering client pages. Thanks to AWS, the frontend can benefit from features such as automatic scalability, ensuring that the system handles increases in user load without interruption.

2. Backend server (Django)
   The backend, built with Django, operates on a separate server also hosted on AWS. This server handles business logic, database communication and API request processing. Keeping the frontend and backend servers separate improves security, performance and the ability to scale each component independently.

3. Load balancers
   To ensure that user traffic is distributed evenly between servers, Kronis uses load balancers provided by AWS. These load balancers allow:

   ○ High availability: If a server fails, the balancer redirects traffic to other active servers.

   ○ Efficient distribution: They distribute the requests among the servers to avoid overloads.

   ○ Scalability: They work in conjunction with the infrastructure to add or remove server instances as needed.

4.1.7 Airtable

Airtable is an online database platform that combines the simplicity of a spreadsheet with the advanced functionalities of a database. It is a powerful tool for project management, task tracking and data organisation, allowing teams to collaborate efficiently and keep information updated in real time. [10]

Airtable's main functionalities

Visual databases: Airtable offers an easy-to-use interface that allows users to manage and organise data visually through tables, calendar views, galleries or kanban. This facilitates quick access to information and collaboration between teams.

Automations: Airtable allows you to create automated workflows to reduce manual effort and streamline processes. For example, you can set up automatic notifications or create new records based on certain events.

Integrations: Airtable integrates with other tools such as Zapier, Slack, Google Drive, and more, allowing for great flexibility in connecting with different platforms and applications.

Real-time collaboration: Several people can work on the same database simultaneously, facilitating constant updating of information and collaboration between team members.

Using Airtable in Kronis

In the Kronis project, Airtable is mainly used for the management of data related to research and development projects, as well as for the monitoring of administrative and operational processes. In addition, it is used to automate certain workflows, such as updating project statuses or assigning tasks, allowing teams to maintain efficient, real-time control over the progress of activities.

With Airtable, Kronis optimises project management and improves productivity by providing a visual and flexible platform to manage tasks, resources and data, integrating seamlessly with other tools and systems used in the project.

4.1.8 Comparison between Tailwind CSS and Bootstrap

In the development of the Kronis platform, two different CSS frameworks were used for the design and layout of the interface, depending on the architecture and technologies used: Bootstrap and Tailwind CSS.

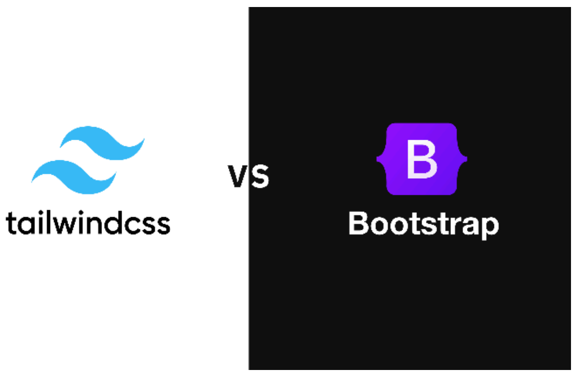


Figure 5 Tailwind vs Bootstrap

Bootstrap on the old Django model with Templates

In the traditional Django model, which used templates to render the frontend, Bootstrap was the chosen option for styling pages. Bootstrap is a predefined, modular CSS framework that offers a wide variety of ready-to-use components and styles, such as buttons, forms, navigation bars, and more. Thanks to its predefined class-based approach, developers could quickly add design and functionality to pages without having to write custom CSS from scratch. However, using Bootstrap also came with limitations in terms of flexibility, as deep customisation of styles required overwriting Bootstrap's default classes.

With the migration of the frontend to Next.js, Tailwind CSS was adopted as the main framework for creating interfaces. Unlike Bootstrap, Tailwind is a utility-first framework, which means that developers can apply specific utility classes directly to HTML elements to modify their styles. Tailwind does not include predefined components like Bootstrap, which allows for more customisation and control over the design. This approach is more flexible, as the design is built from scratch and is easy to adapt to the needs of the project, without predefined design constraints. [4] [5]

Advantages of Tailwind over Bootstrap

1. Greater control and customization: Tailwind offers much more fine-grained control over styling, allowing developers to fully customize the design without having to overwrite large parts of the default classes.

2. Faster and cleaner development: By using specific utility classes for each style property, Tailwind code is often easier to read and maintain. There is no need to search external stylesheets or write additional CSS, which improves productivity.

3. Smaller file size: Tailwind allows you to optimize the size of the final CSS file by using purge during compilation, removing unused classes, resulting in lighter files and faster load time

4.2 Tools

Throughout the development of the Kronis platform, various tools have played a critical role in streamlining workflows, enhancing productivity, and ensuring code quality. PyCharm has been an essential IDE for efficient backend development in Django, offering powerful debugging and testing features. MobaXterm has facilitated seamless remote server access and file management, making it easier to work across different environments. Additionally, Swagger has been indispensable for API documentation and testing, allowing for clear communication and validation of endpoints. Together, these tools have supported a robust and efficient development process.

4.2.1 PyCharm

PyCharm is a highly efficient IDE for Python, ideal for complex projects. It offers advanced features such as code auto-completion, integrated debugging and support for frameworks such as Django. In addition, it allows the management of virtual environments and dependencies, optimising workflow. [11]

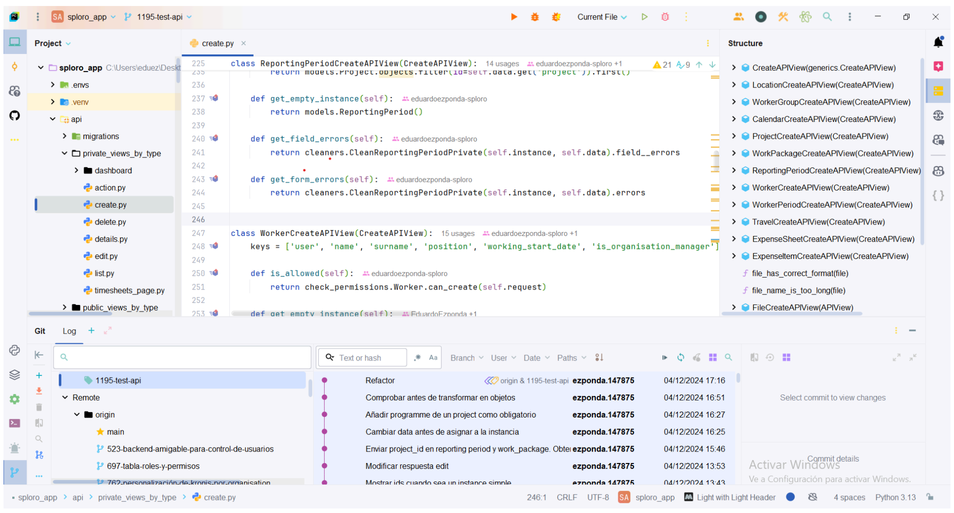
Key features

Keyboard commands: PyCharm allows for quick navigation using keyboard shortcuts, improving productivity.

Plugins: It is possible to extend its functionality with plugins such as Reloadium (automatic reloading of web applications), Key Promoter X (teaches keyboard shortcuts), and JSON Formatter (formats JSON files).

Integration with Git

PyCharm has a robust integration with Git, allowing you to manage repositories directly from the IDE. In addition, it can connect to GitHub Copilot, which enables GitHub Copilot Chat for real-time assistance as you write code, streamlining code generation and troubleshooting.



```
class ReportingPeriodCreateAPIView(CreateAPIView):
        return models.Project.objects.filter(id=self.data.get('project')).first()

    def get_empty_instance(self):
        return models.ReportingPeriod()

    def get_field_errors(self):
        return cleaners.CleanReportingPeriodPrivate(self.instance, self.data).field__errors

    def get_form_errors(self):
        return cleaners.CleanReportingPeriodPrivate(self.instance, self.data).errors


class WorkerCreateAPIView(CreateAPIView):
    keys = ['user', 'name', 'surname', 'position', 'working_start_date', 'is_organisation_manager']

    def is_allowed(self):
        return check_permissions.Worker.can_create(self.request)

    def get_empty_instance(self):
```

```
Refactor                                                 origin & 1195-test-api  ezponda.147875  04/12/2024 17:16
Comprobar antes de transformar en objetos                                        ezponda.147875  04/12/2024 16:51
Añadir programme de un project como obligatorio                                  ezponda.147875  04/12/2024 16:27
Cambiar data antes de asignar a la instancia                                     ezponda.147875  04/12/2024 16:25
Enviar project_id en reporting period y work_package. Obte                       ezponda.147875  04/12/2024 15:46
Modificar respuesta edit                                                         ezponda.147875  04/12/2024 13:53
Mostrar ids cuando sea un instance simple                                        ezponda.147875  04/12/2024 13:43
```

Figure 6 IDE PyCharm

4.2.2 MobaXterm

MobaXterm is a complete tool for remote server access and system management. Thanks to its support for protocols such as SSH and its advanced functionalities for working with files and scripts, it is an ideal choice for tasks related to application deployment.

In the case of Kronis, MobaXterm is used to perform deployments of the latest version of the code to the production environment. This process is carried out quickly and efficiently, taking advantage of its intuitive interface and scripting support. [12]

Deployment Process

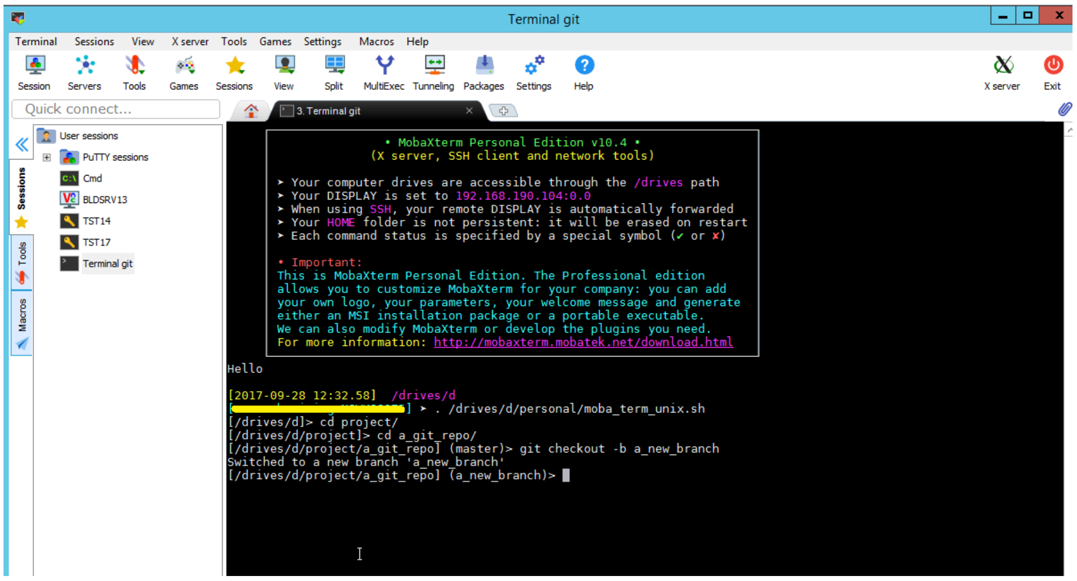
1. Server access via SSH
   With MobaXterm, users connect to the production server via a secure SSH session. The tool allows accessing remote system resources and executing commands directly from its integrated terminal.

2. Execution of the deployment script
   Once inside the server, a pre-configured script is executed that automates the process of updating the code. This script includes the following steps:

   - Navigate to the project directory.
   - Run a git pull from the main branch to get the latest version of the code.
   - Restart frontend or backend services if necessary, ensuring that changes are reflected correctly.

3. Verification of deployment

After the script is executed, basic checks are performed to ensure that the application works as expected in the production environment.



```
        • MobaXterm Personal Edition v10.4 •
      (X server, SSH client and network tools)

 ➤ Your computer drives are accessible through the /drives path
 ➤ Your DISPLAY is set to 192.168.190.104:0.0
 ➤ When using SSH, your remote DISPLAY is automatically forwarded
 ➤ Your HOME folder is not persistent: it will be erased on restart
 ➤ Each command status is specified by a special symbol (✔ or ✘)

 • Important:
 This is MobaXterm Personal Edition. The Professional edition
 allows you to customize MobaXterm for your company: you can add
 your own logo, your parameters, your welcome message and generate
 either an MSI installation package or a portable executable.
 We can also modify MobaXterm or develop the plugins you need.
 For more information: http://mobaxterm.mobatek.net/download.html

Hello

[2017-09-28 12:32.58]  /drives/d
[redacted] ➤ . /drives/d/personal/moba_term_unix.sh
[/drives/d]> cd project/
[/drives/d/project]> cd a_git_repo/
[/drives/d/project/a_git_repo] (master)> git checkout -b a_new_branch
Switched to a new branch 'a_new_branch'
[/drives/d/project/a_git_repo] (a_new_branch)> 
```

Figure 7 MobaXterm

4.2.3 Swagger

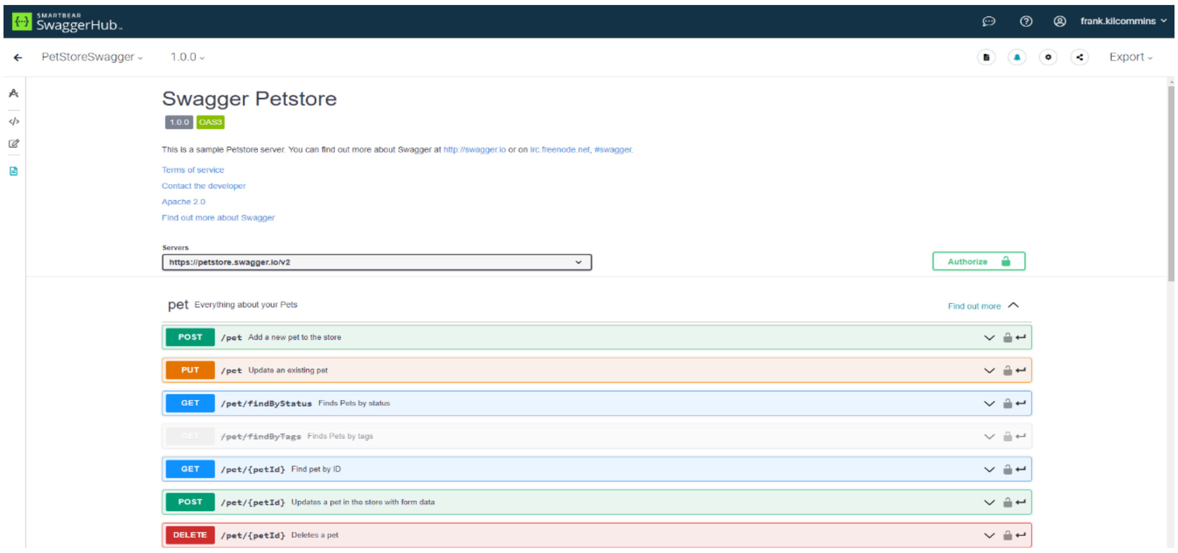
Swagger is a set of accessible and powerful tools designed to simplify the API development process for both large teams and individual developers. It facilitates the entire API lifecycle, from design to documentation, testing and deployment. Developed by SmartBear Software, a leader in software quality solutions, Swagger includes both open source and commercial tools that enable anyone from engineers to product managers to create and manage APIs. [13]

Key features

User-friendly interface: Swagger allows API testing directly from its interface, facilitating the validation of HTTP requests and responses, which improves the development and debugging process.

Integrated testing: Swagger allows API testing directly from its interface, facilitating the validation of HTTP requests and responses, which improves the development and debugging process.

OpenAPI specification: Swagger uses the OpenAPI specification, an open standard for describing the structure and behaviour of APIs, which ensures their readability for both humans and machines.



Swagger Petstore

This is a sample Petstore server. You can find out more about Swagger at http://swagger.io or on irc.freenode.net, #swagger.

Servers: https://petstore.swagger.io/v2

pet Everything about your Pets

POST	`/pet`	Add a new pet to the store
PUT	`/pet`	Update an existing pet
GET	`/pet/findByStatus`	Finds Pets by status
GET	`/pet/findByTags`	Finds Pets by tags
GET	`/pet/{petId}`	Find pet by ID
POST	`/pet/{petId}`	Updates a pet in the store with form data
DELETE	`/pet/{petId}`	Deletes a pet

Figure 8 Swagger

5 Analysis and design

5.1 Database

5.1.1 Relational Model Diagram

The relational model diagram for the database is a complex and intricate representation of how various entities in the system are related to one another. There are many tables within the database, and the diagram shown only captures a small portion of the entire schema, as it is quite expansive. The key to understanding the relationships within the model lies in recognizing how these tables interconnect, with the organisation foreign key being one of the most prevalent throughout the database. This is because, directly or indirectly, most of the tables in the system are linked to a single organization. This structure ensures that all data is consistently grouped within the context of an organization.

This reinforces the idea that the majority of operations, processes, and records within the system are performed for or by a specific organization. One of the tables that is noteworthy in this model is OrganisationLocation, which likely represents the different physical or virtual locations associated with an organization. This table may contain data on offices, departments, or geographical regions, and it serves as an important piece in understanding where various activities are taking place within the organization.

The OrganisationLocation table may be connected to various other tables, possibly through foreign keys, to represent the relationships between locations and other organizational elements. Additionally, Project and WorkPackage tables are also prominently featured. In this schema, WorkPackage and ReportingPeriod tables both contain foreign keys that reference the Project table. This relationship is critical because projects are made up of smaller components called WorkPackages.

Each WorkPackage represents a specific task, deliverable, or phase of the project, and multiple workers can be assigned to one or more work packages. The foreign key relationship ensures that the system can track the allocation of resources and responsibilities within the scope of each project. Moreover, ReportingPeriod serves as a mechanism to track time and performance within a project.

Every worker assigned to a WorkPackage must report their progress or activities within a specific ReportingPeriod. This helps capture time logs, progress updates, and other key performance indicators relevant to the workers' tasks within the WorkPackages they are involved in. These periods are essential for tracking milestones, deadlines, and the overall progress of projects, allowing organizations to have accurate insights into both individual and team-level performance.

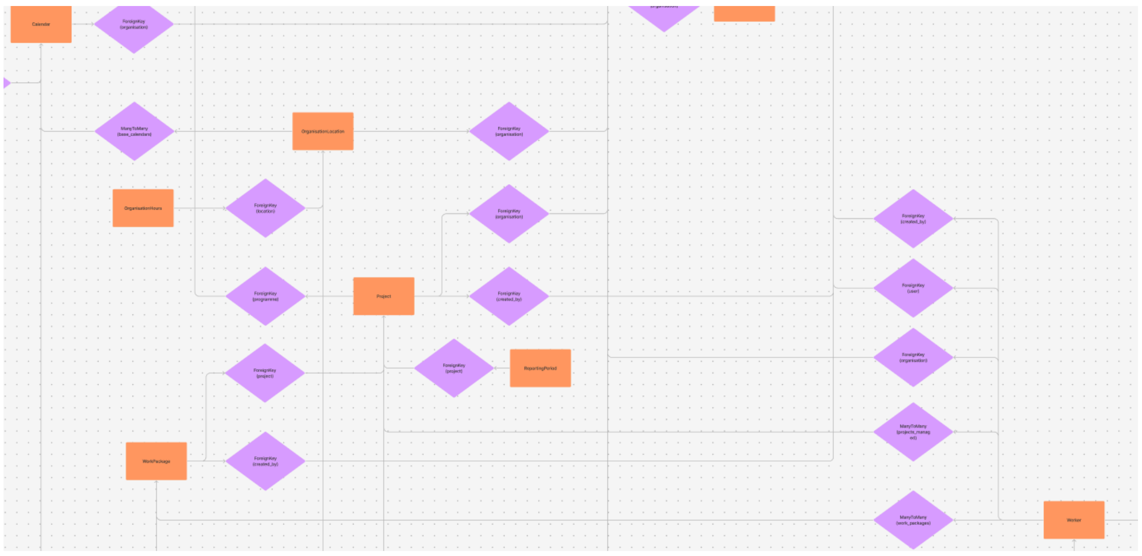


Figure 9 Relational Diagram

5.1.2 Entity-Relationship Diagram

The Entity-Relationship Diagram (ERD) represents the primary models of the Django database, focusing on the key relationships between them. These models define the core structure of the system, ensuring that data is well-organized and interconnected. The relationships primarily follow a 0 to N (one-to-many) pattern, except for those involving Organisation, which are either 1 to 1 or 1 to N (one-to-many). This structured approach ensures that the system maintains data consistency while allowing scalability.

One of the key relationships in the diagram is between ExpenseSheet and ExpenseItem. Each ExpenseItem must always belong to exactly one ExpenseSheet, making this a mandatory one-to-many relationship. However, an ExpenseSheet can contain zero or multiple ExpenseItems, allowing flexibility in expense management. This ensures that expense tracking remains structured, while also permitting ExpenseSheets without any associated items initially.

Similarly, the Project model is closely linked to the WorkPackage model. A Project can have zero or multiple WorkPackages, providing a flexible structure for project management. WorkPackages serve as subdivisions of projects, allowing finer granularity in organizing tasks, budgeting, and reporting. This hierarchy ensures that large-scale projects can be broken down into smaller, manageable components.

The Organisation model plays a central role in structuring the system's entities. Unlike other relationships that follow a 0 to N pattern, all relationships involving Organisation are either 1 to 1 or 1 to N. This means that an Organisation must always be associated with at least one related entity. For example, an Organisation can have one or many OrganisationLocations, ensuring that multiple locations are properly linked to their respective organisations. This prevents orphaned records and guarantees that every location belongs to a defined organisation.

Additional relationships in the diagram involve Worker, WorkerPeriod, ReportingPeriod, FrameworkProgramme, and Calendar. Each of these models is interconnected, ensuring that the system effectively tracks workers, their work periods, reporting structures, and organisational frameworks. This network of relationships allows for a well-integrated data model, ensuring that all aspects of the system—from travel expenses to project management—remain consistent and interdependent.

By structuring the database with these relationships, the system ensures data integrity, efficient querying, and logical organisation of entities. The balance between mandatory and optional relationships allows flexibility while maintaining strict control over critical connections, such as expenses, projects, and organisational data.

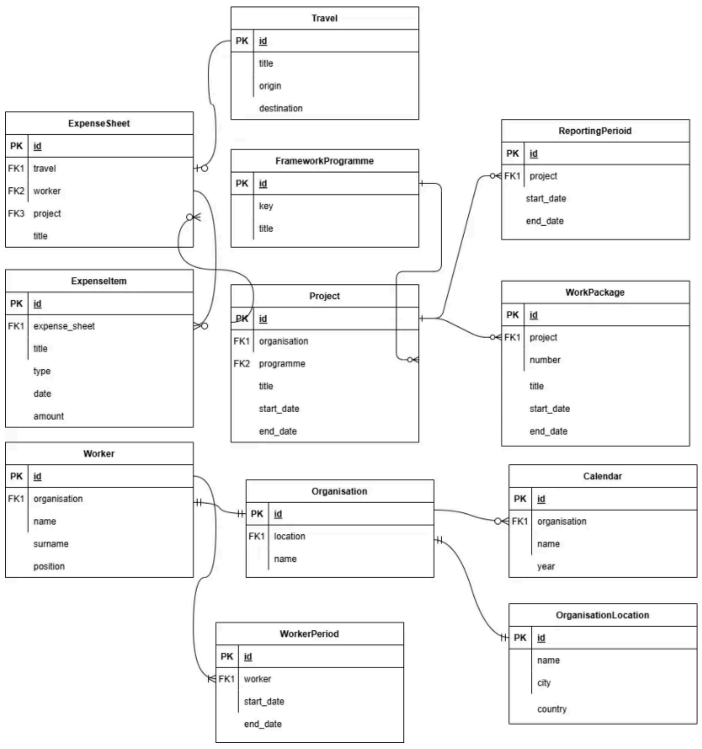


Figure 10 Entity-Relationship Diagram

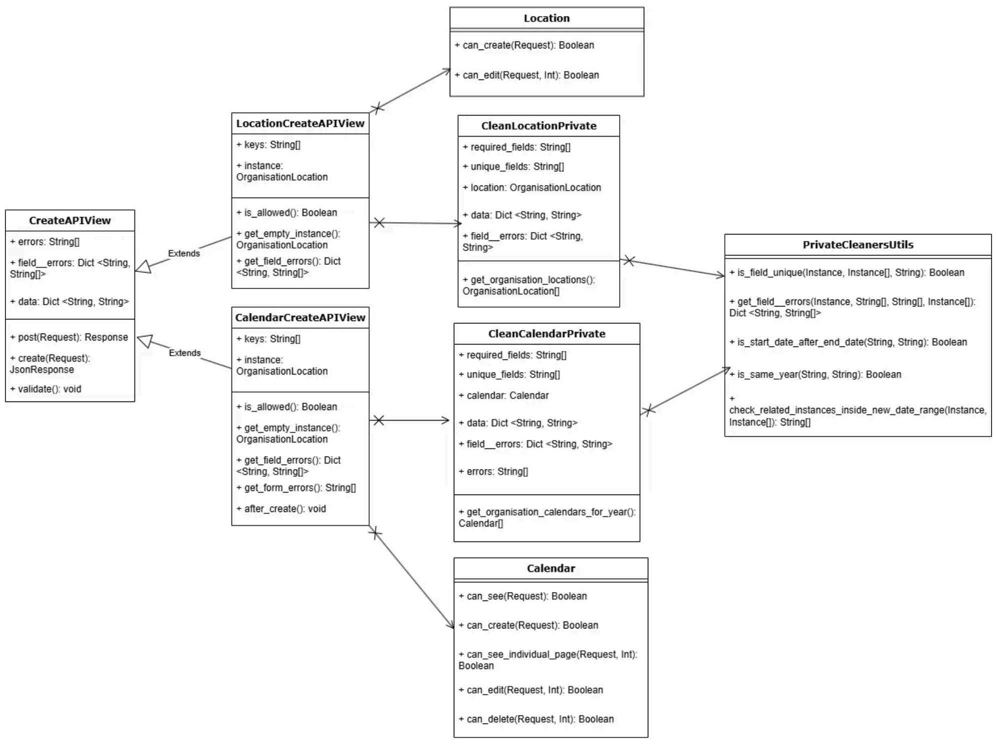
5.2 Diagrams

5.2.1 Class diagram

The class diagram focuses on the Create action of the CRUD methods, showing how different Django views handle object creation. It includes two child views, LocationCreateAPIView and CalendarCreateAPIView, which inherit from the base class CreateAPIView. This base class provides the core functionality for creating instances, and both child views rely on it. The diagram is incomplete, as in the real system, there is one child view for each model that supports creation. These two views were chosen to highlight differences in their implementation.

LocationCreateAPIView and CalendarCreateAPIView differ in their methods. CalendarCreateAPIView has an after_create() method for performing additional operations after creating an instance, while LocationCreateAPIView does not. Similarly, CalendarCreateAPIView uses get_form_errors() to handle general form-wide errors, whereas LocationCreateAPIView only validates field errors, so it doesn't need this method. These differences reflect how views are customized based on the specific needs of their associated models.

Validation is handled by cleaner classes (CleanLocationPrivate and CleanCalendarPrivate) for each view, ensuring field and general errors are checked before saving data. To avoid repeating common logic, a utility class, PrivateCleanersUtils, provides shared methods for cleaners. Additionally, the Location and Calendar classes enforce permissions for their related views, ensuring only authorized users can create instances. This design keeps the system organized, reusable, and secure.



Location
+ can_create(Request): Boolean
+ can_edit(Request, Int): Boolean
LocationCreateAPIView
+ keys: String[]
+ instance: OrganisationLocation
+ is_allowed(): Boolean
+ get_empty_instance(): OrganisationLocation
+ get_field_errors(): Dict <String, String[]>
CleanLocationPrivate
+ required_fields: String[]
+ unique_fields: String[]
+ location: OrganisationLocation
+ data: Dict <String, String>
+ field__errors: Dict <String, String>
+ get_organisation_locations(): OrganisationLocation[]
CreateAPIView
+ errors: String[]
+ field__errors: Dict <String, String[]>
+ data: Dict <String, String>
+ post(Request): Response
+ create(Request): JsonResponse
+ validate(): void
Extends
Extends
PrivateCleanersUtils
+ is_field_unique(Instance, Instance[], String): Boolean
+ get_field__errors(Instance, String[], String[], Instance[]): Dict <String, String[]>
+ is_start_date_after_end_date(String, String): Boolean
+ is_same_year(String, String): Boolean
+ check_related_instances_inside_new_date_range(Instance, Instance[]): String[]
CalendarCreateAPIView
+ keys: String[]
+ instance: OrganisationLocation
+ is_allowed(): Boolean
+ get_empty_instance(): OrganisationLocation
+ get_field_errors(): Dict <String, String[]>
+ get_form_errors(): String[]
+ after_create(): void
CleanCalendarPrivate
+ required_fields: String[]
+ unique_fields: String[]
+ calendar: Calendar
+ data: Dict <String, String>
+ field__errors: Dict <String, String>
+ errors: String[]
+ get_organisation_calendars_for_year(): Calendar[]
Calendar
+ can_see(Request): Boolean
+ can_create(Request): Boolean
+ can_see_individual_page(Request, Int): Boolean
+ can_edit(Request, Int): Boolean
+ can_delete(Request, Int): Boolean

Figure 11 Create Class diagram

5.3 Hexagonal architecture

The Kronis project has started to implement the hexagonal architecture, also known as Ports and Adapters. This approach seeks to separate the core business logic from external dependencies, such as databases, user interfaces or third-party services. Doing so facilitates system evolution and maintenance, as changes to external interfaces do not directly affect the core application. This architecture promotes reusability and independent testing of the different parts of the system.

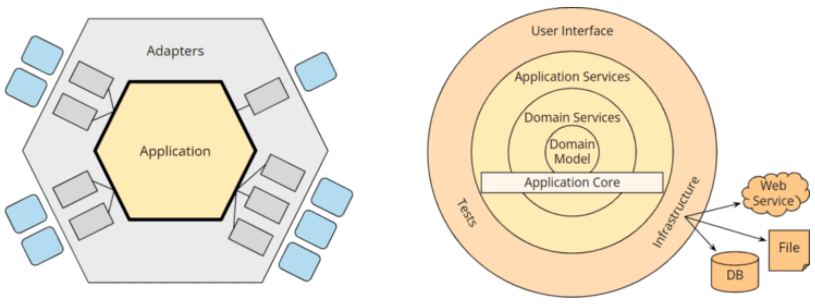


Figure 12 Hexagonal architecture

5.4 Vertical Slice

The vertical slice approach has been adopted, which organises the code so that each ‘slice’ of the system covers all the functionality of a specific use case. Unlike traditional architectures, where layers are grouped by type of function (such as drivers, services or repositories), with vertical slice each functionality has its own structure, ranging from business logic to user interface. This improves clarity, facilitates teamwork and enables complete and independent functionality to be delivered faster and more efficiently.

5.5 Components

5.5.1 Project structure in Django

Django is a robust and versatile web framework that follows the Model-View-Template (MVT) pattern. This pattern organizes the code into three main components:

Models: They represent the logic and structure of the application's data, defined in the form of Python classes that are translated into tables in the database.

Views: They handle the application logic. They receive HTTP requests from the client, interact with the models and return a response (usually a JSON or HTML template).

Templates: They handle the presentation, displaying dynamic data in HTML format for the end user. (Optional, they are not used in our case.)

Flow of a request

The typical flow of a request in Django is as follows:

1. A Client (usually the frontend) makes an HTTP request.
2. The corresponding URL is handled by the router (*urls.py*), which redirects the request to the appropriate view.
3. The view performs operations on the models and returns a response, such as a JSON for APIs or a rendered HTML in the case of using templates.
4. If Django used templates, the model data would be sent to an HTML template file, where it would be dynamically rendered before being sent to the client.

Example with Templates

If instead of using a separate frontend (like Next.js), Django would directly handle views with templates, the structure would be something like:

1. The client accesses a URL such as */profile/*.
2. The router redirects the request to a view called *profile_view*.
3. The view queries the model for the user's profile data and passes this information to an HTML template called *profile.html*.
4. El template *profile.html* renderiza los datos y los devuelve al cliente como una página web completa.

In our case, however, the frontend is decoupled, and Django responds with data in JSON format that the *frontend* uses to display information.

Running Django with Applications

Django organises project development in a modular, app-based structure. An app in Django is a self-contained unit that encapsulates specific functionality, making it easy to reuse across different projects. Each application has its own directory structure, which includes files for models, views, tests, and more.

Project Applications

Users application

The users application is responsible for managing all user-related aspects of the system. It includes:

Authentication management: User registration, login and password recovery.

User profiles: Each user has an associated profile with additional details required for system operations.

Roles and permissions: Allows different roles and permissions to be assigned to access specific functionalities.

Integration with middleware: It uses middleware that extracts the User and Worker objects associated with the authentication token, facilitating the verification of permissions and access to user-specific data.

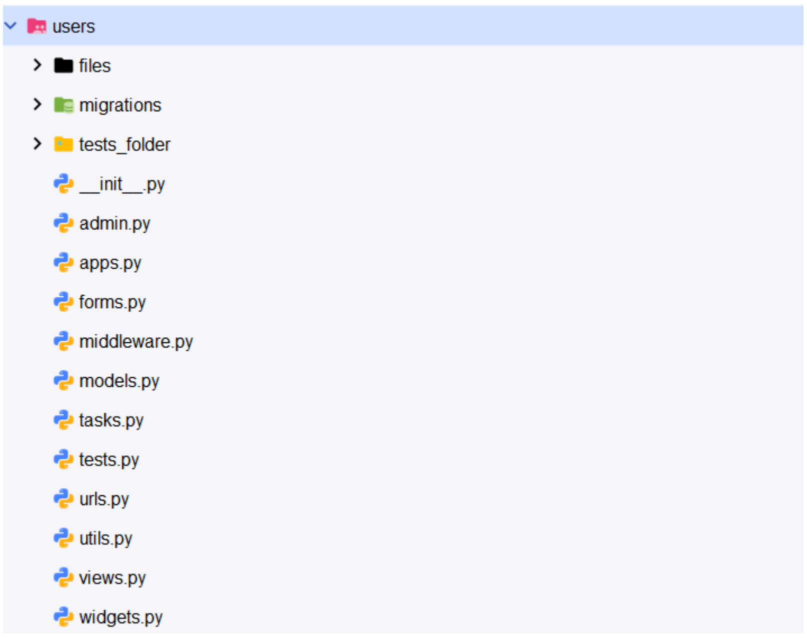

```
users
  files
  migrations
  tests_folder
  __init__.py
  admin.py
  apps.py
  forms.py
  middleware.py
  models.py
  tasks.py
  tests.py
  urls.py
  utils.py
  views.py
  widgets.py
```

Figure 13 Users application

Timesheets application

This application manages most of the operational functionalities related to the organisation, workers and projects. It contains several models, each with specific roles and purposes:

Organisation: Represents the organisations using the system, including attributes such as name, address, city, country and website.

OrganisationLocation: Defines the locations associated with an organisation, with details such as name, address, city and country.

Framework Programme: Model that stores framework programmes with keys and titles.

Project: Manages projects with attributes such as title, acronym, partner organisation, programme, start and end date.

ReportingPeriod: A model that represents a defined time frame within a project, marked by a start date and an end date, for tracking and reporting purposes.

WorkPackage: Organises work within a project into packages, with attributes such as number, title, associated project, dates and assigned person-months.

Worker: Represents the employees, their roles within the organisation, and the relationship with projects and other employees.

WorkerGroup: Allows grouping workers within an organisation to facilitate collective management.

WorkerPeriod: A model that represents a specific time period during which a worker is actively associated with an organization or project.

EmploymentAgreement: Manages labour agreements, including an associated file for reference.

Calendar: Defines an organisation's work schedules, differentiating between working days and public holidays.

Timesheet: Detailed record of the hours worked by each worker on a specific project.

Travel: Manages project-related travel information, including dates, origin, destination, title and participants.

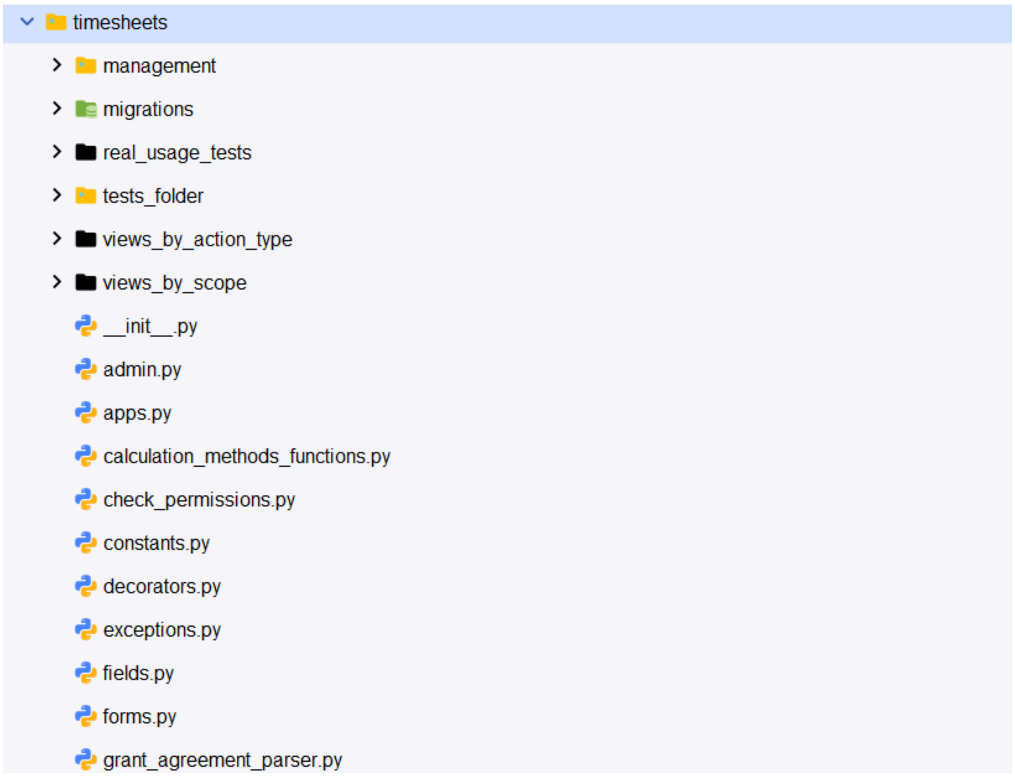
ExpenseSheet: Manages expense sheets for workers, projects and travel, with validation and association of responsible parties.

ExpenseItem: Details the individual elements of an expense sheet, such as title, type, date, amount and associated work package.

Project Applications

Key functionalities:

- Crud actions: Creating, listing, editing and deleting objects
- Analytical graphs: Generation of time, cost and project related charts.
- Subscription management: Changes to the organisation's underwriting plans.
- Staff planning: Assigning employees to projects and tasks.
- Evidence of costs: Creating evidence for reporting personal costs.
- Bulk loading: Allows the creation of objects through files.
- Validation: Verification of expense sheets and time records.



```
timesheets
  management
  migrations
  real_usage_tests
  tests_folder
  views_by_action_type
  views_by_scope
  __init__.py
  admin.py
  apps.py
  calculation_methods_functions.py
  check_permissions.py
  constants.py
  decorators.py
  exceptions.py
  fields.py
  forms.py
  grant_agreement_parser.py
```

Figure 14 Timesheets application

Sploro_app application

This application acts as the core of the project, coordinating the interaction between different applications and providing global configurations. It includes:

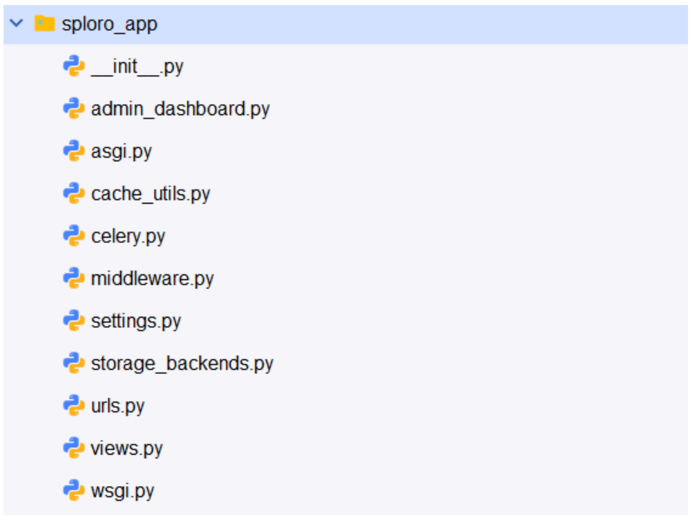
Administrative routes:

- URLs for the access to the administration panel.
- Integration with Silk for performance analysis

Cache management: Configuration to improve system performance and response times.

Error control: Custom drivers for 404, 500, 403 and 400 errors.

- Routing: Defines the primary URLs that redirect to secondary applications.



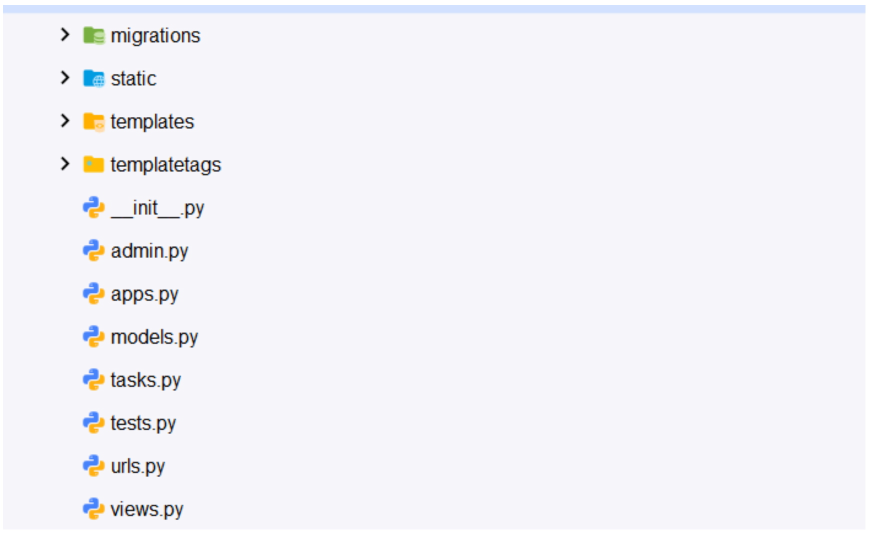
```
splоro_app
    __init__.py
    admin_dashboard.py
    asgi.py
    cache_utils.py
    celery.py
    middleware.py
    settings.py
    storage_backends.py
    urls.py
    views.py
    wsgi.py
```

Figure 15 Sploro_app application

Public application

This application manages the public views of the system, accessible without authentication. It contains:

- Favicon view: Provides the system icon for the browser.
- Home view: Public homepage describing the system and its functionalities.



```
> migrations
> static
> templates
> templatetags
  __init__.py
  admin.py
  apps.py
  models.py
  tasks.py
  tests.py
  urls.py
  views.py
```

Figure 16 Public application

Api application

The api application in the Kronis project plays a fundamental role in the migration of the system from an HTML template-based approach to an architecture that exclusively uses JSON for communication between the frontend and the backend. All requests between the frontend (React) and the backend (Django) are handled through organised and structured endpoints, allowing for efficient, clear and scalable interaction.

Structure of the api folder

Migrations folder

This folder stores the migration files generated by Django for the database. Migrations are incremental versions that reflect changes made to the project's models.

Purpose: They allow to synchronise the status of the models with the database.

Workflow: Each time a change is made to the models (such as adding a new field or modifying a relationship), a new migration is created with the makemigrations command. It is then applied to the database with migrate.

Private_views_by_type folder

This sub-folder organises private views according to the type of CRUD action they perform (create, list, edit, delete). Each file contains the logic associated with a specific functionality of the system.

Kronis dashboard views

Also included within this folder are the views that feed into the Kronis homepage dashboard, where key graphs and analysis related to time, costs, and projects are displayed.

- Objective: Enable the generation of dynamic analytical data, such as graphs of hours worked, cumulative costs, and active project statistics.
- Funcionamiento: Enable the generation of dynamic analytical data, such as graphs of hours worked, cumulative costs, and active project statistics.

Action.py file

This file groups specific actions related to travel management (Travel) in the system. These actions do not fit into the standard Crud operations, as they represent status changes or administrative processes.

1. Close a travel: Mark a trip as completed, preventing subsequent modifications.

2. Request opening of a travel: Allows a user to request the opening of a closed travel.

3. Cancel request to open a travel: Refuses a pending application for reopening.

4. Reopen the travel: Reopens a closed travel, enabling modifications.

5. Accept request to open a travel: Authorises a previously made opening request.

6. Rejecting a request to open a trip: Denies a request to open a trip.

Each of these actions includes validations to ensure that operations are consistent and respect business rules.

Files create.py, delete.py, edit.py and list.py

These files contain the main logic of the Crud operations (Create, Delete, Edit, List) for the objects defined in the timesheet models. These are the details of each file:

- create.py: Defines the views needed to create new model instances. Includes validations to ensure the integrity of the data sent by the user.

- delete.py: Contains views to delete specific records. Uses the object ID as a parameter to locate and delete the instance.

- edit.py: Allows updating attributes of an existing object, validating both the data sent and the state of the object.

- list.py: Provides views to list objects. Supports pagination and dynamic filters to optimise the handling of large volumes of data.

Tests folder

This folder includes unit tests to ensure that Crud endpoints and travel actions work correctly. These are the key features:

- Endpoints Crud: Each operation (create, delete, edit, list) is thoroughly tested.

- Travel actions: The different states and transitions of the trips are validated to ensure consistency.

- Code coverage: The tests verify both success and failure scenarios.

Urls_files folder

This folder organises the URL paths of the endpoints in different files according to functionality.

- Modular structure: Allows each file to handle paths to specific functionality, such as *private_views*.

- Redirection: Each path points to a corresponding view in private_views_by_type, following the pattern */api/private/action/object*

    Action: create, list, edit, delete

File private_serializers.py

Serializers are Django REST Framework classes that convert complex model data into more manageable formats, such as JSON.

Function:

1. Serialise objects from the models to send them to the frontend in JSON format.

2. Deserialize JSON data from the frontend to validate and create or update objects in the database.

Example: A serializer for the Worker model could convert a Worker object into a JSON object with attributes such as name, surname and position.

File urls.py

This main file defines the base path for the private views, connecting them to the specific paths within *private_views_by_type*.

Path base: */api/private/*

- Redirection: Groups all URLs related to CRUD operations and specific actions.

File utils.py

This file contains auxiliary functions used in multiple parts of the application to avoid duplication of code.

Example of common functions:
- Date validation.
- Generation of unique keys for objects.
- Utilities for handling data formats.

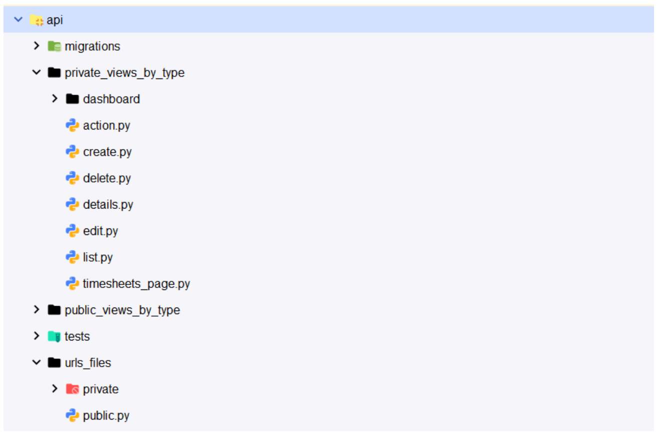


Figure 17 Api application

5.6 Frontend project

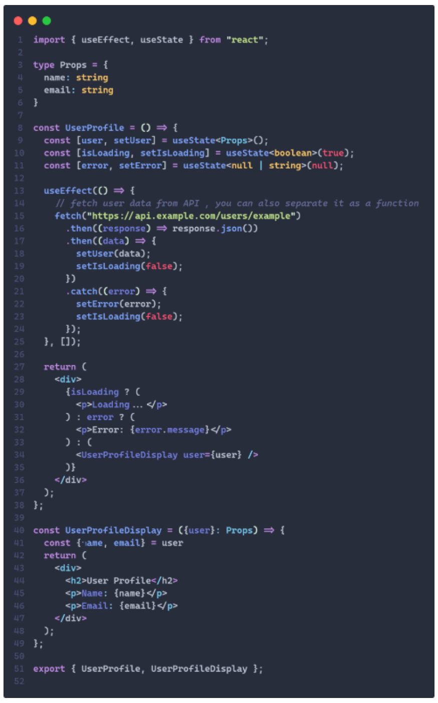
The frontend of Kronis was developed using ReactJS and follows the SOLID principles to ensure a maintainable, scalable, and modular design. Below is a breakdown of how these principles have been applied in the project:

Single Responsibility Principle

Each React component is designed to handle a single responsibility, making the code easier to understand and maintain.

Example: Instead of combining data fetching and rendering logic in a single component, these responsibilities are split. For instance:

- UserProfileFetcher handles API requests and data retrieval.
- UserProfileDisplay focuses on rendering the fetched user data. This separation ensures that each component has a clear and focused purpose.



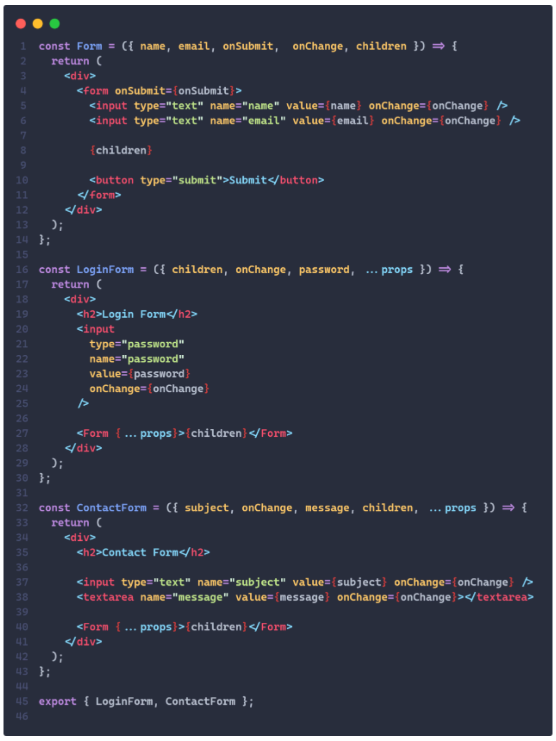
```
import { useEffect, useState } from "react";

type Props = {
  name: string
  email: string
}

const UserProfile = () => {
  const [user, setUser] = useState<Props>();
  const [isLoading, setIsLoading] = useState<boolean>(true);
  const [error, setError] = useState<null | string>(null);

  useEffect(() => {
    // fetch user data from API , you can also separate it as a function
    fetch("https://api.example.com/users/example")
      .then((response) => response.json())
      .then((data) => {
        setUser(data);
        setIsLoading(false);
      })
      .catch((error) => {
        setError(error);
        setIsLoading(false);
      });
  }, []);

  return (
    <div>
      {isLoading ? (
        <p>Loading ... </p>
      ) : error ? (
        <p>Error: {error.message}</p>
      ) : (
        <UserProfileDisplay user={user} />
      )}
    </div>
  );
};

const UserProfileDisplay = ({user}: Props) => {
  const {name, email} = user
  return (
    <div>
      <h2>User Profile</h2>
      <p>Name: {name}</p>
      <p>Email: {email}</p>
    </div>
  );
};

export { UserProfile, UserProfileDisplay };
```

Figure 18 Single Responsibility Principle

Open/Closed Principle (OCP)

Components in the frontend are designed to be open for extension but closed for modification.

Example: A base Form component is created to handle common form functionalities, such as validation and layout. Specialized forms like LoginForm and ContactForm extend this base component, adding specific features without altering the core Form logic. This approach allows for feature additions without risking regressions.



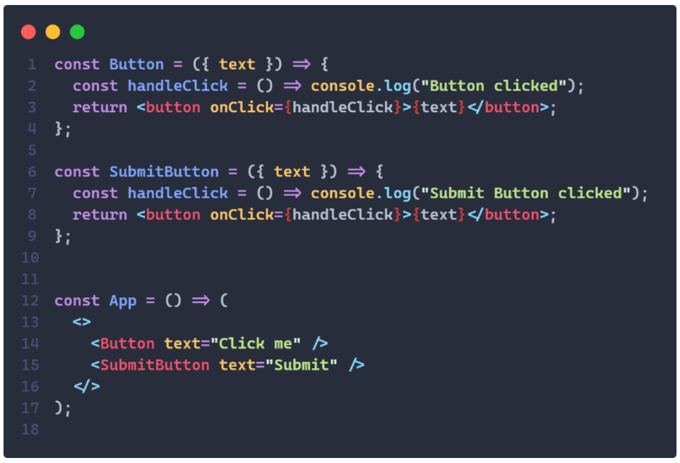
```jsx
const Form = ({ name, email, onSubmit,  onChange, children }) => {
  return (
    <div>
      <form onSubmit={onSubmit}>
        <input type="text" name="name" value={name} onChange={onChange} />
        <input type="text" name="email" value={email} onChange={onChange} />

        {children}

        <button type="submit">Submit</button>
      </form>
    </div>
  );
};

const LoginForm = ({ children, onChange, password, ...props }) => {
  return (
    <div>
      <h2>Login Form</h2>
      <input
        type="password"
        name="password"
        value={password}
        onChange={onChange}
      />

      <Form {...props}>{children}</Form>
    </div>
  );
};

const ContactForm = ({ subject, onChange, message, children, ...props }) => {
  return (
    <div>
      <h2>Contact Form</h2>

      <input type="text" name="subject" value={subject} onChange={onChange} />
      <textarea name="message" value={message} onChange={onChange}></textarea>

      <Form {...props}>{children}</Form>
    </div>
  );
};

export { LoginForm, ContactForm };
```

Figure 19 Open/Closed Principle

Liskov Substitution Principle (LSP)

Child components can seamlessly replace their parent components without disrupting the application's behavior.

Example: The Button component serves as a base for other buttons, such as SubmitButton or CancelButton. These specialized buttons inherit from the Button component and maintain the same interface, ensuring they can be used interchangeably in the UI.



```jsx
const Button = ({ text }) => {
  const handleClick = () => console.log("Button clicked");
  return <button onClick={handleClick}>{text}</button>;
};

const SubmitButton = ({ text }) => {
  const handleClick = () => console.log("Submit Button clicked");
  return <button onClick={handleClick}>{text}</button>;
};


const App = () => (
  <>
    <Button text="Click me" />
    <SubmitButton text="Submit" />
  </>
);
```

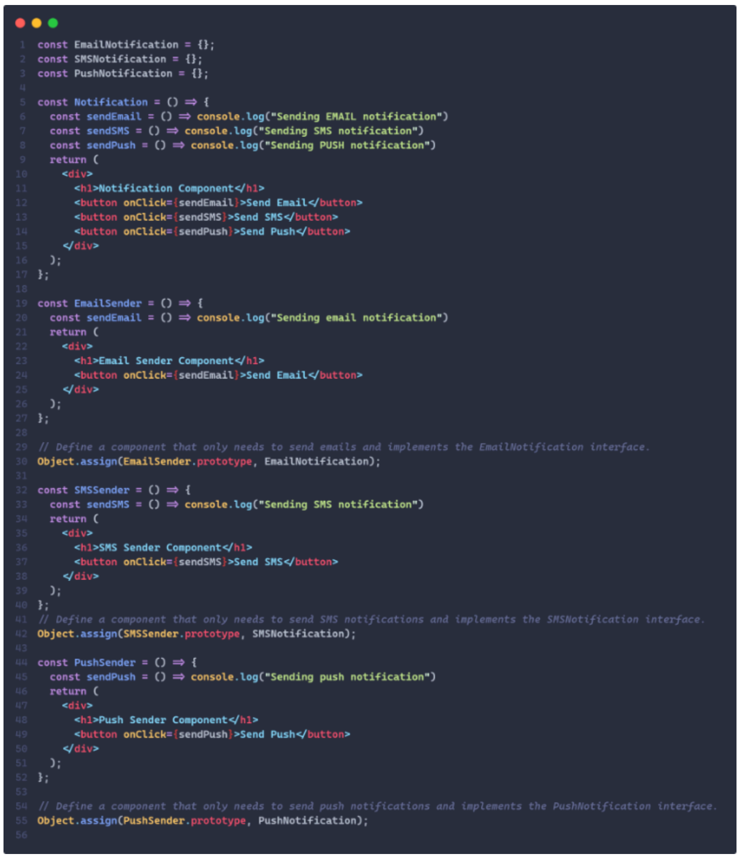
Figure 20 Liskov Substitution Principle

Interface Segregation Principle (ISP)

Components are designed with only the necessary properties and methods required for their specific tasks.

Example: Notification functionalities are split into separate components:

- EmailNotification handles email notifications.
- SMSNotification handles SMS notifications.
- PushNotification handles push notifications. This modular design prevents components from being burdened with unnecessary functionality and keeps the codebase clean.



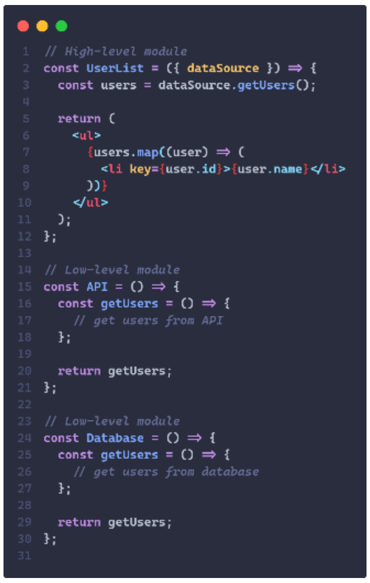
```
const EmailNotification = {};
const SMSNotification = {};
const PushNotification = {};

const Notification = () => {
  const sendEmail = () => console.log("Sending EMAIL notification")
  const sendSMS = () => console.log("Sending SMS notification")
  const sendPush = () => console.log("Sending PUSH notification")
  return (
    <div>
      <h1>Notification Component</h1>
      <button onClick={sendEmail}>Send Email</button>
      <button onClick={sendSMS}>Send SMS</button>
      <button onClick={sendPush}>Send Push</button>
    </div>
  );
};

const EmailSender = () => {
  const sendEmail = () => console.log("Sending email notification")
  return (
    <div>
      <h1>Email Sender Component</h1>
      <button onClick={sendEmail}>Send Email</button>
    </div>
  );
};

// Define a component that only needs to send emails and implements the EmailNotification interface.
Object.assign(EmailSender.prototype, EmailNotification);

const SMSSender = () => {
  const sendSMS = () => console.log("Sending SMS notification")
  return (
    <div>
      <h1>SMS Sender Component</h1>
      <button onClick={sendSMS}>Send SMS</button>
    </div>
  );
};
// Define a component that only needs to send SMS notifications and implements the SMSNotification interface.
Object.assign(SMSSender.prototype, SMSNotification);

const PushSender = () => {
  const sendPush = () => console.log("Sending push notification")
  return (
    <div>
      <h1>Push Sender Component</h1>
      <button onClick={sendPush}>Send Push</button>
    </div>
  );
};

// Define a component that only needs to send push notifications and implements the PushNotification interface.
Object.assign(PushSender.prototype, PushNotification);
```

Figure 21 Interface Segregation Principle

Dependency Inversion Principle (DIP)

The frontend components rely on abstractions rather than concrete implementations, ensuring flexibility and ease of testing.

Example: The UserList component interacts with an abstract data source interface instead of directly relying on an API or database. This abstraction allows UserList to work seamlessly with different data providers (e.g., REST APIs, GraphQL endpoints) without requiring changes to the component itself.



```javascript
// High-level module
const UserList = ({ dataSource }) => {
  const users = dataSource.getUsers();

  return (
    <ul>
      {users.map((user) => (
        <li key={user.id}>{user.name}</li>
      ))}
    </ul>
  );
};

// Low-level module
const API = () => {
  const getUsers = () => {
    // get users from API
  };

  return getUsers;
};

// Low-level module
const Database = () => {
  const getUsers = () => {
    // get users from database
  };

  return getUsers;
};
```

Figure 22 Dependency Inversion principle

This structured approach ensures that the frontend is robust, easy to extend, and aligns with best practices in modern web development. [6]

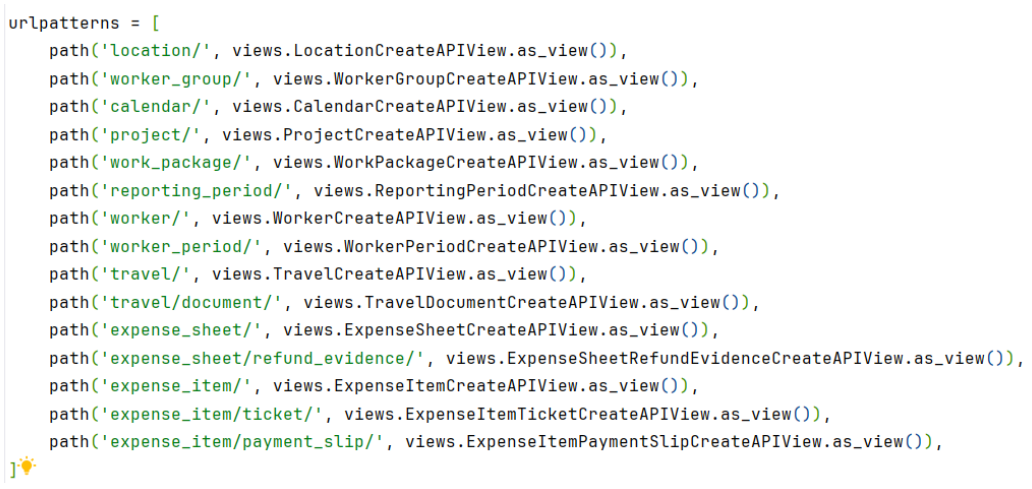
5.7 Project modifications

5.7.1 Object creation (Create)

Among the modifications made to the project, one of the most relevant corresponds to the implementation of the object creation functionality. This section details the design and structure of this functionality, highlighting the advantages of the implemented architecture and explaining the details of the code used.

Definition of URLs for creation

The paths for the object creation actions are defined in a file called *create.py*, located inside the urls_files directory. This file contains the paths under the general structure /api/private/create/object/, where the final part of the URL (‘object’) determines the redirection to the corresponding view.
For example:



```
urlpatterns = [
    path('location/', views.LocationCreateAPIView.as_view()),
    path('worker_group/', views.WorkerGroupCreateAPIView.as_view()),
    path('calendar/', views.CalendarCreateAPIView.as_view()),
    path('project/', views.ProjectCreateAPIView.as_view()),
    path('work_package/', views.WorkPackageCreateAPIView.as_view()),
    path('reporting_period/', views.ReportingPeriodCreateAPIView.as_view()),
    path('worker/', views.WorkerCreateAPIView.as_view()),
    path('worker_period/', views.WorkerPeriodCreateAPIView.as_view()),
    path('travel/', views.TravelCreateAPIView.as_view()),
    path('travel/document/', views.TravelDocumentCreateAPIView.as_view()),
    path('expense_sheet/', views.ExpenseSheetCreateAPIView.as_view()),
    path('expense_sheet/refund_evidence/', views.ExpenseSheetRefundEvidenceCreateAPIView.as_view()),
    path('expense_item/', views.ExpenseItemCreateAPIView.as_view()),
    path('expense_item/ticket/', views.ExpenseItemTicketCreateAPIView.as_view()),
    path('expense_item/payment_slip/', views.ExpenseItemPaymentSlipCreateAPIView.as_view()),
]
```

Figure 23 Url patterns for creation

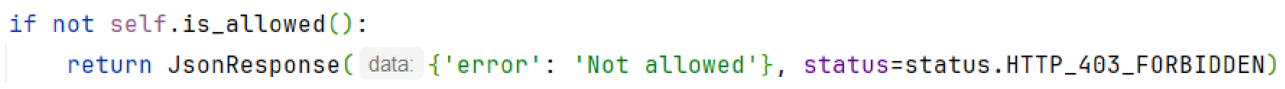
In this way, each type of object has a specific path associated with its respective view.

Architecture of views

In the views based on CreateAPIView classes that we have implemented in the project, the responses returned to the client vary according to the result of the validations, the user's permissions, and the success or failure of the instance creation. These responses are handled through the use of JsonResponse, which allows structured data to be returned in JSON format along with the corresponding HTTP code. The main response return possibilities are described below:

1. Error response for lack of permissions (403 Forbidden)

If the user does not have the necessary permissions to create the object, a response is returned with the error message ‘Not allowed’ and an HTTP status code 403. This case is evaluated by the is_allowed method, which checks whether the necessary authorisation conditions are satisfied.



```
if not self.is_allowed():
    return JsonResponse( data: {'error': 'Not allowed'}, status=status.HTTP_403_FORBIDDEN)
```

Figure 24 Is allowed method for creation

2. Validation error response (400 Bad Request)

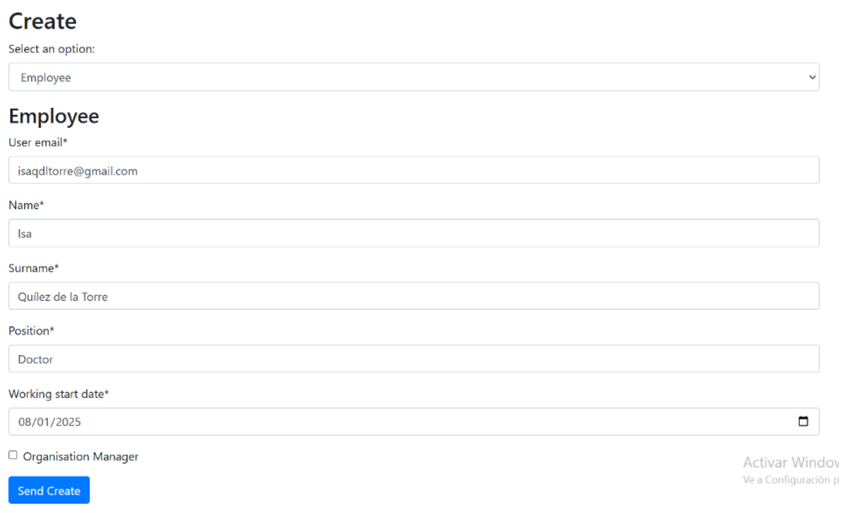
In case errors are detected during the validation process, either at a general level or in specific fields, a response with a status code 400 is returned.

- General errors: They are stored in the errors attribute and are included in the ‘error’ key of the JSON.

- Field errors: They are stored in field__errors, a dictionary where each key represents a specific field and the value corresponds to the associated errors. This format allows the client to identify precisely which data is invalid.

Example of a response:

The image below shows an example where an employee is created with the following data:



Create

Select an option:

Employee

Employee

User email*

isaqdltorre@gmail.com

Name*

Isa

Surname*

Quilez de la Torre

Position*

Doctor

Working start date*

08/01/2025

☐ Organisation Manager

Send Create

Figure 25 Creation of an employee

With this data, a request is made to create an Employee object. The output when this request is sent is:

```
{
  "error": [],
  "field__errors": {
    "user": [
      "Only corporate email addresses are allowed. If you think this is a corporate email, contact us."
    ]
  }
}
```

Figure 26 Error creating employee

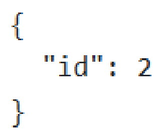
In the errors key, an empty list of general errors is returned. However, the dictionary field_errors is returned, indicating that there is an error with the user field, that only corporate email addresses are allowed, and in this case the Gmail email isaqdltorre@gmail.com is being used.

Therefore, whenever there is at least one error in both errors and field__errors, an HTTP error code will be returned with a detailed message explaining the error.

Successful response (201 Created)

When the validation is successful and the object instance is created without problems, the instance is saved to the database using *self.instance.save()* and a success response is returned with status code 201. The JSON contains the id of the newly created instance, allowing the client to identify the created resource.

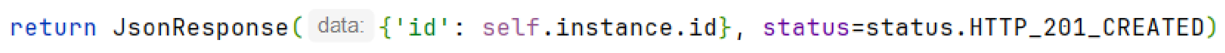
Example of a response:



```
{
  "id": 2
}
```

Figure 27 Creation response

The corresponding code fragment is:



```
return JsonResponse( data: {'id': self.instance.id}, status=status.HTTP_201_CREATED)
```

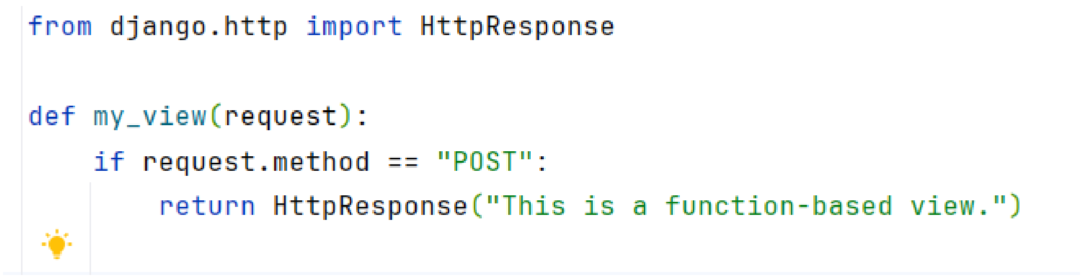
Figure 28 Sending instance id in creation

Types of views in Django and characteristics of project views

In Django, there are two main types of views, each with a different approach to handling HTTP requests:

1. Function-Based Views (FBVs)

These are Python functions that receive a request and return an HTTP response. This approach provides a high degree of flexibility, but requires writing more repetitive code to handle features such as permissions, serialisation or exception handling. Example of a function-based view:

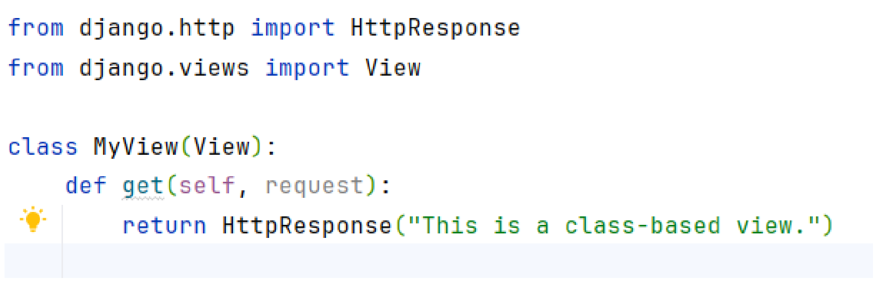


```
from django.http import HttpResponse

def my_view(request):
    if request.method == "POST":
        return HttpResponse("This is a function-based view.")
```

Figure 29 Function-Based View

2. Class-Based Views (CBVs)

Class-based views are classes that represent the HTTP request handling logic. This approach allows code reuse through inheritance, structured code organisation, and overwriting of predefined methods to adjust behaviour as needed.
Basic example:



```python
from django.http import HttpResponse
from django.views import View

class MyView(View):
    def get(self, request):
        return HttpResponse("This is a class-based view.")
```

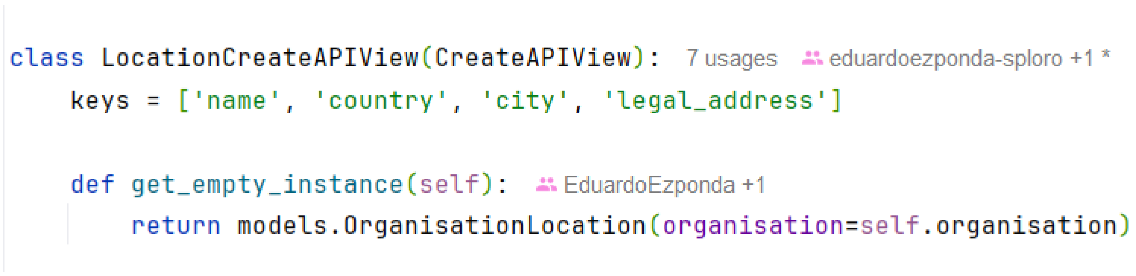
Figure 30 Class-Based View

Characteristics of the project views

In the context of the Kronis project, all creation views are class-based views that inherit from CreateAPIView. These views conform to Django's best practices and are designed to handle object creation efficiently and consistently. As CBVs, they are best suited for large-scale projects like Kronis, where modularity and scalability are essential. This has the following advantages:

- Code reuse: The base CreateAPIView class defines generic methods that are shared by all child views, such as *validate*, *is_allowed*, and *after_create*.
- Extensibility: Child views can override specific methods to customise the behaviour according to the type of object they manage.
- Legibility and organisation: Encapsulating the logic in classes and methods makes the code easier to understand and maintain.

A concrete example of one of the project views would be:



```python
class LocationCreateAPIView(CreateAPIView):  7 usages  eduardoezponda-sploro +1 *
    keys = ['name', 'country', 'city', 'legal_address']

    def get_empty_instance(self):  EduardoEzponda +1
        return models.OrganisationLocation(organisation=self.organisation)
```

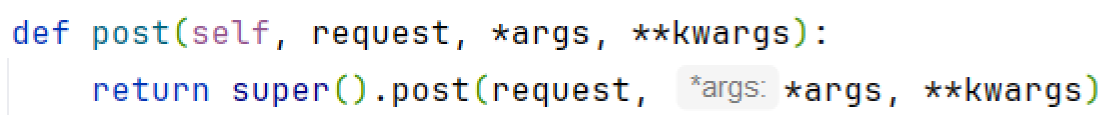
Figure 31 LocationCreateAPIView

Details of the CreateAPIView base class

The CreateAPIView base class establishes the general flow of creation actions, providing methods that can be customized by child views. The most relevant aspects are described below:

Request method

In the project, the HTTP method used for resource creation is POST, in line with REST architecture best practices. This method is specifically used to send data to the server and request the creation of a new resource in the database. The behaviour of the POST method in the project's views is based on the use of the Django Rest Framework (DRF) CreateAPIView base class. This method is overridden in the parent CreateAPIView view of our project to customize its functionality:



```
def post(self, request, *args, **kwargs):
    return super().post(request, *args: *args, **kwargs)
```

Figure 32 Post method for creation

Use of super()

The use of super() in Python allows you to access the methods or attributes of a parent class directly from the child class. In this case, *super().post(request, *args, **kwargs)* calls the post method of DRF's CreateAPIView base class. In this way, the request is initialised, passing through the middleware.

Relation to DRF's CreateAPIView

The parent class that defines this behaviour in our project is an extension of the CreateAPIView class provided by DRF. The latter already includes a default implementation of the post method to handle object creation.

- CreateAPIView in DRF: It is a generic class that combines the logic for resource creation with automatic handling of serialisation, validation and HTTP response. Overriding its post method in our project allows us to customise how data and additional validations are handled before the instance is saved to the database.

Flow of the POST method

The basic flow when making a POST request is as follows:

1. Data reception: The data is sent in the body of the request and captured in *request.data*.

2. Validation of permissions: The *is_allowed* method checks if the user has the necessary permissions to perform the creation.

3. Object Instantiation: *get_empty_instance* is used to create an empty instance of the object to be stored.

4. Data validation: Data are validated by specific methods that check for general errors and errors in individual fields.

5. Save instance: If there are no errors, *instance.save()* is called to save the object to the database.

6. Return of reply: Depending on the result of the validations, a response is returned with the corresponding HTTP code (201, 400, 403).

Validation and permissions

Permission validation is performed through the *is_allowed* method, executed before processing the request. This approach ensures that only authorised users can create resources.

Error structure

The class uses two structures to handle errors:

- field__errors: A dictionary where the keys represent the form fields and the values contain the corresponding error messages. The use of the double underscore (__) helps to differentiate these errors from other types of information.

- errors: Stores general errors, such as inconsistencies in dates (e.g. a start date later than the end date).

Both structures are populated through validations performed in auxiliary classes called Cleaners (more on this later).

Saving and post-processing process

Once the data has been validated and the absence of errors has been confirmed, the instance is saved in the database by means of *self.instance.save()*. Subsequently, the *after_create()* method is executed, which allows additional modifications to be made to related instances, if necessary.

Use of cached_property and difference with property

To optimise access to related data, cached_property is used instead of property. This strategy allows caching the property result once it has been calculated, avoiding redundant operations. In the context of this functionality, the properties obtained by the Worker and Organisation objects from the authenticated user take advantage of this feature because their value does not vary throughout the code. Therefore, the initial calculation of the value will be sufficient. These properties extract the information from the middleware, which manages the authentication token.

Customisation in child views

Child views override various methods to adapt to the particularities of each object:

- is_allowed: Check the conditions of authorisation.

- transform_data_with_related_objects: Gets the related instances via the ids passed in the request.
- get_empty_instance: Provides an empty instance of the corresponding model.
- get_field_errors and get_form_errors: They handle form and model specific errors.
- after_create: Performs additional operations after creation. These operations are optional and depend on the model being created.

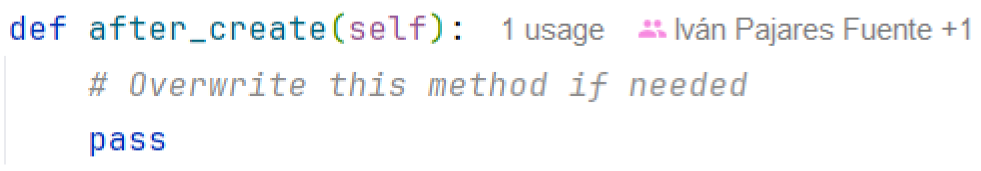
Additionally, each child view defines an array keys, which lists the object's fields in string format.

Parent view methods: Differences between pass and raise NotImplementedError

In the CreateAPIView parent view implementation, some methods have an initial implementation that uses *pass*, while others throw an exception with *raise NotImplementedError*. Both approaches have specific purposes depending on the logic expected in the child views.

Methods with pass

In methods where pass is used, no specific logic is defined in the parent view. This is mainly done when the method is not required to have an implementation in the child view. That is, the method is optional and will not interrupt the flow of the program if it is not overwritten.
Example:



```
def after_create(self):  1 usage  Iván Pajares Fuente +1
    # Overwrite this method if needed
    pass
```

Figure 33 After create method

Functionality of the use of pass

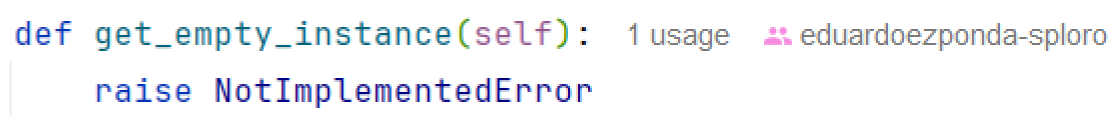
1. Flexibility: Allows child views to implement logic only if necessary.
2. Compatibility: If it is not overwritten, no error will be generated; no further action will be taken.
3. Code reduction: It prevents child views from having to implement empty methods unnecessarily when there is no particular logic to add.

Use cases

Methods such as *after_create*, which perform additional actions after the creation of an object (such as updating related instances), but are not always required.

Methods with raise NotImplementedError

In contrast, methods that include *raise NotImplementedError* are those that must be mandatorily implemented in child views. If a child view does not override one of these methods, the program will throw an error, alerting the developer that a crucial implementation is missing.
Example in CreateAPIView:



```
def get_empty_instance(self):   1 usage   eduardoezponda-sploro
    raise NotImplementedError
```

Figure 34 Raise NotImplementedError

Functionality of the use of raise NotImplementedError

1. Mandatory: Ensures that the child views implement the method with the specific logic for the object they manage.

2. Consistency: Avoid silent errors or unexpected behaviour if a critical method is not defined in the child view.

3. Clarity of design: It serves as an implicit documentation that informs the developer that this method is essential for the functionality.

Use cases

- Methods such as *get_empty_instance*, which are essential to the creation logic, as they provide the base instance of the object to be stored.

- Methods directly affecting validation (*is_allowed*, *transform_data_with_related_objects*).

Validation with Cleaners

In the system design, data validation plays a crucial role in ensuring the consistency and accuracy of the information stored in the database. This section describes how validation is carried out by the Cleaners and their interaction with the child views. Cleaners are responsible for centralising the validation logic, separating it from the main flow of the application and promoting the principle of Single Responsibility.

Cleaners are specialised classes, named after the *CleanObjectPrivate* pattern, where Object is replaced by the name of the model being validated (e.g. *CleanLocationPrivate*). These classes encapsulate the data validation logic specific to each object type. They are accessed through methods defined in the child views, such as *get_field_errors* and *get_form_errors*.

Validation methods in child views

get_field_errors:

This method is used to collect errors related to specific fields of the object. These errors are stored in the dictionary field__errors, where the key is the field name and the value is a message describing the error.

get_form_errors:

This less common method is used to collect general errors not associated with specific fields. These errors are stored in the errors structure, which usually handles problems related to general rules or inconsistencies between several fields.

Operation of validation

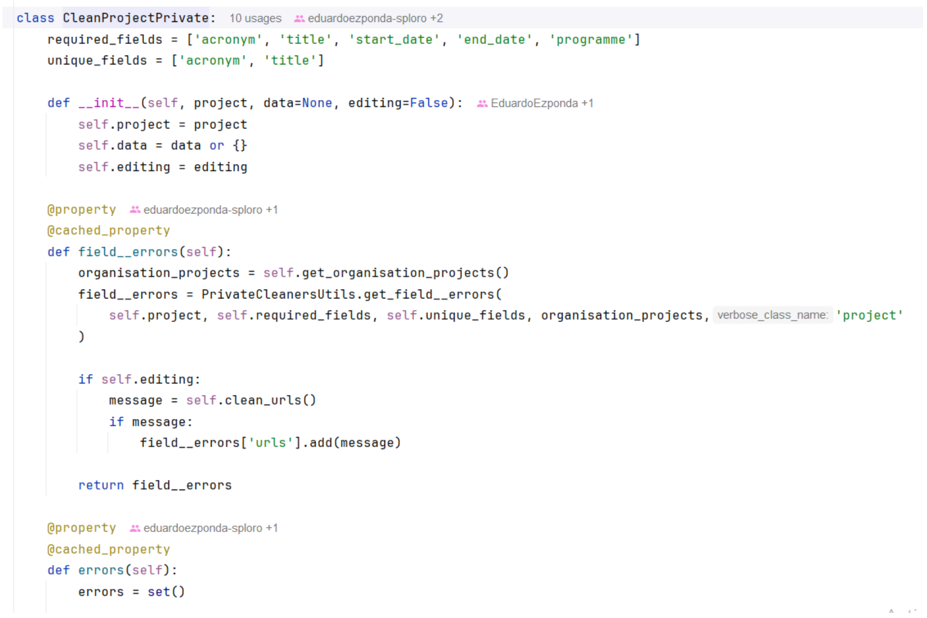
1. Instance of object with request data:
   Before calling the cleaners, an instance of the corresponding model is created and assigned the data sent in the request (usually through the attributes of the object).

2. Parameter editing:
   An *editing* boolean is passed to the cleaner, indicating whether the object is being created (False) or edited (True). This is necessary because validation may vary depending on the context:

   - Creation: May require certain fields to be mandatory or check for uniqueness restrictions.

   - Edition: Allows some fields to be optional or for the current values of the object not to violate business rules.

3. **Implementation of validation methods:**
   The methods in the cleaners validate individual fields and general rules. Any errors detected are added to the field__errors dictionary or to the errors list, as appropriate.

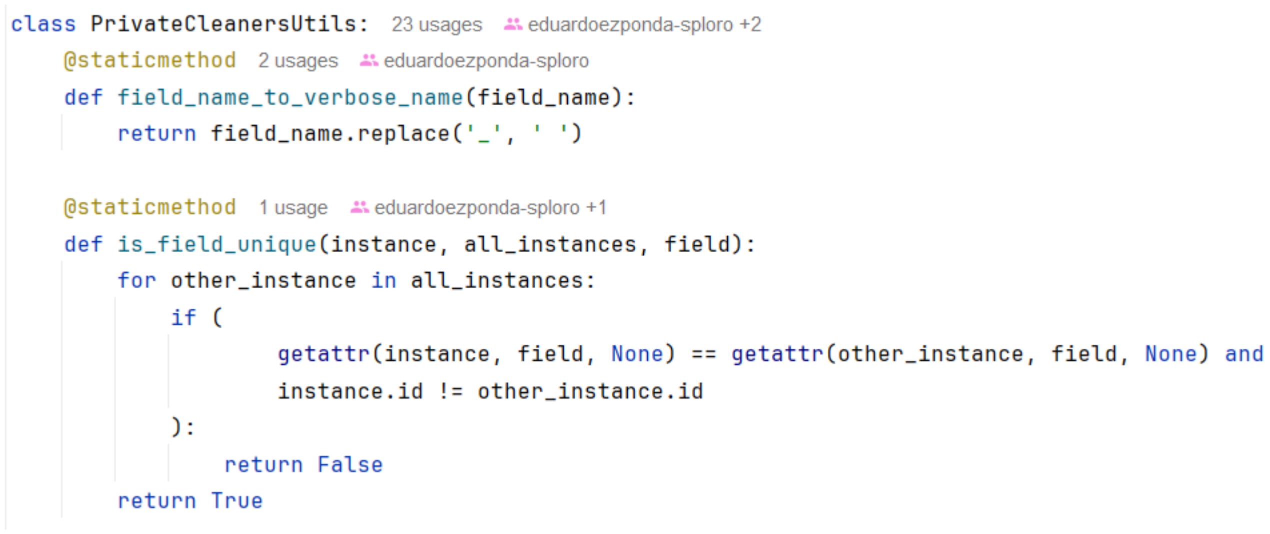


```
class CleanProjectPrivate:  10 usages  eduardoezponda-sploro +2
    required_fields = ['acronym', 'title', 'start_date', 'end_date', 'programme']
    unique_fields = ['acronym', 'title']

    def __init__(self, project, data=None, editing=False):  EduardoEzponda +1
        self.project = project
        self.data = data or {}
        self.editing = editing

    @property  eduardoezponda-sploro +1
    @cached_property
    def field__errors(self):
        organisation_projects = self.get_organisation_projects()
        field__errors = PrivateCleanersUtils.get_field__errors(
            self.project, self.required_fields, self.unique_fields, organisation_projects, verbose_class_name: 'project'
        )

        if self.editing:
            message = self.clean_urls()
            if message:
                field__errors['urls'].add(message)

        return field__errors

    @property  eduardoezponda-sploro +1
    @cached_property
    def errors(self):
        errors = set()
```

Figure 35 CleanProjectPrivate

Class PrivateCleanersUtils

To avoid redundancy in common validation rules, a base class called PrivateCleanersUtils is used. This class implements methods that are reusable between different cleaners. A common validation example is to check that a start date (*start_date*) is before an end date (*end_date*). These centralized utilities not only improve code reusability, but also ensure consistency in validation rules.



```
class PrivateCleanersUtils:  23 usages  eduardoezponda-sploro +2
    @staticmethod  2 usages  eduardoezponda-sploro
    def field_name_to_verbose_name(field_name):
        return field_name.replace('_', ' ')

    @staticmethod  1 usage  eduardoezponda-sploro +1
    def is_field_unique(instance, all_instances, field):
        for other_instance in all_instances:
            if (
                    getattr(instance, field, None) == getattr(other_instance, field, None) and
                    instance.id != other_instance.id
            ):
                return False
        return True
```

Figure 36 PrivateCleanersUtils

Use of constants for error messages

To maintain uniformity in error messages, predefined constants are used. This facilitates the maintenance and localisation of the system, ensuring that all errors follow a consistent format.

Example of field__errors and errors**:**

1. Required field:
   - Message: *"Field is required"*
   - Context: A required field is not present in the submitted data.
   - Structure:
     field__errors['field_name'] = "Field is required"
2. Unique field:
   - Message: *"Another object has the same field"*
   - Context: When the value of a field must be unique and there is already an object in the database with the same value.
   - Structure:
     field__errors['field_name'] = "Another object has the same field"

Differences between field__errors and errors

1. field__errors:
   - Purpose: Contains errors specific to individual fields.
   - Structure: It is a dictionary where the key is the field name and the value is the error message.
   - Example:

```
field__errors = {
    "start_date": "Start date is required",
    "name ": "Another organisation has the same name"
}
```

2. errors:
   - Purpose: Stores general errors related to business rules or validations between various fields.
   - Structure: It is a list of error messages, or a single error message.
   - Example:

```
errors = [
    "Start date must be before end date."
]
```

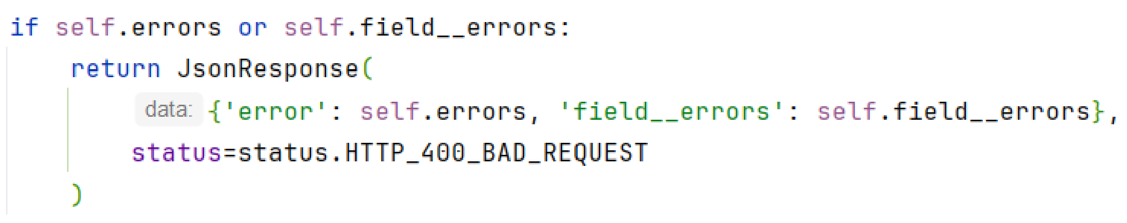
Example of validation flow

1. Instantiation of Cleaners: In a child view, the cleaner is called with the object instance and the boolean editing:

   cleaner = CleanLocationPrivate(instance=self.instance, editing=False)

2. Calling validation methods:
   - *get_field_errors* collects field-specific errors.
   - *get_form_errors* collects general errors.

3. Verification of results: If errors are detected, the flow is stopped and a JSON response is returned with the errors found:



```
if self.errors or self.field__errors:
    return JsonResponse(
        data: {'error': self.errors, 'field__errors': self.field__errors},
        status=status.HTTP_400_BAD_REQUEST
    )
```

Figure 37 Error verification

Advantages of using Cleaners

1. Separation of responsibilities:
   They keep the validation logic separate from the views and the model, improving modularity and maintainability.

2. Re-use:
   Common methods such as *PrivateCleanersUtils* reduce redundancies and ensure consistency.

3. Scalability:
   The structure allows adding new validation rules without affecting the main flow of views.

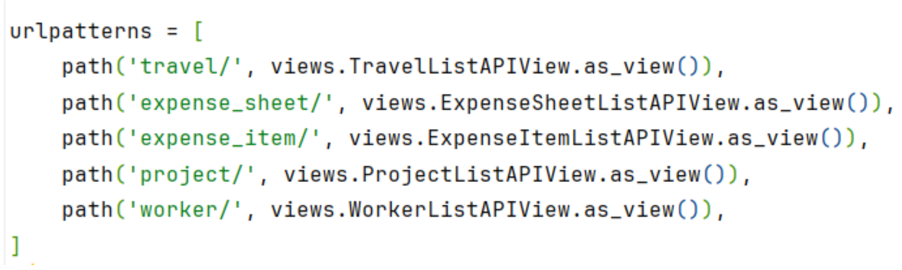
4. Readability:
   Centralising validations in cleaners makes it easier to understand and debug the system..

5. Adaptability:
   The editing parameter allows reusing the same logic for validations in both creation and editing.

5.7.2 List of objects (List)

Definition of URLs for the listing

The paths for the object listing actions are defined in a file called list.py, located inside the urls_files directory. This file contains the paths under the general structure */api/private/list/object/*, where the final part of the URL (‘object’) determines the redirection to the corresponding view.
For example:



```
urlpatterns = [
    path('travel/', views.TravelListAPIView.as_view()),
    path('expense_sheet/', views.ExpenseSheetListAPIView.as_view()),
    path('expense_item/', views.ExpenseItemListAPIView.as_view()),
    path('project/', views.ProjectListAPIView.as_view()),
    path('worker/', views.WorkerListAPIView.as_view()),
]
```

Figure 38 Url patterns for listing

In this way, each type of object has a specific path associated with its respective view.

Architecture of views

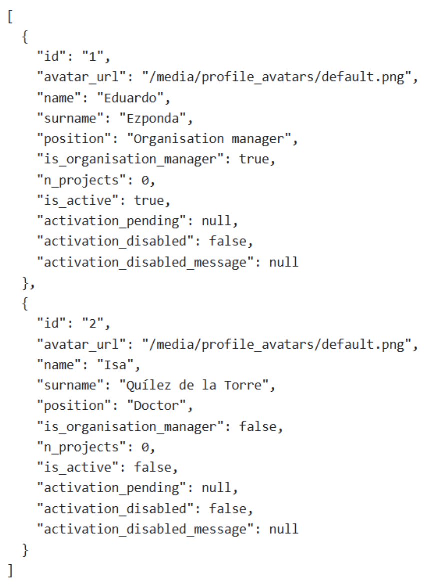
In the views based on ListAPIView classes that we have implemented in the project, the responses returned to the client vary according to the result of the user's permissions. These responses are handled through the use of JsonResponse, which allows structured data to be returned in JSON format along with the corresponding HTTP code. The main response return possibilities are described below:

1. Error response for lack of permissions (403 Forbidden)

If the user does not have the necessary permissions to list the objects, a response is returned with the error message *'Not allowed'* and an HTTP status code 403. This case is evaluated by the *is_allowed* method, which checks if the necessary authorisation conditions are satisfied.

2. Successful response (200 OK)

When the validation is successful, the corresponding objects are queried and the listing is returned together with a successful response with status code 200. The JSON contains the data of the serialised queryset.
Example of response:



```
[
  {
    "id": "1",
    "avatar_url": "/media/profile_avatars/default.png",
    "name": "Eduardo",
    "surname": "Ezponda",
    "position": "Organisation manager",
    "is_organisation_manager": true,
    "n_projects": 0,
    "is_active": true,
    "activation_pending": null,
    "activation_disabled": false,
    "activation_disabled_message": null
  },
  {
    "id": "2",
    "avatar_url": "/media/profile_avatars/default.png",
    "name": "Isa",
    "surname": "Quílez de la Torre",
    "position": "Doctor",
    "is_organisation_manager": false,
    "n_projects": 0,
    "is_active": false,
    "activation_pending": null,
    "activation_disabled": false,
    "activation_disabled_message": null
  }
]
```

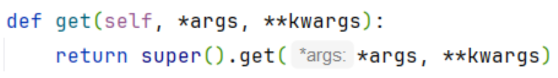
Figure 39 Listing employees

Details of the base class ListAPIView

The ListAPIView base class establishes the general flow of the listing action, providing methods that can be customized by the child views. The most relevant aspects are described below:

Method of the request

In the project, the HTTP method used for resource creation is GET, in line with REST architecture best practices. This method is specifically used to request data from the system. The behaviour of the GET method in the project's views is based on the use of the Django Rest Framework (DRF) ListAPIView base class. This method is overridden in the parent ListAPIView view of our project to customize its functionality:



```
def get(self, *args, **kwargs):
    return super().get( *args: *args, **kwargs)
```

Figure 40 Get method for listing

Flow of the GET method

The basic flow when making a GET request to list objects is as follows:

1. Receipt of the request: The request arrives at the server and is routed to the *list.py* file, where the URL associated with the corresponding child view is defined.

2. Validation of permissions: The *dispatch()* function is called automatically, checking if the user has the necessary permissions for the object listing by means of the *is_allowed* method.

3. Call to super().get(): The *dispatch()* method calls the implementation of the *get()* method of the DRF's ListAPIView base class.

4. Obtaining the queryset: The *get_queryset()* method in the child view is overridden to get the dataset to list.

5. Serialisation of data: The obtained queryset is processed using a previously defined serialiser.

6. Rendering and response: The JSONRenderer processes the serialised data to generate an HTTP response in JSON format. The response is returned with a 200 OK code if everything went correctly, or an error code otherwise.

As with the creation, permissions validation is performed through the *is_allowed* method, executed before processing the request.

Customisation in child views

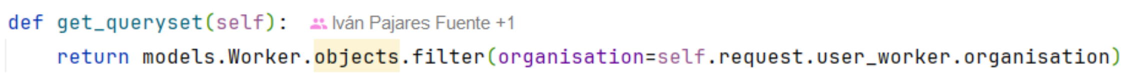
Child views override various methods to adapt to the particularities of each object:

- is_allowed: This method is used to check if the user making the request has permissions to access the corresponding object list. In its implementation:
  - Integration with the check_permissions file:
    - Each model of the timesheets application has a corresponding class in the *check_permissions* file.
    - Within these classes, there are specific functions to validate different types of permissions. In the case of the list, the *can_see_list_page* method is called, which evaluates whether the user has permission to view the list from the User and Worker objects contained in the request.
    - The format of the call is:

      return check_permissions.Objeto.can_see_list_page(self.request)
  - Result of the method:
    - If the *check_permissions* function returns True, access to the listing is allowed.
    - If it is False, a 403 Forbidden error is returned from the *dispatch()* method.

- get_queryset: The *get_queryset()* method is essential to define the data to be returned in the GET request response.
    - Overwriting in the child view:
        - This method is overwritten in each child view, as the data set to be returned depends on the model and the associated specific criteria.

          The method is automatically called by the *get()* of the DRF base class when the request is processed.
    - Execution:
        - Provides the logic for constructing and filtering the data set to be returned. For example, in the case of the child view WorkerListAPIView, the method limits the results to workers belonging to the same organisation as the user making the request:



```
def get_queryset(self):
    return models.Worker.objects.filter(organisation=self.request.user_worker.organisation)
```

Figure 41 Get queryset method

-
    - Relationship with the serialiser:
        - The obtained queryset is passed to the serializer (*serializer_class*) to be transformed into JSON format and returned as a response.

Renderer and serialiser

Renderer

In the base class of the view (ListAPIView), the renderer is defined through the renderer_classes attribute:

- Function of the renderer:
  The renderer converts the view response (a list of data) into the desired format, in this case JSON. This ensures that the response is interpretable by the frontend.

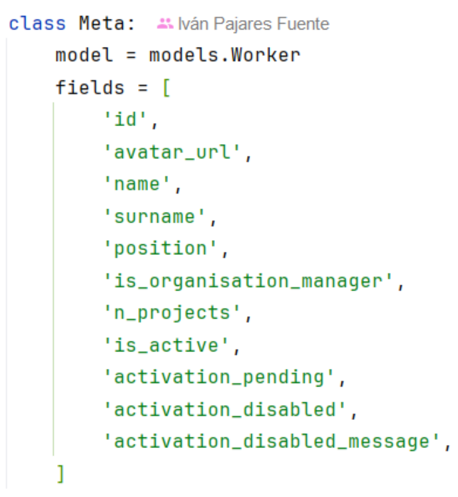
Serialiser

In each child view, a *serializer_class* attribute is defined to specify the serializer used:

- Serialiser function:
  The serialiser takes objects from the Django model (QuerySet) and transforms them into a structured and serialisable format such as JSON.

- Example: *WorkerListSerializer*
  This serialiser is designed to handle the Worker model. It provides custom logic for each field to be included in the response.

- Definition of fields:
  The fields included in the response are defined in the internal *Meta* class:

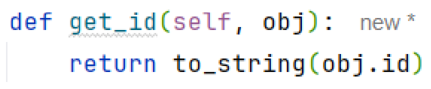


```
class Meta:  Iván Pajares Fuente
    model = models.Worker
    fields = [
        'id',
        'avatar_url',
        'name',
        'surname',
        'position',
        'is_organisation_manager',
        'n_projects',
        'is_active',
        'activation_pending',
        'activation_disabled',
        'activation_disabled_message',
    ]
```

Figure 42 Class meta of serializer

This ensures that only these fields are included in the JSON response, excluding any other attributes of the Worker model.

Use of SerializerMethodField:
Many of the fields in the serialiser use *SerializerMethodField*, which allows you to define a custom function to get the value of each field.
For example:



```
def get_id(self, obj):  new *
    return to_string(obj.id)
```

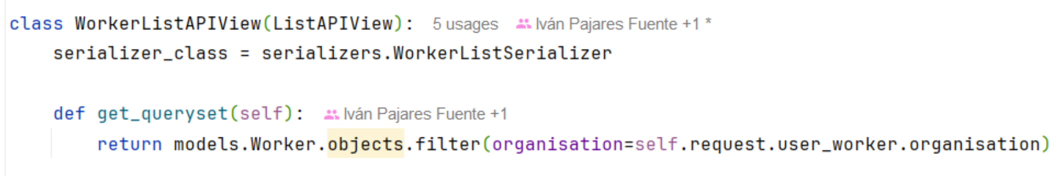
Figure 43 Get id method

Here, *get_id* returns the ID of the object converted to a string format, using an auxiliary function called to_string.

Interaction between renderer and serialiser

1. The get_queryset method provides the data:
   - The child view defines the dataset to be queried using the *get_queryset()* method. The child view defines the dataset to be queried using the *get_queryset()* method.

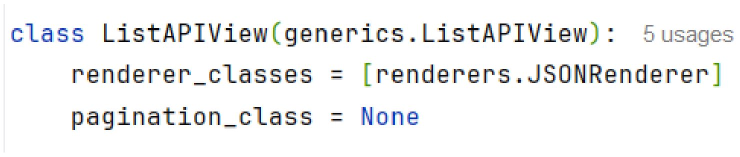
2. The serialiser processes the data:
   - The queryset is passed to the serialiser specified in *serializer_class*, which transforms this data into a serialised structure.



```
class WorkerListAPIView(ListAPIView):  5 usages  Iván Pajares Fuente +1 *
    serializer_class = serializers.WorkerListSerializer

    def get_queryset(self):  Iván Pajares Fuente +1
        return models.Worker.objects.filter(organisation=self.request.user_worker.organisation)
```

Figure 44 WorkerListAPIView serializer

3. The renderer formats the response:
   - Finally, the renderer converts the serialised data into a JSON response that is returned to the client.



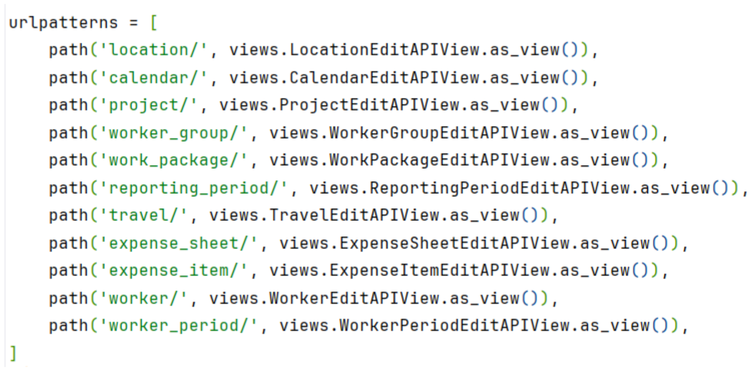
```
class ListAPIView(generics.ListAPIView):  5 usages
    renderer_classes = [renderers.JSONRenderer]
    pagination_class = None
```

Figure 45 ListAPIView renderer

5.7.3 Editing objects (Edit)

Another of the modifications made to the project is the implementation of the object editing functionality. In this section, the design and structure of this functionality is detailed, highlighting the advantages of the implemented architecture and explaining the details of the code used.

The paths for the object creation actions are defined in a file called *edit.py*, located inside the *urls_files* directory. This file contains the paths under the general structure /api/private/edit/object/, where the final part of the URL (‘object’) determines the redirection to the corresponding view.
For example:



```
urlpatterns = [
    path('location/', views.LocationEditAPIView.as_view()),
    path('calendar/', views.CalendarEditAPIView.as_view()),
    path('project/', views.ProjectEditAPIView.as_view()),
    path('worker_group/', views.WorkerGroupEditAPIView.as_view()),
    path('work_package/', views.WorkPackageEditAPIView.as_view()),
    path('reporting_period/', views.ReportingPeriodEditAPIView.as_view()),
    path('travel/', views.TravelEditAPIView.as_view()),
    path('expense_sheet/', views.ExpenseSheetEditAPIView.as_view()),
    path('expense_item/', views.ExpenseItemEditAPIView.as_view()),
    path('worker/', views.WorkerEditAPIView.as_view()),
    path('worker_period/', views.WorkerPeriodEditAPIView.as_view()),
]
```

Figure 46 Url patterns for editing

In this way, each type of object has a specific path associated with its respective view.

Architecture of views

In the views based on EditAPIView classes that we have implemented in the project, the responses returned to the client vary according to the result of the validations, the user's permissions, and the success or failure of the instance edit. These responses are handled through the use of JsonResponse, which allows structured data to be returned in JSON format along with the corresponding HTTP code. The main response return possibilities are described below:

1. Error response due to lack of permissions (403 Forbidden)

If the user does not have the necessary permissions to edit the object, a response is returned with the error message *'Not allowed'* and an HTTP status code 403. This case is evaluated by the *is_allowed* method, which checks if the necessary authorisation conditions are satisfied.

2. Validation Error Response (400 Bad Request)

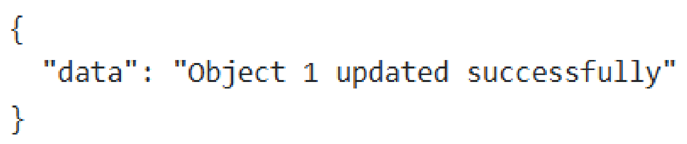
In case errors are detected during the validation process, either at a general level or in specific fields, a response with a status code 400 is returned.

- General errors: They are stored in the errors attribute and are included in the 'error' key of the JSON.

- Field errors: They are stored in field__errors, a dictionary where each key represents a specific field and the value corresponds to the associated errors. This format allows the client to identify precisely which data is invalid.

3. Successful response (200 OK)

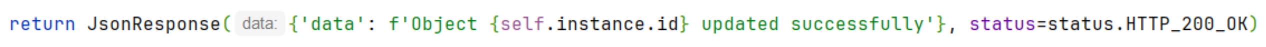
When the validation is successful and the object instance is edited without problems, the instance is saved in the database using *self.instance.save()* and a success response is returned with status code 200. The JSON contains a message with the key data summarising the executed action together with the instance id.
Example of a response:



```
{
  "data": "Object 1 updated successfully"
}
```

Figure 47 Response of a successful edition

The corresponding code fragment is:



```
return JsonResponse( data: {'data': f'Object {self.instance.id} updated successfully'}, status=status.HTTP_200_OK)
```

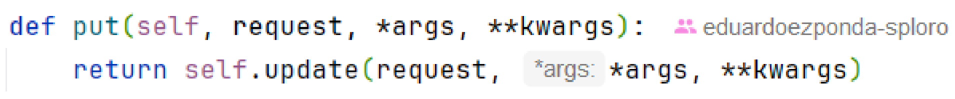
Figure 48 Sending instance in edition

Details of the base class EditAPIView

The EditAPIView base class establishes the general flow of editing actions, providing methods that can be customized by child views. The most relevant aspects are described below:

Method of the request

In the project, the HTTP method used for resource creation is PUT, in line with REST architecture best practices. This method is specifically used to send data to the server and request the edition of a new resource in the database. The behaviour of the PUT method in the project's views is based on the use of the GenericAPIView base class of the Django Rest Framework (DRF). This method is overridden in our project's EditAPIView parent view to customise its functionality:



```
def put(self, request, *args, **kwargs):  eduardoezponda-sploro
    return self.update(request, *args: *args, **kwargs)
```

Figure 49 Put method

Relationship to DRF's GenericAPIView

The parent class that defines this behaviour in our project is an extension of the GenericAPIView class provided by DRF. The latter already includes a default implementation of the PUT method to handle object creation.

- GenericAPIView in DRF: It is a generic class that combines the logic for resource creation with automatic handling of serialisation, validation and HTTP response. Overriding its PUT method in our project allows us to customize how data and additional validations are handled before the instance is saved to the database.

PUT method flow

The basic flow when making a PUT request is as follows:

1. Reception of data: The data is sent in the body of the request and captured in request.data.

2. Validation of permissions: The is_allowed method checks whether the user has the necessary permissions to perform the edit.

3. Object Instantiation: The get_instance method is used to obtain the instance of the object to be edited, which has been previously created.

4. Data validation: Data are validated by specific methods that check for general errors and errors in individual fields.

5. Save instance: If there are no errors, instance.save() is called to save the changes to the object to the database.

6. Return of reply: Depending on the result of the validations, a response is returned with the corresponding HTTP code (200, 400, 403).

Validation and permissions

As with the creation, permissions validation is performed through the is_allowed method, executed before processing the request.

Error structure

The class validates errors through the field__errors and errors structures discussed previously. Both structures are filled through validations performed in auxiliary classes called Cleaners.

Saving and post-processing process

Once the data has been validated and the absence of errors has been confirmed, the instance is saved in the database by means of *self.instance.save()*. Subsequently, the *after_edit()* method is executed, which allows additional modifications to be made to related instances, if necessary.

Customisation in child views

Child views override various methods to adapt to the particularities of each object:

- is_allowed: Check the conditions for authorisation.
- transform_data_with_related_objects: Gets the related instances via the ids passed in the request.
- get_instance: Gets the instance of the object to edit from the id.
- get_field_errors and get_form_errors: They handle form and model specific errors.
- after_edit: Performs additional operations after editing.

Additionally, each child view defines an array keys, which lists the object's fields in string format.

Validation with Cleaners

As with creation, data validation is carried out in a separate class to promote the principle of Single Responsibility Access. Cleaners are used for both creation and editing. Each object has its own private Cleaner class, and they are accessed through the get_field_errors and get_form_errors methods of the child views.

Validation methods in child views

The functionality of the *get_field_errors* and *get_form_errors* methods has been described previously.

Operation of the validation

1. Instance of the object with request data:
Before calling the cleaners, the instance of the corresponding model is obtained from the id, and the data sent in the request is assigned to it (usually through the object attributes).

2. Parameter editing:
The boolean parameter *editing* shall have the value True to refer to the edit action.

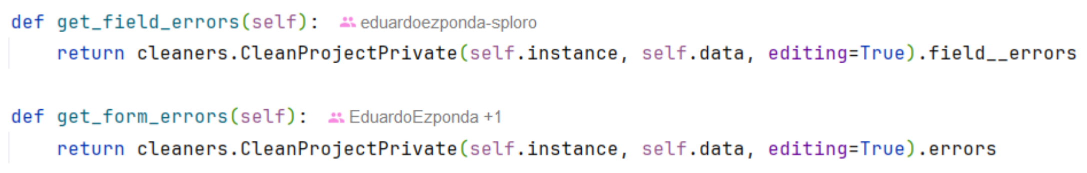
3. Implementation of validation methods:
The methods in the cleaners validate individual fields and general rules. Any errors detected are added to the field__errors dictionary or to the errors list, as appropriate.

Example of validation flow

1. Cleaner instantiation: In a child view, the cleaner is called with the object instance and the boolean editing.

   CleanLocationPrivate(instance=self.instance, editing=True)

2. Validation method calls:
   - *get_field_errors* collects field-specific errors.
   - *get_form_errors* collects general errors.



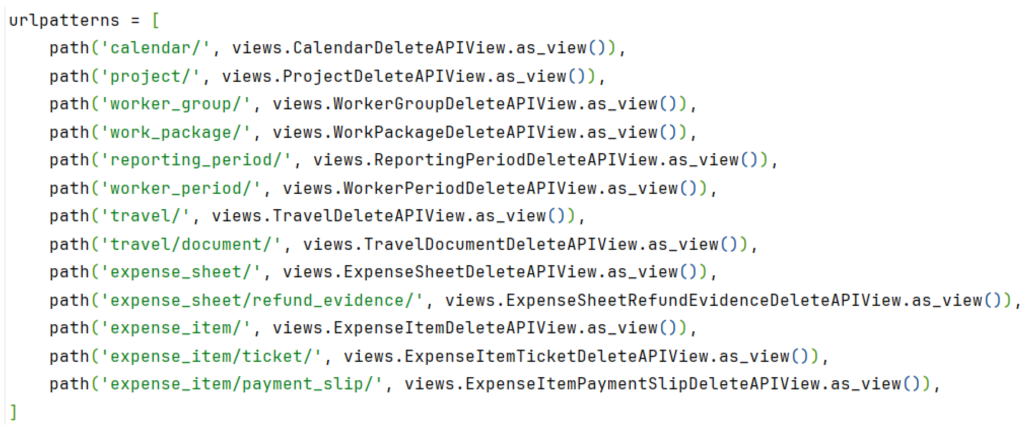
```
def get_field_errors(self):  eduardoezponda-sploro
    return cleaners.CleanProjectPrivate(self.instance, self.data, editing=True).field__errors

def get_form_errors(self):  EduardoEzponda +1
    return cleaners.CleanProjectPrivate(self.instance, self.data, editing=True).errors
```

Figure 50 Get field and form errors

3. Verification of results: If errors are detected, the flow is stopped and a JSON response is returned with the errors found.

5.7.4 Deleting objects (Delete)

The subsection dedicated to the DELETE method deals with the deletion of objects through the associated views. In this case, requests are handled through a file called *delete.py*, which defines the corresponding URL paths. A typical example of path configuration is:



```
urlpatterns = [
    path('calendar/', views.CalendarDeleteAPIView.as_view()),
    path('project/', views.ProjectDeleteAPIView.as_view()),
    path('worker_group/', views.WorkerGroupDeleteAPIView.as_view()),
    path('work_package/', views.WorkPackageDeleteAPIView.as_view()),
    path('reporting_period/', views.ReportingPeriodDeleteAPIView.as_view()),
    path('worker_period/', views.WorkerPeriodDeleteAPIView.as_view()),
    path('travel/', views.TravelDeleteAPIView.as_view()),
    path('travel/document/', views.TravelDocumentDeleteAPIView.as_view()),
    path('expense_sheet/', views.ExpenseSheetDeleteAPIView.as_view()),
    path('expense_sheet/refund_evidence/', views.ExpenseSheetRefundEvidenceDeleteAPIView.as_view()),
    path('expense_item/', views.ExpenseItemDeleteAPIView.as_view()),
    path('expense_item/ticket/', views.ExpenseItemTicketDeleteAPIView.as_view()),
    path('expense_item/payment_slip/', views.ExpenseItemPaymentSlipDeleteAPIView.as_view()),
]
```

Figure 51 Url patterns for deleting

Architecture of the views

The design follows a hierarchical structure with a generic base class called DeleteAPIView. This class provides the core logic for handling DELETE requests. Specific views, such as CalendarDeleteAPIView, inherit from this base class and define specific behaviours for each model.

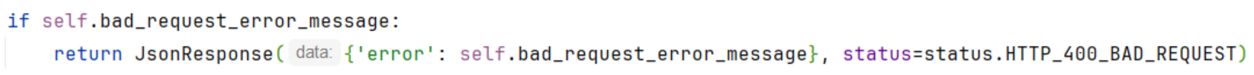
- Data input: DELETE requests send one or more IDs of the objects to be deleted.

- Elimination logic: The DeleteAPIView class has a method called *get_instance()*, which is used from attributes of the child views to search for objects corresponding to the provided IDs.

Error handling

1. No object found: If the *get_instance()* method cannot locate the objects corresponding to the provided IDs, an error message with HTTP status code 404 (*Not Found*) is returned.

2. Generic bad request error: In case a generic error is detected in the request (e.g. missing or badly formatted data), the response includes an error message and an HTTP status code 400 (Bad Request):



```
if self.bad_request_error_message:
    return JsonResponse( data: {'error': self.bad_request_error_message}, status=status.HTTP_400_BAD_REQUEST)
```

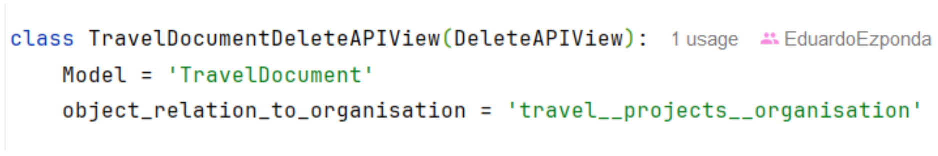
Figure 52 Bad Request Validation

3. Permissions error: If the user does not have the necessary permissions to perform the deletion, an error message is returned with an HTTP status code 403 (*Forbidden*).

The *get_instance()* method in the DeleteAPIView base view is a key part of getting the objects to be safely deleted. In this case, the logic includes a filter based on the associated organisation, which ensures that only objects belonging to the organisation of the user making the request are accessed. This mechanism is essential to prevent unauthorised access to resources from other organisations.

Implementation of the get_instance method

The method uses two attributes defined in the child views:

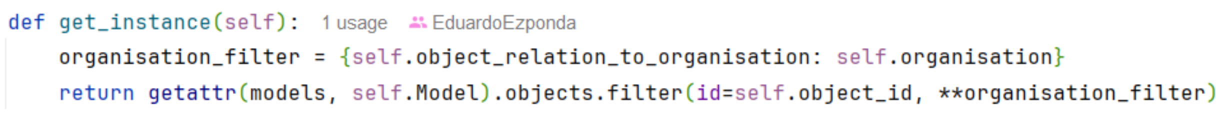


```
class TravelDocumentDeleteAPIView(DeleteAPIView):  1 usage  EduardoEzponda
    Model = 'TravelDocument'
    object_relation_to_organisation = 'travel__projects__organisation'
```

Figure 53 TravelDocumentAPIView

1. model: Specifies the model to which the object to be deleted belongs.

2. object_relation_to_organisation: Defines the relationship between the object model and the organisation model. This attribute allows filtering the objects by the corresponding organisation.



```
def get_instance(self):  1 usage  EduardoEzponda
    organisation_filter = {self.object_relation_to_organisation: self.organisation}
    return getattr(models, self.Model).objects.filter(id=self.object_id, **organisation_filter)
```

Figure 54 Get instance in deleting

Security: Filter objects using the object_relation_to_organisation attribute, which ensures that objects belong to the same organisation as the user of the request. This filter is crucial to avoid deleting objects from other organisations.

Use of ** characters

The ** operator in ***{self.object_relation_to_organisation: organisation}* is used to unpack a dictionary into keyword arguments. In this case, *self.object_relation_to_organisation* is a string representing the name of the organisation-related field (e.g. organisation), and organisation is the instance of the Organisation model.

The expression ***{self.object_relation_to_organisation: organisation}* generates an equivalent dynamic argument in case the object has a direct relation to the Organisation model, as follows:

organisation=organisation

This makes the filter flexible and reusable for any child view, as the organisation-related field can vary between models.

Benefits of this implementation

1. Re-use: Child views only need to define the model and *object_relation_to_organisation* attributes, while the object retrieval logic is centralised in the base class.

2. Security: Filtering by organisation ensures that users cannot interact with objects from other organisations, while respecting access policies.

3. Flexibility: The use of the ** operator allows the method to be adapted to different models and relationships without the need to modify the main logic.

6 Implementation

6.1 Development of a web interface to the API

In addition to the tasks performed during my internship, I have developed a web interface that facilitates interaction with the project's API. This development is not part of the functionalities required by the company, but an extension of my work, designed to show in a clear and structured way the possible interactions with the API.

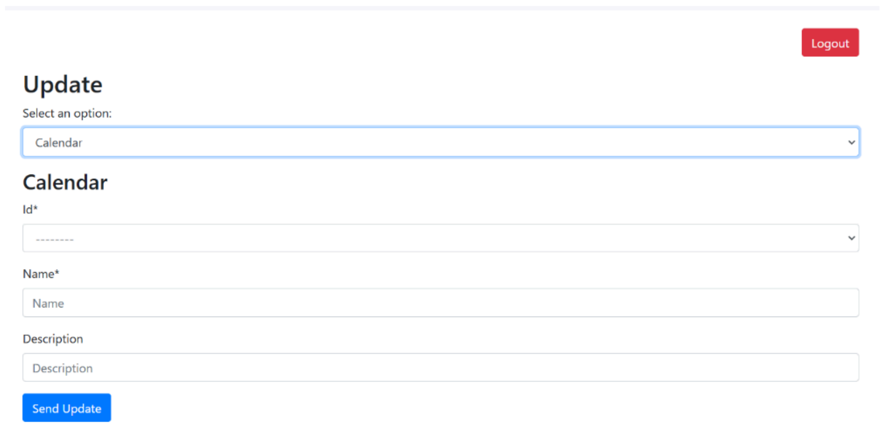
Interface characteristics

1. Endpoint management and CRUD methods
   The interface allows you to select the endpoint to which you want to make the request via a drop-down menu. Subsequently, the HTTP method to be used is selected:

   - GET: To consult data.
   - POST: To create new resources.
   - PUT: To update existing records.
   - DELETE: To delete data.

   This design organises and simplifies interaction with the API, making it easy to test different endpoints.

2. Dynamic data insertion
   The interface provides adaptive fields depending on the selected method and endpoint. This ensures that only the necessary data is displayed for each type of operation, avoiding errors and improving the user experience.


Logout

Update

Select an option:

Calendar

Calendar

Id*

Name*

Name

Description

Description

Send Update

Figure 55 Interface dynamic fields

3. Visualisation of responses
   After each request, the interface displays the response message provided by the backend. This message can be either a success or an error message:

   - Success messages: Confirm the operation performed, such as the creation of a resource or a successful query.

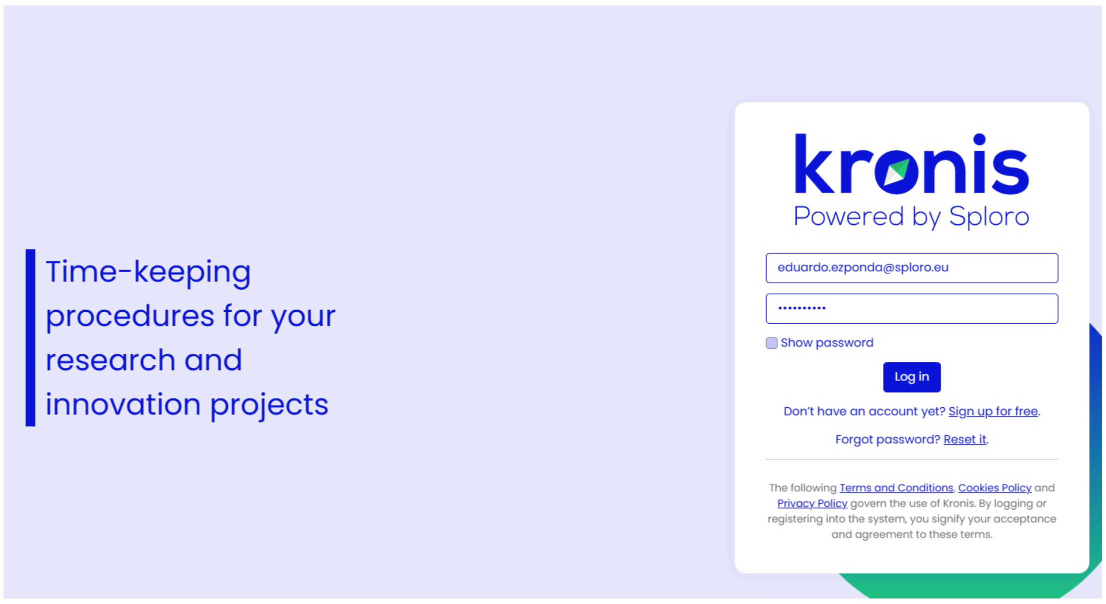
   - Error messages: They detail the type of failure, whether due to invalid data, authentication problems or server errors. They are displayed clearly and with sufficient information to identify and correct the problem.

4. Authentication and security
   The interface requires the user to log in correctly before accessing its functionalities. To this end, the system includes:

   - Login screen: Where user credentials are validated.

   - Secure redirection: After a successful login, the user is taken directly to the interface, where he/she can start testing with the API.

   In addition, all requests made from the interface are authenticated by a session token.


Time-keeping procedures for your research and innovation projects
kronis
Powered by Sploro
Show password
Log in

eduardo.ezponda@sploro.eu

••••••••••

Don't have an account yet? Sign up for free.

Forgot password? Reset it.

The following Terms and Conditions, Cookies Policy and Privacy Policy govern the use of Kronis. By logging or registering into the system, you signify your acceptance and agreement to these terms.

Figure 56 Kronis login interface

Organisation of the interface

The interface includes a navigation menu that organises the available functionalities according to CRUD actions.

- Create: To add new objects.

- Read: To consult a list of existing objects.

- Update: To modify a given object.

- Delete: To delete objects.

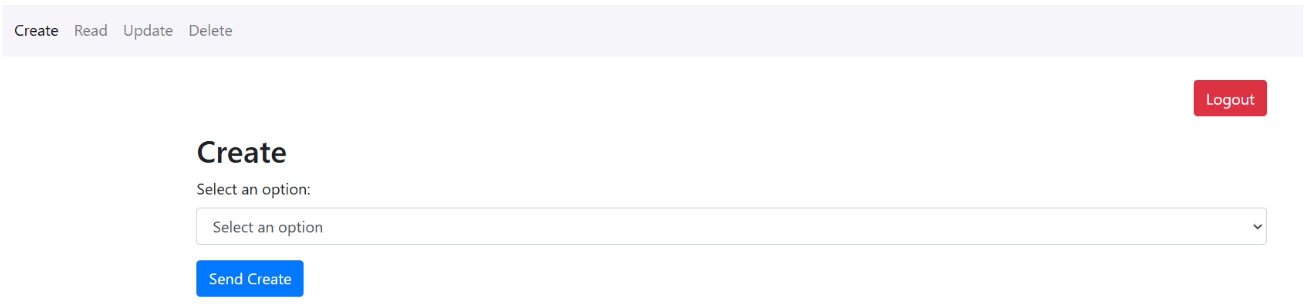


Figure 57 Organisation of the interface

Each section has a specific layout to guide the user, displaying the required fields and the available options in a clear way.

Benefits of the tool

1. Ease of use:
   Simplifies interaction with the API, especially for users with less technical experience.

2. Clarity of information:
   It provides detailed messages about the operations performed, helping to understand both successes and failures.

3. Flexibility:
   The modular structure of the interface allows it to be easily adapted to new endpoints or API methods in the future.

4. Security:
   It ensures that only authenticated users can interact with the API, protecting the system from unauthorized access.

Adaptations of the Kronis project for local implementation

In the process of setting up the Kronis project to run in my local environment, I have made several modifications to the code to ensure that it does not rely on sensitive data that was previously in the global.env file. However, the file is still present with some basic values that are necessary, such as ports and minimal settings related to development containers.

Modifications made to the code

1. Adaptation to the partial use of the global.env file

- Originally, the *global.env* file contained sensitive variables such as API keys, external service configurations and credentials.

- I have modified the code so that if any of these variables are not defined, safe defaults are used or functionality that depended on them is omitted. In this way, the project is not broken by the absence of unnecessary data for the local environment.

2. Reduction of external dependencias

- Parts of the code that depended on advanced configurations, such as third-party services or complex integrations, have been disabled or adapted. This includes email handling and advanced authentication configurations.

3. Error checking and defaults

- I have added validations in the code to handle the absence of certain environment variables. For example, instead of failing if an API key is not found, the system now throws a warning message or uses a neutral value that does not affect local execution.

Process for implementing the project locally

The project can be run on my local machine by following a series of steps that include installing dependencies, preparing the database, and running the server. Below, I detail the necessary commands:

1. Install dependencies
   To install all required libraries:

   ```
   pip install -r requirements.txt
   ```

2. Configure the global.env
   Although the amount of data in the global.env file has been reduced, some basic settings need to be maintained. These are automatically loaded when the project is started.

3. Prepare the database
   The project models require migrations to set up the database:

   - Create migrations:
     ```
     python manage.py makemigrations
     ```

   - Apply migrations:
     ```
     python manage.py migrate
     ```

4. Run the local server
   With the environment set up, I can start the server using:

   ```
   python manage.py runserver
   ```

   This gets the backend up and running at *http://127.0.0.1:8000*.

5. Other useful commands
   - Create a superuser:
     If I need access to the administration panel:
     ```
     python manage.py createsuperuser
     ```
   - Check the project status:
     To check configurations:
     ```
     python manage.py check
     ```

Conclusion

These modifications have allowed me to keep the global.env file in the project, but in a simplified form and without sensitive information. Adapting the code to handle the absence of the original data ensures that the system remains functional in my local environment, without compromising the security of Splorotech-related information.

7 Tests

7.1 Unit testing

During the development of the project, I have implemented unit tests to ensure the correct functioning of the API endpoints without the need to access the database, which I have achieved through the use of mocks. This has allowed me to avoid unnecessary dependencies, simplify the test environment and ensure that the functionalities of each endpoint are verified in an isolated and efficient way.

Unit test structure

I've created a folder called tests inside the api application, and inside it I've included a file for each view corresponding to each model of the timesheets application. These files are named following the Django convention, that is, they all start with test_. Likewise, the test functions within each file must also start with test_, allowing the Django testing framework to detect them automatically.

Each test file contains:

- Types of tests: Each file includes four classes, one for each CRUD operation (Create, Read, Update, Delete). These classes group the tests of each corresponding method, which makes them easier to organise and read.

- SetUp: A setUp method that initialises common data, such as user instances, organisation and test data. This ensures that each test starts from a clean, predefined state.

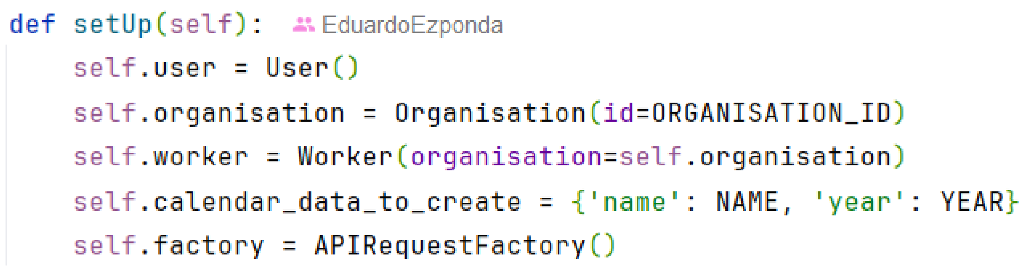
- Constants: Constants used in tests (e.g. error messages or test data) are defined at the top of each file for easy reuse. In addition, I have created a constants.py file where I store repeated messages or values that are used in multiple tests.

- Mocks: To avoid interaction with the database, I used mocks. This has allowed me to simulate class, method and data behaviours, ensuring that the tests focus exclusively on the endpoint logic.

Code example

The following is a representative example of the tests carried out:

1. Data preparation: In the setUp method, I initialise data needed for testing, such as user, organisation and test data to create a new object. I also use APIRequestFactory to generate simulated HTTP requests.



```python
def setUp(self):  EduardoEzponda
    self.user = User()
    self.organisation = Organisation(id=ORGANISATION_ID)
    self.worker = Worker(organisation=self.organisation)
    self.calendar_data_to_create = {'name': NAME, 'year': YEAR}
    self.factory = APIRequestFactory()
```

Figure 58 Set Up for unit testing

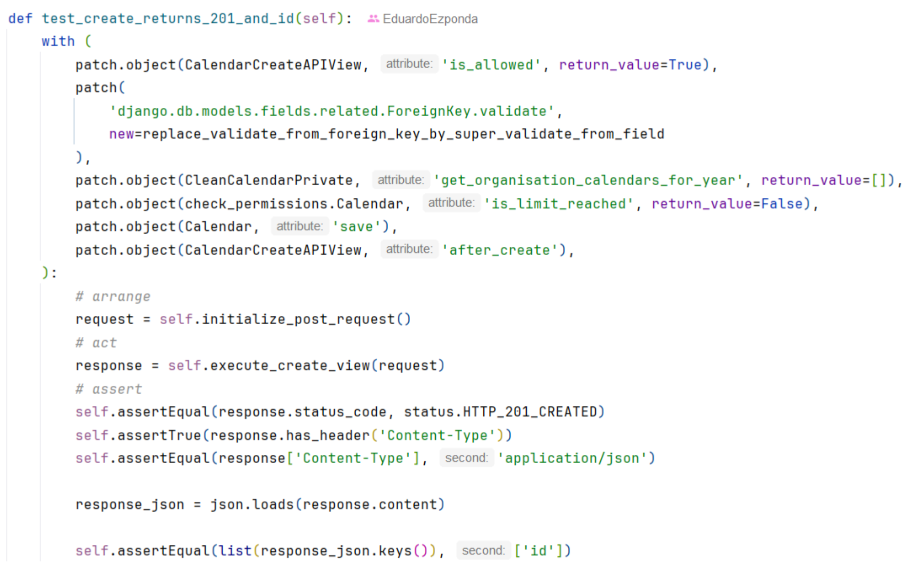
2. Mocking of functionalities: Use patch to replace real methods with simulated versions during testing. This allows you to control the behaviour of external dependencies, such as foreign key validation or access to specific functions.

3. Test execution: To run the tests, I use the command:

   python manage.py test

   This finds and executes all test files within the project (in any application that has tests).

Benefits of the testing strategy



```python
def test_create_returns_201_and_id(self):  EduardoEzponda
    with (
        patch.object(CalendarCreateAPIView,  attribute: 'is_allowed', return_value=True),
        patch(
            'django.db.models.fields.related.ForeignKey.validate',
            new=replace_validate_from_foreign_key_by_super_validate_from_field
        ),
        patch.object(CleanCalendarPrivate,  attribute: 'get_organisation_calendars_for_year', return_value=[]),
        patch.object(check_permissions.Calendar,  attribute: 'is_limit_reached', return_value=False),
        patch.object(Calendar,  attribute: 'save'),
        patch.object(CalendarCreateAPIView,  attribute: 'after_create'),
    ):
        # arrange
        request = self.initialize_post_request()
        # act
        response = self.execute_create_view(request)
        # assert
        self.assertEqual(response.status_code, status.HTTP_201_CREATED)
        self.assertTrue(response.has_header('Content-Type'))
        self.assertEqual(response['Content-Type'],  second: 'application/json')

        response_json = json.loads(response.content)

        self.assertEqual(list(response_json.keys()),  second: ['id'])
```

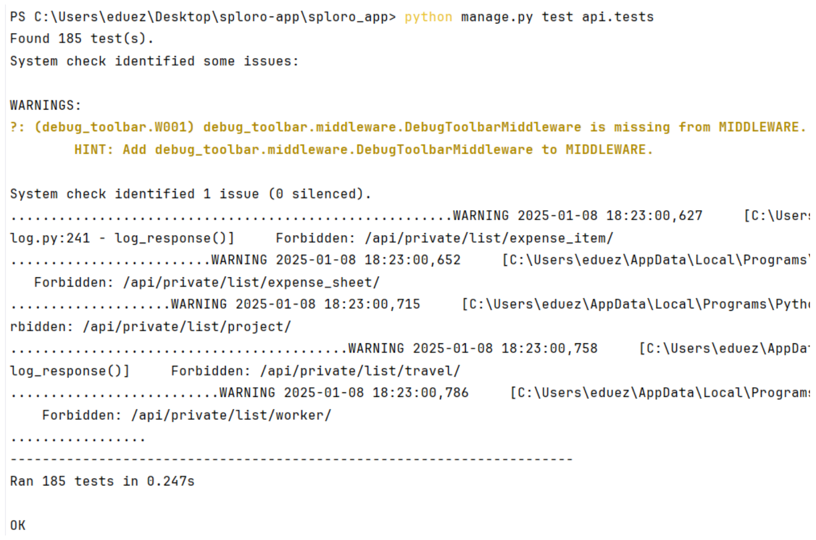
Figure 59 Test example for a calendar creation

The use of unit testing with mocks has been a key to:

- Eliminate external dependencies: The tests do not require an active database, which makes them easy to run in any environment.
- Modularity and clarity: The organisation in classes by CRUD method and the use of centralised constants improve the clarity and maintainability of the code.
- Reliability: It ensures that each functionality is tested in isolation, detecting errors before they are integrated into the system.

This approach has allowed me to ensure the quality of the code and the robustness of the developed endpoints.

In conclusion, in my case I have run all the tests of the api application to validate the correct functioning of all the code. This process has been done both during the internship and independently, making sure that each part of the application complies with the requirements and does not generate errors. In the image below, you can see the execution of the command python manage.py test api.tests, where 186 tests have been executed correctly, and the output shows that everything has worked without problems, with a final result of OK. This confirms that the code is fully validated and ready for implementation.



```
PS C:\Users\eduez\Desktop\splorо-app\sploro_app> python manage.py test api.tests
Found 185 test(s).
System check identified some issues:

WARNINGS:
?: (debug_toolbar.W001) debug_toolbar.middleware.DebugToolbarMiddleware is missing from MIDDLEWARE.
        HINT: Add debug_toolbar.middleware.DebugToolbarMiddleware to MIDDLEWARE.

System check identified 1 issue (0 silenced).
.......................................................WARNING 2025-01-08 18:23:00,627     [C:\User
log.py:241 - log_response()]     Forbidden: /api/private/list/expense_item/
.........................WARNING 2025-01-08 18:23:00,652     [C:\Users\eduez\AppData\Local\Programs
   Forbidden: /api/private/list/expense_sheet/
....................WARNING 2025-01-08 18:23:00,715     [C:\Users\eduez\AppData\Local\Programs\Pyth
rbidden: /api/private/list/project/
..........................................WARNING 2025-01-08 18:23:00,758     [C:\Users\eduez\AppDa
log_response()]     Forbidden: /api/private/list/travel/
..........................WARNING 2025-01-08 18:23:00,786     [C:\Users\eduez\AppData\Local\Program
    Forbidden: /api/private/list/worker/
.................
----------------------------------------------------------------------
Ran 185 tests in 0.247s

OK
```

Figure 60 Execution of tests

7.2 Load testing

Load testing is a critical aspect of evaluating a system’s performance under expected and peak workloads. However, in this project, conducting load tests was not feasible due to the lack of a stable infrastructure at the time of development. The company was in a transitional phase, awaiting the arrival of a new CTO with expertise in cloud technologies. As a result, the existing infrastructure was temporary and scheduled for replacement, making any performance benchmarks obsolete in the near future.

At the time of my work, the backend was still running on an infrastructure that was soon to be deprecated. Performing load tests on this setup would have provided insights that would no longer be relevant after the migration. Since the frontend migration from Django templates to Next.js was also incomplete, it was unclear how the final system architecture would impact performance. Therefore, the focus was placed on ensuring the backend’s API functionality and compatibility with the new frontend rather than optimizing performance on a soon-to-be-discarded environment.

In a real-world scenario, load testing is typically conducted when the final production environment is defined, ensuring that test results accurately reflect system behavior under real conditions. Given that the company was transitioning to a new cloud-based infrastructure, load testing would only be meaningful once the new environment was fully set up. Future iterations of this project will likely include load and stress testing to assess performance in the definitive cloud infrastructure.

By prioritizing functional validation during this stage, the project ensured a smooth backend migration while leaving performance optimizations for a more stable and relevant infrastructure.

8 Evaluation and conclusion

8.1 Evaluation

During my involvement in the development of this project, I gained a deep understanding of how to adapt and optimize a backend to cope with a frontend migration, as was the case with Kronis. This process involved not only creating new API endpoints, but also improving the efficiency of the system by integrating tools such as Silk, resulting in a more robust backend ready for future extensions.

In addition, I have reinforced my knowledge of Django, exploring its capabilities in depth and adapting my development to best practices. The use of Git to manage code versions has been essential to maintain an orderly and collaborative workflow, allowing me to make changes and tests in a controlled manner. Added to this is the use of PyCharm as a development environment, which has greatly facilitated my work thanks to its debugging tools and its integration with GitHub Copilot. This last tool has been a key support to speed up code writing and receive intelligent suggestions, which has improved both my productivity and my continuous learning.

On the other hand, the development of a complementary web interface has been one of the most enriching experiences. This interface has allowed me to show more easily the inputs and outputs of the endpoints created, as well as to make customised requests to the API. By integrating functionalities such as the selection of CRUD methods, the handling of data through drop-downs and the display of backend responses (either success messages or specific errors), I have achieved an environment that facilitates both the validation of my work and its presentation in an academic context.

This project has not only allowed me to advance my technical skills, but also to apply what I have learned in a practical environment. The combination of my efforts in adapting the backend and creating this interface reflects my commitment to offer complete and functional solutions, as well as consolidating knowledge that will be useful in future developments.

8.2 Conclusion

In conclusion, Kronis is a platform designed to optimize the management of projects and human resources in organisations, a goal that reflects Splorotech's mission to offer innovative and efficient solutions to its clients. Kronis' future objectives include expanding its functionalities, improving its integration with other platforms and strengthening its presence in the market as an essential tool for business management.

My work has contributed significantly to the development of Kronis by facilitating the transition to a more modern and efficient frontend, adapting the backend for this migration. In addition, the creation of new endpoints and the implementation of a complementary web interface have improved the accessibility and clarity of the project's functionalities. These contributions not only strengthen the technical structure of Kronis, but also lay the foundations for sustained growth and evolution in the future.

8.3 Future lines

As for future lines, Kronis does not have a limit of estimated hours of development, as its evolution will depend on the subsidy management needs required by the European Union. Splorotech will adapt to implement in Kronis all the necessary functionalities to meet these requirements. Furthermore, Splorotech's future is oriented towards the automation of all the phases involved in the management of subsidised innovation projects, consolidating itself as a benchmark in integral and efficient solutions for this type of initiative.

9 Bibliography

9.1 Techniques

[1] ”Django”, Django Software Foundation, [Online]. Disponible en:
https://www.w3schools.com/django/django_intro.php
https://www.djangoproject.com/

[2] ”NextJS”, Vercel, [Online]. Disponible en:
https://nextjs.org/docs

[3] ”Silk”, Microsoft, [Online]. Disponible en:
https://silk.us/platform-overview/

[4] ”Tailwind CSS v4.0”, Tailwind Labs, [Online]. Disponible en:
https://www.geeksforgeeks.org/introduction-to-tailwind-css/

[5] ”Bootstrap”, Twitter, [Online]. Disponible en:
https://getbootstrap.com/docs/5.0/getting-started/introduction/

[6] ”Solid Principles NextJS”, [Online]. Disponible en:
https://www.linkedin.com/pulse/solid-principles-reactjs-oleksii-bortnytskyi/?trackingId
=YurE3exMTaOV2TfzDpNVKQ%3D%3D

9.2 Tools

[7] “Excel”, Microsoft, [Online]. Disponible en: https://www.coursera.org/articles/what-is-excel

[8] “Docker”, Docker, Inc., [Online]. Disponible en: https://docs.docker.com/get-started/docker-overview/

[9] “Amazon Web Services”, Amazon, [Online]. Disponible en: https://aws.amazon.com/es/what-is-aws/

[10] “Airtable”, Formagrid, Inc., [Online]. Disponible en: https://www.airtable.com/

[11] “Pycharm 2024.3.2”, JetBrains, [Online]. Disponible en: https://www.geeksforgeeks.org/what-is-pycharm/

[12] “MobaXterm”, Mobatek, [Online]. Disponible en: https://mobaxterm.mobatek.net/

[13] “Swagger”, SMARTBEAR, [Online]. Disponible en: https://swagger.io/