BABEŞ BOLYAI UNIVERSITY CLUJ NAPOCA
FACULTY OF MATHEMATICS AND COMPUTER SCIENCE
SPECIALIZATION COMPUTER SCIENCE IN GERMAN

DIPLOMA THESIS

Online Document Management System for the Romanian National Opera of Cluj-Napoca

Supervisor
Assoc. Prof. Săcărea Christian

Author
Ţurcan Olga

2020

UNIVERSITATEA BABEŞ-BOLYAI CLUJ-NAPOCA
FACULTATEA DE MATEMATICĂ ŞI INFORMATICĂ
SPECIALIZAREA INFORMATICĂ ÎN LIMBA GERMANĂ

LUCRARE DE LICENŢĂ

Online Document Management System for the Romanian National Opera of Cluj-Napoca

Conducător ştiinţific
Conf. Dr. Săcărea Christian

Absolvent
Ţurcan Olga

2020

BABEŞ-BOLYAI UNIVERSITÄT CLUJ-NAPOCA
FAKULTÄT FÜR MATHEMATIK UND INFORMATIK
INFORMATIK IN DEUTSCHER SPRACHE

BACHELORARBEIT

Online Document Management System for the Romanian National Opera of Cluj-Napoca

Betreuer
Dozent Dr. Săcărea Christian

Eingereicht von
Ţurcan Olga

2020

Abstract

The purpose of this thesis is to design and develop an online document management system for the National Romanian Opera of Cluj-Napoca. The system is meant to represent a digital alternative to the current old-fashioned registry system. It automates the process of registering documents, offers the possibility of uploading files and provides features that simplify the process of sharing documents with their recipients. By integrating email notifications, the system improves transparency as each user is constantly aware of the latest status of the documents that an employee has to manage. The application also takes into consideration the sensitivity of the managed data, offering secure solutions and granting access to the data only to authorized users. The main challenge of designing this system was not just to transform the current document management process in a digital one, but rather offer a solution that would meet all requirements in a way that would simplify the process as well as improve its efficiency.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Context

An institution like the National Romanian Opera of Cluj-Napoca keeps track of a lot of documents. Employees from different departments including but not limited to the economic, marketing or human resources departments, send and receive documents as part of their work on a daily basis. The institution should track each single incoming and outgoing document, identify each one by an unique registry number and know whether it has reached its recipient as well as the processing status of the document.

Currently, for achieving this task, the institution maintains a physical register in the form of a catalog. Each document has its corresponding row in the register, which was manually added at the moment of registration. This takes time and effort which could be spared by using an automated solution.

1.2 Motivation

A lot of documents means plenty of information, and working with a huge amount of information stored only on paper is usually a difficult and tedious task. This could be simplified by introducing a digital solution into the document management process, which would considerably improve its efficiency. In addition, manual document administration is prone to human errors, which could be avoided by automating some parts of the process.

1.3 Original contributions

The main contribution of this thesis is designing and developing a document management application which would be used by the National Romanian Opera of Cluj-Napoca. This includes a considerable amount of research related to all stages of a software product lifecycle. As part of the requirements engineering process, discussions with the Opera employees were conducted to get first hand information regarding their expectations for the application. Another big part of the research was related to software architecture and technologies. This was needed because choosing the best approach is relevant when developing an application that would ultimately be used by a real client. Last but not least, the implementation of the app had to take into consideration security, performance and usability, to make interacting with the app both efficient and intuitive.

1.4 Structure of the thesis

The thesis is structured in the following way:

- *Chapter One* presents the **Introduction**, defining the context from which we start designing our application and our motivation.
- *Chapter Two* shows the current **State of the Art**, presenting related work and similar applications.
- *Chapter Three* exposes the **Scientific Problem**, describing theoretical concepts that we used in our research and development.
- *Chapter Four* presents the **Proposed approach**, defining the requirements of the final application and argumenting the technology choices.
- *Chapter Five* describes the **Application**, specifying its use cases, presenting the architecture design, as well as illustrating the implementation results.
- *Chapter Six* defines the **Conclusion** of the entire thesis, as well as specifying our future steps.

Chapter 2

State of the Art

Researching the existence of similar applications has proven that online document management systems are quite popular among both public institutions and private companies. Those who already embraced the digitalization of their document archives affirm that it helped keeping documents more organized and significantly improved search effectiveness. In addition, such systems offer significant advantages in terms of information retrieval, security and lower cost of operations.

A fair amount of research has been conducted on designing and developing web-based distributed administration system services. The article "Optimization for Web-based Online Document Management" [13] describes in detail a "web-based document life-cycle management model" which handles documents from an institute library beginning with their creation till reaching archived state. It focuses on effectiveness and real-time tracking of the document workflow routine, constant availability of the service regardless of network delays, version control and archivation of the documents using a cloud-based solution.

While some institutions choose to develop their own digital document management platforms, others turn to already implemented solutions. A good example of such a solution is Regista - "A complete application for registry and document management" [2]. With more than 500 different clients in 40 romanian counties, it has a total of more than 7 million documents registered through their application. It offers features like automating the handling of the register of incoming and outgoing documents, classification and distribution of documents, all of which should considerably improve time efficiency. In addition, it facilitates information sharing between the document issuer and recipient, simplifying the way requests are made and responses are emitted, which again speeds up the process of solving document related tasks. Their clients count a broad spectrum of institutions from the industry, education, public administration and healthcare systems among many others, which only demonstrates that the digitalization of the document management process is a growing trend in various fields.

Chapter 3

Scientific Problem

3.1 Problem definition

Before defining the proposed solution, we should first illustrate the problem and analyze it in detail. Based on this in-depth analysis, decisions will be made to choose the best fitted approach from a software development perspective.

Software is a vast term generally used to refer to applications, scripts and programs that run on a device. Yet there are different kinds of software, each one having its own challenges and things to be taken into consideration. Designing and implementing an operating system would involve low-level programming and hardware interaction. On the other hand, this becomes absolutely unnecessary if you're developing an eCommerce software product. Indeed, there are practices and patterns that could be applied to all kinds of software, but many are relevant for only one particular field. That is why, as a first step, we should describe the main characteristics of the application that will be built and identify under what type of software it fits.

First of all, our application would involve persistent data. The data about documents which is added by the users should be stored for several years, allowing users to access it, filter it, generate data reports and make conclusions based on them.

As it will be used by the majority of the employees rather than a single person, the application should ensure that many people can access the data concurrently without causing errors. This is especially important for web applications which have thousands or millions of users, but even for a smaller application like ours, concurrency should be handled in a way that would exclude integrity problems and assure data consistency.

Then there is the most important part of the application: its business logic. What data should be stored for each document? What input values are considered valid for each field? Who is allowed to view and edit the data? Which functionalities should the app provide? The answer to all these questions can only be found by analyzing the requirements of the client and turning them into business rules that would serve as the functional engine of the application.

Last but not least, the application needs to provide a GUI[1] that would present the data to the user. The challenge here would be to design the application keeping in mind that usability is as important as functionality. A software system meant to improve the effectiveness of processes in an organization should also offer efficiency in its usage. To bring users satisfaction from interacting with the app, the GUI must provide all the desired features, yet keep things simple and intuitive.

The identified characteristics are also described in [4] as aspects that define Enterprise Applications as a software type. Enterprise Applications are software solutions that provide business logic to handle processes of an organization in a way that would improve efficiency and productivity. Some people think of a large system when hearing the term "Enterprise Application". However, it is important to keep in mind that not all enterprise applications are large and the value they bring to the organization does not depend on their size. What is also important to understand is that even an application for a small organization, with a few users and relatively simple logic, needs to be built in such a way that it will be maintainable and extensible. It could happen that in a couple of years the organization would like to add other processes to the app, thus we have to make sure it allows adding new functionality without having to reimplement already existing logic. This can only be achieved by taking into account well-known design patterns qualified for our purpose.

3.2 Theoretical foundations

In this section of the thesis we will analyze the theoretical foundations that will further help us design and build the application keeping in mind all the challenges that were previously mentioned.

3.2.1 Why a Web distributed system?

A first and important decision we have to take is whether to develop a desktop application or a web-based one. Indeed, a traditional desktop app doesn't need constant Internet connection in order to be used. It could do all of its processing on the locally installed app and only sync data with the database once in a while when it has Internet connection. However, this seems to be its biggest and only advantage. Web applications, on the other hand, have a wide range of advantages. To start with, a web application runs on a web browser, which means there is no need to install additional software on the computer. This makes it accessible from a numerous devices and operating systems. In addition, it gives users freedom to access the app from anywhere, anytime, via any device with an Internet connection, allowing them to solve urgent

[1]Graphical User Interface

tasks without a trip to the office. App maintenance is also less complicated - once an update is made on the host server, users can access it without manual updates on their computer. These are the main reasons why building a web application was chosen over a classical desktop one.

3.2.2 Layer architecture

A web application architecture is based on a *client-server* model, which is a distributed application system that divides tasks between the server - the provider of resources, and the client, which communicates with the server via requests made over the network. As stated in [10], the simplest shape a client-server architecture can take is a *two-tier* architecture. This originated back in the 90s long before the rise of the popularity of web applications. Back then, it was used in database applications so that the server layer was responsible for data storage and retrieval while the client handled data manipulation and presentation. Nowadays however, applications are much more complex, which calls for the necessity of adding an additional layer responsible for the business logic. This is known as the *three-tier architecture*, which consists of three primary layers: Presentation Layer, Application Layer and Database Layer (See Figure 3.1).

Each of the layers has its own set of responsibilities. The Presentation Layer, in our case a web-browser GUI, handles the display of information as well as user input, such as mouse clicks, keyboard hits and http requests. The Database Layer is primarily responsible for storing persistent data. The middle-tier, called the Application Layer, is the means of communication between the other two layers and is fully responsible for all the business logic of the application. This includes validation of the data that comes from the UI, performing calculations based on user input and execution of different business flows depending on requests received from the Presentation Layer.

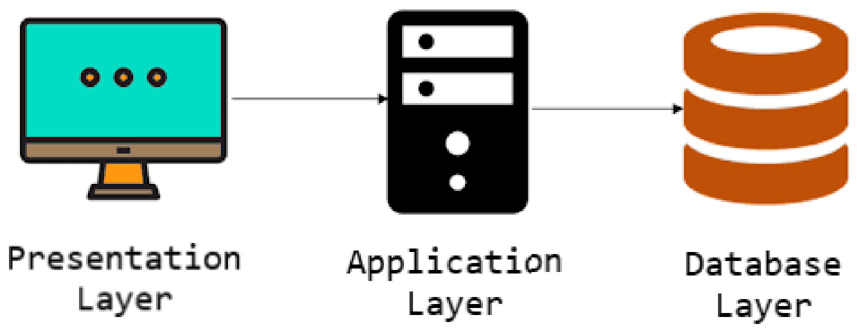


Figure 3.1: The three-tier architecture

The three-tier architecture represents the simplest layering scheme that could be successfully applied to an enterprise application. However, if the application is quite complex, it would indeed need additional mediating layers. This raises the question of how to further split the layers and how the dependencies between layers should be managed. The Software Engineering field has seen quite a few architecture models which serve exactly this purpose.

The Clean Architecture model seen in Figure 3.2 is, as described in [7], an attempt to integrate and combine several architectures having as end goal the separation of concerns, which results in modular systems composed of fairly independent components.

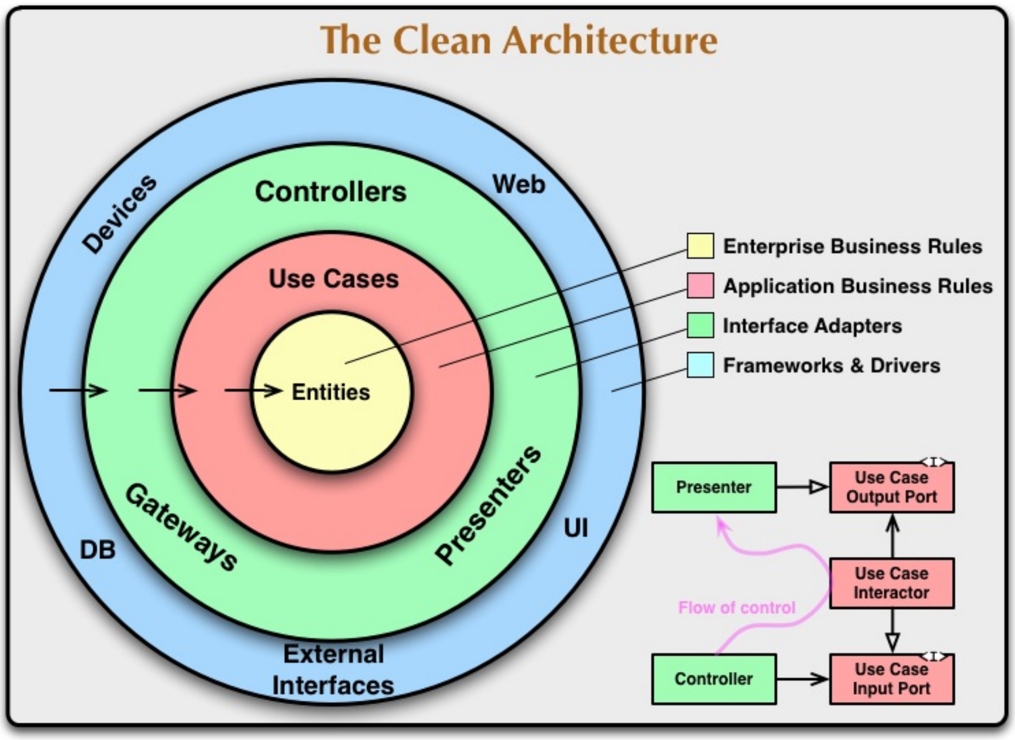


Figure 3.2: The clean architecture [2]

The circles in Figure 3.2 show different components of a software application:

- **Entities** represent the business objects of the application.
- **Use cases** encapsulate the business logic and is not affected by changes to the database or the UI.
- **Interface adapters** act as converters between data formats and handle communication between use cases and entities with the database or the web GUI.
- **Frameworks and drivers** is a layer composed mainly of tools - be it the database or the web framework.

[2]Image source: [7]

Of course, a clean architecture would imply a low coupling of layers, that would permit the components to remain independent of each other. The business layer should be independent of the database, allowing it to be replaced with a different data source as long as the model objects keep the same structure. At the same time, it should also be independent of the UI. Thus, the UI could change, be redesigned or reimplemented using another framework without even touching the business layer. This separation also contributes to making the business layer highly testable on its own, without involving the actual database or the UI.

To achieve this low coupling, an important rule must be kept in mind when defining dependencies between modules and layers. This is described in [7] as "The Dependency Rule", which states that "Source code dependencies must point only inward". To clarify this, let's look again at Figure 3.2. According to the rule, components from an inner circle should not know about components from an outer circle. From a code perspective, this means that any software entity should use only those software entities that were declared in an inner circle. As an example, a controller, which belongs to the Interface Adapters group, cannot use an entity defined in the Web Presentation layer, which belongs to Frameworks and Drivers. Instead, it is the responsibility of the Web Presentation layer to make a request to the controller, which will then invoke the Application layer and so on.

Building dependencies according to this rule will result in a layered design with a structure that resembles a *directed acyclic graph* (DAG), meaning that it has no cycles. This will ensure that the components are independent of one another, which will make them easier to develop, maintain and extend.

3.2.3 Client-server communication: Representational State Transfer

In the previous subsection, we have talked about the separation of client and server so that each of them could be individually developed and changed without affecting the other part. However, the only way to achieve this modularity is to establish a communication standard between the server and the client.

This could be accomplished by using Representational State Transfer (REST), which is an architectural style for providing standards for communication between web systems. It was designed as an alternative to Simple Object Access Protocol (SOAP), which has been, for a long time, the mainly used architectural style for building web services. However, due to its simplicity, lightweightness and scalability, REST has quickly gained popularity and is nowadays the most popular approach chosen for web service development.

The decoupling of the client and server is one of the 6 constraints of the REST architectural style. The communication is made via the defined Application Programming Interface (API) using requests and responses. Clients will send requests using the HTTP protocol and wait for responses. The server will receive those requests and do whatever the client needs, i.e. query the database or perform some other operations. When the REST service will finish processing the request, it will send back a response to the client (See Figure 3.3). It is worth mentioning that the communication is always initiated by the client. The server will never share information about its resources on its own: it will wait and only send responses as a reaction to incoming requests.

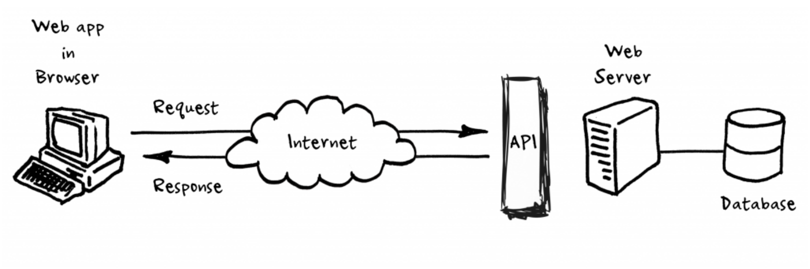


Figure 3.3: The client-server communication using a REST API [3]

What operation will be triggered on the server is determined by the information the client has to provide while making the request:

- The **endpoint**, which is the Uniform Resource Identifier (URI) for the requested resource.
- The **HTTP method** that tells the server which operation needs to be performed on the resource. (See Table 3.1)

GET	read a resource
POST	create a new resource
PUT	update an existing resource
DELETE	delete a resource

Table 3.1: The basic HTTP verbs

Another important REST constraint says that the client-server communication is **stateless**, meaning that each single client request should contain all the information that the server needs for processing. It does not store a history of the requests and thus cannot use some previous context. Session state management is therefore entirely the client's responsibility. This is one of the main reasons why RESTful web services are so lightweight, scalable and easy to implement.

3.2.4 Usability in the context of Enterprise Application Design

We had already mentioned in the previous section that usability is an important ingredient for a successful application. Now is the time to analyze in more detail what usability means and how it should be achieved in our application.

[3]Image source: [3]

In regard to web applications and software applications in general, usability is defined as the ease at which an average person can use the software to benefit from its functionality. This includes a wide range of characteristics: how intuitive the UI is, how memorable the workflows are, how time-efficient each operation is, as well as how much satisfaction it brings to the user. In the context of enterprise applications all these seem especially important, as tools made to improve efficiency become useless if they are not intuitive to use.

It may seem at first glance that achieving usability is only a matter of design, a correct placement of widgets on a page and a tasteful combination of colors. However, usability is much more than that. Some decisions made to improve usability can even influence the backend implementation of the system. Consider for example the application which is the subject of this thesis. As one of its primary functions will be tracking all documents in the organization, it will have to display all the existing data, so the UI will most probably contain a table presenting the data. What will start as a couple of documents to be displayed will quickly grow to be a large collection of data registered into the system. Yet displaying all the data on one single page will be hard to interpret for the human eye. A best practice for avoiding this is using pagination to split the documents into multiple pages, thus limiting the number of rows displayed on a single page [5]. This is exactly the case when for achieving a task which will improve usability, we will have to also change the backend implementation to support handling paginated requests.

Another way of dealing with a lot of data as an alternative to pagination is loading the next chunk of data when the user scrolls to the bottom of the screen. This, however, makes the user unaware of the total amount of data and provides less control over navigation between data rows. This makes it more suitable for apps with an endless stream of data that the user should consume, while pagination is more suitable for enterprise applications like ours.

Another thing to keep in mind is the responsiveness of the application. After the client sends a request, receiving a response will occur with a certain delay called **response time** - the time it will take the server to process the request. The amount of time needed will depend on the nature of the request, as well as the latency - the minimum delay that is present even when no processing from the server is required. While there are a number of things to be considered when implementing the application that could improve overall performance and minimize response time, you could also do something to improve the experience on the UI side. Improve **responsiveness**, which represents the time in which the system acknowledges the request. Responsiveness and response time are the same if during a request processing the system is waiting without any indicator to the user. However, adding a loading spinner will improve responsiveness as the user will be aware that the request is being processed. This will improve the overall experience, even if the response time will be the same.

Chapter 4

Proposed approach

4.1 Feature specification

Now that we have analyzed the technical requirements and identified some of the challenges that could be encountered, we can continue with defining the solution for the described problem. Our approach would involve designing and developing a web application, keeping in mind its advantages over desktop applications, which we described in the previous section. The app would include two major parts: a web service developed in Java using the Spring Boot Framework and a client app built using the ReactJS Framework. The web service will define a REST API which will be used for communication between the client and the server. It will also handle the business logic and manage the persistence of data, which will be achieved by using a MySQL database. The client app will be the one that will run in the user's browser. As REST is stateless by definition, the client app will have to manage the session state. This will be achieved with the help of Redux - a predictable state container for JavaScript apps.

One important thing to keep in mind is that the application will be used for managing real documents of a public institution, which could potentially involve sensitive data. To secure our app, we will use features offered by Spring Security to enable JSON Web Token Authentication and manage access control.

4.2 Technologies used

In the following subsections of this chapter we will describe each of the chosen technologies in detail, with a strong focus on features that were decisive in choosing it over alternative technologies from the same branch.

4.2.1 The Spring Framework

The Spring Framework is an application framework for the Java platform which offers a wide range of features for modern Java-based applications. According to a survey from 2018, Spring is the most popular Web Framework worldwide in the JVM ecosystem (See Figure 4.1). It provides a large set of functionality and tools, making it suitable for a big variety of applications. In addition, its flexibility and focus on ease of use allow developers to focus on the application business logic rather than configurations and boilerplate code, thus increasing velocity, productivity and development speed.

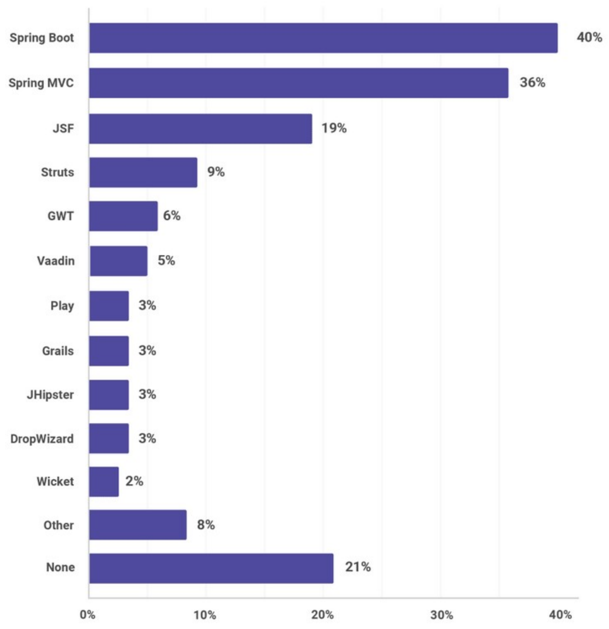


Figure 4.1: The most popular Java Web Frameworks according to the JVM Ecosystem report 2018 [1]

One of the most important features offered by Spring is *Inversion of Control* (IoC), achieved by *Dependency Injection* (DI). According to this principle, objects are not responsible for creating their dependencies. Instead, they should be "injected" into the object through arguments passed to the constructor, setter methods or factory methods. In contrast to the classical way where each object is locating and instantiating all of its dependencies, the DI approach has several advantages. The most important one is achieving loose coupling, making it easier to interchange distinct implementations and maintain program modularity.

To achieve this, the Spring Framework uses an *IoC container*. It is responsible for the instantiation, configuration and managing of the *beans* - objects that will be used by the application (See Figure 4.2). In order to know how each bean should be created and which

[1]Image source: https://snyk.io/blog/jvm-ecosystem-report-2018-platform-application

dependencies injected, the IoC container needs additional configuration information. This is provided as metadata in form of XML, Java Configuration classes or Java annotations. In our project, we are going to use Java annotations as it is the most easy and straightforward way to configure beans.

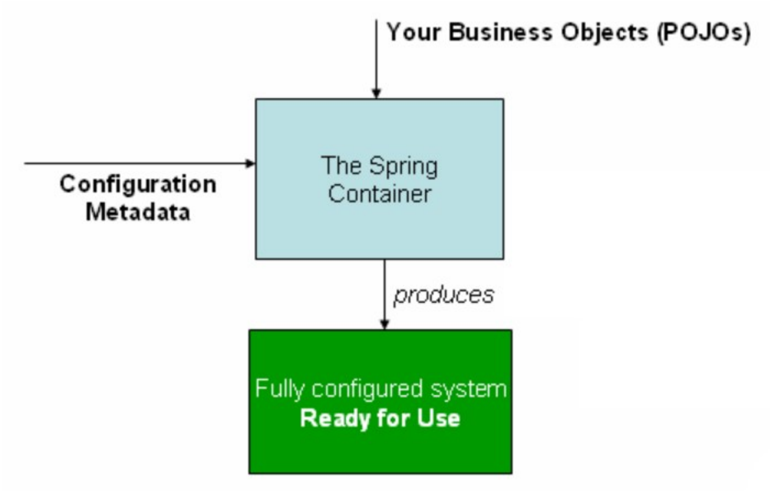


Figure 4.2: The Spring IoC Container [2]

Another advantage of Spring is that it uses well-known design patterns which are implemented under the hood:

- All beans created by Spring are by default **Singletons**. This means that the IoC container creates only one instance for a bean and caches it so that it could then be used in multiple different places.
- Spring uses the **Factory Method Pattern** for bean creation. The `BeanFactory` class defines a list of `getBean()` methods, all of each are factory methods that could be used to create a bean with the desired configuration. This functionality is extended by the `ApplicationContext` class, which adds the possibility of creating beans using external configurations like XML files or Java annotations.
- Spring uses the **Proxy Pattern** to control access to its beans. One typical example where this is necessary is when using transactions. If we annotate a method as `@Transactional`, Spring guarantees its atomic execution and transactional consistency (hence the name). This is only possible by intercepting the call through a proxy object that wraps our bean and would control the method execution.

[2]Image source: [1]

- Spring uses the **Template Method Pattern** to minimize the amount of boilerplate code that is repeated again and again throughout projects. The `JdbcTemplate` class is a perfect example, providing an easy and intuitive way to execute database queries using predefined templates.

4.2.2 Spring Boot

The Spring Framework provides a wide range of features for building enterprise and web applications, which makes it suitable for almost any type of use case. However, with the increasing amount of Spring features embedded into an application, the complexity of developing it also increases. Configurations grow in size and get difficult to maintain.

Spring Boot was designed to solve exactly this problem. The module, while maintaining all of the power of its parent, significantly reduces the amount of configurations the developer has to make. It eliminates the need to write those parts of the code that always tend to be the same across multiple projects, like configuring a data source or a transaction manager. It also tends to simplify the deployment process, offering an embedded server which is ready to run as an alternative to manually setting up a deployment server.

As for managing external dependencies, Spring Boot comes with the new concept of *starters* - dependency descriptors which bundle together multiple dependencies that are usually used together for achieving a task. Our project is a good example that illustrates the usage of this feature. Developing a web service requires multiple external modules: we would use features from different Spring modules, but then we would also need libraries that would take care of parameter validation, JSON conversions and server embedding. Spring Boot assumes this set of libraries would be required in most web services and wraps them all in a `spring-boot-starter-web` dependency. By including this dependency in our project, it transitively imports all dependencies it contains, saving us the effort of including each one manually.

All these features result in an effortless development of web applications, which has made Spring Boot a widely popular choice for developing RESTful web services.

4.2.3 Spring Security and JWT

The Spring Security framework provides *authentication* and *authorization* support for Spring-based applications. We will try to explain this concepts on the example of our application. One of its main features is storing documents in an online archive. Authentication will provide users access to the document archive, so that they can see the entire list of registered documents and basic details like the person who submitted them or the registration date. On the other hand, authorization will give a user permission to perform more specific actions, like download the file content associated with the document or mark it as resolved.

Traditionally, Spring Security operates by creating a session and storing cookies on the client. However, this contradicts the principles of REST architecture.

To avoid this, we are going to use JSON Web Tokens which handle authentication in a stateless way and can be easily integrated with Spring Security. As stated in [9], a JWT is just a string consisting of three encoded parts which are separated by a dot:

- The **header** is used to indicate the hashing algorithm used to sign the token. In our application we use the *HS512* - a Hash-based Message Authentication Code algorithm which uses the SHA-512 cryptographic hash function and a secret cryptographic key to generate the code.
- The **payload** is used to store the user's *claims* - information about the user like authorization data, roles and permissions.
- The **signature** is the final section of the token used for additional verifications.

The way JWT secures the application is simple and intuitive (4.3). When the user enters his or her credentials in the login form, they are sent to the server for verification. If the provided credentials are correct, the server queries the database for information about the user, generates a JWT and stores the user information in it. The server has to also specify an expiration time for the token. After being signed and hashed, the JWT is sent to the client as a response to the authentication request. As the server is stateless, it is the client's responsibility to store the token until it expires.

Afterwards, every request from the client to the server has to contain the JWT token specified in an Authorization header. The server will extract the token, validate the signature and get the user data from the JWT payload. Based on that data, the server will decide if the user is or is not authorized to perform the desired request.

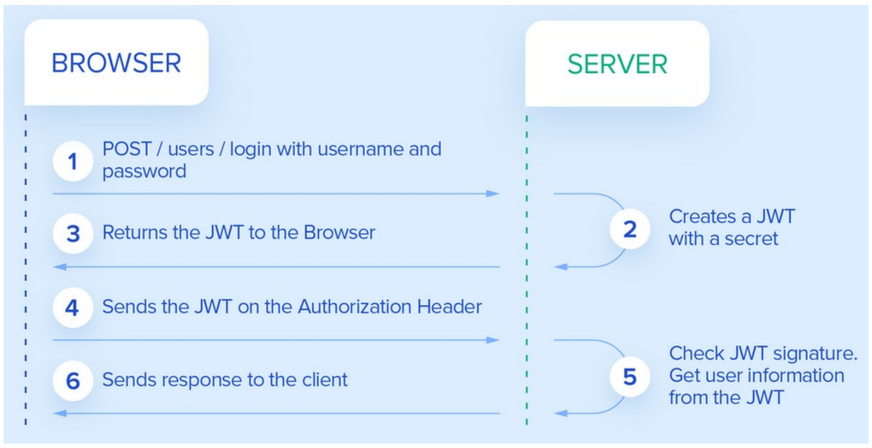

BROWSER
SERVER
1 POST / users / login with username and password
2 Creates a JWT with a secret
3 Returns the JWT to the Browser
4 Sends the JWT on the Authorization Header
5 Check JWT signature. Get user information from the JWT
6 Sends response to the client

Figure 4.3: Creation and usage of Json Web Tokens [3]

4.2.4 MySQL Database

An important decision we had to make was whether to choose a *relational* or a *non-relational* database for persisting our data. Relational databases have been successfully used in enterprise applications for decades, however the NoSQL movement of the recent years has questioned whether the relational model is the best approach in all situations. Indeed, relational databases weren't designed to be used in an Object Oriented context. There are numerous techniques of mapping domain objects into database tables and columns. It is especially cumbersome when concepts like inheritance or user-defined data types are involved. The normalization of the obtained database model usually tends to split the information about an entity across multiple tables. We then need to make use of the foreign keys and perform joins to reconstruct our entity, which are costly operations. This is the main reason why, in the Big Data Era, non-relational databases are preferred for storing huge amounts of data. A NoSQL database will store information as JSON documents instead of rows spread across multiple tables, which will provide a considerable advantage in scalability and performance.

The drawback is that NoSQL databases don't treat operations as transactions. This makes the non-relational model a less fitted choice if you have operations that depend on each other and all of them must either execute successfully or not at all (See [6]). In our application, ensuring consistency of the persisted data is key. We want to enforce data integrity by maintaining the *ACID* principles (Atomicity, Consistency, Isolation, Durability). That is the reason why a relational database seems the most fitted choice for our application.

From the vast amount of relational database technologies available, we chose to use **MySQL**, which is an open-source relational database management system. To compensate for the speed - the field where non-relational databases are performing better than relational ones, we will research techniques to improve MySQL query performance described in [11]. One basic technique for increasing the speed of operations is using *indexes* - data structures that boost data management efficiency and become increasingly important as the size of managed data grows. There are also different types of indexes to consider. Tree-based indexes are more suited for fields which will be accessed with range queries or for sorting the query results. Thus, it would make sense to create a *B-Tree* index on fields based on which we will sort documents in our application - like, for instance, the document registry number. Hash-based indexes, on the other hand, perform better for direct access. In our database, a *Hash* index would be useful for the email column of the user table, as the user retrieval at login time will be performed based on email instead of the id.

[3] Image source: [8]

4.2.5 Spring JDBC

Next thing to consider is how the web service is going to connect to the database. Spring boot offers support for multiple Object Relational Mapping (ORM) frameworks, such as Hibernate and JPA. This are sometimes preferred because the developer doesn't have to think about how the data is stored. The mapping between the object-oriented world and the database is handled entirely by the framework. However, this might not always result in fully optimized queries. This is why, for our application, we chose the **Spring JBDC** Database Access. With this approach, the Spring Framework is still fully responsible for managing low-level details like opening and closing the connection, preparing and executing the statement and handling transactions. What we get instead is having full control over SQL statements and query parameters. The `NamedParameterJdbcTemplate` also offers the possibility to provide names for each parameter instead of the classical unintuitive placeholders (`?`), which results in a more readable and less error-prone code.

4.2.6 ReactJS Framework

With the rise of the popularity of web applications, front-end technologies have also become a rather popular topic. In contrast to using plain old HTML and JavaScript, modern UI frameworks strive to offer solutions for achieving a responsive and interactive user interface with minimum effort.

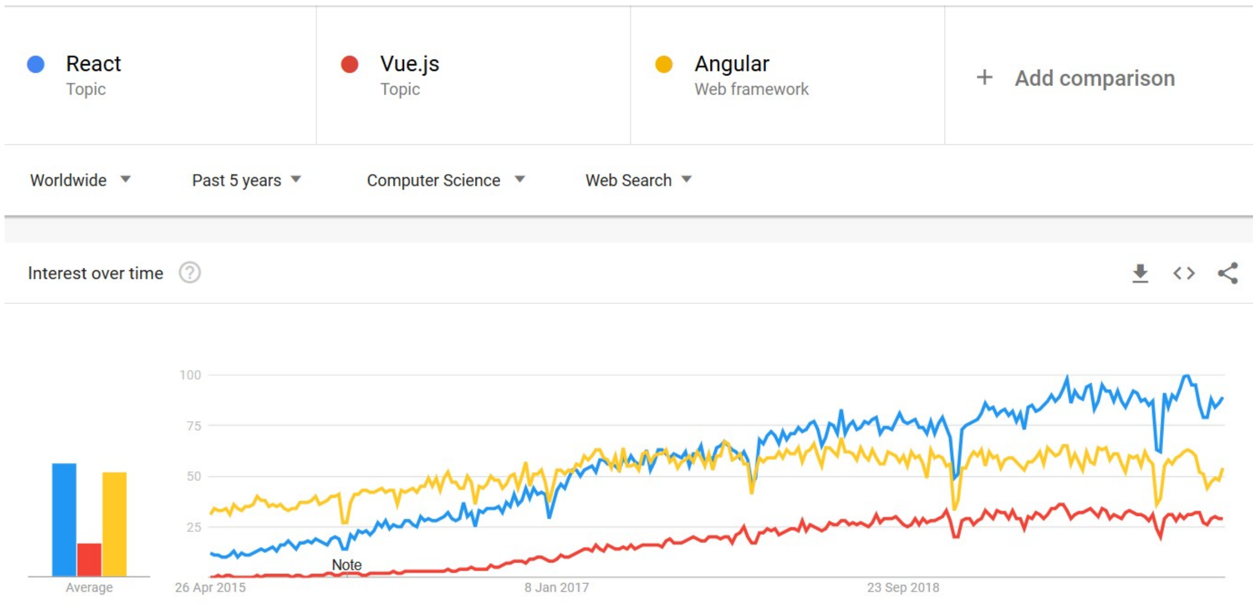


Figure 4.4: Popularity of JavaScript front-end frameworks in the last 5 years according to Google Trends

React, **Vue.js** and **Angular** are 3 popular JavaScript frameworks that have been constantly present in front-end technologies tops in the last years. According to Google Trends (Figure 4.4), the popularity of Angular has been falling since 2015, while the interest for both React and Vue has been rising. However, React has a greater popularity, since Vue is a younger technology with a smaller community.

All of them tend to offer similar features like component-based architecture that results in reusable components, easy syntax and high performance. However, for our application we chose to use React as our UI framework. In comparison to Angular, it is more lightweight and thus suitable for systems that aren't too complex. As for Vue.js, it is a progressive fast-changing framework which is steadily gaining popularity, but React has the advantage of being on the market for a longer time, which guarantees reliability of the resulting software.

4.2.7 Redux

According to the official documentation, Redux is *"a predictable state container for JavaScript Apps"*. To put this in simple words, it is a library for managing application state, that can be used standalone or in connection with libraries like React. As described in [12], state is an overall broad concept that could refer to anything that is present and can be modified in the browser. Entity state is data retrieved from the backend - like the list of all documents or the object describing the user that is currently logged in to the application. On the other hand, view state is not stored in the backend and is only related to the UI, like the flag that tells if a dropdown is open or not.

When speaking about state management, we mean initializing, modifying and deleting state. For very small applications, this could be easily handled only by using React's local state. However, as the size and complexity of the application grows, so does the effort needed to manage its state. To make things easier, we used Redux, which handles state management in a predictable and elegant way using only 3 main building parts (See Figure 4.5):

- The **store** is the most important part, which holds the entire application state as a global object. It is a singleton, meaning that there is only one Redux store per application.

- **Actions** are events that send data from the application to the store. To execute an action in Redux is called to *dispatch*, and it should be done when you want to change the state. Dispatching an action requires providing an *action type* and an additional *payload* - information being sent to the store. Once an action is dispatched, it will pass through all reducers.

- **Reducers** are pure functions that are responsible for updating the state. They take as input the current application state and a dispatched action, and, based on the action type and payload, return the next state. It is worth noting that reducers return a new state object without mutating the current state, which always results in a predictable outcome.

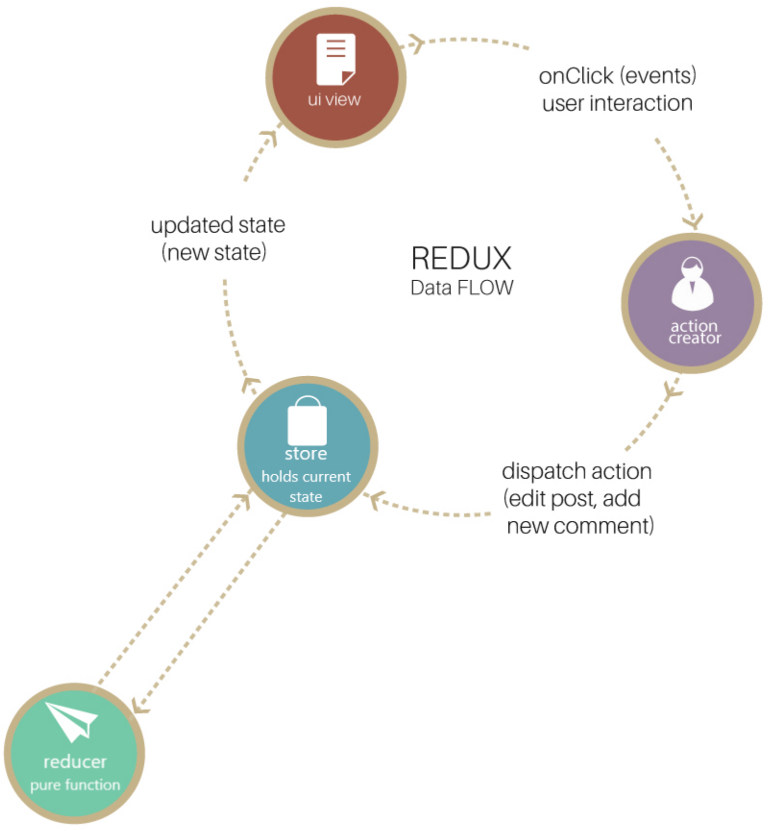


Figure 4.5: Redux Data Flow [4]

[4]Image source: https://cloud.netlifyusercontent.com/assets/344dbf88-fdf9-42bb-adb4-46f01eedd629/44e53712-498d-400f-8967-8ed0dbdb90a1/new-redux-data-flow-large-opt.png

Chapter 5

Application

5.1 Use cases

Before diving into the application architecture and implementation details, let us first describe the use cases to get a complete overview of the functionality the app has to provide (See Figure 5.1). The first and the most important use case is registering documents into the system. This would include automatically generating a registration number for each document. It is an important part of automating the current process, in which for registering a new document, the employee has to contact the registry administrator and ask for him to provide an available registration number. For the registration to be complete, the user has to specify a destination for the document. In most cases, the document would be internal, meaning that the destination would be other employees of the institution. In this case, the document should be sent to the specified recipients, which get an email notification and could then view the received documents in their account. However, some documents could have an external destination, like another institution or organization. In that case, the destination is specified so the document could be easily tracked in the future. Analogically, the issuer could specify an external source of the document, if it has one.

The user could upload an electronic version of the document, either immediately after the registration, or in any point of time in the future. Additionally, the upload feature is available for all recipients of a document.

The logic behind the document management process relies mainly on two actions: resolving and archiving a document. Basically, when a user receives a document, it means that he or she is expected to perform a task related to it. It could be an action as trivial as acknowledging that the document was received, or something more complex like reading and approving the document, uploading an edited version of the document or sending it further to other users. Regardless of this, the **resolve** action is meant to finalize the interaction between a user and a document.

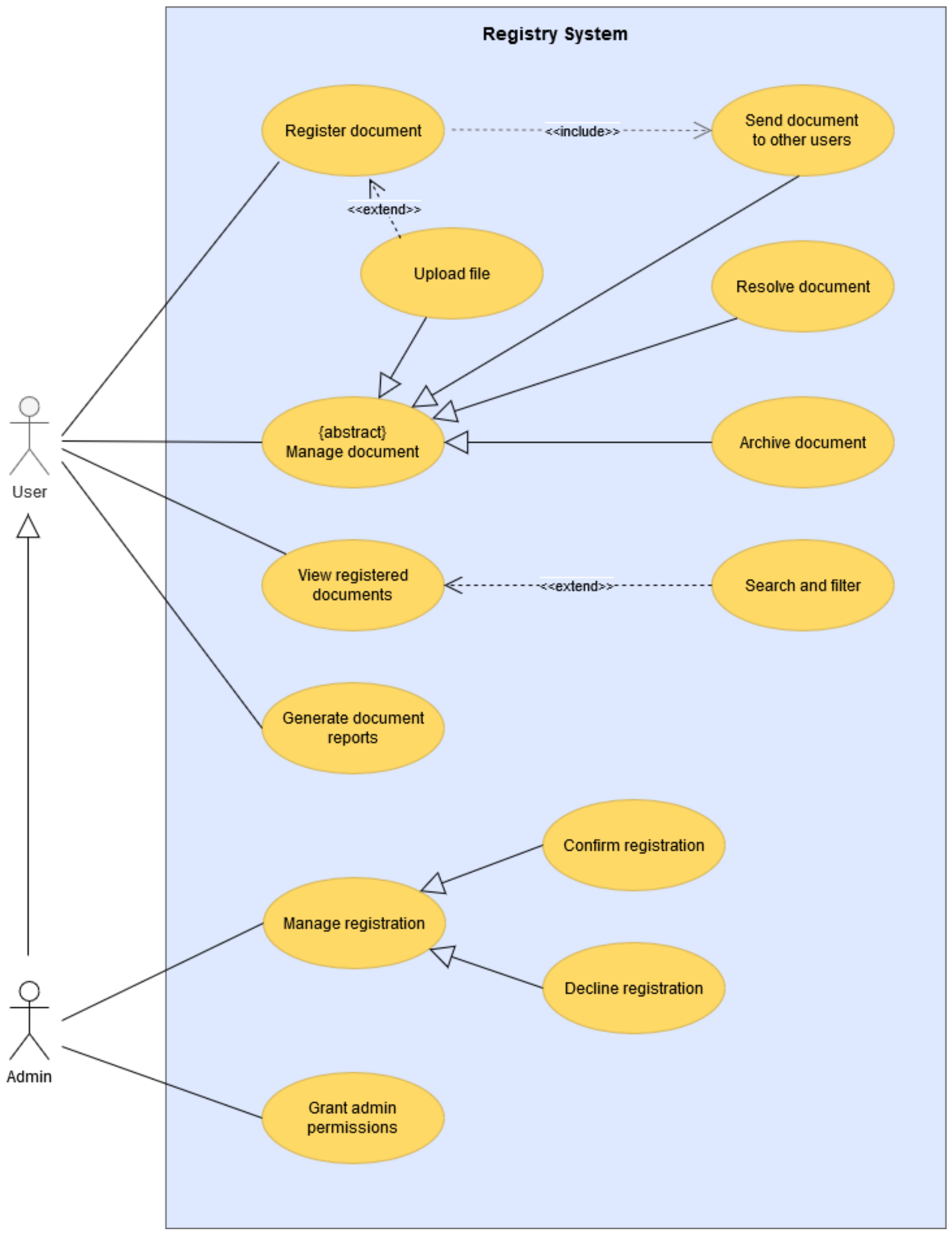


Figure 5.1: Registry System Use Cases

In simpler terms, marking a document as resolved is telling its issuer that all necessary actions from the receiver were taken.

The **archive** action, on the other hand, is meant to complete the document lifecycle and can only be performed by the document issuer. After a document is marked as archived, it could no longer get resolved or sent to other users. Resolving and archiving a document are closely related actions. After all, it seems rather logical that the user should only archive a document after it has been resolved by its receiver. However, the logic gets more complicated in case multiple receivers are specified. It could be that all of them have to take different actions on the document. It could also be that getting the document resolved by only one of

Name	Resolve document
Short description	A user marks a document as resolved .
Precondition	The user is logged in to the system. The user is a receiver of the document.
Postcondition	The document shows as resolved by the user in the app. The document issuer receives an email notification.
Error situations	The server could not perform the operation.
System state in the event of an error	Document is not marked as resolved by the user.
Actors	User
Trigger	All document related actions needed from the user were performed.
Standard process	(1) User selects the document. (2) User introduces a resolve message (optional). (3) User confirms the resolve action.
Alternative process	(3') User cancels the resolve action.

Table 5.1: Use case description for *Resolve document*

them would be sufficient, if, for example, they are employees of the same department. In this case, getting a document resolved by a receiver is not a required constraint for the document to get archived. Similarly, when a document is resolved, it does not necessarily imply that it should immediately get archived. That is why we have not constrained the ability to archive a document based on its resolved status, but rather let the issuer do it when he or she deems it necessary. The *Resolve document* and *Archive document* use cases are described in detail in Table 5.1 and 5.2.

Name	Archive document
Short description	A user marks a document as archived.
Precondition	The user is logged in to the system. The user is the issuer of the document.
Postcondition	The document shows as archived in the app. The document cannot be send to other users or resolved.
Error situations	The server could not perform the operation.
System state in the event of an error	Document is not marked as archived by the user.
Actors	User
Trigger	The document issuer considers that all actions related to the document are complete.
Standard process	(1) User selects the document. (2) User introduces an archiving message (optional). (3) User confirms the archive action.
Alternative process	(3') User cancels the archive action.

Table 5.2: Use case description for *Archive document*

Aside from managing owned and received documents, users should also have access to the entire register containing all documents cataloged into the system. This should include the ability to search for a specific document or filter the list based on different criteria. Another important feature would be generating a report in PDF or XLS format that would either include all documents or only a certain subset matching the criteria specified by the user. The reports could then be printed and physically archived in the institution and should therefore replace the register that is currently filled out by hand.

Having a separate account for each user would imply the necessity of a registration process. As it is an internal app, a flow in which the user provides credentials and the account gets automatically created would not be sufficient. We have to also verify that the person requesting the account is an employee of the institution to which the application belongs. To achieve this, we would add the **Admin** role to our system. Admins would inherit all document management rights that the simple users have, but will also have special permissions that would allow them to manage the registration of new users.

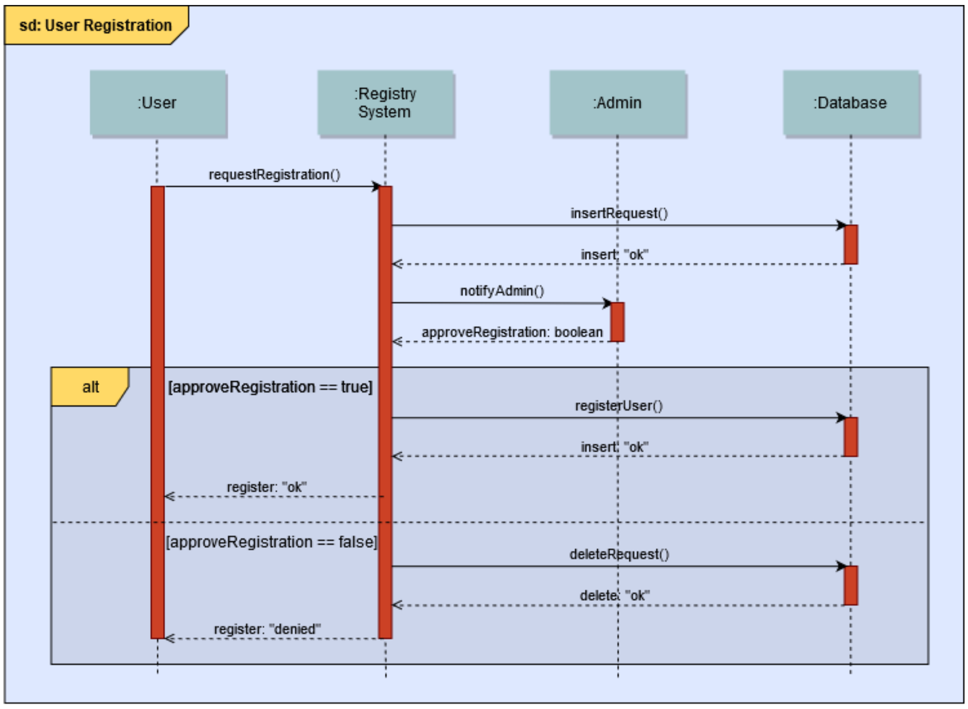


Figure 5.2: Sequence diagram for the *User Registration* flow

As seen in Figure 5.2, the registration process begins with a user sending a request to the system to register a new account. The system then inserts the request into the database and sends an email notification to each of the Admins telling them that a new registration has been requested. The admins could then either confirm or decline the registration. In case of confirmation, the registration is completed within the database and the user is notified that the registration was successful. He or she can, from that moment, access the newly created account using the login credentials provided at the time of registration. In case an admin denies the registration, all information about the user is deleted from the database and an email notification is sent to inform the user that his request was denied.

5.2 Architecture

The Registry System Application has a layered architecture (See Figure 5.3) which closely follows the principles of layered architecture design described in 3.2.2.

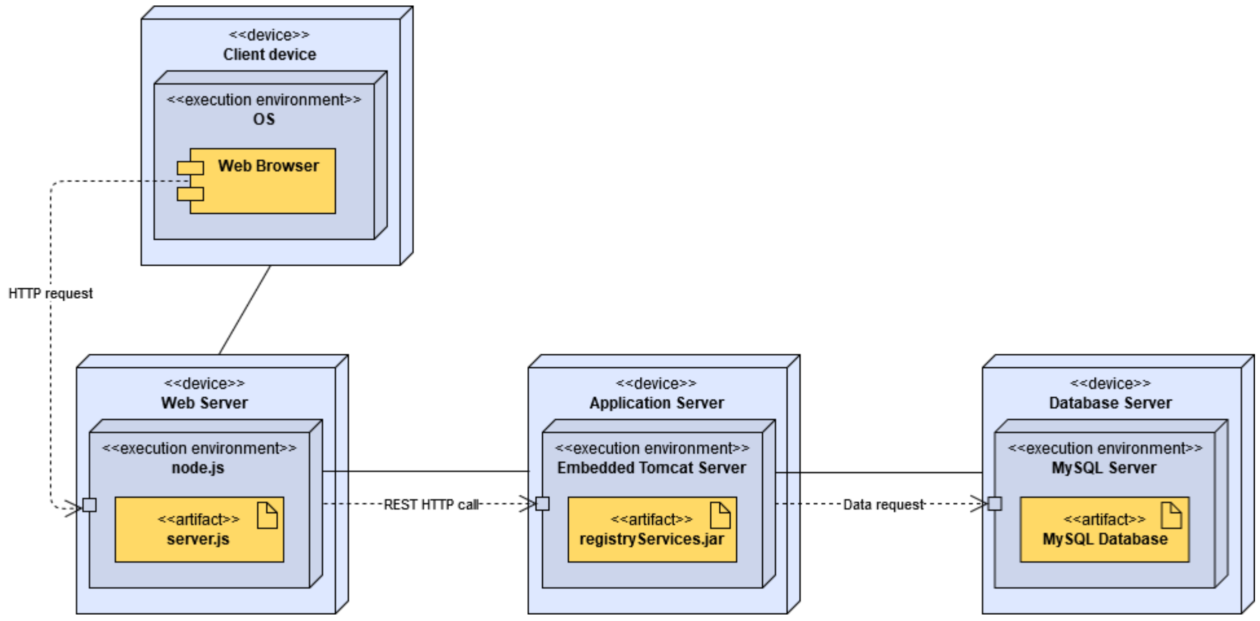


Figure 5.3: Registry System Deployment Diagram

- The **persistence layer** is presented by a MySQL database. It connects to the application layer which queries the database for inserting, retrieving and updating data.
- The **application layer** is a Spring Boot Application, which defines the domain, encapsulates the business logic and exposes a REST API which is used by the presentation layer to manipulate the data.
- The **presentation layer** is a React JS application which connects to the Spring Boot backend through HTTP requests. The Web GUI can be accessed by the user on any device from a Browser.

In the following subsections, we are going to analyze the architecture of each component in detail.

5.2.1 Database

Searching for a database schema that would support all required use cases, we came up with the design presented in Figure 5.4. First of all, we have the **user** table, which, as the name suggests, will store information about the users of our app, like their name and department within the institution. An additional boolean field will be used to determine whether the user registration has been confirmed. The user table will also store authentication related data - the credentials the user provided at the moment of registration. One thing to mention here is that the password is encoded prior to storing it in the database using the `BCryptPasswordEncoder` provided by the Spring Security Framework. This assures that user passwords are not exposed even to persons that have direct access to the database.

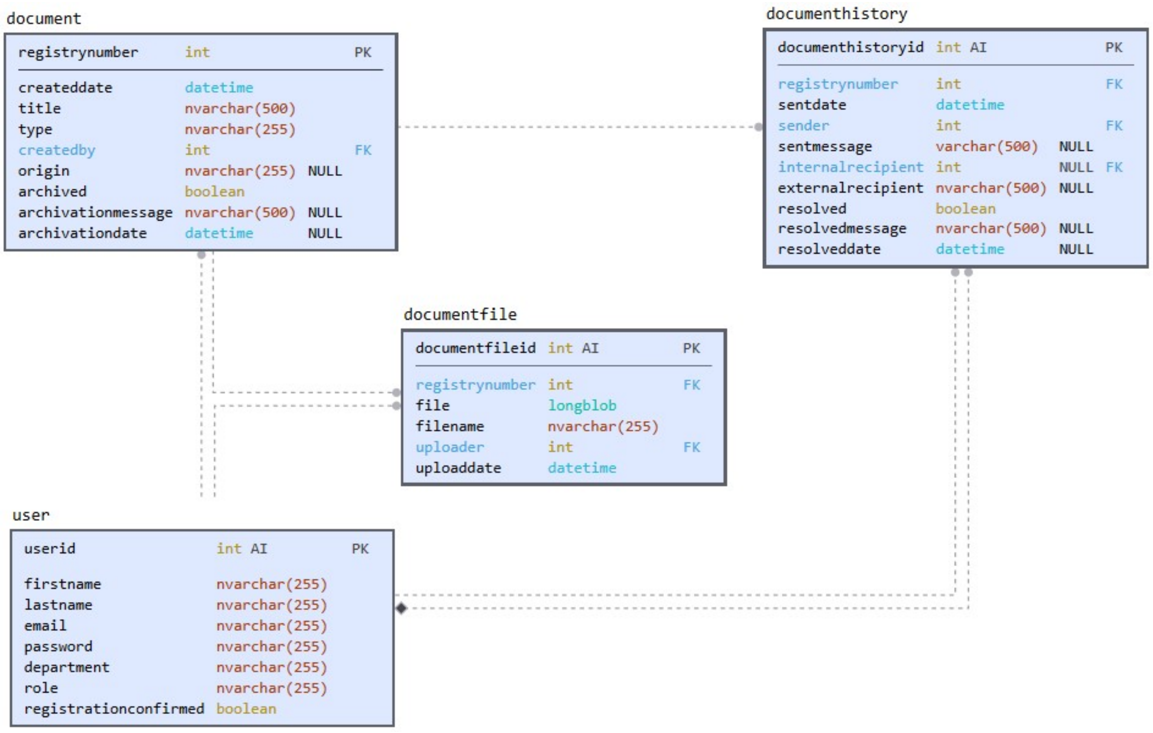


Figure 5.4: Registry System Database Diagram

The **document** table will store all registered documents, having the registration number as primary key. It is also the only primary key of this database that has no auto increment option set on it. Instead, we generate the next available number from code, each time a new document is registered into the system. This allows us more control over the way the number is generated. We already mentioned when describing use cases that a document can either be internal, or have external origin or destination, meaning that it is coming or should be sent to another institution. This information would be stored in the type field. One important thing to mention here is that a document cannot have external source and destination at the same time as it would technically not be related to the institution's registry anymore. That is why there are only three valid document types - internal, origin external and destination external. The table will also store a flag that will show the document's state as archived/not archived.

As an internal document could have multiple other users as recipients, another table is needed to store this m:n relationship. We decided to call this table **documenthistory**, as it reflects not only the recipients of a document, but also the *resolved* status of the document set by a specific user. In case a document has external destination, only one entry for this document will be created in the documenthistory table, with the destination value stored in the *externalrecipient* column.

Last but not least, we have a separate table for storing the uploaded files that represent the electronic version of the document. We only support uploading a single file per document now, but this has been kept as a separate table so that the design could support uploading multiple files if the feature would be desired in the future.

5.2.2 Spring Boot Application

As we already mentioned, the backend logic of our Registry System is represented by a Spring Boot Application. It is divided in 4 independent modules which interact to provide the full functionality of the app.

- The **Model** is the first module in this list. It defines the application domain, that is the classes that represent our use case, like `User` and `Document`.
- The **Persistence** module is the one that is responsible for interacting with the database. It defines and implements *Repository* interfaces which retrieve, insert and update data using the Spring `NamedParameterJdbcTemplate` class. It also defines methods for mapping domain objects into JDBC parameters, as well as JDBC result sets into domain objects.
- The **Business** module is the biggest and most important one, because it encapsulates the business logic of the entire application. It defines and implements *Service* interfaces which provide the necessary logic for the desired functionality. It is the module that is handling the server-side validation of user input, translation of the domain objects into Data Transfer Objects (DTOs), as well as mapping exceptions that occur throughout the program to user defined business exceptions. This is also the module that handles all the security related logic.
- The **Web** module contains *Controller* classes that define the REST API which will be used by the client. It maps a method to a specific URL (endpoint) that would be accessed in an HTTP request and a HTTP Response Status that should be returned.

For exception handling, we define a `@RestControllerAdvice` annotated class that will globally handle all exceptions that are thrown to the Controller layer (See 5.1). The way this works is that in this class we define a method for each exception type that could occur. This method then gets called each time an exception of that type is thrown. Just imagine that five different methods can throw an `EntityNotFoundException`. Instead of handling all those five cases separately, we only define a single method which returns a meaningful error message and the 404 HTTP status. In this way we centralize the place where errors are handled which provides better readability and significantly reduces the amount of repetitive boilerplate code.

```
@RestControllerAdvice
public class ControllerExceptionHandler {

    @ExceptionHandler(EntityNotFoundException.class)
    @ResponseStatus(HttpStatus.NOT_FOUND)
    public ErrorResponse handleException(EntityNotFoundException e) {
        return new ErrorResponse(e.getMessage(), e.getCause());
    }

    //other @ExceptionHandler methods follow
}
```

Listing 5.1: Simplified example of global exception handling class

With the same goal of reducing boilerplate code in mind, we have used the **Lombok** Java library which provides great shortcuts in form of annotations for most used Java code snippets. As an example, we used the `@Data` annotation on our model classes, which generates all the repetitive code that is normally associated with simple POJOs (Plain Old Java Objects) and beans: getters for all fields, setters for all non-final fields, `toString`, `equals` and `hashCode` implementations involving the fields of the class, and a constructor that initializes all required arguments. Another feature that we used from Lombok is the `@Slf4j` annotation, which automatically generates a Logger associated to the class it is used on.

An important part of the application is the security logic implemented using the features provided by the Spring Security framework. An unauthenticated user would have access only to the registration and login pages. All other endpoints would need a valid JWT token to be accessed. In some cases, it matters not only *if* the user is authenticated, but also *who* the user is. A good example is the `archiveDocument` endpoint which should only be called by the document issuer. All the other users are unauthorized to perform this method on that specific document. In that case, in addition to the JWT token validation, we need to manually verify the user's identity. Of course, after integrating the backend with the frontend app, a user won't have any way of triggering an unauthorized endpoint from the UI. For instance, we won't display *Archive* buttons for documents issued by another users as the one that is currently logged in. However, the Spring Boot app is a standalone application that should not necessarily be tied to the frontend. Its endpoints could be triggered by directly accessing the URL they are associated with. This is why it is important to perform server-side validations such as verifying the identity of a user that is performing a request.

5.2.3 ReactJS Application

The presentation layer of our Registry System consists of a ReactJS application with Redux as a state container. The whole application state is kept into a global *store*, making the app behavior predictable and easy to understand. REST HTTP requests sent to the backend are performed by the Redux *actions*, keeping data fetching code separate from the rest of the logic. The retrieved data is then passed to the *reducers*, which update the application state. It is worth mentioning that with this approach, each time a reducer updates some object from the application state that is showing on the currently displayed page, the page automatically rerenders, meaning that it updates the displayed information. This assures that the data displayed is always up to date without the necessity of a refresh or other interaction from the client.

For building the UI, we used the **React Bootstrap** frontend library. It integrates the classical Bootstrap stylesheets into ready to use react components that follow the latest UI trends. Bootstrap was designed for developing responsive and mobile-first websites. We took advantage of this feature to customize our UI so that it would be accessible from all kinds of devices, from mobile to tablets to PCs. In the following subsection, while providing screenshots of the app, we will sometimes include both the PC and mobile version to demonstrate how the app adapts to different screen sizes.

5.3 Feature implementation

Now that we have described the use cases and architecture, we can finally present the resulting application. As we mentioned before, a user would have access to the app by logging in to his or her account (See Figure 5.5)

After logging in, the user is redirected to the main page of the application, where all registered documents can be viewed (See Figure 5.7). The documents are presented in form of a table, which shows the document registration number, issuer, date of registration, title, recipients and state (archived or not archived). Designing this table, we had in mind the idea that it should give an overview about each document and include only the most relevant information so that the table won't get cluttered and difficult to read. That is why, for internal issuers and

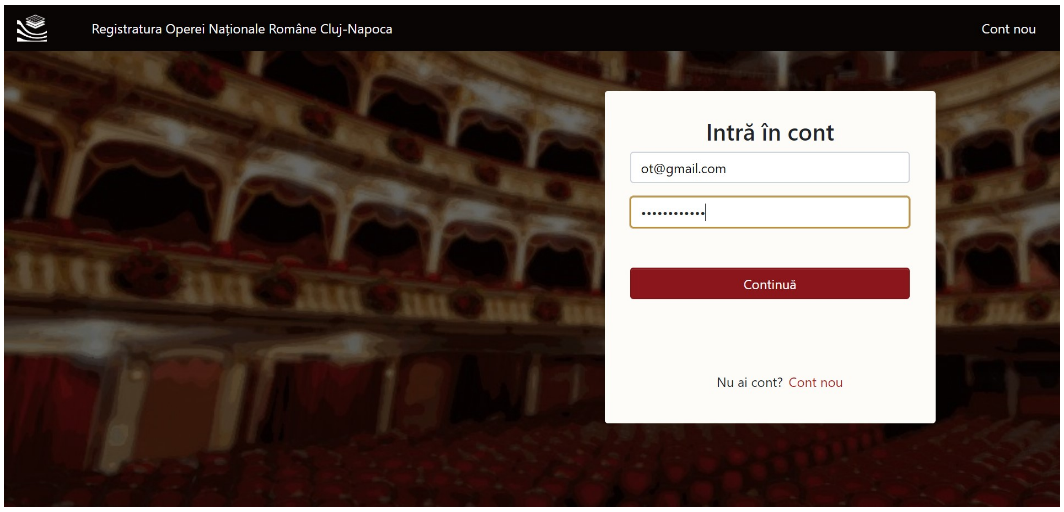
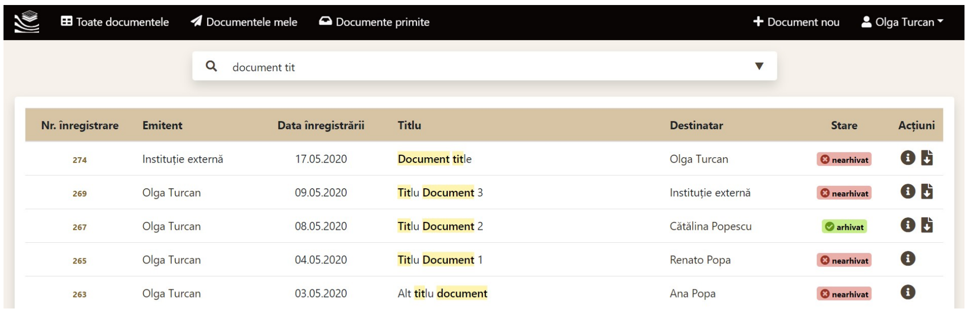


Figure 5.5: Login page

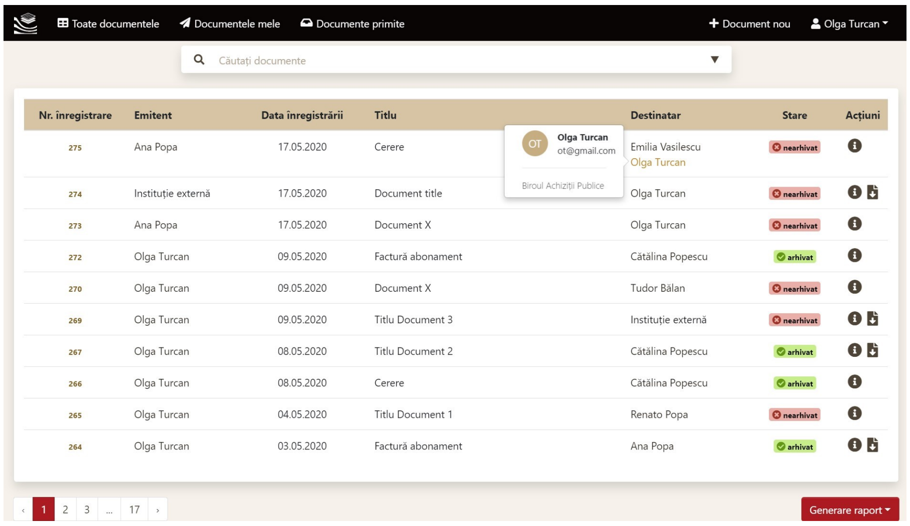
recipients, only the full name is presented in the table, but if you hover on the name, additional information like the employee's email and department are displayed as a popup. Additional details like the date the document was sent to each recipient, whether and when it was resolved and the archiving date are displayed in a modal dialog that opens when the user clicks the information button. Also, for documents that were issued or received by the logged user and have a file previously uploaded, a button for downloading the attached file will be displayed. And of course, since the total number of documents will grow more and more, the request that retrieves the document list from the backend is paginated, meaning that we load and display only 10 documents at a time.

The page also includes a search bar which you can use to search documents by registration number or title. If you type in some text, all documents containing the search term in the title will be shown and the exact place where the search term appears will be highlighted. If you enter more than a single word, the system will search for documents which contain all those terms in the title, but without taking into consideration the order they appear in (See Figure 5.6).



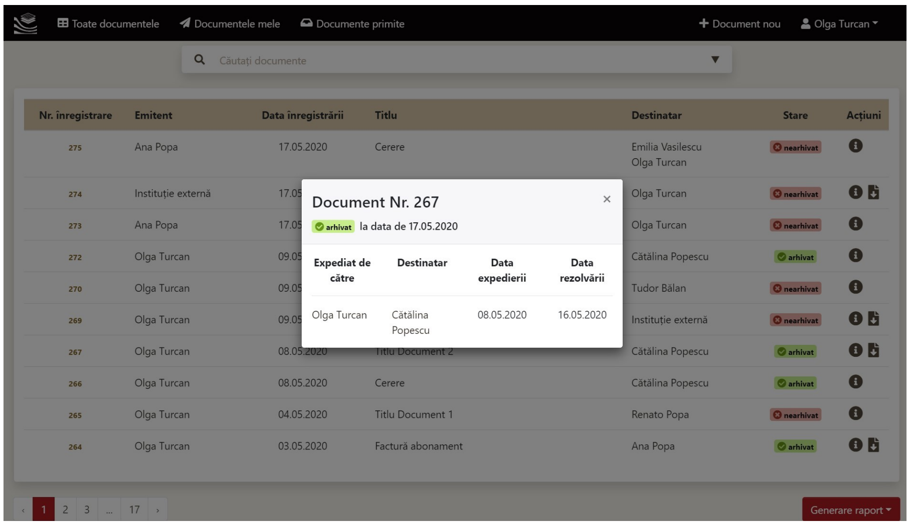
Nr. înregistrare	Emitent	Data înregistrării	Titlu	Destinatar	Stare	Acțiuni
274	Instituție externă	17.05.2020	Document title	Olga Turcan	nearhivat	
269	Olga Turcan	09.05.2020	Titlu Document 3	Instituție externă	nearhivat	
267	Olga Turcan	08.05.2020	Titlu Document 2	Cătălina Popescu	arhivat	
265	Olga Turcan	04.05.2020	Titlu Document 1	Renato Popa	nearhivat	
263	Olga Turcan	03.05.2020	Alt titlu document	Ana Popa	nearhivat	

Figure 5.6: Document search



Nr. înregistrare	Emitent	Data înregistrării	Titlu	Destinatar	Stare	Acțiuni
275	Ana Popa	17.05.2020	Cerere	Emilia Vasilescu Olga Turcan	nearhivat	
274	Instituție externă	17.05.2020	Document title	Olga Turcan	nearhivat	
273	Ana Popa	17.05.2020	Document X	Olga Turcan	nearhivat	
272	Olga Turcan	09.05.2020	Factură abonament	Cătălina Popescu	arhivat	
270	Olga Turcan	09.05.2020	Document X	Tudor Bălan	nearhivat	
269	Olga Turcan	09.05.2020	Titlu Document 3	Instituție externă	nearhivat	
267	Olga Turcan	08.05.2020	Titlu Document 2	Cătălina Popescu	arhivat	
266	Olga Turcan	08.05.2020	Cerere	Cătălina Popescu	arhivat	
265	Olga Turcan	04.05.2020	Titlu Document 1	Renato Popa	nearhivat	
264	Olga Turcan	03.05.2020	Factură abonament	Ana Popa	arhivat	

(a) Overview



Document Nr. 267

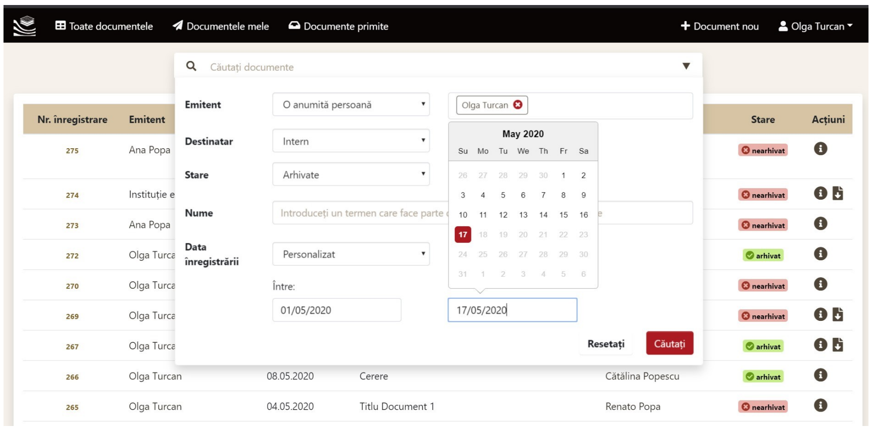
arhivat la data de 17.05.2020

Expediat de către	Destinatar	Data expedierii	Data rezolvării
Olga Turcan	Cătălina Popescu	08.05.2020	16.05.2020

(b) Details

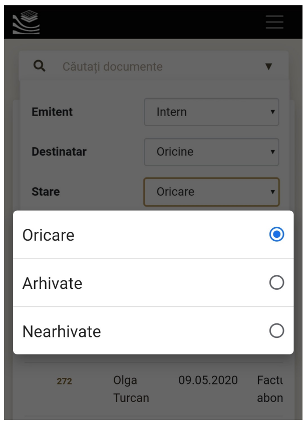
Figure 5.7: Viewing all registered documents

The system also provides the functionality of an advanced search, accessible by expanding the search bar dropdown (See Figure 5.8). A wide range of criteria could be used to filter the search result. You could for example search for documents issued only by external institutions, or only by internal users of the app. You could narrow the search even more, searching for documents issued by a specific list of users. The same options are available for specifying recipients. You could choose to search for documents based on their state, in case, for example, you want to see all documents that are not archived yet. There is also the option to show only those documents that were registered within a specific timeframe. The system offers some predefined options, like yesterday, in the last 7 days and in the last month, but you could also enter whatever date interval you like. Finally, you could still specify a string if you wish to search by title but narrow your results using additional criteria.



Emitent: O anumită persoană — Olga Turcan
Destinatar: Intern
Stare: Arhivate
Nume: Introduceți un termen care face parte
Data înregistrării: Personalizat
Între: 01/05/2020 — 17/05/2020

(a) On desktop



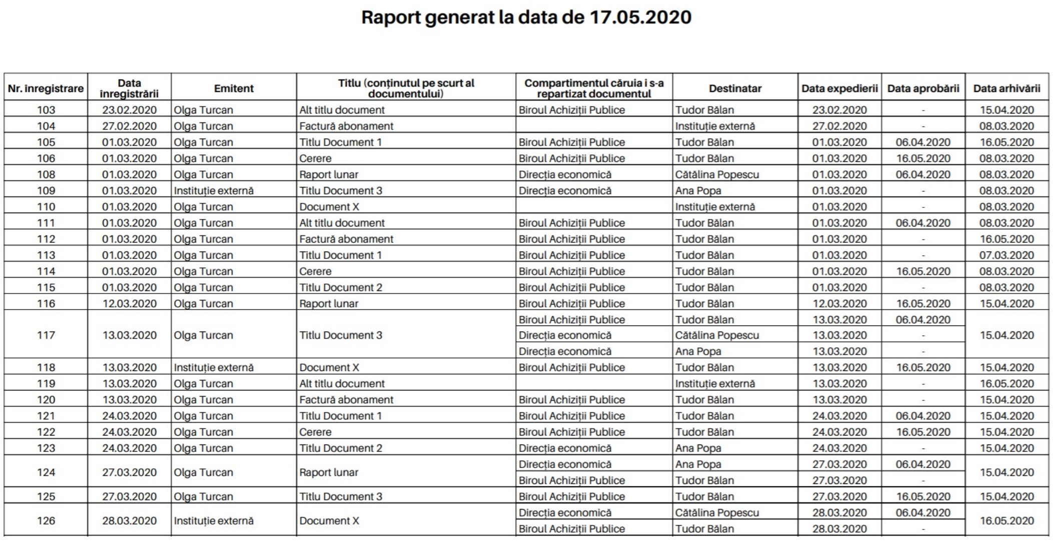
Emitent: Intern
Destinatar: Oricine
Stare: Oricare

[✓ Oricare]
Arhivate
Nearhivate

(b) On mobile

Figure 5.8: Advanced document search

The app provides the functionality of generating a report in PDF or XLS format, containing all documents that meet the criteria specified in the search. The report could be printed so that a physical copy of the document list would be saved. The table from the report differs slightly from the one in the app, containing some columns that present additional information (See Figure 5.9). These were added so the report would include the whole information about a document, but also to resemble the format of the manual register that is currently used to keep track of the documents within the institution.

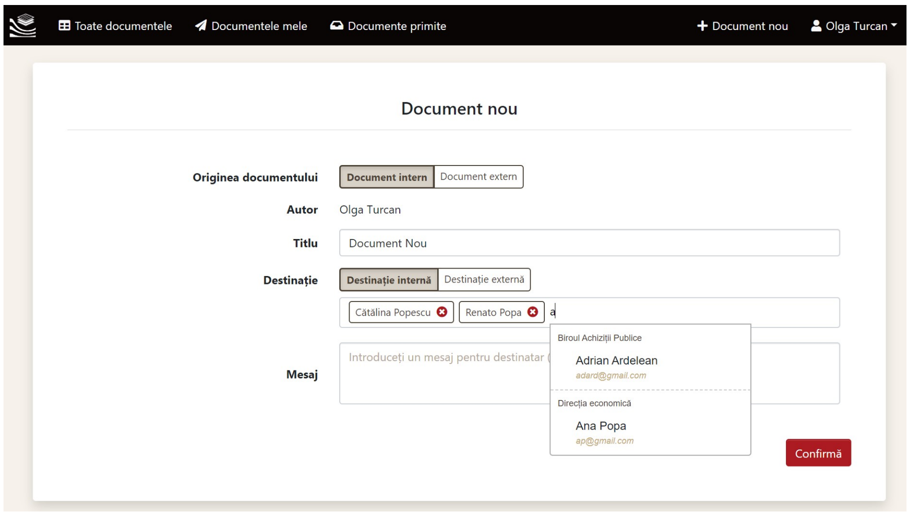


Raport generat la data de 17.05.2020

Nr. înregistrare	Data înregistrării	Emitent	Titlu (conținutul pe scurt al documentului)	Compartimentul căruia i s-a repartizat documentul	Destinatar	Data expedierii	Data aprobării	Data arhivării
103	23.02.2020	Olga Turcan	Alt titlu document	Biroul Achiziții Publice	Tudor Bălan	23.02.2020	-	15.04.2020
104	27.02.2020	Olga Turcan	Factură abonament		Instituție externă	27.02.2020	-	08.03.2020
105	01.03.2020	Olga Turcan	Titlu Document 1	Biroul Achiziții Publice	Tudor Bălan	01.03.2020	06.04.2020	16.05.2020
106	01.03.2020	Olga Turcan	Cerere	Biroul Achiziții Publice	Tudor Bălan	01.03.2020	16.05.2020	08.03.2020
108	01.03.2020	Olga Turcan	Raport lunar	Direcția economică	Cătălina Popescu	01.03.2020	06.04.2020	08.03.2020
109	01.03.2020	Instituție externă	Titlu Document 3	Direcția economică	Ana Popa	01.03.2020	-	08.03.2020
110	01.03.2020	Olga Turcan	Document X		Instituție externă	01.03.2020	-	08.03.2020
111	01.03.2020	Olga Turcan	Alt titlu document	Biroul Achiziții Publice	Tudor Bălan	01.03.2020	06.04.2020	08.03.2020
112	01.03.2020	Olga Turcan	Factură abonament	Biroul Achiziții Publice	Tudor Bălan	01.03.2020	-	16.05.2020
113	01.03.2020	Olga Turcan	Titlu Document 1	Biroul Achiziții Publice	Tudor Bălan	01.03.2020	-	07.03.2020
114	01.03.2020	Olga Turcan	Cerere	Biroul Achiziții Publice	Tudor Bălan	01.03.2020	16.05.2020	08.03.2020
115	01.03.2020	Olga Turcan	Titlu Document 2	Biroul Achiziții Publice	Tudor Bălan	01.03.2020	-	08.03.2020
116	12.03.2020	Olga Turcan	Raport lunar	Biroul Achiziții Publice	Tudor Bălan	12.03.2020	16.05.2020	15.04.2020
117	13.03.2020	Olga Turcan	Titlu Document 3	Biroul Achiziții Publice	Tudor Bălan	13.03.2020	06.04.2020	15.04.2020
				Direcția economică	Cătălina Popescu	13.03.2020	-	
				Direcția economică	Ana Popa	13.03.2020	-	
118	13.03.2020	Instituție externă	Document X	Biroul Achiziții Publice	Tudor Bălan	13.03.2020	16.05.2020	15.04.2020
119	13.03.2020	Olga Turcan	Alt titlu document		Instituție externă	13.03.2020	-	16.05.2020
120	13.03.2020	Olga Turcan	Factură abonament	Biroul Achiziții Publice	Tudor Bălan	13.03.2020	-	15.04.2020
121	24.03.2020	Olga Turcan	Titlu Document 1	Biroul Achiziții Publice	Tudor Bălan	24.03.2020	06.04.2020	15.04.2020
122	24.03.2020	Olga Turcan	Cerere	Biroul Achiziții Publice	Tudor Bălan	24.03.2020	16.05.2020	15.04.2020
123	24.03.2020	Olga Turcan	Titlu Document 2	Direcția economică	Ana Popa	24.03.2020	-	15.04.2020
124	27.03.2020	Olga Turcan	Raport lunar	Direcția economică	Ana Popa	27.03.2020	06.04.2020	15.04.2020
				Biroul Achiziții Publice	Tudor Bălan	27.03.2020	-	
125	27.03.2020	Olga Turcan	Titlu Document 3	Biroul Achiziții Publice	Tudor Bălan	27.03.2020	16.05.2020	15.04.2020
126	28.03.2020	Instituție externă	Document X	Direcția economică	Cătălina Popescu	28.03.2020	06.04.2020	16.05.2020
				Biroul Achiziții Publice	Tudor Bălan	28.03.2020	-	

Figure 5.9: Generated report in PDF format

Another important page is the one that allows registering new documents into the system. The user is prompted to complete input fields related to the document. All of them are mandatory, except the message which is an optional field for the user to send a comment to the document receivers. The origin and destination types of the document could be changed from internal to external using toggle buttons. When specifying recipients, search suggestions would appear based on the string the user types (See Figure 5.10). The suggestions would show all users with a matching name or surname, grouped by department. After the registration is confirmed, the user is asked to upload a file for the newly created document. If this step is skipped, the file can still be uploaded later.



Document nou

Originea documentului: Document intern | Document extern
Autor: Olga Turcan
Titlu: Document Nou
Destinație: Destinație internă | Destinație externă
Cătălina Popescu, Renato Popa, a

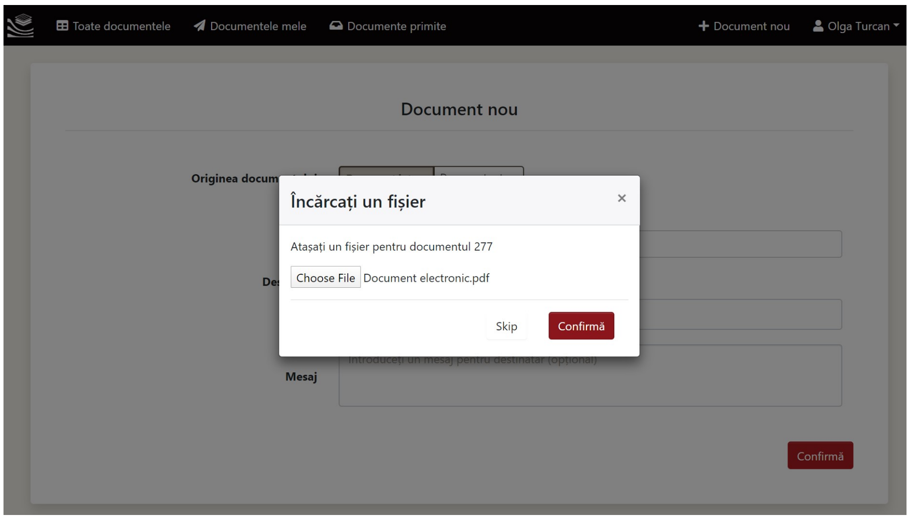
Biroul Achiziții Publice
Adrian Ardelean
adard@gmail.com

Direcția economică
Ana Popa
ap@gmail.com

Mesaj: Introduceți un mesaj pentru destinatar (

Confirmă

(a) Completing input fields



Document nou

Încărcați un fișier

Atașați un fișier pentru documentul 277

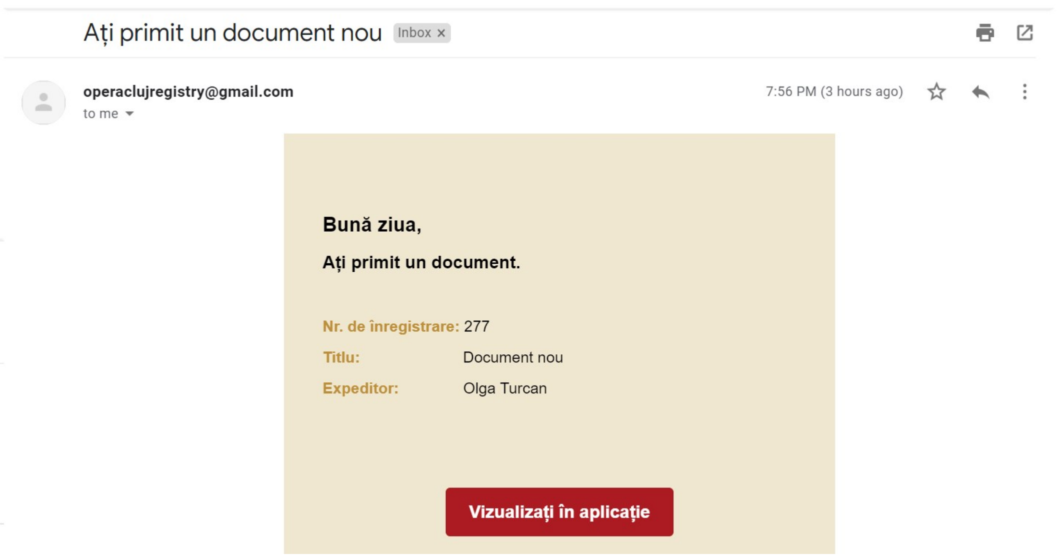
Choose File Document electronic.pdf

Skip Confirmă

(b) Uploading a file

Figure 5.10: The *Create new document* flow

For each registered document with internal destination, an email notification is sent to each recipient to inform that a new document has been sent to them (See Figure 5.11). To view the document, the user can click the *View in app* button from the email, which will redirect him to the *Received Documents* page. Another option is to manually navigate to it.



Ați primit un document nou

operaclujregistry@gmail.com
to me

7:56 PM (3 hours ago)

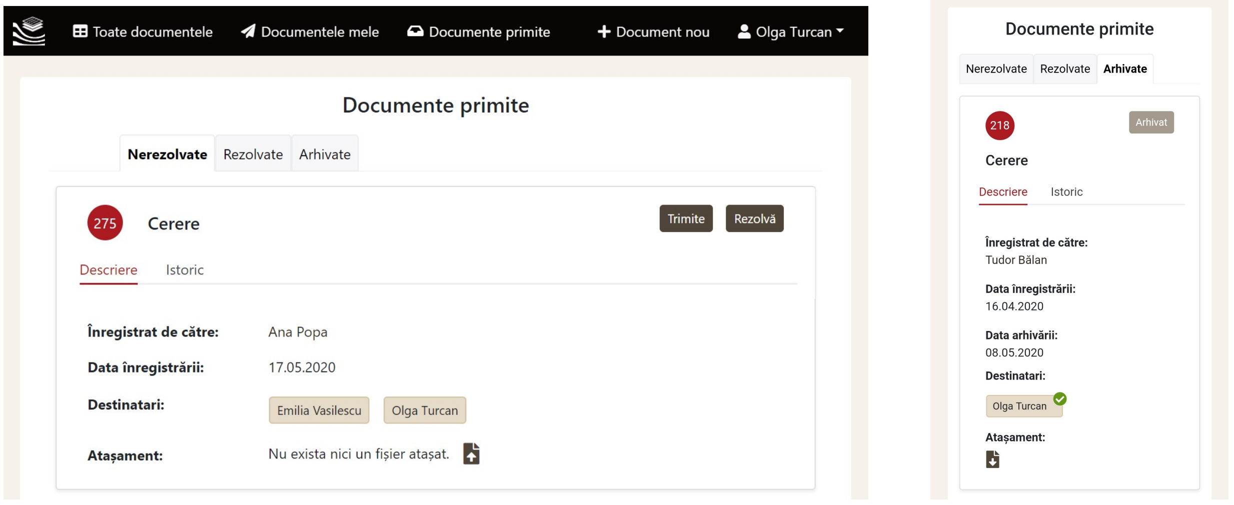
Bună ziua,

Ați primit un document.

Nr. de înregistrare: 277
Titlu: Document nou
Expeditor: Olga Turcan

Figure 5.11: Email notification when a document is received

The Received Documents page is split into three different tabs (See Figure 5.12). The first tab shows documents that the user had received but not yet resolved.



Documente primite

275 Cerere

Înregistrat de către: Ana Popa
Data înregistrării: 17.05.2020
Destinatari: Emilia Vasilescu, Olga Turcan
Atașament: Nu exista nici un fișier atașat.

Documente primite

218 Cerere

Înregistrat de către: Tudor Bălan
Data înregistrării: 16.04.2020
Data arhivării: 08.05.2020
Destinatari: Olga Turcan
Atașament:

(a) On desktop (b) On mobile

Figure 5.12: The *Received documents* page

An action that the user can take is to further send the document to another user. In that case, suggestions will be displayed in the same way they were at document creation, only this time excluding users that already received the document (See Figure 5.13)

After all necessary actions related to the document were performed, the user can mark the document as resolved by clicking the **Resolve** button, in which case a modal dialog would be shown asking for confirmation and offering the possibility to specify an optional resolving comment. After that, the document would move to the second tab, that looks almost the same

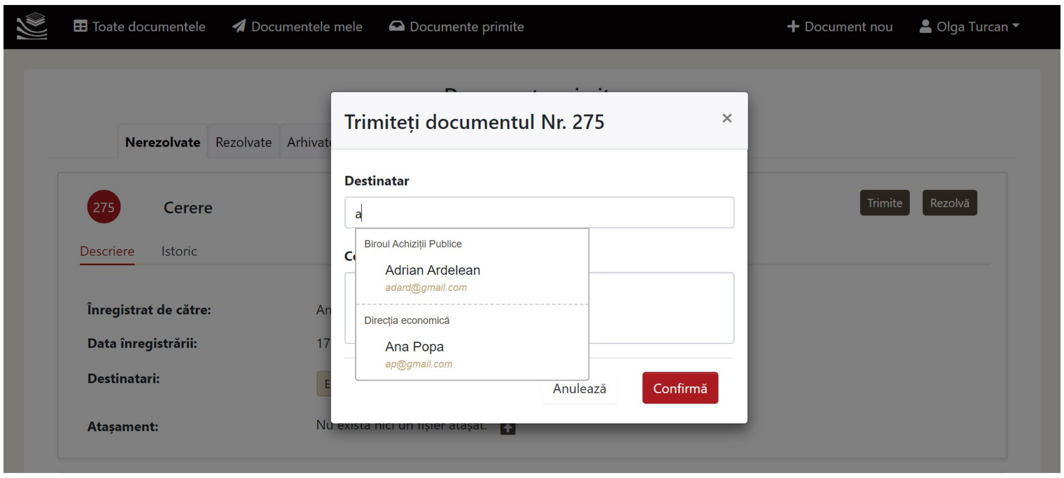


Figure 5.13: Sending a received document to other users

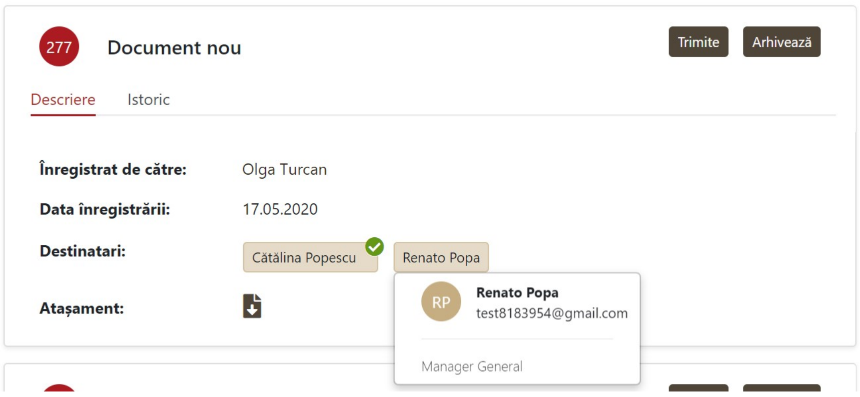
except the Resolve button is disabled since the action was already taken.

The last tab shows documents that were already archived by their issuers. It is logical to assume that this tab contains documents that the user marked as resolved at some previous point. However, it may also include documents that had multiple recipients and got archived after being resolved by another user. Keeping this kind of documents in a separate tab is especially important to let the user focus on the documents that need action.

In addition to managing received documents, the user can also access the documents he or she created under the *My Documents* page. This shows documents in a similar layout, this time in only two tabs to separate the archived ones from the rest. The Resolve button is replaced by the Archive button, although their appearance and behavior are quite similar.

The resemblance doesn't end here. Other than the different action buttons in the upper right corner, the document cards from both pages have the same design and functionality. The card contains two tabs. The first one displays general information about the document. Additional information about the issuer or receivers could be viewed by hovering on their names to display a popup (See Figure 5.14). The user can download files attached to the document, or upload one if no file present. Also, if a document got resolved by someone, a green checkmark is placed near the receiver's name to indicate this. It is especially useful for tracking the status of created documents. The second tab shows all actions performed on the document from its creation to the state it is in now in form of a timeline. Actions are sorted descending by date and include the custom messages given by the users at the moment of creation, resolving or archiving.

While we tried to design our system as intuitive as possible, users may still need guidance when they first start using the app, especially since they won't have any technical background. To assist them throughout this process, we have created a help page that is meant to familiarize the users with all the features the app offers. It provides detailed description of the steps that


277 Document nou

Trimite Arhivează

Descriere Istoric

Înregistrat de către: Olga Turcan

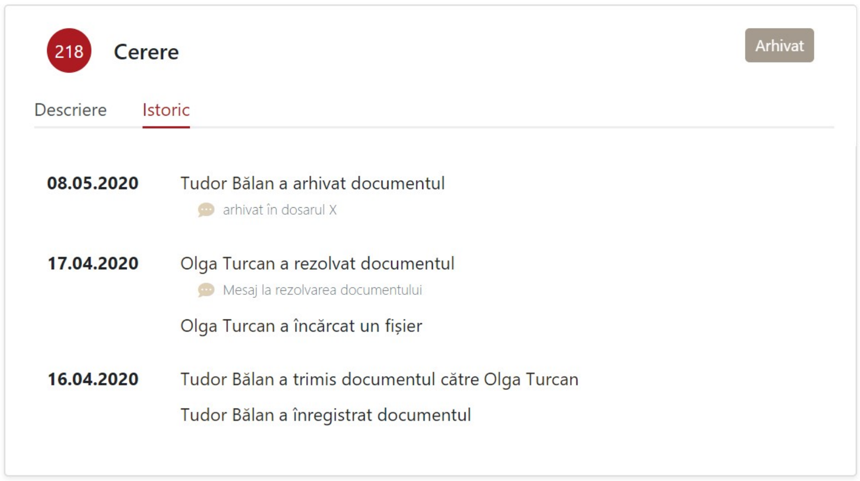
Data înregistrării: 17.05.2020

Destinatari: Cătălina Popescu Renato Popa

Atașament:

RP Renato Popa
test8183954@gmail.com
Manager General

(a) Description tab


218 Cerere

Arhivat

Descriere Istoric

08.05.2020 Tudor Bălan a arhivat documentul
arhivat în dosarul X

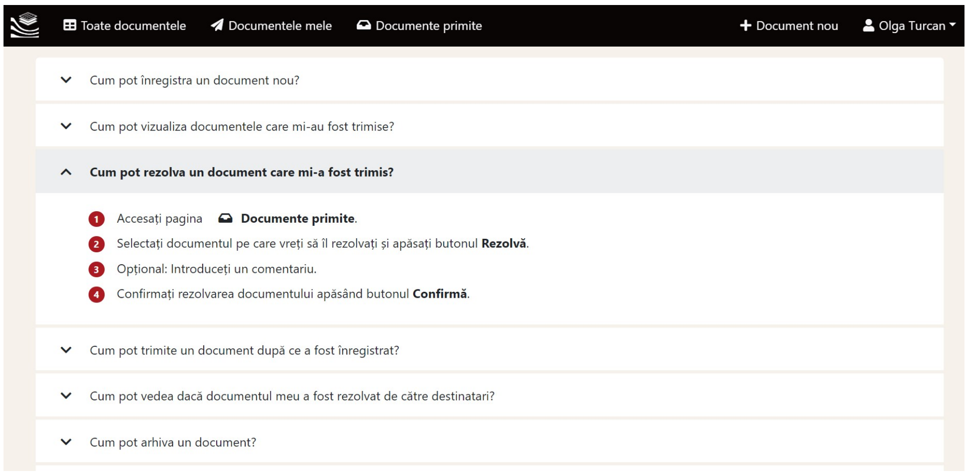
17.04.2020 Olga Turcan a rezolvat documentul
Mesaj la rezolvarea documentului
Olga Turcan a încărcat un fișier

16.04.2020 Tudor Bălan a trimis documentul către Olga Turcan
Tudor Bălan a înregistrat documentul

(b) History tab

Figure 5.14: Example of a document card

should be performed for all tasks that we described in this chapter, like creating, resolving or archiving a document, uploading files or generating reports (See Figure 5.15).


Toate documentele Documentele mele Documente primite Document nou Olga Turcan

Cum pot înregistra un document nou?

Cum pot vizualiza documentele care mi-au fost trimise?

Cum pot rezolva un document care mi-a fost trimis?

1. Accesați pagina **Documente primite**.
2. Selectați documentul pe care vreți să îl rezolvați și apăsați butonul **Rezolvă**.
3. Opțional: Introduceți un comentariu.
4. Confirmați rezolvarea documentului apăsând butonul **Confirmă**.

Cum pot trimite un document după ce a fost înregistrat?

Cum pot vedea dacă documentul meu a fost rezolvat de către destinatari?

Cum pot arhiva un document?

Figure 5.15: Registry System Help page

Chapter 6

Conclusions

The application that is the result of the research and development conducted in this thesis successfully manages to address the requirements regarding the document management process. It represents a good alternative to the current flow, minimizing human effort and need for interaction by automating parts of the process. In addition, the system provides great features for tracking documents and their status. This makes it easier to identify which status a particular document has at any given time, but it also allows to manage the register of documents as a whole, quickly and efficiently getting all the desired information and statistics from the app.

Designing and implementing the Registry System had its own challenges. We had to make the app secure so that users would trust their data with it. We had to keep performance in mind while implementing the application logic. Last but not least, we had to design the application in a way that offers a good user experience. After all, regardless of the complex logic it provides, an app is not going to raise efficiency if the user is frustrated and confused using it. That is why we tried making all the features as intuitive and easy to learn as possible.

As a next step, we plan to deploy our Registry System Application, either on a physical server or a cloud platform, after which the application would be ready to be used by the opera employees.

The main achievement of this thesis is that it is meant to digitalize an existing process in a public institution, which, if we look at the bigger picture, is a small piece in the digitalization and modernization of the Cluj-Napoca city.

Bibliography

[1] Spring framework documentation. https://docs.spring.io/spring-framework/docs/current/spring-framework-reference/core.html.

[2] Zitek COM. Regista - a complete application for registry and document management. https://regista.ro, 2014-2020.

[3] Robert Drummond. How to build a restful api on a raspberry pi in javascript. http://www.robert-drummond.com/2013/05/08/how-to-build-a-restful-web-api-on-a-raspberry-pi-in-javascript-2.

[4] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional, 2002.

[5] James Jacobs. Modern enterprise ui design - part 1: Tables. https://medium.com/pulsar/modern-enterprise-ui-design-part-1-tables-ad8ee1b9feb, 2018.

[6] Peter Lake and Paul Crowther. *Concise Guide to Databases: A Practical Introduction*. Undergraduate Topics in Computer Science. Springer-Verlag London, 1 edition, 2013.

[7] Robert C. Martin. *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Robert C. Martin Series. Prentice Hall, 1 edition, 2017.

[8] Dejan Milosevic. Rest security with jwt using java and spring security. https://www.toptal.com/java/rest-security-with-jwt-spring-security-and-java.

[9] Raja CSP Raman and Ludovic Dewailly. *Building RESTful Web Services with Spring 5*. Packt Publishing, 2nd edition, 2018.

[10] George Reese. *Database Programming with JDBC and Java*. O'Reilly Media, 2 edition, 2000.

[11] Baron Schwartz, Peter Zaitsev, Vadim Tkachenko, Jeremy Zawodny D., Arjen Lentz, and Derek J. Balling. *High Performance MySQL*. O'Reilly Media, 2 edition, 2008.

[12] Robin Wieruch. *Taming the State in React. Your journey to master Redux and MobX*. LeanPub, 2017.

[13] Cheng Wen Zhi, Yang Yi, Zhang Liao, and Lian Li. Optimization for web-based online document management. *Advanced Materials Research*, 756-759, 9 2013.