INSTITUTO POLITÉCNICO DE LISBOA

Instituto Superior de Engenharia de Lisboa
Escola Superior de Tecnologia da Saúde de Lisboa

Design and development of process control and management application for pre-clinical laboratories

Ricardo Luís Carvalho Monteiro

Thesis to obtain the Master's Degree in Biomedical Engineering

Supervisors
Nuno Soares Domingues (ISEL)
Pedro Miguens Matutino (ISEL)

Examination Committee

Chairperson: Manuel José de Matos

Members of the Committee: Markus Maeurer
Nuno Soares Domingues

September 2024

INSTITUTO POLITÉCNICO DE LISBOA

Instituto Superior de Engenharia de Lisboa
Escola Superior de Tecnologia da Saúde de Lisboa

coley

Design and development of process control and management application for pre-clinical laboratories

Ricardo Luís Carvalho Monteiro

Thesis to obtain the Master's Degree in Biomedical Engineering

Supervisors
Nuno Soares Domingues (ISEL)
Pedro Miguens Matutino (ISEL)

September 2024

Acknowledgements

I thank my beloved wife Diana for the love, affection and care for those around her.

I thank my daughter Sara. You fill my heart with joy, love and the desire to practice engineering to make the world a little better.

I would like to thank my supervisors, Prof. Nuno Domingues and Prof. Pedro Miguens Matutino, for the wisdom, patience, experience and encouragement that were given over the months that were applied in preparing this thesis.

I would like to thank God, and His Son, Jesus Christ, for being my source of strength, my Redeemer, for the gift of knowledge and for giving me the opportunity to develop this work.

Resumo

As origens dos sistemas de gestão de informação laboratorial (*Laboratory Information Management Systems* - LIMS) remontam à década de 1970, quando a necessidade de gestão eficiente de dados e de manutenção de registos nos laboratórios se tornou cada vez mais evidente. Os laboratórios, que tradicionalmente se baseavam em métodos manuais para a introdução de dados, rastreio de amostras e elaboração de relatórios, estavam a enfrentar desafios na gestão do enorme volume de dados gerados. Isto levou a ineficiências, erros e atrasos nos relatórios das análises clínicas.

As primeiras versões de LIMS eram rudimentares em comparação com os sistemas atualmente utilizados. Centravam-se principalmente na automatização do controlo de amostras e no fornecimento de capacidades básicas de armazenamento de dados. Estes sistemas eram frequentemente construídos à medida das necessidades específicas de cada laboratório, o que limitava a sua escalabilidade e adaptabilidade. A década de 1980 marcou um ponto de viragem significativo, uma vez que os avanços tecnológicos na computação e no desenvolvimento de software permitiram soluções LIMS mais abrangentes. O uso de software de registo e controlo de informação genérico como folhas de cálculo ainda é prevalente em muitos laboratório de investigação biomédica e prática clínica. Com o avanço das tecnologias *web* e bases de dados, têm surgido nas últimas décadas LIMS com base nestas tecnologias que visam facilitar a gestão laboratorial.

Na década de 1990 deu-se um aumento da popularidade dos LIMS à medida que os fornecedores de software comercial reconheceram o potencial do mercado. Com a integração de interfaces gráficas de utilizador, bases de dados relacionais e capacidades de ligação em rede, os LIMS tornaram-se mais fáceis de utilizar e acessíveis. Estes sistemas começaram a oferecer funcionalidades para além do armazenamento de dados, incluindo a integração de instrumentos, a gestão do fluxo de trabalho e o controlo de qualidade.

Com o início do século XXI, os LIMS evoluíram para plataformas complexas e integradas, servindo uma vasta gama de tipos de laboratórios, incluindo laboratórios farmacêuticos, ambientais, clínicos e de investigação. Uma das principais tendências durante este período foi a mudança para soluções LIMS baseadas na Web e na nuvem, permitindo o acesso remoto, a colaboração e a partilha de dados. Esta mudança também abordou os desafios de segurança e conformidade de dados, críticos em indústrias regulamentadas.

Neste trabalho, propõe-se o desenvolvimento de um protótipo de aplicação *web* a ser aplicado na gestão e controlo do fluxo de trabalho de um laboratório dedicado à investigação e práticas clínica em imunoterapia (Mauerer lab, Fundação Champalimaud). A aplicação foi designada de *coley*, em homenagem a William B. Coley, um dos pioneiros da imunoterapia. A arquitectura da aplicação segue a constituição de uma aplicação web simples de implementar, composta por uma *frontend*, *backend* e uma base de dados. O *frontend* consiste na interface gráfica com a qual o utilizador interagirá, sendo que esta interface terá a informação disponível gerida pelo *backend* que por sua vez irá fazer pedidos à base de dados consoante a interação do utilizador com a aplicação.

O desenvolvimento da aplicação terá como base a implementação de ferramentas *open-source*. Para a interface gráfica, utilizou-se o *ReactJS*, conhecida biblioteca *frontend* de código aberto, com base em *JavaScript*, criada pelo *Facebook*. O *ReactJS* é, no campo do desenvolvimento *web*, a ferramenta de criação de interfaces mais popular. A sua abordagem oferece uma mudança de paradigma na forma como as interfaces de utilizador são construídas e mantidas. No coração do *ReactJS* estão vários conceitos fundamentais que o diferenciam das abordagens tradicionais de desenvolvimento Web. Um dos princípios-chave é a programação declarativa, em que os programadores descrevem o estado desejado da UI (*user interface*) e o *React* encarrega-se de atualizar eficientemente a UI real para corresponder a esse estado. Essa abordagem elimina a necessidade de código imperativo para manipular o *DOM* (*Document Object Model*) diretamente, levando a um código mais legível e de fácil manutenção. Para alem da abordagem da programação declarativa, a otimização virtual do *DOM*, a arquitetura baseada em componentes e o ecossistema próspero impulsionaram o *React* para a vanguarda do desenvolvimento Web moderno.

No lado do *backend*, utilizou-se a biblioteca *Django*, com base na linguagem *Python*. *Python Django*, é uma ferramenta de código aberto, surgiu como uma escolha popular para programadores *web* devido à sua ênfase no princípio DRY (*Don't Repeat Yourself*) e à facilidade que oferece na construção de aplicações *web* complexas. O *Django* segue o padrão arquitetural *Model-View-Controller* (MVC), que é adaptado no *Django* como o padrão *Model-View-Template* (MVT). Os componentes integrais do *Django* são modelos (*models*), visualizações (*views*), e roteamento de URL (*urls*). O *Django* também inclui um sistema interno de autenticação e autorização, formando uma base de segurança robusta para aplicações Web. A autenticação confirma as identidades dos utilizadores através de métodos como nome de utilizador-senha e autenticação de terceiros. Autorização, por outro lado, controla o acesso do utilizador a diferentes partes da aplicação baseado em papéis, permissões e regras. Ao integrar o sistema de autenticação embutido do *Django* e a gestão de permissões, os programadores podem garantir interacções seguras, proteger dados sensíveis e regular as acções dos

utilizadores de forma eficaz. A abordagem coesa desta estrutura à gestão de utilizadores ajuda os programadores a encontrar um equilíbrio entre a conveniência do utilizador e a segurança dos dados, crucial para a criação de soluções Web fiáveis e conformes.

Como sistema de gestão de base de dados (SGBD) optou-se pela ferramenta *PostgreSQL*. O *PostgreSQL*, um sistema de gestão de base de dados relacional (*Relational DataBase Management System* - RDBMS) de código aberto que oferece uma plataforma robusta e rica em recursos para armazenar, gerir e recuperar dados. Com uma história que remonta ao início dos anos 80, o *PostgreSQL* evoluiu para uma solução versátil que satisfaz as exigências de diversas aplicações e indústrias. Em sua essência, o *PostgreSQL* incorpora os princípios da colaboração de código aberto. Nascido do projeto Ingres da Universidade da Califórnia, Berkeley, a jornada do *PostgreSQL* começou com um foco na extensibilidade e na adesão aos padrões SQL. Esta filosofia lançou as bases para uma comunidade próspera que continua a contribuir para o seu crescimento. A arquitetura do *PostgreSQL* é um testemunho da sua eficiência e escalabilidade. Seguindo um modelo cliente-servidor, emprega um *cluster* de processos para lidar com tarefas, como processamento de consultas, manutenção em segundo plano e gerenciamento de transações. O coração do seu controlo de concorrência reside no sistema de Controlo de Concorrência Multi-Versão (MVCC), que permite elevados níveis de atividade concorrente, mantendo a consistência dos dados.

Nos últimos anos, a tecnologia de contentores transformou a forma como as aplicações são desenvolvidas, implementadas e geridas. No meio desta evolução, o *Docker* emergiu como uma das ferramentas mais influentes e amplamente adoptadas, permitindo aos programadores criar, empacotar e distribuir aplicações juntamente com as suas dependências de forma eficiente e consistente. Com a crescente adoção em todo o setor de TI, compreender os princípios, benefícios e casos de uso do *Docker* é fundamental para se manter atualizado no cenário tecnológico em constante evolução. O *Docker* é uma plataforma de código aberto que automatiza a implantação de aplicativos em contentores leves e portáteis. Um contentor é uma unidade autónoma de software que aloja o código de uma aplicação, bibliotecas, dependências e outras configurações necessárias para executar a aplicação de forma isolada e consistente em qualquer ambiente. Esta capacidade é conseguida através da virtualização ao nível do sistema operativo. O protótipo deste trabalho é executado num abiente de contentores, de forma a facilitar a transferência do ambiente de desenvolvimento para o ambiente de produção.

Palavras-chave: Sistemas de gestão da informação laboratorial (LIMS), Imunologia, *React*, *Django*, PostgreSQL, *Docker*

Abstract

The use of generic information recording and control software such as Excel is still prevalent in many biomedical research laboratories and clinical practices. With the advance of web technologies and databases, Laboratory Information Management Systems (LIMS) based on these technologies have emerged in recent decades to facilitate laboratory management. This project consists in the development of a web application prototype to be applied to the management and control of the workflow of a laboratory dedicated to immunotherapy research and clinical practice (Mauerer lab, Champalimaud Foundation). The application has been named *coley*, in honor of William B. *coley*, one of the pioneers of immunotherapy. The development of the proposed application will be based on open-source tools. The project consisted on the conception, develpment and implementation of a simple web application to implement, consisting on a graphic interface, backend and a database. For the graphic interface, is implemented in *React*JS, a well-known open-source *frontend* library based on JavaScript, created by Facebook. The backend side uses the *Django* library, based on the Python language. The DataBase Management System (DBMS) is deployed in *PostgreSQL* engine. The use of the tools enabled the development of a simple laboratory information management system, which allows users with different roles in the system to register samples and control their respective sub-samples, namely, cuts. This feature is essential for tracking and controlling sample results. For each cut it is possible to upload a result in xlsx format, corresponding to a test carried out on the cut, allowing results to be recorded and downloaded quickly and centrally. Future developments could include functionalities such as data analysis, trial creation and user performance control.

Keywords: Laboratory information management systems (LIMS), *React*, *Django*, PostgreSQL, *Docker*, Immunology

Contents

Index of Tables

Table of Figures

Abbreviation list

ACID – Atomicity, Consistency, Isolation and Durability

AI – Artificial Intelligence

API – Application Programming Interface

AWS – Amazon Web Services

DOM – Document Object Model

IoT – Internet of Things

LIMS – Laboratory Information Management System

MVC - Model-View-Controller

PaaS – Platform-as-a-Service

PLpg – Procedural Language Postgres

RBAC - Role-Bases Access Control

REST – Representational State Transfer

SQL – Structured Query Language

SSL – Secure Sockets Layer

URL – Uniform Resource Locator

1. Introduction

In scientific progress, Laboratory Information Management Systems (LIMS) stand as a pivotal technological advancement that revolutionized the way laboratories manage, organize, and analyze their data. This sophisticated software solution has a rich history that parallels the growth of laboratory science itself. The origins of LIMS can be traced back to the 1970s when the need for efficient data management and record-keeping in laboratories became increasingly evident. Laboratories, which had traditionally relied on manual methods for data entry, sample tracking, and reporting, were facing challenges in managing the sheer volume of data generated by experiments. This led to inefficiencies, errors, and delays in analysis and reporting. The early versions of LIMS were rudimentary compared to the advanced systems in use today. They primarily focused on automating sample tracking and providing basic data storage capabilities. These systems were often custom-built for specific laboratory needs, limiting their scalability and adaptability. The 1980s marked a significant turning point as technological advancements in computing and software development allowed for more comprehensive LIMS solutions. The 1990s witnessed a surge in the popularity of LIMS as commercial software vendors recognized the market potential. With the integration of graphical *user interface*s, relational databases, and networking capabilities, LIMS became more user-friendly and accessible. These systems began to offer functionalities beyond data storage, including instrument integration, workflow management, and quality control (Gibbon, 1996; Prasad & Bodhe, 2012). As the 21st century dawned, LIMS evolved into complex and integrated platforms, catering to a wide array of laboratory types, including pharmaceutical, environmental, clinical, and research labs. One of the key trends during this period was the move towards web-based and cloud-based LIMS solutions, allowing for remote access, collaboration, and data sharing (Prasad & Bodhe, 2012). This shift also addressed the challenges of data security and compliance, critical in regulated industries. The modern LIMS landscape is characterized by its multifaceted capabilities. These systems not only handle data storage and management but also facilitate laboratory automation, experimental design, compliance with regulatory standards, and even advanced data analysis through integration with other software tools. LIMS now play a pivotal role in ensuring data integrity, traceability, and reproducibility, which are paramount in scientific research and industries with stringent quality control requirements. (Sepulveda & Young, 2013) Furthermore, the integration of LIMS with other emerging technologies like artificial intelligence (AI) and the Internet of Things (IoT) has opened new frontiers for data-driven decision-making in laboratories. AI-powered LIMS can help in predictive analytics, anomaly detection, and optimization of experimental conditions (Cui & Zhang, 2021; Niazi et al., 2019). IoT-enabled LIMS can directly collect data from instruments, minimizing manual entry and reducing the risk of human

errors (Fushshilat et al., 2018; Yin et al., 2022). Looking ahead, the future of LIMS holds even greater promises. As laboratories continue to generate massive volumes of data, the demand for efficient and intelligent data management solutions will only intensify. LIMS will likely become more intuitive, leveraging machine learning to provide insights and recommendations based on patterns in data (Cabitza & Banfi, 2018). Interoperability with other laboratory equipment and systems will also be a key focus, enabling seamless data flow and enhancing overall laboratory efficiency. The history of LIMS is one of relentless evolution and adaptation. From its humble beginnings as a basic data storage tool to its current status as a sophisticated data management and analysis platform, LIMS have been a driving force in shaping modern laboratory practices. As technology continues to progress, LIMS will remain an indispensable tool, enabling scientists to focus on innovation rather than administrative tasks, and ultimately accelerating the pace of scientific discovery.

In this work, we developed a prototype web application to be applied to the management and control of the workflow of a laboratory dedicated to immunotherapy research and clinical practice (Mauerer lab, Champalimaud Foundation). The application has been given the name *coley*, in honor of William B. Coley, one of the pioneers of immunotherapy (Parish, 2003). The development of the application will be based on the implementation of open-source tools.

Building a web application is a multifaceted endeavor that encompasses various approaches, technologies, and considerations. Understanding these diverse approaches can empower you to make informed decisions and streamline your development process. Below, we'll explore key approaches in building a web app. One of the fundamental decisions when building a web app is choosing a frontend framework. These frameworks provide a structured way to create the *user interface* and handle user interactions. Some popular frontend frameworks include *React*, Angular, and Vue.js. They offer reusable components, routing, state management, and more, which can significantly speed up development and improve maintainability. The backend is responsible for server-side logic, database management, and handling frontend requests. Technologies like Node.js (Javascript), Ruby on Rails (*Ruby/Javascript*), *Django* (Python), and ASP.NET are commonly used. Data storage and management are critical components of a web app. You can opt for relational databases like *MySQL* or *PostgreSQL*, *NoSQL* databases like *MongoDB*, or hybrid solutions like *Firebase* or *Amazon DynamoDB*. APIs (Application Programming Interfaces) facilitate communication between the frontend and backend of a web app. Hosting a web app can be simplified with *Docker*. You can deploy *Docker* containers to cloud platforms like *AWS*, *Azure*, and *Google Cloud*, ensuring consistent deployment and scalability across different cloud providers. Platform-as-a-Service (PaaS) providers like *Heroku* and *Vercel* also support *Docker*-based deployments.

For this work, we will use *frontend*, *backend*, database and containerization open-source technologies. Below, we describe each of these tools and frameworks in more detail, along with the reasoning behind choosing them instead of the others available.

1.1. Objectives

The objectives for this work are to develop a database schema for an application to be used in an immunotherapty lab setting, with a RBAC and result file upload and download systems; to develop an API were data could be retrived regarding user, sample and result information; to develop an interface were the user could interact with the data, at the different levels of access, with the option of uploading and downloading result files and configure the deployment of the application using *Docker*.

1.2. Main contributions

This research project represents an advancement in the field of laboratory data management through the development and implementation of a LIMS that facilitated data storage, management, accessability and security. The primary contribution of this research is the successful creation of a comprehensive LIMS – *coley* - that effectively addresses the need of modern laboratories. The system provides functionalities for user creation, role atribution, sample tracking and result file uploading.

By utilizing *React*, *Django*, and *PostgreSQL*, *coley* LIMS leverages the latest advancements in web development and database management. *React*'s component-based architecture offers a dynamic and interactive *user interface*, while *Django* provides a secure and robust *backend* framework. *PostgreSQL* ensures reliable and structured data storage and retrieval. The *React*-based *frontend* employs intuitive interfaces, interactive components, and real-time updates to facilitate seamless interaction with the system.

The *coley* LIMS architecture is designed with scalability in mind, allowing for the accommodation of a wide range of laboratory sizes and types. The modular structure of *React* and *Django*, coupled with *PostgreSQL*'s scalability features, ensures that the system can adapt to evolving laboratory requirements.

The integration of *Django* and *PostgreSQL* ensures robust data management. *Django*'s Object-Relational Mapping (ORM) system facilitates efficient data modeling, validation, and complex queries. *PostgreSQL*'s ACID-compliant nature guarantees data integrity, even in high-throughput laboratory environments. Development of database-side function represented an improvement on the data retrieval, relieving the backend of unecessary effort, since the data is obtained and filtered directly in the database, ready to be fetched from the backend and presented in the frontend.

The source code of the *coley* LIMS is available as an open-source project. This contribution aims to foster collaboration within the scientific community, allowing for further customization, improvement, and adaptation of the system to suit specific research needs.

1.3. Outline

The present document is as follows:

Chapter 1 identifies the goals, introduce the theme and briefly describes the developed and expected contributions to the problem identified.

Chapter 2 describes *coley* LIMS architecture following the implementation of the system, starting with the database, followed by a description of the *backend*, the *frontend* and the deployment (containerization).

Chapter 3 contains a description of the final application at the time of the writing of this thesis.

Chapter 4 describes de conclusions and future work, providing a comprehensive discussion on the key findings, implications, and potential future directions stemming from this research.

2. *coley* LIMS architecture

A web application architecture diagram presents a layout with all the software components - databases, applications, and middleware - and how they interact. It defines how the data is delivered through HTTP and ensures that the communication beteween the client and *backend* servers can understand. Moreover, it also ensures that valid data is present in all user requests. It creates and manages records while providing permission-based access and authentication. In software engineering, defining an application architecture is an important initial step in a project. The architecture lays out all the software components, including databases, applications and other middleware and its interactions. It also defines how data is transmitted between components and how its demonstrated to the end user. In the following section, the architecture of coley is described.

2.1. Functional requirements & types of users

The clear definition of functional requirements is a crucial step in the successfull building of a software solution. Ensuring the application meets the precisely described needs of the client is a key approach in delivering a software that allows the workflow of the client to be comparatively better than the solution implemented before the proposed application. In this case, the main goal of *coley* is to facilitate and centralize the storage and management of result files that are generated by the analytical instruments in the *xlsx*[1] format. With this goal in mind, the first functional requirement is user authentication and authorization. The system shall provide secure user authentication using unique usernames and passwords. The app shall implement Role-Based Access Control (RBAC) (Bertino, 2003) with predefined roles such as Admin, Supervisor, Technician and Student. The roles were determined beforehand in discussions with the client, according to the laboratory workflow. The Admin will have administrator access to the application, with the possibility of creating new users. The Supervisor and Technician are not able to create new users, but can upload and view, download and track result files. The Student role will only be able to view, download and track result files.

1 *xlsx* refers to the format attributed to Microsoft Excel Open XML Spreadsheet files. It was introduced by Microsoft with the release of Microsoft Office 2007. Based on structure organized according to the Open Packaging Conventions.

The registration of patients is another functional requirement. This is the first step in the app workflow. Patient registration will be implemented using a form, where patients details will be inserted and commited to the database upon confirmation.

The app will also allow the creation of sample(s) from a given patient. A sample can be registered through a form, where there will be a field where the patients existent in the database will appear. From a given patient, data regarding the sample will be prompted to the user, such as the origin of the sample, the tumor from which the sample is retrieved, the tissue type of the sample, the date of registration, the temperature in which the sample will be stored and the container where the sample will be stored.

From a sample, the app shall also allow cut creation. From a sample, cut can be generated, that represent a subsample from an original sample. The data associated with a cut is the same as the origin sample data, the purpose of the cut (i.e., the assay or experiments to be performed) and the is registrated. The app shall also allow the uploading of result files associated with a given cut. Each cut can have one or more result files associated. For result uploading, the user (Admin, Supervisor or Technician roles) will fill a form, prompting the cut for which the result file will be created. The date of the result file creation and the actual file will also be prompted in the form.

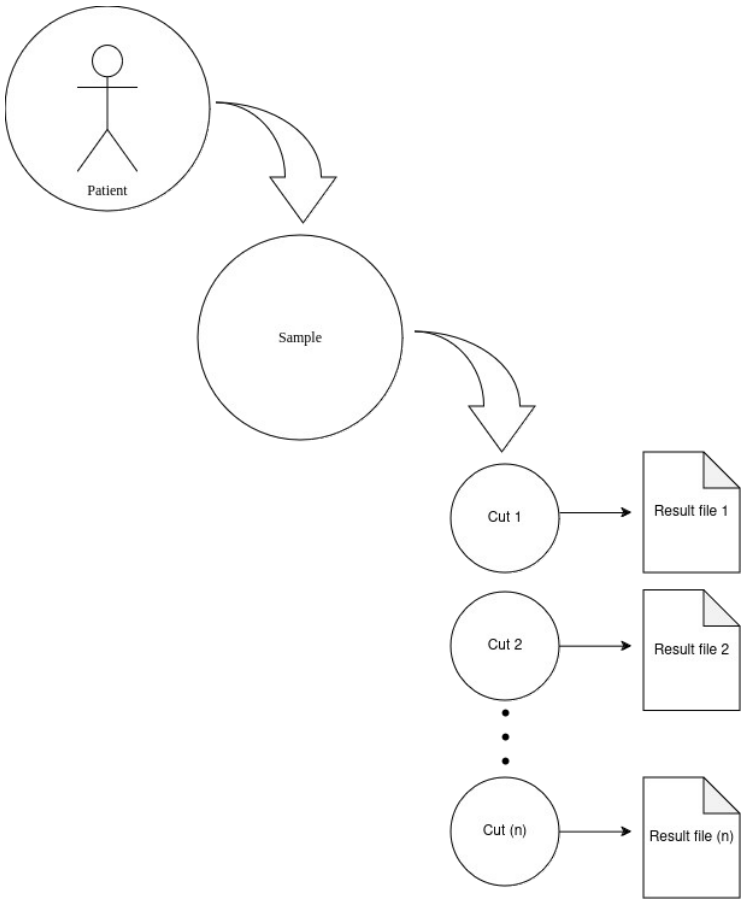

Figure 1: General overview of the application.

Still regarding the results, the app will allow the showing of all the results present in the system and their subsequent download. For each file, the user that uploaded it, the date of upload, cut and sample of origin will be present for the user to visualize (Figure 1).

2.2.Architecture overview

The *frontend* of *coley* LIMS provides the *user interface* for interacting with the system. It includes web-based interfaces for sample entry, result retrieval, reporting, and administrative functions. UI components may be developed using technologies like HTML, CSS, JavaScript, and modern *frontend* libraries or frameworks (e.g., *React*, Angular, or Vue.js) (Patrylo & Miłosz, 2017; Rawat & Mahajan, 2020).

The *backend* of coley LIMS is the layer that handles the processing and business logic of the *coley* LIMS. It includes modules for oatient registration, sample creation, cut creation. The applicationp logic can be implemented using *backend* frameworks (e.g., *Django*, Flask, .NET) and programming languages like Python, Java, or C# (Kaluža et al., 2019).

The DBMS is responsible for data storage, retrieval, and management. LIMS typically use a relational database for structured data storage. Common choices include *PostgreSQL*, MySQL, or Oracle. The database schema is designed to accommodate sample metadata, test results, user information, and other relevant data (Chamberlin, 1976; Diallo et al., 1999; Stonebraker & Rowe, 1986). The API acts as a bridge between the *frontend* and the *backend*. It defines the endpoints and protocols for communication, allowing the UI to request data and perform actions from the *backend*. RESTful APIs are commonly used for their simplicity and scalability (Ehsan et al., 2022). The application diagram is depicted on Figure 2.

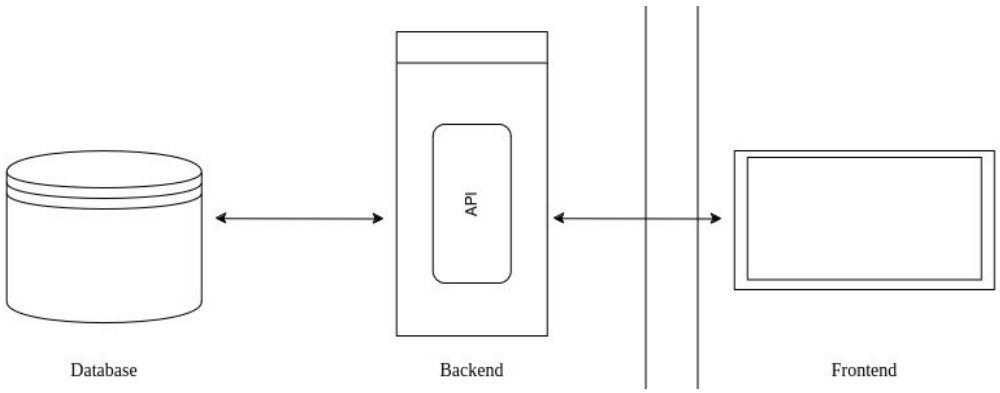

Figure 2: *coley* LIMS basic architecture

Having particularities among diferent laboratories and such, the development of a tool like is proposed in the present thesis should be applied to a case study. Therefore, *coley* is generic for many clients but it was designed accordingly to the Immunotherapy/Immunosurgery laboratory at Botton-Champalimaud Pancreatic Cancer Centre. As such, the basic wokflow of the application was designed in dialog with the client, in order to assess the functional needs for the system and its adaptation to the laboratory routine and workflow. With this in mind, the diagram in Figure 3 represents the basic workflow of the application. The start screen is the Log in Screen. If the user successfully logs in, there will be a screen with the role options for the user. The user will only be able to choose one role for each session. After confirming the role, the user moves on to the main dashboard, where he will be prompted with several options offered by the system. If the user is an Admin, there will an option to create new users. All other roles, except the Student role, are able to create patients, create samples, create cuts, upload results and download results. The Student role is able only to view and download result files.

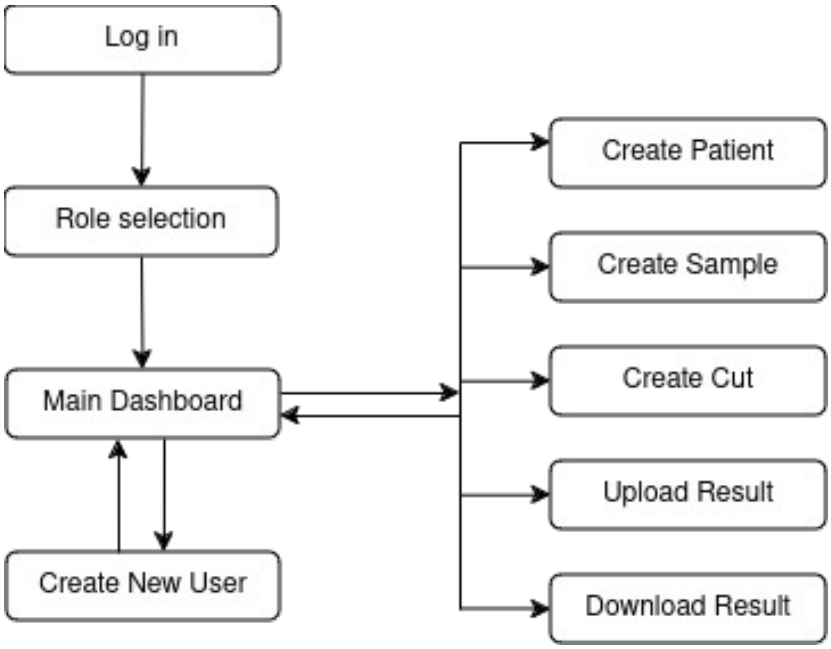

Figure 3: Workflow diagram of coley, with the basic functionalities and navigation present in the application.

3. *coley* LIMS implementation

In this chapter, we will provide an detailed description of the actual implementation of *coley*, diving into the different components described in Chapter 2. We will start by describing the database system used in the application. After, there will be a description of the backend and the *frontend*. Closing this chapter, there will be section describing the deployment of *coley*.

3.1.Database

The schema for *coley* was designed to efficiently store and manage various types of laboratory data. This included entities such as samples, analysis, results, users, patients and associated metadata. Relationships between different entities were established using foreign keys to ensure data integrity. In *Django*'s authentication and permission system, several tables within the database handle the intricate details of user management, permissions, groups, and sessions. *auth_user* table serves as the central repository for user-related information (Gagliardi, 2021; Gore et al., 2021; Kaluža et al., 2019). The schema for this system is described in Figure 4. It stores user-specific data such as username, password (usually hashed for security), email, first and last names, whether the user is active, and when the user joined. This table forms the backbone of user management in *Django*. *auth_group* describes collections of users that can share common permissions. The *auth_group* table stores group names and serves as a container for assigning permissions collectively to multiple users at once. The *auth_permission* table holds details about individual permissions that users or groups can possess. Each permission is linked to a specific model and can grant actions like view, add, change, or delete for that particular model. Permissions here are created when models are defined and can be assigned to users or groups, allowing fine-grained control over who can do what within the application.

The *auth_user_groups* intermediary table establishes a Many-to-Many relationship between users and groups. It functions as a bridge, allowing users to be associated with one or more groups, thereby inheriting the permissions associated with those groups. The *auth_user_user_permissions* table is similar to *auth_user_groups*, this table also implements a Many-to-Many relationship, but it links users directly to specific permissions without involving groups.

This mechanism enables individual users to possess distinct permissions that might not be shared by other users in the same group. *Django_session*: This table manages user sessions within the *Django* application. It keeps track of session data such as session key, user ID (if authenticated), IP address, user agent, and last activity timestamp. Sessions are crucial for maintaining user authentication across multiple requests, allowing users to stay logged in until they explicitly log out or

their session expires. These tables collectively form the scaffolding for user authentication, permission assignment, and session management within *Django*. They provide a robust foundation for developers to control access, manage user groups, and maintain secure user sessions within their applications (Gagliardi, 2021; Gore et al., 2021).

Also, there are the tables specific for *coley's* specific functionality. The current version contemplates 9 tables: *sample, tumor_type, tissue_type, temperature, container, patients, cut,* and *analysis.* The *user* table present on the diagram corresponds to the *auth_user* table from the *Django*'s authentication system. The *sample* table describes the data related to each sample that is received by the laboratory. It is connected in a One-to-One fashion with auxiliary tables such as *tumor_type*, *tissue_type*, *temperature* and *container.* It is also conected to the *patients* and *user* table, where for each sample, a patient and user has to be associated - the patient from which the sample is originated and the *coley* user that registered it in the system. The auxiliary tables mentioned before describe the type of tumor the sample consists of, the type of tissue (i.e. blood, skin, connective tissue, etc), the temperature at which the sample will be stored and the physical container where the sample will be stored. The *patient* table contains the data related to the patients from which the samples will derive. The *cut* table has information related to the cuts originated from a given sample and the *analysis* table pertains to the information regarding the result file associated with a given cut. The schema with the tables specific to *coley* is described in Figure 5.

The chosen DBMS should allow relation among entities, wide option in attribute datatype, have extensive documentation and wide community and be open-source. With this in mind, *PostgreSQL* was chosen, since it gathers all the requirements mentioned previously and is growing to be one of the most popular DBMS used in several real-world applications and businesses.

The development of Procedures and Functions significantly enhances the database's capabilities by allowing users to encapsulate complex logic and operations within the database itself. Procedures are a set of SQL statements that can be executed with a single call. They offer a powerful way to handle tasks that require multiple steps, transactions, or conditional processing. By creating procedures, developers can streamline processes, improve code modularity, and reduce redundancy in database operations. Functions, on the other hand, are similar to procedures but can return a value. They encapsulate reusable blocks of code and enable dynamic data processing within queries. This makes it possible to perform intricate calculations, data transformations, and custom operations directly within the database environment. Functions also support parameterization, allowing for versatile and adaptable solutions. With this in mind, several Procedures and Functions were developed

to retrived, process and filter data to be used by the API and exposed to the *frontend*. A summary of all the functions and procedures is presented in Tables 1 and 2.

PostgreSQL, an open-source relational database management system, was selected as the database engine. *PostgreSQL* is an open-source database management system (DBMS) that incorporates the relational approach for its databases and makes use of the Structured Query Language (SQL), a domain language specifically developed to manage data. It's history dates back to the early 1980's and it has evolved into a versatile solution that meets the demands of diverse application in multiple industries. It has emerged from the Berkeley's Ingres project, from the University of California and began with a focus on extensibility and adherence to SQL standards (Held et al., 1975). This philosophy laid the foundation for a thriving community that continues to its growth.

The architecture of *PostgreSQL* follows a client-server model and employs a cluster of processes to handle tasks, such as query processing, background maintenance, and transaction management. The heart of its concurrency control lies in the Multi-Version Concurrency (MVCC) system, which enables high levels of concurrent activity while maintaning data consistency. A defining feature of *PostgreSQL* is its emphasis on data integrity and extensibility. With support for various data types, including geometric, textual, and JSON, *PostgreSQL* enables developers to model complex real-world scenarios accurately. The extensibility is further amplified by custom operators, functions, and procedural languages, providing a robust framework for building specialized functionality. Query optimization is a cornerstone of *PostgreSQL*'s performance prowess. The query planner employs a cost-based approach to evaluate multiple execution strategies and chooses the most efficient one. This process, coupled with advanced indexing mechanisms and techniques like table partitioning, results in responsive query performance even for complex operations. Moreover, *PostgreSQL*'s commitment to adherence to SQL standards fosters portability and compatibility. Developers can seamlessly migrate from other RDBMS platforms or integrate *PostgreSQL* into existing systems, reducing the barriers to adoption and encouraging the utilization of its advanced features.

In recent years, *PostgreSQL* has embraced modern trends, staying relevant in a data landscape dominated by big data and NoSQL solutions. Its support for JSON data types and advanced indexing mechanisms for text search and geospatial data enables it to handle a wide variety of data structures. This flexibility, combined with its robust ACID compliance, positions *PostgreSQL* as a viable option for hybrid applications. Security is a paramount concern in any database system, and *PostgreSQL* addresses it diligently. Role-based access control, SSL encryption, and comprehensive auditing capabilities help safeguard sensitive information, making it suitable for applications requiring stringent security measures.

Real-world applications testify to *PostgreSQL*'s prowess. It has found a home in industries spanning e-commerce, finance, healthcare, and geospatial analysis. Well-known organizations rely on *PostgreSQL* for its ability to manage large datasets and complex queries while maintaining data integrity and reliability. Looking ahead, *PostgreSQL* continues to evolve. Ongoing research and development focus on enhancing performance through parallelism and optimizing query execution. With the rise of cloud computing, *PostgreSQL*'s adaptability is evident as it integrates with various cloud platforms, providing scalable solutions for data storage and management (Rowe & Stonebraker, 1987; Stonebraker & Rowe, 1986).

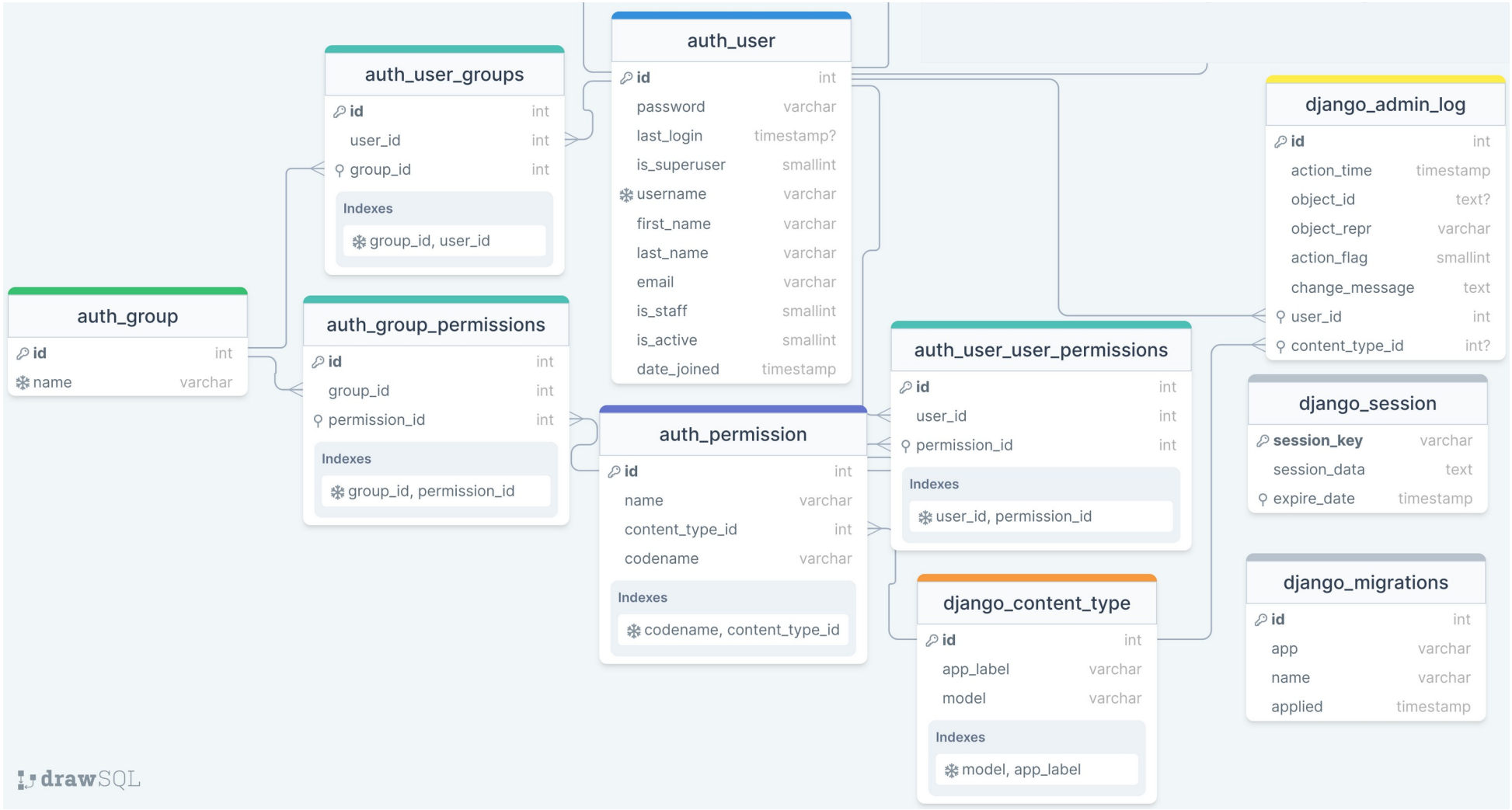

Figure 4: *Django*'s built-in authentication system used from the RBAC.

Table 1 - PLpg/SQL auxiliary functions implemented in *coley* database

PLpg/SQL functions	Description
authenticated_user(userid integer)	Retrieves the data associated with the current logged in user
get_all_cuts_from_sample(sampleid integer)	Retrieves all the cuts associated with a given sample
sample_information(sampleid integer)	Retrieves the selected sample information
sample_list_for_user(userid integer)	Retrieved the samples registeres by a given user
user_available_roles(userid integer)	Retrieves the roles available for a user
user_list()	Retrieved a list of the current users registered in the database

Table 2 - Postgres Procedures implemented in *coley* database.

Postgres Procedures	Description
create_new_patient(name text, dob date, gender text)	Registers a new patient in the database
create_new_user(username character varying, firstname character varying, lastname character varying, lastname character varying, e-mail character varying, password character varying, roles numeric[])	Registers a new user for the application, with a given list of roles chosen upon creation by the admin
register_new_analysis(userid integer, cutid integer, submitdate text, xlsxfile text)	Submit an analysis file
register_new_sample(userid integer, origin text, patientid integer, tumortypeid integer, tissuetypeid integer, temperatureid integer, containerid integer, location json, entrydate timestamp without timezone)	Register a new sample
register_new_cut(userid integer, prps, sampleid, cutdate)	Register a new cut from a sample

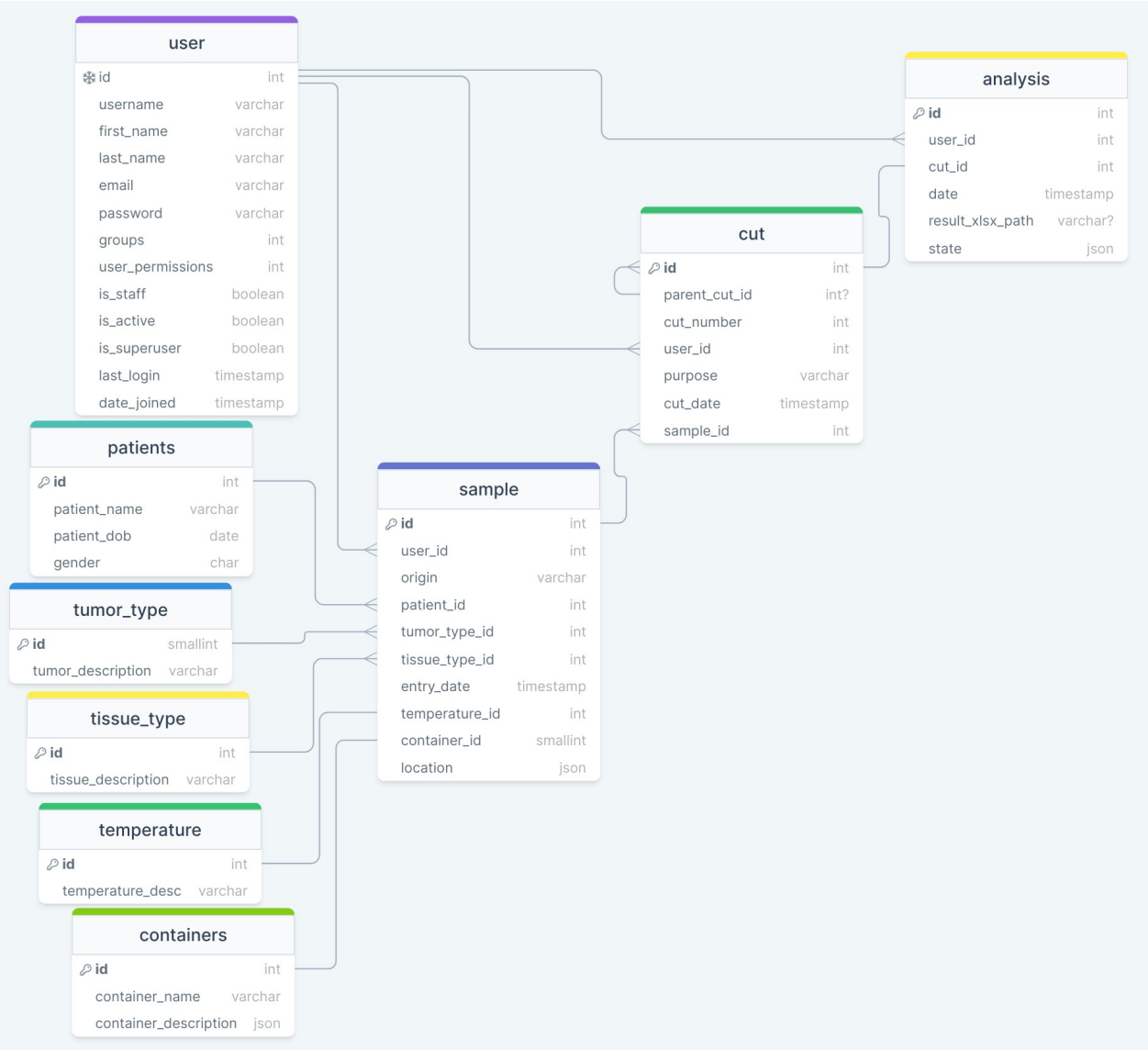

Figure 5: *coley* specific database schema

3.2. Backend

For the backend, the system chosen should have the ability of seamlessly build a REST API, to allow authentication and authorization, allow the upload and download of files and be of open-source licensing. Having all this in mind, *Django* was the framework chosen (Gagliardi, 2021; Gore et al., 2021). *Django* is a high-level Python web framework. *Django*, an open-source web framework, is renowned for its efficiency, scalability, and versatility in web development. This versatile Python framework has evolved significantly since its inception in 2005, thanks to its history, a multitude of features, and dedicated community support.

Since its inception, *Django* has undergone several version updates, with each iteration introducing new features and enhancements. At its core, *Django* embraces the Model-View-Template (MVT) architectural pattern, which shares a kinship with the widely acknowledged Model-View-Controller (MVC) pattern. In *Django*'s paradigm, Models serve as the bedrock for defining the data structure, Views orchestrate user interaction and encapsulate business logic, while Templates govern the presentation layer. This strategic separation of concerns empowers developers to craft code that is clean, organized, and intuitively maintainable.

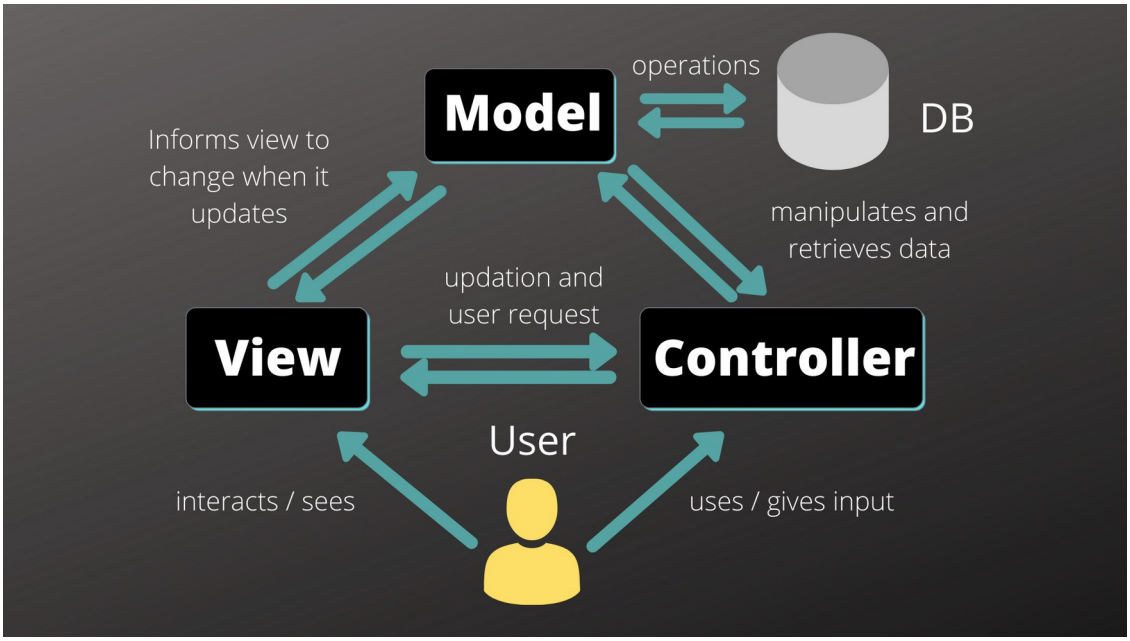

Figure 6: *Django*'s MVC architecture (Sharma, 2022).

Django's adaptability is one of its most striking attributes. Its proficiency extends across a diverse spectrum of web applications, spanning from dynamic content management systems (CMS) to intricate e-commerce platforms and even the intricate frameworks underpinning social media

networks. This versatility arises from *Django*'s modularity and its innate capacity to integrate seamlessly with an extensive array of third-party packages, rendering it an exceptional choice for a multitude of projects (Gore et al., 2021; Kaluža et al., 2019).

Django allows the construction of API's in its native form, although there are libraries specifically developed for API (Gagliardi, 2021). In this work, an API was developed without the use of libraries, and its implementation was tested using Postman (Soni & Ranga, 2019). Below, we describe each endpoint created with a short summary depicted in Table 3.

The authentication system was successfully implemented, allowing users to register, log in, and recover passwords. Role-based access control (RBAC) was enforced, with different levels of permissions assigned to admin, supervisor, technician and student roles. This ensured that users only had access to the functionalities relevant to their roles (Bertino, 2003).

The RBAC implementation was built upon the *Django*'s built-in authentication system. It is based in the tables on the database, the *auth_group* table shown in Figure 7 and the *auth_user_groups* table shown in Figure 8.

	id [PK] integer	name character varying (150)
1	1	Admin
2	2	Supervisor
3	3	Technician
4	4	Student

Figure 7: *auth_group* table from *Django*'s built-in authentication system.

	id [PK] bigint	user_id integer	group_id integer
1	1	1	1
2	2	1	2
3	3	1	3
4	4	1	4

Figure 8: *auth_user_groups* table, where the user has its roles defined.

Table 3 - API endpoints (a more detailed documentation fo the API is available in Annex 2)

endpoint	description
login/	Called at login, returns the status of login, an error and a variable used to open the session,
logout/	Called at logout, returns the success of the logging out.
create_user/	Called at user creation, posts the new user data to the database and returns a success or error message
user_roles/	Called at a successfull log in, returns the list of roles available for the logged user.
new_sample/	Called at sample creation, posts the data for a sample created from a given patient.
new_cut/	Called at cut creation, posts the data for a cut created from a given sample.
new_patient/	Called at patient creation, posts the data for a newly created patient.
patient_list/	Returns the patients available in the database.
tissuetypes/	Returns the tissue types available in the database.
tumortypes/	Returns the tumortypes available in the database.
temperatures/	Returns the temperatures available in the database.
containers/	Returns the containers available in the database.
samples/	Returns the samples available in the database.
cuts_from_sample/	Returns cuts available for a giver sample present in the database.
result_file_upload/	Called at file submission, returns the success status of the submission of the file.
get_analysis/	Returns all data available in the database for all the result files submitted to the systems.
get_users/	Returns the list of users available in the database.
download_file/	Called at file download, launched the file download or returns an error message.

Django's group definition and the role definition in our project overlap in their meaning. The *auth_group* table contains the available roles by the system, with each corresponding unique identifier. The *auth_user_groups* table defined the roles available for each user, and each user-role association has its unique identifier.

From the frontend, the RBAC is implemented through the UseEffect hook and the *location.state*, and is described in more detail in the next section.

3.3. Frontend

For the *frontend*, we consider that the library or framework should allow the building of simple and aestetically appealing *user interface*s, easy to understand and implement and, at the same time, allow for flexibility and customization; it should allow to easily build an interface with a responsive design. Additionally, it should have a thiving community with detailed documentation, be used in real-world settings and be open-source. With all this in mind, the frontend of *coley* was developed using *React*, a widely used JavaScript library for building *user interface*s. In the ever-evolving world of web development, *ReactJS* has emerged as a groundbreaking technology that has redefined how web applications are built and maintained. Developed by Facebook, *ReactJS* is an open-source JavaScript library that has gained immense popularity due to its declarative approach, component-based architecture, and efficient rendering. This text aims to provide an overview of *ReactJS*, exploring its core concepts, benefits, and its impact on modern web development and demonstrate how *coley* interface was build with this library. At the core of *ReactJS* lie several core concepts that differentiate it from traditional web development approaches. One of the key principles is declarative programming, where developers describe the desired UI state, and *React* takes care of efficiently updating the actual UI to match that state. This approach eliminates the need for imperative code to manipulate the DOM directly, leading to more maintainable and readable code. The Virtual DOM is another fundamental concept that underpins *React*'s efficiency. Rather than directly manipulating the real DOM, *React* creates an abstract representation of it in memory, known as the Virtual DOM. When the state of a component changes, *React* calculates the difference between the previous and current Virtual DOM states and updates only the necessary parts of the actual DOM. This process significantly reduces the performance overhead associated with traditional DOM manipulation. *React*'s component-based architecture is a cornerstone of its development philosophy. Developers break down *user interface*s into smaller, reusable components. These components encapsulate specific functionality and can be easily composed to create complex UIs. This modularity promotes code reusability, readability, and maintainability. Components can be categorized into two types: class components and functional components. Class components have been a traditional way of creating components and managing local state. However, with the introduction of *React* Hooks, functional components have gained prominence. Hooks allow developers to manage state and side effects within functional components, eliminating the need for class components in many cases. *React* follows a unidirectional data flow pattern, ensuring a clear and predictable flow of data through an application. In this pattern, data flows

from parent components to child components through props (short for properties). This strict flow of data simplifies debugging and makes the application's behavior more predictable, as changes in state only affect the components that directly depend on that state. State management is achieved by using the useState hook to define and manage component-specific state, or by employing external state management libraries like Redux or MobX for more complex scenarios. This unidirectional data flow promotes separation of concerns and enhances the overall architecture of the application. *React*JS development involves several key steps. First, developers design the *user interface* by creating reusable components. These components can be as simple as buttons or as complex as entire sections of a webpage. The JSX (JavaScript XML) syntax allows developers to write HTML-like code within JavaScript, making UI creation intuitive and efficient. Next, developers define the application's state and props, which determine how the UI behaves and appears. State defines the internal data of a component, while props are passed down from parent to child components. As the state changes, *React* automatically re-renders the components to reflect the updated state. Finally, developers can leverage various lifecycle methods or hooks to control the behavior of components at different stages of their lifecycle. *React*JS has fostered a vibrant ecosystem of libraries, tools, and resources that complement its capabilities. *React* Router, for example, facilitates the creation of single-page applications with dynamic routing, enabling seamless navigation between different views. Material-UI and Ant Design offer pre-designed UI components, enabling developers to create visually appealing interfaces with ease. Moreover, the *React* community actively contributes to the growth of the ecosystem by developing and sharing open-source packages, extensions, and best practices. This collaborative nature has led to the creation of a wealth of resources, tutorials, and forums where developers can seek guidance and exchange knowledge. *React*JS has undoubtedly left an indelible mark on web development, offering a paradigm shift in how *user interface*s are constructed and maintained. Its declarative approach, virtual DOM optimization, component-based architecture, and thriving ecosystem have propelled it to the forefront of modern web development. *React*'s influence extends beyond its technical prowess; it has also shaped the way developers approach UI design and coding practices. As the digital landscape continues to evolve, *React*JS remains a testament to the power of innovation in shaping the tools that drive the modern web forward (Rawat & Mahajan, 2020).

The implementation of RBAC in the frontend makes use of the *React* hook *useEffect* and the *React* Router hook *useLocation* (Ganatra, 2018). At log in, a variable *isLogged* is attributed a boolean value of *true*, that is transported throughout the navigation in the application with *useLocation*. The *useLocation React* Router hook allows the application to access the location object that represents the active URL. The value of this object changes whener the user navigates to a new URL, allowing the trigger of events whenever the URL changes. *UseLocation* is a good method for making sure the user

is logged in before accessing the different screens corresponding to the different functionalities of the application. This verification is made with *useEffect*. This hook allow the attachment of side effects to components of the application. These side effects include, for example, data fetching, reading from local storage, updating the DOM and timers. In *coley*, a *useEffect* is used to verify the value of the variable *isLogged* mentioned previously, upon component rendering on the DOM. Once the value of isLogged is verified:

- if true, the DOM renders the screen associated with the URL

- If false, the user is navigated to the Login page.

The functionalities restricted ot the Admin and Supervisor/Technician in the Main Dashboard were implemented using conditional rendering with embbeded JSX expressions. The inline code allows for a specific functionality to be present or absent depending on the role chosen by the user. With this, the logged user will see only the functionalities available to its role.

3.4. Deployment

For deployment, the containerization tool *Docker* was used to containerize the application, and facilitate the deployment to production (Rad et al., 2017). With this in mind, two *Dockerfiles* were created, for the backend and frontend. Given the architecture of *coley* being composed of distinct frontend, backend and database servers, *Docker* compose was used to deploy *coley* as a multicomponent application (Ibrahim et al., 2021). This was achieved with a *Docker*-compose file, in YAML format.[2] After setting the appropriate commands, the containerized application can be accessed in the localhost of the machine running *Docker* Desktop. After all images are running correctly, the database can be created using an SQL script provided in the source code. The script is used to create the appropriate tables, set the essential data required for accessing and use the application seemlessly. Any development should be replicated by other authors and investigators. Thereby, the instructions to replicate the system can be found in the Annex A.3.

3.5. Results

The starting page for the app is the login page depicted in Figure 9. A minimalistic approach was taken, were online the username and password fields are presented, along with the login button,

2 YAML, from YAML is Not a Markup Language, is a data serialization language intended for human readability. It is used for configuration files and very commonly used for *Docker* containerization. (Fonseca & Simões, 2007)

all centered on the DOM. After logging in, a user is able to choose the role he wants to proceed, from the available roles for that given user, as seen in Figure 10. More screens and forms were implemented such as the Main Dashboard, User Creation, Patient Creation, Sample Creation, Cut Creation, Result upload and Result Download. All the interactions with the database flow seemlessly through the *Django backend* and all the core functionalities behave as expected. After choosing a role, a dashboard will appear. This dashboard will have different presentations for different roles, with the biggest difference being the *Admin* role allowed to create new user, through the button Create New User. On the top of the Dashboard, one can also see the current user logged in and the respective role attributed in the session, seen in Figure 11.

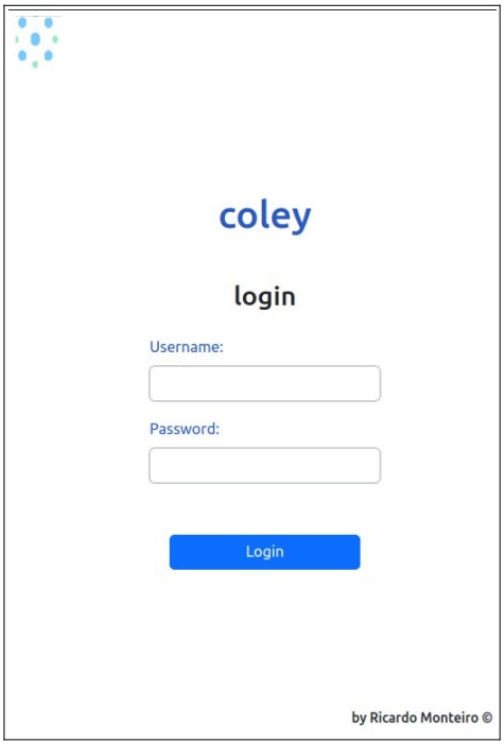

Figure 9: Login page.

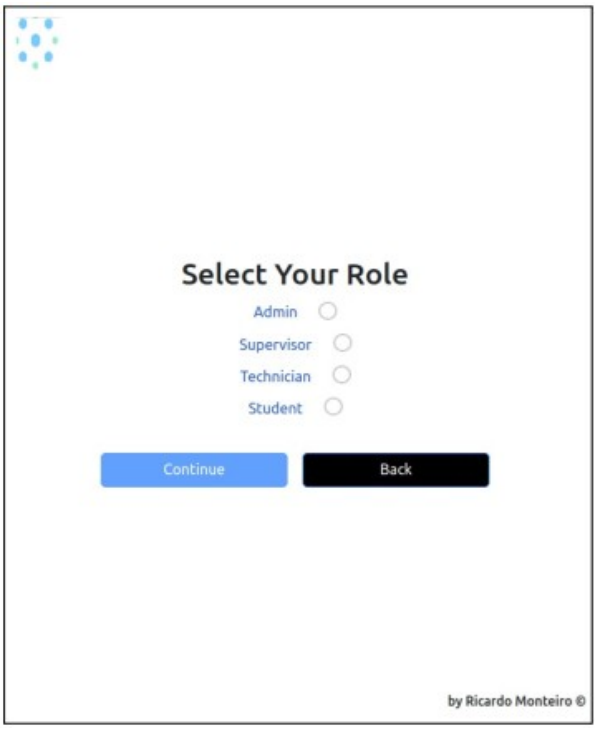

Figure 10: Different roles allowed in the system.

Also, there are all the different options of interacting with *coley* in the version existing at the time of writing this thesis. There are the options: Create new patients, Create new samples, Create new cuts, Create new analysis, Create new users and Results. The New patient button will open a form for the new patient being registered. This form includes a *Name*, *Date of Birth* and *Gender* (Figure 12).

Next, there is the form for sample registration opened with the button New sample (Figure 13). This form comprises of a *Patient* form where the sample was taken (hence, the patient creation is necessary prior to sample registration). An open field is present to describe the origin of the sample,

i.e., the hospital, clinic or laboratory from which the sample was sent to the lab. Next, there's the tumor type dropdown, which allows the selection of the type of tumor for the sample. The tumor types are defined in the database. This is also true for the field of the tissue type, the temperature and the container where the sample will be stored. All these fields are present in this form as dropdown menus.

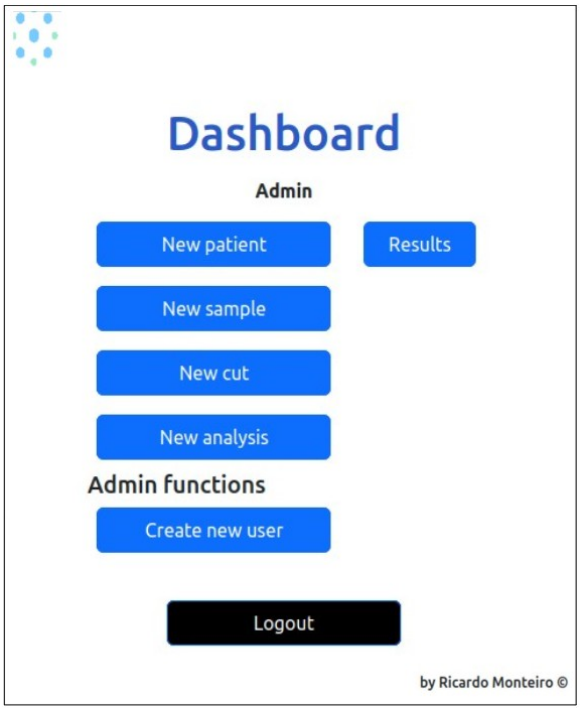

Figure 11: Options for interacting with coley as an admin user.

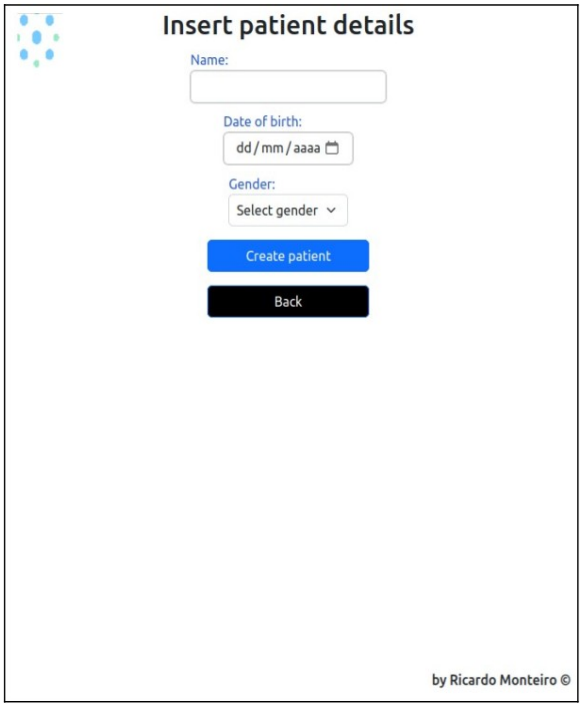

Figure 12: Create patient form.

There's also a date input field, which allows the user to associate a date with the sample registration. By default, the date is the current day. From a sample, the user can create a cut, from which a result file will be uploaded. According the the lab workflow, the cut is a segment of a sample to be analysed and processed through a given protocol. With the New cut button, the cut creation screen will appear. The user will be prompted to select a sample from which to create the cut, describe the cut's purpose (i.e. the type of assay, analysis or protocol to be applied to the cut) and the cut creation date (Figure 14). The New analysis button opens the form for *Analysis* file submission. This is a crucial part of the application, since it allows for the uploading of result files. For file submition, the form requires an original sample and cut. Also, associated with a file there will be a submission date. By default, it has the current day (Figure 15).

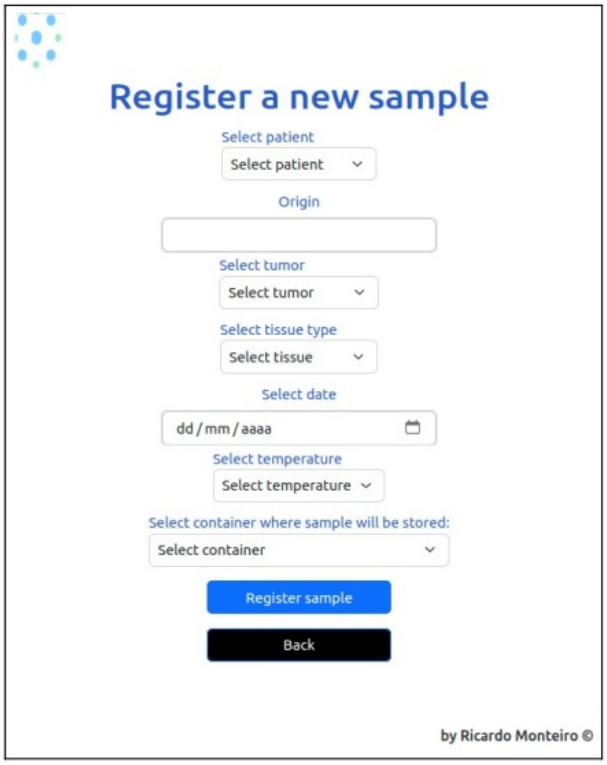

Figure 13: Sample registration form.

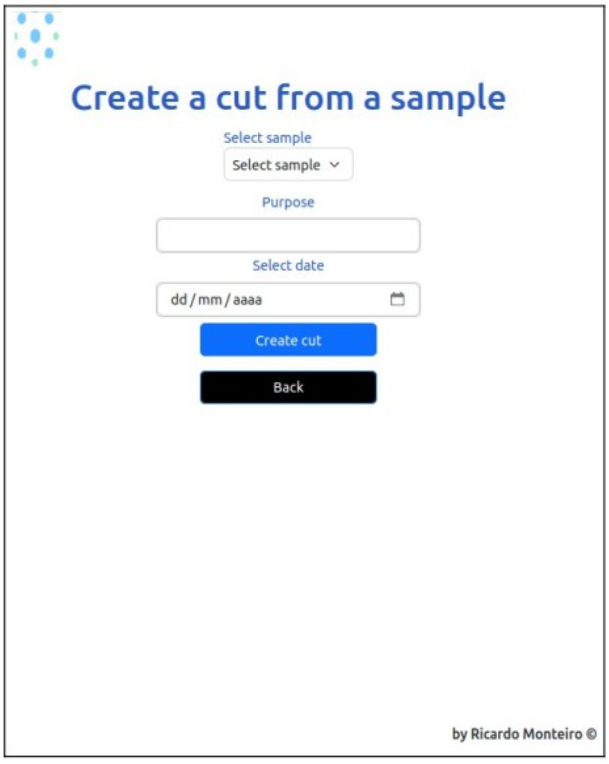

Figure 14: Cut creation form.

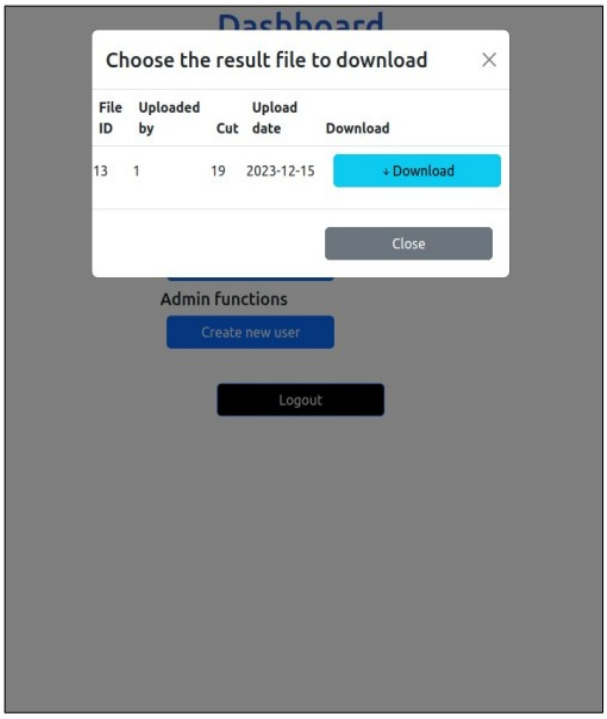

Figure 15: Result file download modal.

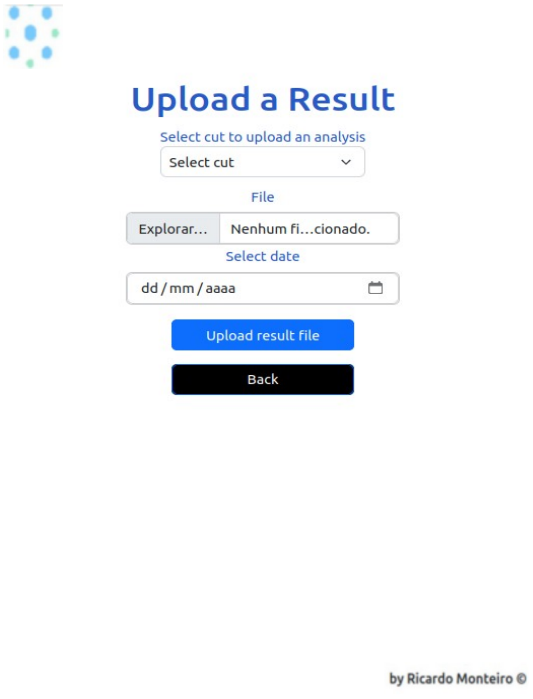

Figure 16: Result file upload form.

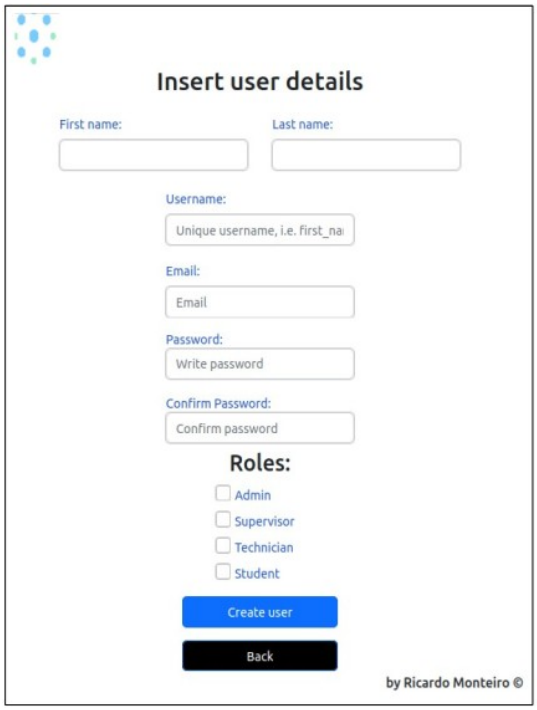

Figure 17: User creation form.

Next, the user can see the results files currently present in the system. After clicking in the Results button, a modal will appear with a table containing information regarding each file in the system, with a button that allows the download of the respective file (Figure 16). The Create user button (only allowed to users with Admin role) will open a form to register new users. This form includes the name of the new user (first and last name), the username, e-mail, password and the roles attributed to the new user (Figure 17).

4. Conclusion and future work

4.1. Conclusion

The development of *coley* LIMS using *React*, *Django*, and *PostgreSQL* represents a significant advancement in the field of laboratory data management. The *coley* LIMS, developed using *React* for the *frontend*, *Django* for the *backend*, and *PostgreSQL* as the database system demonstrated a high level of functionality and usability. The *React* framework allowed for the creation of a dynamic and responsive *user interface*, enabling smooth navigation and interaction. The incorporation of *Django* provided a robust backend structure that facilitated seamless data management and processing. *PostgreSQL* played a critical role in ensuring data integrity, security, and scalability. The use of *PostgreSQL*, known for its ability to manage extensive datasets efficiently, proved instrumental in achieving this goal. Additionally, *React*'s virtual DOM and efficient rendering mechanisms contributed to a smooth user experience, even with complex and data-intensive operations. The modular architecture of *React* and *Django* allowed for a high degree of flexibility in system design and functionality. Components could be easily extended, modified, or replaced, facilitating customization to meet specific laboratory requirements. This adaptability is crucial in accommodating the diverse needs of various scientific disciplines and research areas. Data security is paramount in laboratory environments, where sensitive information and proprietary research data are handled. In case of security failure, private patient data could be leaked and important data for research could be lost. The combination of *Django*'s robust authentication system and *PostgreSQL*'s advanced security features ensured that only authorized users could access, modify, or delete critical information. Furthermore, the implementation of secure communication protocols and encryption mechanisms added an extra layer of protection to the system. Throughout the development process, the users initial consultation played a crucial role in shaping the system's features and functionality.

In conclusion, the development of a LIMS using *React*, *Django*, and *PostgreSQL* represents a significant achievement in laboratory data management. The system's functionality, performance, and flexibility make it a valuable tool for researchers and laboratory personnel. By addressing key challenges and considering future enhancements, this LIMS has the potential to significantly advance scientific research and data management practices. The code for the application is available at https://github.com/ricmonteiro/coley.

4.2. Future work

While *coley* LIMS demonstrated notable strengths, it is important to acknowledge certain challenges and limitations. These include potential compatibility issues with older legacy systems, the learning curve associated with adopting a new system, and the need for ongoing maintenance and support.

Several avenues for future research and development exist. Adding the feature of visualizing the values associated with the results submited directly in the application could improve it's usability. Machine Learning Integration: Exploring the integration of machine learning algorithms to automate data analysis and improve decision-making processes within *coley* LIMS. Cloud Integration and Distributed Computing: Investigating the feasibility of migrating the LIMS to a cloud-based infrastructure to enhance scalability and accessibility. Integration with IoT Devices: Exploring the integration of Internet of Things (IoT) devices for real-time data acquisition and monitoring. Features such as study creation, sample and cuts visualization, patient/study associated results, data visualization directly in the application, app user monitoring and other can be of great value for a laboratory dedicated to preclinical development. These features are included in many modern LIMS and would represent a valuable enhancement of our current system version. A safer and more comprehensive documentation will also allow for a improved depth and richness in data analysis since clinical samples often undergo so-called exploratory assays in addition to highly quality-controlled and validated assays (performed in standard clinical laboratories). *coley* represents an improvement of the often utilized excel platform in preclinical development towards a safer, use-friendly sample analysis/documentation system.

5. References

Bertino, E. (2003). RBAC models - Concepts and trends. *Computers and Security*, *22*(6), 511–514. https://doi.org/10.1016/S0167-4048(03)00609-6

Cabitza, F., & Banfi, G. (2018). Machine learning in laboratory medicine: Waiting for the flood? *Clinical Chemistry and Laboratory Medicine*, *56*(4), 516–524. https://doi.org/10.1515/cclm-2017-0287

Chamberlin, D. D. (1976). Relational Data-Base Management Systems. *ACM Computing Surveys (CSUR)*, *8*(1), 43–66. https://doi.org/10.1145/356662.356665

Cui, M., & Zhang, D. Y. (2021). Artificial intelligence and computational pathology. *Laboratory Investigation*, *101*(4), 412–422. https://doi.org/10.1038/s41374-020-00514-0

Diallo, B., Travere, J. M., & Mazoyer, B. (1999). A review of database management systems suitable for neuroimaging. *Methods of Information in Medicine*, *38*(2), 132–139.

Ehsan, A., Abuhaliqa, M. A. M. E., Catal, C., & Mishra, D. (2022). RESTful API Testing Methodologies: Rationale, Challenges, and Solution Directions. *Applied Sciences (Switzerland)*, *12*(9). https://doi.org/10.3390/app12094369

European Commission. (2017). IVDR - Regulation (EU) 2017/746 on in-vitro diagnostic medical devices. *Official Journal of the European Union*, *60*(April 2014), 1–175. 1. European Commission. MDR - Regulation (EU) 2017/746 on in-vitro diagnostic medical devices. Off J Eur Union [Internet]. 2017;60(April 2014):1–175. Available from: https://www.emergogroup.com/sites/default/files/europe-medical-devices-regulation.pdf%0Ah

Fonseca, R., & Simões, A. (2007). Alternativas ao XML: YAML e JSON. *XATA 2007-5ª Conferência Nacional Em XML*, *5*(Aplicações e Tecnologias Aplicadas), 33–46. http://repositorium.sdum.uminho.pt/bitstream/1822/6230/1/xmlyamljson07.pdf

Fushshilat, I., Rahmat, A., Somantri, Y., & Haritman, E. (2018). Laboratory management: Digital laboratory information system (DLIS) concept. *IOP Conference Series: Materials Science and Engineering*, *434*(1). https://doi.org/10.1088/1757-899X/434/1/012286

Gagliardi, V. (2021). Decoupled *Django*. In *Decoupled Django*. https://doi.org/10.1007/978-1-4842-7144-5

Ganatra, S. (2018). *React Router Quick Start Guide: Routing in React Applications Made Easy*. Packt Publishing Ltd.

Gibbon, G. A. (1996). A brief history of LIMS. *Laboratory Automation and Information Management*, *32*(1), 1–5. https://doi.org/10.1016/1381-141X(95)00024-K

Gore, H., Singh, R. K., Singh, A., Singh, A. P., Shabaz, M., Singh, B. K., & Jagota, V. (2021). *Django*: Web development simple & fast. *Annals of the Romanian Society for Cell Biology*, *25*(6), 4576–4585.

Ibrahim, M. H., Sayagh, M., & Hassan, A. E. (2021). A study of how *Docker* Compose is used to compose multi-component systems. *Empirical Software Engineering*, *26*(6), 128. https://doi.org/10.1007/s10664-021-10025-1

Kaluža, M., Kalanj, M., & Vukelić, B. (2019). A comparison of *backend* frameworks for web application development. *Zbornik Veleučilišta u Rijeci*, *7*(1), 317–332. https://doi.org/10.31784/zvr.7.1.10

Niazi, M. K. K., Parwani, A. V., & Gurcan, M. N. (2019). Digital pathology and artificial intelligence. *The Lancet Oncology*, *20*(5), e253–e261. https://doi.org/10.1016/S1470-2045(19)30154-8

Parish, C. R. (2003). Cancer immunotherapy: the past, the present and the future. *Immunology and Cell Biology*, *81*(2), 106–113.

Patrylo, N., & Miłosz, M. (2017). Comparison of AngularJS and VueJS frameworks efficiency. *Journal of Computer Sciences Institute*, *5*, 204–207. https://doi.org/10.35784/jcsi.622

Prasad, P. J., & Bodhe, G. L. (2012). Trends in laboratory information management system. *Chemometrics and Intelligent Laboratory Systems*, *118*, 187–192. https://doi.org/10.1016/j.chemolab.2012.07.001

Rawat, P., & Mahajan, A. N. (2020). *React*JS: A Modern Web Development Framework. *International Journal of Innovative Science and Research Technology*, *5*(11), 698–702. www.ijisrt.com

Rowe, L. A., & Stonebraker, M. R. (1987). The POSTGRES Data Model. *Proceedings of the 13th International Conference on Very Large Data Bases*, *4871*, 83–96. http://db.cs.berkeley.edu/papers/ERL-M87-13.pdf

Sepulveda, J. L., & Young, D. S. (2013). The ideal laboratory information system. *Archives of Pathology and Laboratory Medicine*, *137*(8), 1129–1140. https://doi.org/10.5858/arpa.2012-0362-RA

Soni, A., & Ranga, V. (2019). API features individualizing of web services: REST and SOAP. *International Journal of Innovative Technology and Exploring Engineering*, *8*(9 Special Issue), 664–671. https://doi.org/10.35940/ijitee.I1107.0789S19

Stonebraker, M., & Rowe, L. A. (1986). The Design of POSTGRES. *ACM SIGMOD Record*, *15*(2), 340–355. https://doi.org/10.1145/16856.16888

Yin, J., Li, Q., He, Y., Du, X., & Song, G. A. O. (2022). Intelligent Laboratory Management System Based on NB-IoT. *ACM International Conference Proceeding Series*, 119–125. https://doi.org/10.1145/3577065.3577087

A. Annexes

A.1. Annex 1 – Postgres Functions and Procedures

A.1.1. PLpg/SQL functions

NOTE: The code segment of the following tables contains data from a test version of the application and serves only as a demonstration for the data that the given function will return. Hence, it will vary when data from a real-world scenario is inserted.

authenticated_user

Function designation and arguments	**Description**
authenticated_user(userid integer)	Retrieves the data associated with the current logged in user

Code

```
BEGIN

RETURN (SELECT json_build_object('id',id,'first_name', first_name, 'last_name',
last_name,'username', username) FROM auth_user WHERE id = userid);

END
```

Returns


```
{
  "id": 1,
  "first_name": "First name (Admin)",
  "last_name": "Last name (Admin)",
  "username": "admin"
}
```

get_all_cuts_from_sample

Function designation and arguments	**Description**
get_all_cuts_from_sample(sampleid integer)	Retrieves all the cuts associated with a given sample

Code

```
DECLARE
result JSON;
BEGIN

	SELECT json_agg(cuts) INTO result
	FROM (SELECT id, purpose, user_id, cut_date FROM cut WHERE sample_id = sampleid)
cuts;
	RETURN result;

END;
```

Returns


```
[
  {
    "id": 19,
    "purpose": "TGF-alpha",
    "user_id": 1,
    "cut_date": "2023-12-15T00:00:00"
  },
  {
    "id": 20,
    "purpose": "TFG-alpha",
    "user_id": 1,
    "cut_date": "2023-12-18T00:00:00"
  }
]
```

sample_information

Function designation and arguments	**Description**
sample_information(sampleid integer)	Retrieves the selected sample information

Code

```
BEGIN
RETURN (SELECT json_agg(json_build_object('id',id, 'user',user_id,'origin',origin))
        FROM sample WHERE id = sampleid);
END;
```

Returns


```
[
  {
    "id": 8,
    "user": 1,
    "origin": "HFVX"
  }
]
```

sample_list_for_user

Function designation and arguments	Description
sample_list_for_user(userid integer)	Retrieved the samples registered by a given user.

Code

```
BEGIN

RETURN (SELECT json_agg(json_build_object('sample',t)) FROM sample t WHERE t.user_id = userid);

END;
```

Returns


```
[
  {
    "sample": {
      "id": 8,
      "user_id": 1,
      "origin": "HFVX",
      "patient_id": 21,
      "tumor_type_id": 2,
      "entry_date": "2023-12-15",
      "temperature_id": 4,
      "container_id": 1,
      "location": null,
      "tissue_type_id": 1
    }
  }
]
```

user_available_roles

Function designation and arguments	Description
user_available_roles(userid integer)	Retrieves the roles available for a user

Code

```
BEGIN
RETURN (SELECT json_agg(json_build_object('role_id', group_id, 'role', name)) FROM
auth_user_groups
INNER JOIN auth_group ON auth_user_groups.group_id = auth_group.id WHERE
auth_user_groups.user_id = userid);
END
```

Returns


```
[
  {
    "role_id": 1,
    "role": "Admin"
  },
  {
    "role_id": 2,
    "role": "Supervisor"
  },
  {
    "role_id": 3,
    "role": "Technician"
  },
  {
    "role_id": 4,
    "role": "Student"
  }
]
```

user_list

Function designation and arguments	**Description**
user_list()	Retrieves a list of the current users registered in the database

Code

```
BEGIN
RETURN (SELECT
json_agg(json_build_object('id',id,'user',json_build_object('username',username,'password',password,
'first_name', first_name, 'last_name', last_name,'email', email,'last_login', last_login,'is_superuser',
is_superuser, 'is_staff', is_staff, 'is_active', is_active, 'date_joined', date_joined)))
        FROM auth_user);
END
```

Returns


```
[
  {
    "id": 1,
    "user": {
      "username": "admin",
      "password":
"pbkdf2_sha256$390000$k4Zp8ru2ts2WysHR3b4ih0$RJCQl0EYCOnWM/u7t4FnKgSOFAVwlYkn
QPcNaBjjUW0=",
      "first_name": "First name (Admin)",
      "last_name": "Last name (Admin)",
      "email": "",
      "last_login": "2023-12-18T13:18:27.591198+00:00",
      "is_superuser": true,
      "is_staff": true,
      "is_active": true,
      "date_joined": "2023-11-21T20:38:13.426796+00:00"
    }
  }
]
```

A.1.2. Postgres Procedures

create_new_patient

Procedure designation and arguments	Description
create_new_patient(name text, dob date, gender text)	Registers a new patient in the database

Code

```
BEGIN
        INSERT INTO patients(patient_name, patient_dob, gender)
        VALUES (name, dob, pgender);

COMMIT;
RAISE INFO 'Patient registered successfully';
END;
```

create_new_user

Procedure designation and arguments	**Description**
create_new_user(username character varying, firstname character varying, lastname character varying, lastname character varying, e-mail character varying, password character varying, roles numeric[])	Registers a new user for the application, with a given list of roles chosen upon creation by the admin

Code

```
DECLARE
	i numeric;
	newuserid numeric;
BEGIN

	INSERT INTO auth_user (is_superuser, is_staff, is_active, date_joined, username, first_name, last_name, email, password) /* introduced password must be hashed */
	VALUES ('false','true','true', CURRENT_TIMESTAMP, username, firstname, lastname, email, password);

/*
 * add roles to user
 */
	newuserid := (SELECT currval(pg_get_serial_sequence('auth_user', 'id')));
	FOREACH i IN ARRAY roles LOOP
		INSERT INTO auth_user_groups (user_id, group_id)
		VALUES (newuserid, i);
		RAISE NOTICE 'Inserted role % in user %', i, newuserid;
	END LOOP;

	COMMIT;
	RAISE INFO 'User created successfully';

END
```

register_new_analysis

Procedure designation and arguments	**Description**
register_new_analysis(userid integer, cutid integer, submitdate text, xlsxfile text)	Submit an analysis file

Code

```
BEGIN

    INSERT INTO analysis (user_id, cut_id, submit_date, result_xlsx_path)
    VALUES (userid, cutid, submitdate, xlsxfile);

COMMIT;
RAISE INFO 'Analysis submition procedure created successfully';
END;
```

register_new_sample

Procedure designation and arguments	**Description**
register_new_sample(userid integer, origin text, patientid integer, tumortypeid integer, tissuetypeid integer, temperatureid integer, containerid integer, location json, entrydate timestamp without timezone)	Register a new sample

Code

```
BEGIN

    INSERT INTO sample (user_id, origin, patient_id, tumor_type_id, tissue_type_id, entry_date, temperature_id, container_id, "location")
    VALUES (userid, origin, patientid, tumortypeid, tissuetypeid, entrydate, temperatureid, containerid, "location");

COMMIT;
RAISE INFO 'Sample registered successfully';
END
```

register_new_cut

Procedure designation and arguments	**Description**
register_new_cut(userid integer, prps, sampleid, cutdate)	Register a new cut from a sample

Code

```
BEGIN

    INSERT INTO cut(user_id, purpose, sample_id, cut_date)
    VALUES (userid, prps, sampleid, cutdate);

COMMIT;
RAISE INFO 'Cut registered successfully';
END;
```

A.2. Annex 2 - API endpoints

Endpoint	Type
login/	POST

Response

```
{
    "success": true,
    "error": "Invalid username or password",
    "is_logged": true
}
```

Endpoint	Type
logout/	POST

Response

```
{
    "success": true
}
```

Endpoint	Type
create_user/	POST

Response

```
{
   "success":true,
   "message":"User created successfully!"
}
```

Endpoint	Type
user_roles/	GET

Response


```
[
   [
      {
         "role_id": 1,
         "role": "Admin"
      },
      {
         "role_id": 2,
         "role": "Supervisor"
      },
      {
         "role_id": 3,
         "role": "Technician"
      },
      {
         "role_id": 4,
         "role": "Student"
      }
   ]
]
```


Endpoint	Type
new_sample/	POST

Response


```
{
  "success":true,
  "message":"Sample created!"
}
```


Endpoint	Type
new_cut/	POST

Response


```
{
  "success":true,
  "message":"Cut created!"
}
```

Endpoint	Type
new_patient/	POST

Response


```
{
  "success":true,
  "message":"Patient registered successfully!"
}
```


Endpoint	Type
patient_list/	GET

Response


```
{
   "success": true,
   "message": "Patients retrieved!",
   "data": [
      [
         {
            "id": 21,
            "patient_name": "John Doe",
            "patient_dob": "2002-04-02",
            "gender": "male
"
         }
      ]
   ]
}
```

Endpoint	Type
tissuetypes/	GET

Response


```
{
   "success": true,
   "message": "Tissues retrieved!",
   "data": [
      [
         {
            "id": 1,
            "tissue_description": "blood"
         }
      ],
      [
         {
            "id": 2,
            "tissue_description": "liver"
         }
      ],
      [
         {
            "id": 3,
            "tissue_description": "connective tissue"
         }
      ],
      [
         {
            "id": 4,
            "tissue_description": "skin"
         }
      ]
   ]
}
```

Endpoint	Type
tumortypes/	GET

Response


```
{
   "success": true,
   "message": "Tumors retrieved!",
   "data": [
      [
         {
            "id": 2,
            "tumor_description": "liver cancer"
         }
      ],
      [
         {
            "id": 3,
            "tumor_description": "pancreatic cancer"
         }
      ],
      [
         {
            "id": 4,
            "tumor_description": "bladder cancer"
         }
      ]
   ]
}
```

Endpoint	Type
temperatures/	GET

Response


```
{
   "success": true,
   "message": "Temperatures retrieved!",
   "data": [
      [
         {
            "id": 4,
            "temperature_desc": "-80ºC"
         }
      ],
      [
         {
            "id": 5,
            "temperature_desc": "4ºC"
         }
      ],
      [
         {
            "id": 6,
            "temperature_desc": "other"
         }
      ]
   ]
}
```

Endpoint	Type
containers/	GET

Response


```
{
   "success": true,
   "message": "Containers retrieved!",
   "data": [
      [
         {
            "id": 1,
            "container_name": "main freezer",
            "container_description": {
               "drawer 1": {
                  "box 1": {
                     "slot 1": 0,
                     "slot 2": 0,
                     "slot 3": 0,
                     "slot 4": 0,
                     "slot 5": 0,
                     "slot 6": 0
                  },
                  "box 2": {
                     "slot 1": 0,
                     "slot 2": 0,
                     "slot 3": 0,
                     "slot 4": 0,
                     "slot 5": 0,
                     "slot 6": 0
                  },
                  "box 3": {
                     "slot 1": 0,
                     "slot 2": 0,
                     "slot 3": 0,
                     "slot 4": 0,
                     "slot 5": 0,
                     "slot 6": 0
                  }
               },
               "drawer 2": {
                  "box 1": {
                     "slot 1": 0,
                     "slot 2": 0,
                     "slot 3": 0,
                     "slot 4": 0,
                     "slot 5": 0,
                     "slot 6": 0
                  },
                  "box 2": {
                     "slot 1": 0,
                     "slot 2": 0,
                     "slot 3": 0,
                     "slot 4": 0,
                     "slot 5": 0,
                     "slot 6": 0
```

```
                },
                "box 3": {
                    "slot 1": 0,
                    "slot 2": 0,
                    "slot 3": 0,
                    "slot 4": 0,
                    "slot 5": 0,
                    "slot 6": 0
                }
            }
        }
    }
    ],
    [
        {
            "id": 2,
            "container_name": "secondary freezer",
            "container_description": {}
        }
    ]
]
}
```


Endpoint	**Type**
samples/	GET

Response


```
{
    "success": true,
    "message": "Samples retrieved successfully",
    "data": [
        [
            {
                "id": 8,
                "user_id": 1,
                "origin": "HFVX",
                "patient_id": 21,
                "tumor_type_id": 2,
                "entry_date": "2023-12-15",
                "temperature_id": 4,
                "container_id": 1,
                "location": null,
                "tissue_type_id": 1
            }
        ]
    ]
}
```

Endpoint	Type
cuts_from_sample/	GET

Response


```
{
   "success": true,
   "message": "Cuts from a given sample retrieved!",
   "data": [
      [
         {
            "id": 19,
            "purpose": "TGF-alpha",
            "user_id": 1,
            "cut_date": "2023-12-15T00:00:00"
         },
         {
            "id": 20,
            "purpose": "TFG-alpha",
            "user_id": 1,
            "cut_date": "2023-12-18T00:00:00"
         }
      ]
   ]
}
```


Endpoint	Type
result_file_upload/	GET

Response


```
{
  "success":true,
  "message":"Analysis result submitted!"
}
```

Endpoint	Type
get_analysis/	GET

Response


```
{
   "success": true,
   "message": "Analysis retrieved successfully!",
   "data": [
      [
         13,
         1,
         19,
         "/home/ricardo/Documents/coley/coleyapp/media/2023-12-
15_user_1_cut_19_filesubmitiontest.xlsx",
         "2023-12-15"
      ]
   ]
}
```


Endpoint	Type
get_users/	GET

Response


```
{
   "success": true,
   "message": "Users retrieved successfully",
   "data": [
      [
         [
            {
               "id": 1,
               "user": {
                  "username": "admin",
                  "password":
"pbkdf2_sha256$390000$k4Zp8ru2ts2WysHR3b4ih0$RJCQl0EYCOnWM/u7t4FnKgSOFAVwlYkn
QPcNaBjjUW0=",
                  "first_name": "First name (Admin)",
                  "last_name": "Last name (Admin)",
                  "email": "",
                  "last_login": "2023-12-18T14:09:06.543737+00:00",
                  "is_superuser": true,
                  "is_staff": true,
                  "is_active": true,
                  "date_joined": "2023-11-21T20:38:13.426796+00:00"
               }
            }
         ]
      ]
   ]
}
```

A.3. Annex 3 - Deployment instructions

- Download *Docker* desktop for you Operating System (https://www.*Docker*.com/products/*Docker*-desktop/);

- Download the *coley* repo to your local machine (https://github.com/ricmonteiro/coley);

- Open a terminal window and go to the repo folder;

- Run the commands:
  - Docker compose build;
  - Docker compose up -d;

- Go to the database container in *Docker* Desktop. On the terminal, go to the *home/coleydata* folder. Run the commands:
  - psql -U postgres;
  - \i init.sql;

- Go to the the *funcionalities* folder;

- Run all the scripts for database functions and procedures creation:
  - \i **script_name.sql**

- Go to http://localhost:3000;

- *coley* is now running on your local machine.