**Universidade de Aveiro**
**2019**

Departamento de Eletrónica, Telecomunicações e Informática

**ANDRÉ EMANUEL CAJÚS DE CARVALHO**

# SOLUÇÃO SCAFFOLDING DINÂMICA BASEADA EM REGRAS DE NEGÓCIO

# DYNAMIC SCAFFOLDING SOLUTION BASED ON BUSINESS RULES

**Universidade de Aveiro**
**2019**

Departamento de Eletrónica, Telecomunicações e Informática

**ANDRÉ EMANUEL CAJÚS DE CARVALHO**

# SOLUÇÃO SCAFFOLDING DINÂMICA BASEADA EM REGRAS DE NEGÓCIO

# DYNAMIC SCAFFOLDING SOLUTION BASED ON BUSINESS RULES

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia Informática, realizada sob a orientação científica do Doutor Cláudio Jorge Viera Teixeira, professor equiparado a investigador auxiliar do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro, e coorientação do Doutor Joaquim Sousa Pinto, professor auxiliar do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro

**o júri / the jury**

presidente / president

Professor Doutor António Manuel Melo de Sousa Pereira

Professor Catedrático, Universidade de Aveiro

vogais / examiners committee

Professor Doutor Ricardo Filipe Gonçalves Martinho

Professor Adjunto, Escola Superior de Tecnologia e Gestão de Leiria

Doutor Cláudio Jorge Vieira Teixeira

Equiparado a Investigador Auxiliar, Universidade de Aveiro

**Agradecimentos / Acknowledgments**

Obrigado ao meu orientador, o professor Cláudio Teixeira, ao qual gostaria de agradecer esta oportunidade, e por todo o apoio que me deu durante este percurso. Também gostaria de agradecer ao professor Sousa Pinto, pela orientação, disponibilidade e simpatia.

Um agradecimento especial ao Gonçalo, Luís, João, Andreia, Clony, Renato, Saramago e ao Aricson, pois foram os amigos que mais me acompanharam durante esta jornada.

**Palavras Chave** Scaffolding, Back-End, Frameworks, Django, MVT, ORM

**Resumo**

Existe uma grande variedade de arquiteturas e estruturas usadas para ajudar na criação de soluções web. Algumas ferramentas fornecem modelos e mecanismos para acelerar este processo, que designamos como *scaffolding*. O nosso objetivo é desenhar uma solução que facilite a criação de aplicativos de back-end, apresentando maneiras de interagir prontamente com as bases de dados, fornecendo mecanismos para executar tarefas sem a necessidade de produzir sistemas de back-end a partir do zero, criando soluções web em tempo real. Depois de escolher a arquitetura apropriada, e uma *framework* para construir esta ferramenta, estudámos as soluções de *scaffolding* e os seus mecanismos, para compreender o que implementar e incluir na nossa solução. Posteriormente, estabelecemos um gerador *scaffolding*, para controlar bases de dados, e que fornece maneiras simples de aplicar regras de negócios. Como resultado, usando um modelo de dados, os utilizadores podem gerar a estrutura de um projeto sem possuir conhecimentos em programação. O produto do nosso gerador, é um sistema que visa ajudar na criação e atualização de dados, para que a sua inserção seja intuitiva e validada, garantindo que o que é introduzido respeite e esteja alinhado com o que a base de dados espera. A nossa ferramenta também funciona como suporte para soluções mais elaboradas, uma vez que a sua estrutura facilita a adição de lógica de negócio. Embora existam soluções semelhantes, esta traz um novo conceito, relativo aos mecanismos de apresentação das ligações entre os dados, e possui recursos que soluções idênticas não fornecem.

**Keywords** Scaffolding, Back-end, Frameworks, Django, MVT, ORM

**Abstract** There is a wide range of architectures and frameworks used to aid in creating web solutions. Some tools provide templates and mechanisms to hasten this process and which we address as *scaffolding*. We aim to design a solution that facilitates creating back-end applications, by presenting ways to interact with the database out-of-the-box, giving enclosed mechanisms to perform tasks without the need to produce back-end systems from scratch, creating complete web solutions on the fly. After choosing an appropriate architecture and a framework to build such a tool, we have studied *scaffolding* solutions and their mechanisms, to understand how to design it and what to include. Subsequently, we establish a *scaffolding* solution to control databases that provide simple ways to apply business rules. As a result, by using a database or a model, users can generate a project structure without programming skills. The product of our generator it is a system that aims to help with creating and updating data so that its insertion is intuitive and validated, ensuring that it respects and is in line with what the database expects. Our tool also works as the basis for more elaborate solutions, since its structure facilitates adding further business logic. Even though there are similar solutions, it brings a new concept regarding the data interconnection presentation mechanisms, and it holds features that solutions alike do not.

# List of contents

# List of figures

# List of tables

# List of acronyms

| | |
|---|---|
| **API** | Application Programming Interface |
| **CRUD** | Create, Read, Update and Delete |
| **CMS** | Content Management Systems |
| **CSS** | Cascading Style Sheets |
| **DRY** | Don’t Repeat Yourself |
| **DBSM** | Database Management System |
| **EVM** | Erlangs Virtual Machines |
| **FTP** | File Transfer Protocol |
| **GNU** | General Public License |
| **HTML** | Hypertext Markup Language |
| **HTTP** | Hypertext Transfer Protocol |
| **IT** | Information Technology |
| **JS** | JavaScript |
| **JSON** | JavaScript Object Notation |
| **KISS** | Keep It Simple, Stupid |
| **MTP** | Mail Transfer Protocol |
| **MVT** | Model View Template |
| **OWASP** | Open Web Application Security Project |
| **ORDBMS** | Object-Relational Database Management System |
| **ORM** | Object-Relational Mapping |
| **PHP** | Hypertext Processor |
| **PIP** | Pip Installs Packages |
| **QA** | Quality Assurance |
| **RDBMS** | Relational Database Management System |
| **SMTP** | Simple Mail Transfer Protocol |
| **SQL** | Structured Query Language |
| **SaaS** | Software as a Service |
| **SOA** | Service-Oriented Architecture |
| **WSGI** | Web Server Gateway Interface |
| **UC** | Use Cases |
| **UX** | User Experience |
| **URL** | Uniform Resource Locator |
| **VCS** | Version Control System |
| **VM** | Virtual Machines |

# 1. INTRODUCTION

[1]

*In this chapter, we present a concise overview of the concepts that are the starting point for our work. Then we will contextualize the specific subject of interest, followed by the main objectives that we aim to achieve, and we finish with a summary of the dissertation structure.*

## 1.1. Overview

In today's world, there is a drastic change in every field because of the fast advances in technology [1]. The computer was significant for the rise of the Internet and web applications, which are the base of most of today's software systems. Companies use software to promote their business which automate processes and provide various tools [2]. Subsequently, web-applications bring different notions related to their creation and deployment.

There are usually three main components when building a web-application. The user interface, which is the view of the application, often created with Hypertext Markup Language *(HTML)*,

[1] *https://i.pinimg.com/originals/4f/76/af/4f76af37862675f52669451b3715ec21.jpg*

Cascading Style Sheets *(CSS)* and *JavaScript (JS)*. The server-side, which is the core component of the application and where the whole business logic lies, and the database, which stores the data.

Concerning the server-side, there is a wide range of architectures and frameworks used to aid in creating web solutions. Even though they are different, they all live in the separation's assumption between the layers of data, interface, and control of the actions performed. They vary, however, in the way they relate their elements. There is a demand to facilitate building the constituent components of the existing frameworks concerning those architectures to improve development time and achieve better solutions.

When building an application, one of the best tools for software development is the Integrated Development Environment (*IDE*). They abstract away the complexity, working as an interface between the developer and the computing infrastructure. Most of the *IDEs* give an easy way to extend plugins, manage dependencies, control builds, manage requirements, provide test environments, between others. [3] These tools have the goal to increase development speed and to reduce development mistakes as they present tools to automate tasks, such as refactoring, correction, auto-complete, and compilation errors [4].

Furthermore, when building projects, there are templates and generators available on the web that fasten the development time. They provide an enormous advantage by setting aside the need for programmers to design everything from the beginning by supplying ready-to-use solutions or supporting tools.

## 1.2. Contextualization

To manage data, a web-application relies on a front-end to implement the semantics and on a back-end Database Management System (*DBMS*) to store and process data. There are various frameworks to build such database-backed web applications. They provide a direct way to interact with the database via an Application Programming Interface (*API*), which pulls, saves, or changes data. [5] These operations must comply with specific rules based on the structure of the underlying database towards the entities and their respective constraints. These systems aim to help create and update the data so that its insertion is intuitive and validated, ensuring that it respects the data structure and is in line with what the databases expect.

Built such a platform may not be trivial, as it is necessary to create mechanisms that facilitate the user's perception of the context when performing operations on the system. Besides, there may be a whole set of business rules embedded in this kind of system, like features and other inherent mechanisms.

## 1.3. Objectives

The motivation for this dissertation is to formulate a tool that facilitates creating back-end applications, by presenting ways to interact with the database out-of-the-box, giving enclosed mechanisms to perform tasks without the need to build systems from scratch, creating complete web solutions on the fly.

First, we want to analyze systems that adopt similar ideas to design our solution the best way possible, choosing the best architecture, framework, and database technology.

A significant requirement for this system is to include the right business rules in a practical and agile fashion after generating those back-end solutions, and the ease to use that generating tool. Another necessity is the intuitive graphical interface made available, as it abstracts away the internal operations from the users.

We expect that our solution will transpose different data models and generate complete back-end solutions to manage data while facilitating introducing new business rules if required.

## 1.4. Work Methodology

The method of work rests on a group of steps that reflect how we reach our solution, which are:

i. Define the problem;
ii. Search and compare existing solutions, like architectures, frameworks, database technologies, *scaffolding* methodologies, among others;
iii. Specify the functional and nonfunctional requirements based on our study and on what we intend to include in the default solution;
iv. Develop the solution according to the previous steps;
v. Test and repeat the whole process until reaching the intended solution.

## 1.5. Outline

After the introduction, we structure the dissertation by six correlated chapters.

The second chapter, the literature review, presents the fundamental concepts of our work. We start by explaining what it is a Content Management System (*CMS*), and then we choose the right tools and mechanisms to build the best solution possible. We will focus on the various system architectures, comparing them, and deciding the one that satisfies our needs. Following, we will analyze frameworks to compare and choose the right one according to a set of criteria. Finally, we will define the *DBSM* that best fits it. Thus, with our defined structure, we will investigate *scaffolding* methodologies

presenting similar solutions available on the web, giving focus to the ones associated with the chosen framework.

In the third chapter, the system requirements, we will build our functional and non-functional system requirements, and present the use cases.

The fourth chapter, the system implementation, starts by detailing the adopted technologies, the system itself, the script which generates our entire back-end solutions, and presents an overview.

In the fifth chapter, we introduce the solution interface, exposing a product which was build based on our system.

The sixth chapter refers to the conclusions and future work.

# 2. LITERATURE REVIEW

[2]

*The literature review of this project has the purpose of conducting a study regarding the concepts related to our dissertation objectives. First, we explain what CMSs are and why we did not use them. Afterward, we define the essential concepts to build our solution from scratch, presenting the existing web solution architectures, the frameworks that underpin the chosen architecture, and the DBMS that best suits it. Finally, we introduce the scaffolding concept and highlight similar works so we can conceive a strategy for our dissertation purposes.*

## 2.1. Content Management Systems

When building a dynamic application, the first thing that comes to mind is the *CMSs* systems, as they provide ways to create and upload content of a website. *CMSs* are a set of tools that enable users with no technical skills to maintain websites. The most popular *CMSs* are *WordPress* and *Joomla*, which provide mechanisms to store data using relational databases as they support Structured Query Language (*SQL*). [6]

[2] *http://img2.rtve.es/v/5256299?w=1600&preview=1559489727541.jpg*

*CMSs* provide an administration area, where it is possible to add, delete, and change functionalities or any other aspect. A *CMS* entails two foundations, the Content Management Application (*CMA*), which facilitates content creation, modification, and removal, and the Content Delivery Application (*CDA*), that uses and compiles that information on the website. [7]

Open-source *CMSs* have great communities around them, which present organizations with a wide diversity of resources to support, develop, improve, and innovate their platforms.

*WordPress* is the most popular *CMS* [8]. Other well-known *CMSs* include *Joomla*, *Shopify*, and *Drupal*, even though they hold a much lower market share. In this chapter, we introduce these solutions, reviewing their main advantages and disadvantages.

### 2.1.1. Overview

1. ***WordPress***

*WordPress* supports over 60 million websites built on top of over 50.000 available plugins that allow users to tailor sites towards their needs [8], encouraging entire industries to use it. With the *WordPress* Representational State Transfer (*REST*) *API*, it is now possible to use *WordPress* as a headless *CMS* to build web apps on top of, while enjoying all its core back-end functionalities like content and user management [9].

*WordPress* has many advantages. For example, it is possible to install *WordPress* for free on customer servers, there are many free template themes, the service provides packages with plans towards the customer needs, and it works on mobile phones and tablets. Another exceptional advantage is the performance since it uses minimal Hypertext Processor (*PHP)* code, which allows websites to load fast. It has a few disadvantages, often associated with the many updates and a hard learning curve for inexperienced users. [10]

2. ***Joomla***

*Joomla CMS* is a *PHP* solution [11]. It is an open-source *CMS* based on a *model-view-controller* (*MVC*) web application framework used for building applications [7]. *Joomla* is a great option when creating interactive language websites. This *CMS* has many extensions available, such as forums, calendars, search, ad serving tools, and much more. *Joomla* is very efficient based on its components, modules, templates, and add-ons found on *Joomla.org*. [12] Additionally, it has an intuitive management interface to control every feature and functionality [8].

*Joomla* brings many advantages. It is easy to install but not as easy as *WordPress*, as it takes a few minutes to have a working script on a server. The plugins are free, and there are plenty of them available on the website homepage. Although not as popular as *WordPress*, this *CMS* contains many tools and tutorials online. Other advantages are the Uniform Resource Locators *(URLs)* taxonomy, the

updates, and the perks of the management and advanced administration. The limitations relate to the limited template options, server resources, and efficiency. [13]

*3.* ***Drupal***

*Drupal* is another *CMS* open-source used to manage from single-user blogs to multilingual sites with thousands of users [14]. *Drupal* requires basic technical knowledge. Like the others, it is a popular *CMS* among developers, and it comes with core modules that can become contributed modules. [15] It is a good option as it comes with standard features like simple content authoring, performance, and security [8].

It has advantages as a broad range of content types, providing ways for advanced users to create complex projects. Other features include flexibility, resources, and the great community around it. As for disadvantages, it is very time consuming, not user-friendly, and requires advanced knowledge to install and maintain. [14], [16]

*4.* ***Shopify***

*Shopify*, unlike the others, it is not open-source and requires a fee. It helps to kick-start a business with simple drag-and-drop functionalities. *Shopify* has a great community of users that use this solution to power their e-commerce stores. [17]

This *CMS* provides one of the best *UX/UI* for e-commerce platforms, excellent app integration with Software as a Service (*SaaS*), and it is very reliable and secure. One of the most significant advantages comes from end-to-end integration, as its e-commerce solutions workflow is straightforward and much better than any other solution in that matter. However, it has disadvantages like the ecosystem of paid services that help customers to support their infrastructure and the lack of customizability for those customers themselves. [18]

### 2.1.2. Considerations

In this section, we introduced some of the most popular *CMSs* available in the market, showing their advantages and disadvantages. Our research aims to explain how to build similar systems by understanding their base architectures. It is essential to realize that we propose a generating back-end tool to manipulate data, automating the need for taking care of the complexities of the underlying database. That is the main reason we approach these solutions, since *CMS* provides ways to manage back-end functionalities of a site. So, we try to understand if *CMSs* bring any advantages and all the tools to achieve what we intend, putting aside the need to reinvent the wheel.

Although *CMSs* give all the tools to produce a great solution, they do not meet our needs. They have constraints that we will not find in a solution built from scratch. Creating our solution empowers flexibility and comprehensibility, promoting total control of our results. They also do not present mechanisms to produce management platforms out-of-the-box, creating ready-to-use solutions

based solely on databases. Meaning that, given a data model, we need a solution that can dynamically adapt and instantly provide a management platform. Furthermore, we aim to create a structure where it is easy to apply customizations and specific business logic. With our custom template, based on an appropriate design and technologies, we want to ease those adjustments, leading to a flexible business logic solution, targeted only for management purposes.

## 2.2. Software Architectural Patterns

With the amount of software increasing each day, software architectures are raising to support stakeholders to define high-level structures of a system [19]. They come as the fundamental structure of software, which establishes the relations and properties among their elements.

Software architecture is essential when building a successful solution by promoting systemic qualities such as performance, security, availability, modifiability, and so on. Designing any software architecture involves deciding on many variables, and once made, are usually very difficult and expensive to change. [20]

The article [20] includes a survey taken by 809 participants from 39 different countries. The questionnaire asks people to point the software architectural patterns most used in their projects.

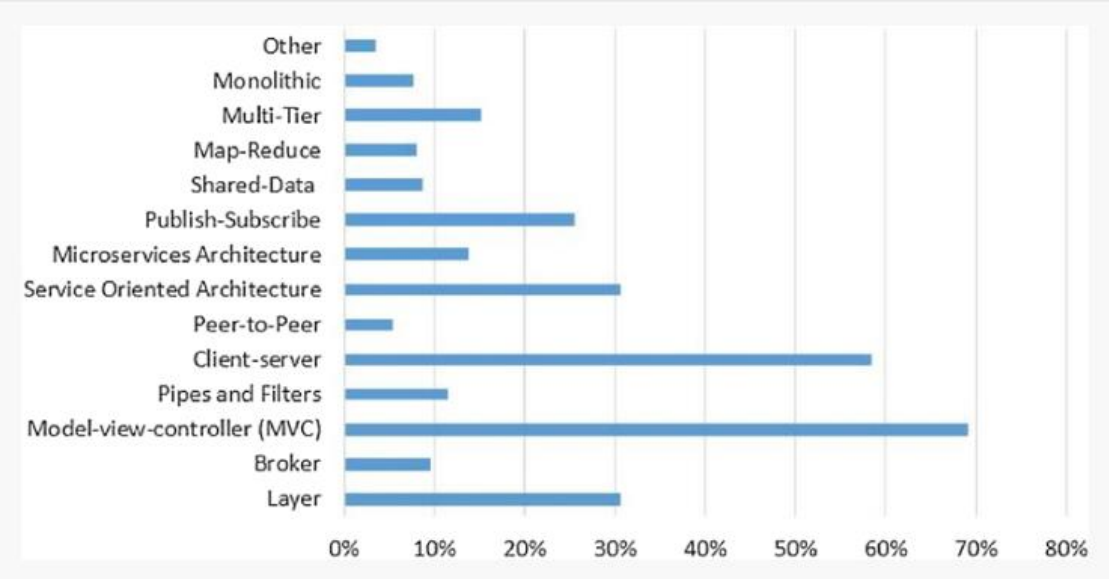



Figure 1 - Software Architectural Patterns answers [20]

During this section and based on some results of the survey (*e.g., Figure 1*), we will review and describe well-known architectures that are the core of most software solutions. With this analysis, we intend to understand their advantages, disadvantages, and usages. Afterward, we plan to select the one that best suits our objectives.

### 2.2.1. Overview

*1.* ***Model-View-Controller***

*MVC* is the basis for most modern web frameworks, and it is the best-known software pattern for implementing user interfaces and web applications. In 1970, Trygve Reenskaug introduces this architecture, which is one of the first in the Model-View (*MV*) patterns family [21]. *MVC* has the purpose of dividing the application model, user input, and visual feedback [22].

The pattern separates the application into three components. The *model* handles data storage. The *view* instantiates the *controller*, to display the *model* data and forward the input commands to the *controller*. The *controller* updates the *model* state when there is an event. [23]

Several frameworks use this architecture. *Django*, *Ruby* on *Rails*, *and ASP.NET* which have a thin client, meaning almost all the logic is on the server. And there are other *MVC* frameworks, like *EmberJS*, *Backbone*, and *AngularJS*, that allow executing part of the *MVC* components by the client.

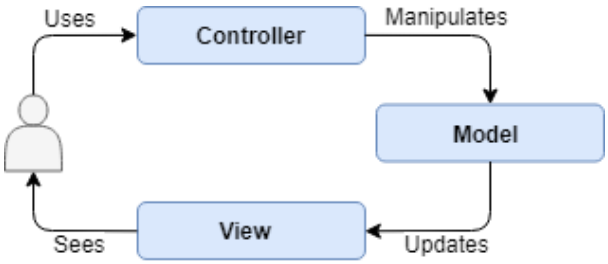



Figure 2 - *MVC*

This architecture has several advantages. It provides a fast development process because it supports parallel development, meaning that one programmer can work in a *controller* while others can work on a *view*. Developers can change a *model* without affecting all the project architecture. This architecture is easy to maintain, as it is possible to provide multiple *views* for a *model* and to reuse components. Concluding that, it is modular, flexible, and reusable [24].

As for limitations, it requires knowledge in several technologies, the complexity can increase over time, and it is not suitable for small applications because of performance reasons. The *view* depends both on the *controller* and the *model*. And the *models* do too much work, as they inform the controller about the data and prepare the result for the *view*.

Worldwide Web (*www*) applications in major programming languages use this architecture [25].

*2.* ***Client-server***

Client-server systems are popular and are the basis of several solutions these days. The client-server is a software architecture composed of the *client* and the *server* where *clients* send requests

while the *server* responds to those requests. Standardized protocols emerge for the communications between the *client* and *servers*, including File Transfer Protocol (*FTP*), Simple Mail Transfer Protocol (*SMTP*), and Hypertext Transfer Protocol (*HTTP*). [26] To speed up rate performances, in this architecture environment, the *server* processes the data and sends the results to the *clients* [27]. Figure 3 translates this architecture.

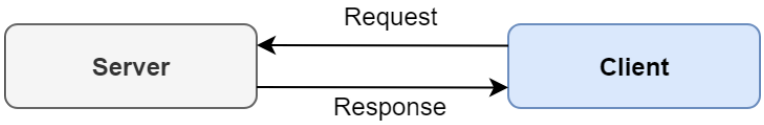



Figure 3 - Client-server Architecture

Client-server systems hold several benefits. They store data on a *server*, which makes it easier to protect. It is possible to replace or enhance elements, which promotes scalability. And they are easy to maintain due to the encapsulation of standalone computers.

This architecture also has significant drawbacks. For example, when a *server* gets too many requests, it can become overloaded or when a *server* fails, the *client* requests cannot get a response from that *server*.

Developers apply this architecture towards file transfer, which allows storing files on a *server*, for mailing transfer using the Mail Transfer Protocol (*MTP*), and banking [26] [25].

### *3. Service Oriented*

Service-Oriented Architecture (*SOA*) is a group of services that can combine and communicate with each other over networks on-demand. It is possible to access them without knowing the underlying system platform. [28]

This architecture couples' various services to build an application and allows them to recombine in the desired ways, to improve or implement business requirements. It also provides a reliable and flexible way to implement technologies for customers and service providers. [29]

Services are program units that can execute different functions assigned to them. They can run in distinct environments like hardware, operative systems (*OS*), or even in multiple programming languages. They are easy to add, combine, reuse, or replace. [30]

This approach provides several benefits, as it promotes the reuse of services, their easy maintenance, service independence, among others.

The most prominent disadvantages result from possible high cost and the number of requests can lead to the need of high bandwidth servers.

*4.* ***Layered***

Layer architecture or *n-tier* architecture is the most common software architecture pattern, and therefore, the choice for most business applications development. This pattern holds horizontal *layers* that perform specific roles.

Programmers separate it into four standard *layers*, making the surrounding work abstracted from the rest of the *layers*. [31] Figure 4 translates this architecture.

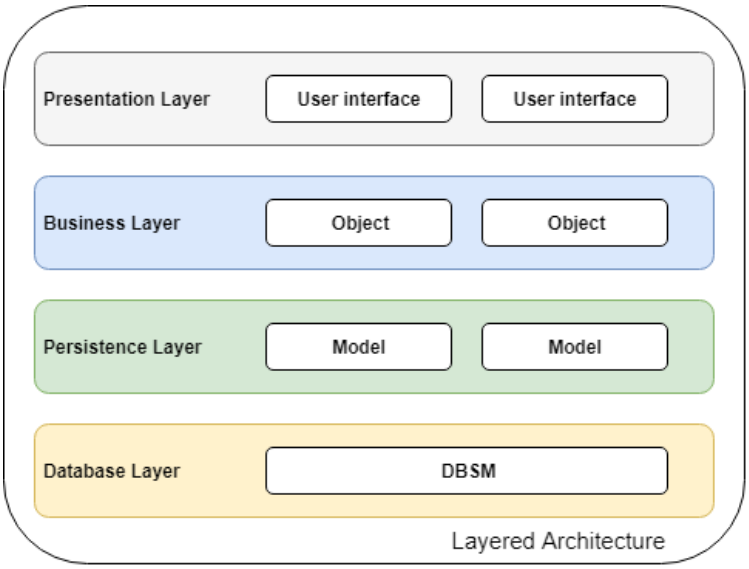



Figure 4 – Layered

This architecture presents several disadvantages, it is not very extensible, it does not perform well when there are too many *layers*, which affects high performance if needed, layer isolation can lead to worse architecture module interpretation, and one of the worst reasons is that small changes can require complex redeployments. Overall, it is a great pattern, considering it shares the concerns within the pre-defined *layers*, which makes it maintainable, testable, and updatable, as it is possible to change the *layers*.

This architecture is the basis for general desktop applications and e-commerce web applications, working great towards applications that require maintainability and testability. [25]

*5.* ***Microkernel***

Microkernel architecture is used to implement product-based applications, which are applications packed and distributed as a third-party product. Microkernel comprises a pair of components, the *core* system, and the *plug-in* modules, dividing the application logic between the independent modules and the *core* system. [31]


Figure 5 - Microkernel

This pattern has advantages, as improving systems development, and since it does not need additional features, it allows performance streamlining. [31] Another advantage is the modularity and customization of this architecture. Microkernels are very modular by nature, being possible to change or remove *modules* without changing the *kernel*.

Concerning the disadvantages, the *plug-ins* require the *microkernel* to know of their installation, and adjustments can be very challenging once the *plug-ins* have dependencies.

This architecture is the best alternative when applications possess a fixed set of core routines that require frequent updates, when there is a clear division between those basic runtimes, and when there is the need to build tools used by a wide variety of individuals.

### *6. Microservices*

James Lewis introduced the concept of microservices in 2012 [32]. Since then, more and more companies have been using it to build distributed systems. The microservices architecture intends to develop standalone application systems with a small set of *services*, where each *service* runs in its process and uses the *HTTP* resource *API* mechanism to communicate with each other [33]. A microservice is a small application with a single responsibility, which is easy to deploy, scale, and test [34].

Microservices have many advantages in today's era of cloud computer and containerization. It is possible to deploy microservices on several platforms, with different developer tools and programming languages. One of the most prominent benefits is that microservices use *APIs* and communication protocols to interact, but without relying on each other. This individual responsibility is essential, as it leads to better scalability and higher failure tolerance.

Nevertheless, microservices also have weaknesses. It is hard to perform a global test to their distributed components, solutions can become expensive because of license costs for third-party apps, and it is essential to control the microservices as they can become harder to manage and integrate.

There is a reasonable use of the microservice frameworks in the industry, including *Netflix*, *Spring Cloud*, and *Ali's Dubbo*. [32]

### 2.2.2. Considerations

After examining the most practiced software architectures, we can now compare the results and reach conclusions. This process was fundamental to determine the most appropriate design for our solution.

First, it is worth mentioning the differences between layered and *MVC* architectures. These architectures may seem similar, but they are not. In the layered, the client layer never interacts with the data layer and needs to pass through the mediator layers, which makes it a linear architecture. While the *MVC* architecture is a triangular architecture where the *view* sends the updates to the *controller*, this will update the *model*, and the *view* receives the update by the *model*. In the layered architecture, the user interacts with the presentation layer, and in MVC, with the controller with the help of the view. Another significant difference between the two is that *MVC* is for presentation, while the layered architecture focuses on whole systems. It is possible to use the *MVC* architecture in the presentation layer of the layered architecture, as they can complement each other and not replace each other.

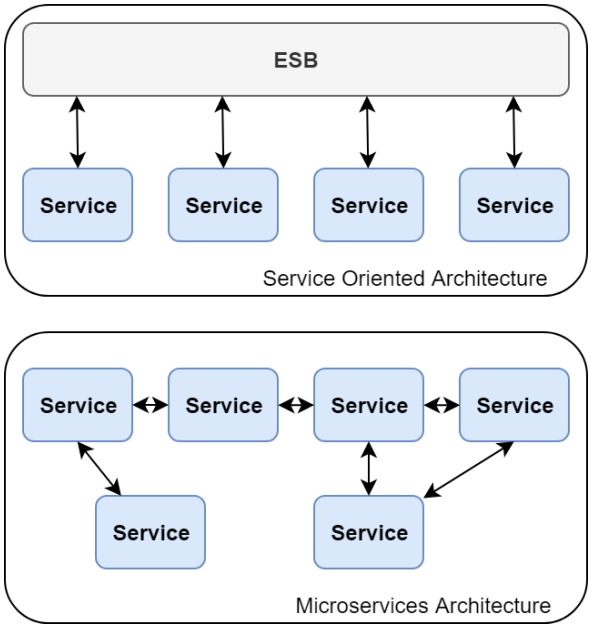



Figure 6 - SOA vs Microservices

*SOA* and Microservices, although similar and with identical components, are distinct architectures regarding their services and the protocols. The *SOA* service is loosely coupled and cohesive, while the microservices are independent and uncoupled. *SOA* has centralized governance, and microservices have decentralized, being governance regarded as adding strategic value. Furthermore, in terms of communication, there are also differences, as both use different protocols. *SOA* uses *HTTP*, *JMS*, *SOAP*, and *XML*, while microservices use *HTTP*, *REST*, and *JSON*. Overall, on *SOA*, the components are decoupled and connected by an Enterprise Service Bus protocol

used for communication, while microservices, deliver one functionality and communicate with lightweight protocols among them (*e.g.*, *Figure 6*). [35]

With this research towards the chosen architectures, we got a general idea to determine in which architecture we want to develop our solution. Since we aim to create a back-end solution to manage databases, with system-wide modifications easy to maintain and easy to reuse, we selected the *MVC* architecture, which is in the core of most web frameworks, and it fulfills our needs. For now, we are only aiming to create the generator itself. The other architectures can come in handy when deploying or integrating the *scaffolding* result in a production environment. Meaning that, for example, *SOA* and *MVC* can coexist, an *MCV* app can connect to the application server in an *SOA* system, working at a lower level of abstraction, since *SOA* breaks down the application rather than creating monolithic applications by working at a higher level.

## 2.3. Web Frameworks

Simplifying the complexity of software involves dividing a problem into smaller problems. In this context, standards emerge, which promote the reusability and quality of developed software by making systems easier to build and understand. [36]

Being aware of the latest technological advances is very important, as the needs are changing. The back-end frameworks follow this permutability, and with it comes the need to perceive the most suitable ones depending on the demands of the target customers.

During this section, we aim to present the most adopted back-end frameworks to provide an understanding in which circumstances to implement each of them and to determine the one that better suits our objectives.

However, we need to decide which frameworks to study and compare, as in today's era, there are many possibilities. We started by analyzing trends, and to assist this process, we used two online rankings from *GitHub* [37] and *Hotframeworks* [38].

First, we began by examining the most used server-side frameworks on *GitHub.* The presented list still had several entries, so we reduced it to one of each respective programming language.

*GitHub* provides information concerning all the frameworks, like the *stars*, which shows the number of saved repositories, the *forks*, which are the number of copies of the repository, and their activity graph. In Table 1, we present the main results.

The other frameworks were not significant when weighing the number of *stars*, *forks*, updates over time, and activity graphs. We only choose one from each programming language, and since *Flask* was close to *Django* with 44.229 stars and 12.689 forks, we will also address it during this framework review.

Table 1 - Most used backend frameworks by programming language (GitHub)

| **Language** | **Framework** | **Star** | **Fork** |
|---|---|---|---|
| Python | Django | 42,643 | 18,362 |
| Java | Spring | 30,629 | 19,642 |
| JavaScript | Express.js | 44,600 | 7,482 |
| Ruby | Ruby on Rails | 43,585 | 17,515 |
| PHP | Laravel | 53,420 | 16,381 |

To support this decision, we search for another platform for more feedback about the most used frameworks, which leads us to *Hotframeworks.* This platform is attractive as it updates in real-time both the *GitHub* and Stack Overflow average scores. On this platform, we can see the chosen back-end frameworks scores in order of their total score: *Ruby on Rails* (94), *Django* (92), *Laravel* (91), *Spring* (89), *Express.js* (87) and *Flask* (84). By examining the results, we could conclude that the previous select back-end frameworks were still on top, so we stick with them. We also include the *Phoenix* framework, with a lower rate (71), but that it was appealing to us for being written on *Elixir*.

In the following chapter, we will use these ranks as the basis to select which frameworks to study.

### 2.3.1. Overview

*1.* ***Django***

*Django* is an easy-to-use *Python* back-end framework used to build high-efficient, portable, reusable, and extendable applications [39], [40].

*Django* has a large community of contributors who develop new third-party applications for the framework. Besides, it holds the Pip Installs Packages (*PIP*), a package manager which is very useful for replicating the development environment and to track the containing packages [41].

It is the most popular *Python* web frameworks with default Object-Relational Mapping (*ORM*), alongside the excellent documentation, the support, and it is open-source. Like many, it is an *ORM* framework used by developers as one of the most desired ways to manage data and provides a conceptual abstraction for mapping database records to objects in object-oriented languages. [42]

*Django* framework has a Model-View-Template (*MVT*) architecture. *Model* is the data source that contains the fields and behaviors of the data, which are after mapped to the database tables. The *view* is a *Python* function that takes a web request and returns a web response. The *template* contains the static parts of the desired *HTML* output and the syntax to present the rendered content. [43]

*2.* ***Express.js***

*Express.js*, written in *JS*, is a *Node.js* web application development framework that provides a wrapper on the lower-level node interface. This framework presents a more consistent service implementation instead of using *Node.js*, giving ways to handle routing and *HTTP* operations. [39] Several *Node.js* frameworks use *Express*, including *Feathers*, *ItemsAPI*, *Poet*, *Kraken*, *MEAN*, *Sails*, *Kites*, *Nestjs*, and so on. [44]

As stated, *Express* is a *Node.js* web application, meaning that we need to explain what is *Node.js* and then the *Express* framework itself. *Node.js* is an open-source, cross-platform, runtime environment that enables developers to build server-side tools and applications in *JS*. From a web server development's viewpoint it has advantages as high performance, it uses plain old *JS*, it uses the node package manager (*NPM*) that provides thousands of packages, is portable among all the major operative systems (*OS*), and it possesses a great community around it. *Express* is the most popular *Node.js* web framework. Although it is a minimalist framework, the developers have already created many packages to address almost every need. The main parts of its architecture comprise routes, middleware, error handling, and *template* code. [45] Basically, *Node.js* is a run-time environment for building server-side applications, while *Express.js* is a framework based on *Node*, with added features for building web-applications.

*3.* ***Flask***

*Flask* is a powerful, sophisticated, and lightweight web framework written in *Python* that supports relation databases, like the *SQLite* database [46]. It provides a simple *template* for web development since it runs on top of the Web Server Gateway Interface (*WSGI)* toolkit and *Jinja2* template engine [47]. *Flask* is a microframework, and it is a multi-platform, which eliminates the restrictions on working with only one operating system [48], [49].

Out of the box, it comes with several advantages, like a built-in development server, debugging, unit testing, *RESTful* request dispatching, support for secure cookies, *WSGI* compliant, Unicode based, and it possesses excellent documentation. It is also straightforward to set up, using the command "pip install *flask*" to install, "export *FLASK*_APP=app.py" to export the environment variable, and "*flask* run" to run the app.

In comparison with *Django*, it is better suited to develop microservices and smaller projects. This makes it an excellent solution for *Python* programmers who do not need all the functionalities and perks that come with *Django*. Overall, *Flask* is flexible, lightweight, and offers fast prototyping.

*4.* ***Laravel***

*Laravel* is a popular web application framework based on *PHP* programming language and the best compared to other *PHP* frameworks [50]. This framework, created by Taylor Otwell, is open-

source and follows an *MVC* architectural pattern [51]. It comes with elegant syntax, *Blade Templating*, *ORM*, and *Bundles* that support system functions [52].

It is a robust framework, providing authentication, routing, session manager, caching, and more of the broad *MVC* components. It has exceptional database migration tools and integrated unit testing. These aspects make it an excellent framework for developers to build a complex application. [51]

*5.* ***Phoenix***

*Phoenix* framework uses an *MVC* pattern on the server-side, being great because of its concurrency and functional programming. It uses *Elixir* programming language and runs on Erlangs Virtual Machines (*EVM*). [53]

Many of its components are like the ones from *Ruby on Rails* or *Django*, with distinct parts and different roles. Controllers provide actions (functions), to handle requests that adapt the data to pass into the *views* and to perform redirects. *Views* render the *templates* and act as a presentation layer. The *templates* hold the content displayed in response, which is pre-compiled and fast. [54]

This framework works great with smart applications that require efficient socket connections and API-based machine-to-machine interactions [55].

Even though it has an elegant and great design, it requires technical skills in *Elixir*, which can be preponderant in the time of choosing the most suitable framework.

*6.* ***Ruby on Rails***

*Ruby on Rails* works with *Ruby* programming and is based on *MVC* architecture. This web framework facilitates the development and maintenance of projects, as its focused on agile development, centered on *don't repeat yourself* (*DRY*) and *Keep It Simple, Stupid* (*KISS*) software principles. [56]

This framework has several components. The *Active Record* with backup from the *Active Model* handles the *model*, the *Action Controller* manages the *controller*, and the *Action View* controls the *view*. Besides, the framework has other components. An *Action Mailer* that sends *HTML* and plaintext emails, an *Action Cable* that does updates over WebSocket's, an *Active Support* that enhances the language, and, the *Railties* which provides commands and generators to bind them all together. [57]

*7.* ***Spring***

*Spring MVC* is a lightweight, robust *Java* application framework. It provides support to additional frameworks like *Hibernate*, *Struts*, *EJB*, *Tapestry*, and more. [58] This framework presents handy components to develop web applications like *servlets*, *Java server pages*, *Java beans*, and business logic. [59]

All is architecture runs around the *DispatcherServlet* that handles all the *HTTP* requests and responses. Like the previous frameworks, the business model contains the business logic. The *controller*, called *DispatcherServlet*, matches the *URI* to the *controller* class, takes the requests, and calls the proper service method that sets the data *model* and returns the *view* to de *DispatcherServlet*. The *DispatcherServlet*, with help from the *ViewResolver*, captures the proper *view* for the request. [59]

### 2.3.2. Considerations

When developing a back-end platform, and after this framework review, there are many things to consider. We need to emphasize that we intend to build a dynamic back-end application generator based on the database structure to fit establish business rules, and which later can manage the data for separate applications.

Taking into consideration the main frameworks, which include *Django*, *Express*, *Laravel*, *Ruby on Rails* and *Spring*, deciding which back-end framework is the most suitable to accomplish our goal is subjective since all of them give excellent support to implement what we propose.

It is necessary to define criteria to select the fittest framework. We gather four main measures to proceed with our decision, which are: the learning curve and how fast it is to develop and achieve a good solution; documentation and community; security and reliability; and the performance overall.

To make these comparisons, we create a table with three different classifications, "2" for good, "1" for average, and "0" for bad. The next points translate the criteria:

1. ***Learning curve and fast development***

To test the learning curve, we came across *StackShare* [60], which provides a comparison between the frameworks. We use this platform as it presents a great notion of the frameworks without the necessity to implement a project and analyze all its functionalities. *StackShare* came as a reliable answer since are the developers who judge the frameworks they develop with, presenting a comprehensive opinion to fill the criteria.

According to StackShare, *Django* and *Rails* are easy to learn and faster for developing, followed by *Express* and *Laravel*, which are also straightforward, and *Spring* is the most difficult to learn. Therefore, we give a score of 2 to *Django* and *Rails*, 1 to *Laravel* and *Express*, and 0 to *Spring*.

*Django* and *Rails* are also mature frameworks with more stuff out-of-the-box than the others are.

2. ***Documentation and community***

When looking at the official documentation of all the frameworks, we can verify that they are all well documented and structured, even though some might be easier to explore than others.

As for the community, we can base our criteria from the *Hotframeworks* and *GitHub*, since they translate it. We rated 2 points for all the frameworks.

*3.* ***Security and reliability***

Security is necessary when developing an application for a business. Even though they present current updates, the security and reliability are not 100%. The Open Web Application Security Project (*OWASP)* is the most useful standard for secure web development. In 2010, 2013, and 2017, *OWASP* composed a guide for "The Ten Most Critical Web Application Security Risks", which can be a good read. In this section, we did not make a broad review of each framework. [61] We based the framework's security and reliability on the intensive and current security updates shown on their logs, giving a score of 2 to all.

*4.* ***Performance***

It is hard to compare framework performances, as there are so many factors that influence it, like request sizes, queries, and so on. Test each framework in our environment and see which one fits our needs is the best option. Considering that we are examining many options and that task would be hard to accomplish, we have found *TechEmpower* benchmarks.

*TechEmpower* makes different tests (single query, multi queries, fortunes, data updates, and more) by measuring several web application performances. We chose Fortune, which contains tests regarding the "*ORM, database connectivity, dynamic-size collections, sorting, server-side templates, XSS countermeasures, and character encoding*". [62]

The scores from *TechEmpower* for best performance were 12.381 for *Django*, 40.659 for *Express*, 5.484 for *Laravel*, 8.807 for *Ruby on Rails*, and 30.891 for *Spring*. Meaning, in top average score, the *Express* and *Spring* score is 2, *Django* score is 1, since it was far ahead of *Laravel* and *Ruby* on *Rails*, which we score with 0. [62]

*5.* ***Overall***

After completing a light evaluation and applying our criteria (*e.g., Table 2*), *Django* and *Express* got the same result. We proceeded with *Django* since it had a better average on *GitHub* and a more solid score on *Hotframeworks*. And because we need to build a generator, that must be easy to manage and change. So, we select the learning curve and fast development over performance.

It is important to emphasize that these scores only translate our criteria and that each framework has its advantages and disadvantages according to the problem and the development team requirements. Even though we did not perform a thorough comparison between the frameworks, this study was enriching, considering we could study and learn about each of the frameworks used nowadays.

Table 2 - Comparison criteria for the most used backend frameworks

| Frameworks / Criteria | Django | Express | Laravel | Ruby on Rails | Spring |
|---|---|---|---|---|---|
| Learning curve and fast development | 2 | 1 | 1 | 2 | 0 |
| Documentation and community | 2 | 2 | 2 | 2 | 2 |
| Security and reliability | 2 | 2 | 2 | 2 | 2 |
| Performance | 1 | 2 | 0 | 0 | 2 |
| **Score:** | 7 | 7 | 5 | 6 | 6 |

## 2.4. Database Management System

Following the previous section, we have chosen *Django* to implement our system. Now it's time to determine the appropriate *DBMS*.

Even though some people call *DBMSs* databases, they are different. A database is nothing more than organized information that is manageable. A *DBMS* is a computer program that allows the interaction with the databases by providing ways to create and manage them. [63]

*Django* has support for *DBMS* like *PostgreSQL*, *MySQL*, *SQLite*, and *Oracle*, to which it delivers detailed documentation. Besides them, it is also possible to use back-ends provided by 3rd-parties, that allow the usage of other solutions, such as *SAP SQL Anywhere*, *IBM DB2*, *Microsoft SQL Server*, *Firebird*, and *ODBC*. During this section, we will do a brief introduction to the main *DBMS* supported by the *Django* official documentation.

### 2.4.1. Overview

1. ***MySQL***

*MySQL* is the most used relational database management system (*RDBMS*). It was born in Sweden in 1995, is open-source under the terms of the General Public License (*GNU*), and owned by the *Oracle* Corporation. [64] Just like other *RDBMS*, it uses tables, constraints, triggers, stored procedures, and *views*.

Major organizations like *GitHub*, *NASA*, *Tesla*, *Netflix*, *WeChat*, *Facebook*, *Twitter*, *YouTube*, and *Spotify use it.*

2. ***Oracle***

*Oracle* Database is an object-relational database management system (*ORDBMS*), the development process started in 1977 which makes it one of the oldest *SQL* databases, and it is nowadays between the most used and reliable *RDBMS* [65]. *Oracle RDBMS-based* rests on its availability, robustness, storage optimization, indexing, and data integrity capabilities [66].

Well-known companies that practice this multi-model database are *CAIRN India*, *Capcom* Co., *Coca*-Cola *FEMSA*, *ENEL*, *MTU Aero Engines*, and *National Foods Australia*.

*3.* ***PostgreSQL***

*PostgreSQL RDBMS* is one of the prevailing open source systems in the world [67]. It has over 15 years of updates, runs on all major operating systems, contains programming interfaces for all major programming languages, underpins a large part of the *SQL* standards, and offers many innovative highlights, like, complex *SQL* questions, *SQL* sub-chooses, foreign keys, and triggers [68]. Even though is a relational database, *PostgreSQL* is an object-relational database, meaning that it also includes features such as table inheritance and function overloading.

Examples of companies that work with it are *Apple*, *IMDB*, *Macworld*, *Debian*, *Fujitsu*, *Red Hat*, *Sun Microsystem*, *Cisco*, and *Skype*.

*4.* ***SQLite***

*SQLite* is an embedded *SQL* database engine, it does not separate the server process, and performs the read and write to the cross-platform disk files. Even though it is free, it has perks such as a small library size, compiler optimizations, and regular tests. [69]

Examples of companies are *Adobe*, *Airbus*, *Apple*, *Bosh*, *Dropbox*, *Facebook*, *Flame*, *Microsoft*, and *McAfee*.

### 2.4.2. Considerations

There are many similarities and differences between these possibilities, but it is important to refer that we have already selected the architecture and framework we will work on, so we should decide considering them. Choosing an *RDBMS* is not as simple as picking the fastest one, there is a necessity to analyze what they offer, and the tools provided considering what best suits our scenario and specially the framework. There is a table on Appendix A, which translates the *DBMS* based on their official pages and the *Django* documentation.

We opt for *PostgreSQL* because of the *Django* documentation referring to the supported versions [70] and the migrations [71].

In terms of versions, only *PostgreSQL* and *SQLite* had support concerning their most up-to-date versions. *SQLite* is an excellent alternative for an applications that are mostly read-only [70].

In terms of migrations, the schema support varies among them. *PostgreSQL* is the most fit of all the *DBMSs*, being the only downside when adding columns with a default value, as there is the need to do a full rewrite of the table. *SQLite* does not have much built-in schema support, so *Django* must emulate it, and even if this process goes well, it can be slow or buggy. As for *MySQL*, there is a lack of support for transactions around schema alteration operations, which leads to manual

procedures towards the migrations. There is also a limit to the length of columns, tables, and indexes that can lead to failure in the migration process. [71]

## 2.5. Scaffolding

In 1976, Wood, Bruner, and Ross introduced the concept of *scaffolding*. They described it as a process that supports someone to engage in activities that would be beyond their unassisted efforts [72].

Even though this concept might lead people to think of the temporary structure that helps to construct or repair buildings, that is not the case. As *scaffolding* is essential for the duration of the constructions, the term *scaffolding*, and its ideology, it is also vital to assist people in other areas, empowering the acquirement of new skills, notions, or levels of understanding.

Several frameworks support this technique, which comes as a meta-programming method to build part of the software applications. These *scaffolding* methodologies work toward most of the described back-end frameworks. A well-known example of *scaffolding* are the controllers of *ASP.NET*, which include basic *CRUD* operation methods serving as the foundation to create more elaborate methods [73].

During this section, we will present different *scaffolding* methodologies, and in the next chapter, we will detail the ones related to the chosen framework.

Before we proceed with the search for each framework *scaffolding* solution, we look for repositories that might aggregate all of them together. The best repository we found was Yeoman. Yeoman is a great generic *scaffolding* system that helps people getting started on new projects and maintain existing ones. All the *scaffolding* notions revolve around generators, which are plugins in the Yeoman environment. These generators help to create projects, modules, packages, and so on. In the "*Discovering generators*" section, it is possible to search and retrieve the intended generators (*e.g.*, *Figure 7*). [74]

Even though at this point, we already selected *Django* as the framework that best fits our needs, it is a good practice to study and analyze what others offer to reach a more desirable solution. The idea is to gather information so we can practice similar *scaffolding* notions in our project. There will be two main *scaffolding* strategies that we will address for each framework, which are:

i. *Scaffolding* the base components of projects, like CRUD methods, and other operations, which helps programmers to have a head start;
ii. *Scaffolding* ready-to-use solutions, which only need few modifications.

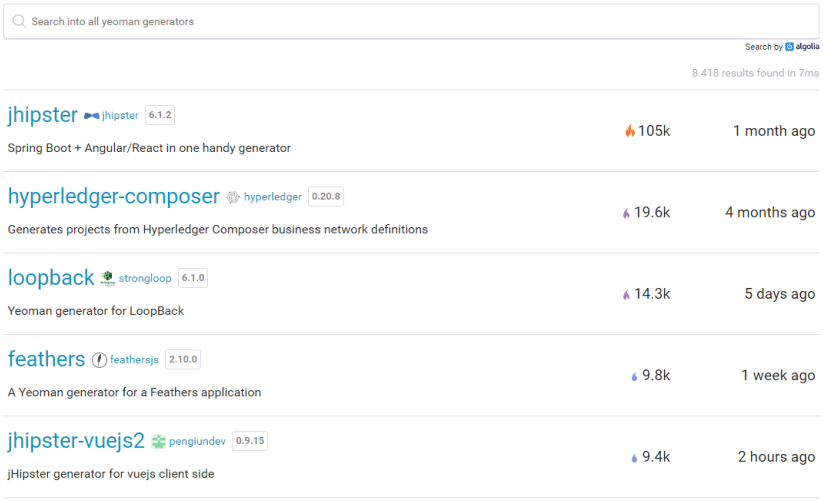



Figure 7 - Yeoman generators [74]

### 2.5.1. Express

1. ***Base Structure***

*Express* provides in its website a generator tool, the *Express-generator*, which can help to create fast application skeletons. This generator creates an application structure like *directories*, *routes*, *views*, and so on. [75]

As shown in the generator page link, it takes a few commands to generate the solution. We find this solution interesting since it is fast and provides a default project template with the help of the generators. We consider it only a base structure solution, since it provides the default template and not the control over the components of the database.

With this generator, it is possible to choose the engine (*dust*, *ejs*, *hbs*, *jade*, *pug*, *twig*, and *vash*), and a compilator of *CSS* (*less*, *stylus*, *compass*, *sass*) rather than the default plain *CSS*, and which we found interesting. The product of the generator is very appealing as it presents everything in an *app.js* file, builds the *package.json*, sets the style, and more. This solution is a helpful *scaffolding* solution, even though it is minimalistic.

2. ***Full Solution***

*Express-admin*, is built on top of *Hogan.js*, *Express*, *MySQL*, and *Bootstrap*. This solution comes as a tool to create a friendly administrative interface for *MySQL*, *MariaDB*, *SQLite*, or *PostgreSQL*. It is the best solution we found, that uses *Express* and provides a minimalistic admin to control data, which is translated by Figure 8. [76]

To use this interface, the user must configure *JSON* files after setting the project. There are a lot of necessary configurations depending on the database specifications, so there is the need to read and follow the documentation to achieve the desired solution.

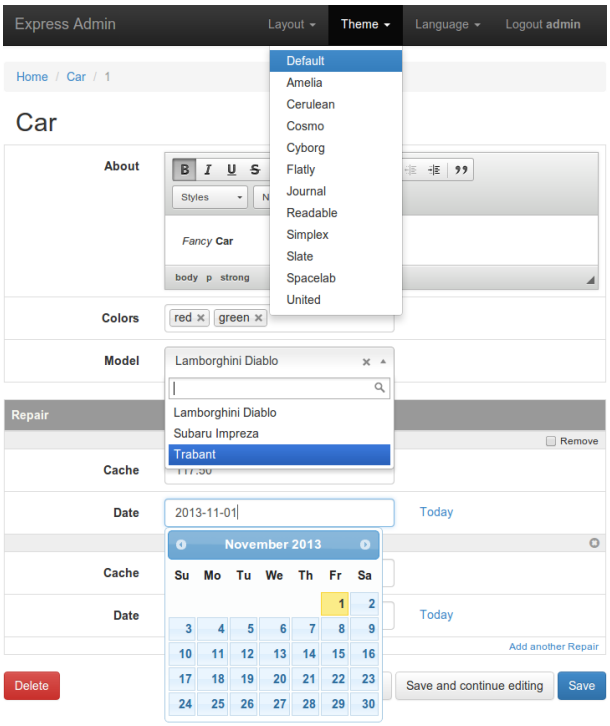



Figure 8 -*Express-admin* interface [77]

## 2.5.2. Laravel

### *1.* ***Base Structure***

It is possible to implement many functionalities through code generators using the *scaffold* property [78].

*Laravel's* previous versions had a library called *Laravel*-4-Generators. Its functionality is to provide a variety of generators to speed up the development process. [79]

Nowadays, the framework includes several generators out-of-the-box. These code generators come from the *Artisan* commands, which is a command-line interface incorporated in *Laravel* and provides direct support towards the application construction. [80]

Considering the current generators do not include everything, there is a new version called *Laravel-5-Extended-Generators* which include three additional methods (*make:migration:schema, make:migration:pivot, and make:seed*). [81]

From the current *Laravel* generators method and the additional *GitHub* Library, it is imperative to study and understand the purpose of each generator, as there are several generators in the

"*php artisan list*", with different keywords, that produce different *scaffolding* results. These generators aim to create not only the project structure but also some base code of each of the components. Meaning that the generators are amazing for composing a project structure, but they don't provide a complete back-end solution for performing *CRUD* and other operations. Another essential point to consider is the value of the chronological study we have made. Since in the beginning, *Laravel* did not include any official generators, and even after they became part of the official documentation, *Laravel-5-Extended-Generators* arrive to complement it. With this, we can conclude that there are always ways to improve *scaffolding* methodologies towards frameworks.

### 2. ***Full Solution***

For *Laravel's* complete solution generators, we have found two outstanding options, *InfyOm Laravel Generator* and *Voyager*.

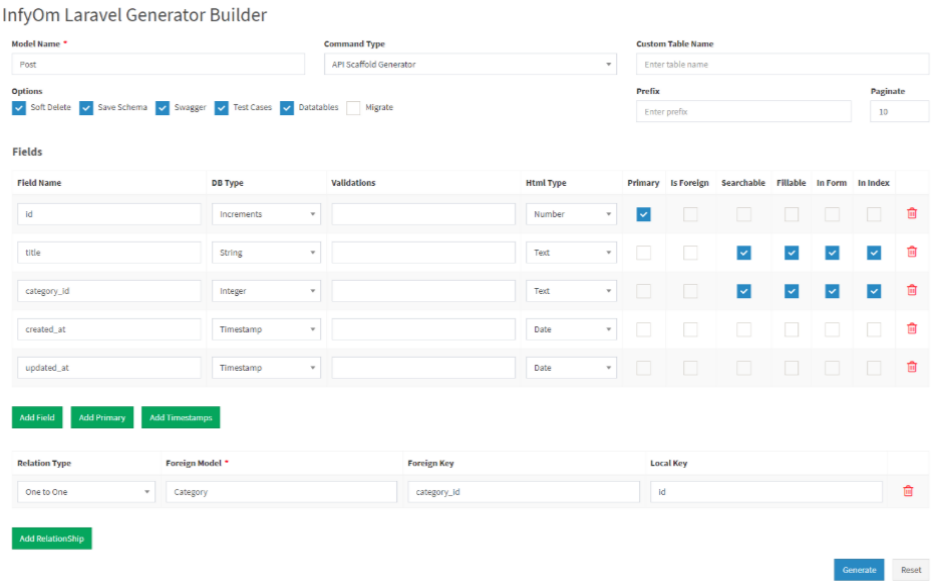



Figure 9 - *Laravel* Generator interface [82]

*Laravel Generator* provides an admin panel that is ready in minutes for *CRUD*, *APIs*, test cases, and *Swagger*. This generator possesses many parameters and options to choose from, but it does not abstract the database from the average user, requiring a broad knowledge of the database (*e.g.*, *Figure 9*). [82]

While *Voyager* is not so customizable, it is much easier to work within a user point of view as there is not the need to know in such detail the database constraints. *Voyager* provides *Laravel* with an Admin Package. This package includes ready-to-use *CRUD* operations and other functionalities (*e.g.*, *Figure 10*). [83]

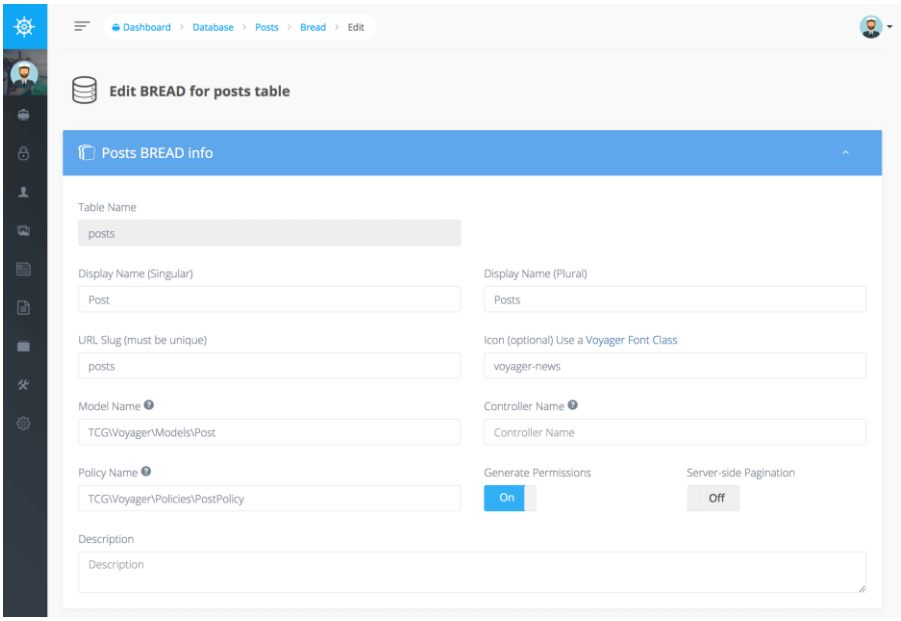



Figure 10 - Voyager admin interface [83]

### 2.5.3. Spring

#### *1.* ***Base Structure***

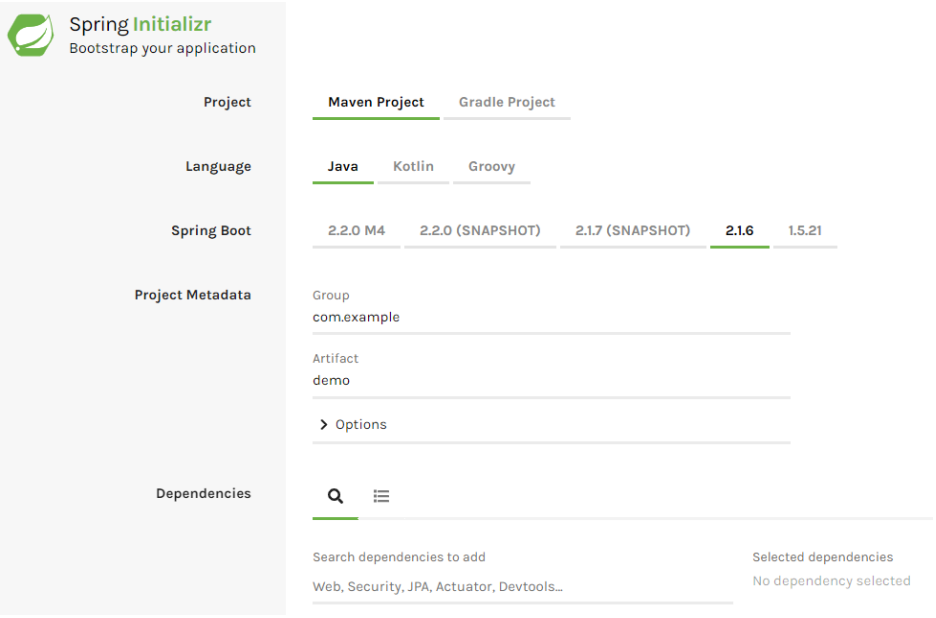



Figure 11 - *Spring* Initializr [84]

Considering up-to-date solutions, we came across *Spring Boot*. This *Spring* extension facilitates creating runnable *Spring-based* applications. It eliminates the need to make configurations for setting up a *String* project. [85] With the use of *Spring Initializr*, provided by *Spring*, it is even possible to

create *Spring Boot* projects. There is only the need to provide project details, like the project, language, *Spring Boot* version, and Dependencies. The dependencies can become handy, as it is possible to select them and add to the *Spring Boot* Project (core, web, template, database connection, cloud, and more). After setting the right requirements for the project, it is possible to download a *zip*, which contains a *Spring* boot project based on those details (*e.g., Figure 11*). The output project only depends on *Java*, and it can be the basis to produce exceptional structured standalone *Spring* applications. [84]

### 2.5.4. Ruby on Rails

*1.* ***Base Structure***

This framework allows generating interfaces related to controllers with *models* marked by the *scaffolding* keyword [78].

In *Ruby on Rails'* official guide, it says, "*Scaffolding is a quick way to generate some major pieces of an application. If you want to create the models, views, and controllers for a new resource in a single operation, scaffolding is the tool for the job.*" The *scaffold* creates all the *controllers*, *models*, and *views*, which can be mold to the system requirements. [86] To use the *scaffolding* functionality provided by *Ruby on Rails*, it is necessary to follow a group of steps provided by the official page documentation.

It is crucial to keep in mind that this *scaffolding* method presents different ways to generate the application by providing additional control parameters.

*2.* ***Full Solution***

This framework has several ready-to-use admin interfaces to perform many actions. There are admin interfaces like:

i. *Activeadmin*, is the central administration solution for *Ruby on Rails*;
ii. *Administrate*, which is alike the existing admins but tries to provide a better experience for site admins and facilitates customization;
iii. *Rails_admin*, provides an easy-to-use interface for managing data;
iv. And there are plenty more like *active_scaffold*, *trestle*, *forest* and so on.

Since there are plenty of possibilities, we will go for the main used admin, the *activeadmin*. This admin brings lots of functionalities which we can take notes when building ours. It provides global navigations, scopes, index styles, content downloads, user authentication, *CRUD*, and filters. We could go broader on the possibilities, but the main points that we want to highlight are the capabilities of the platform. [87]

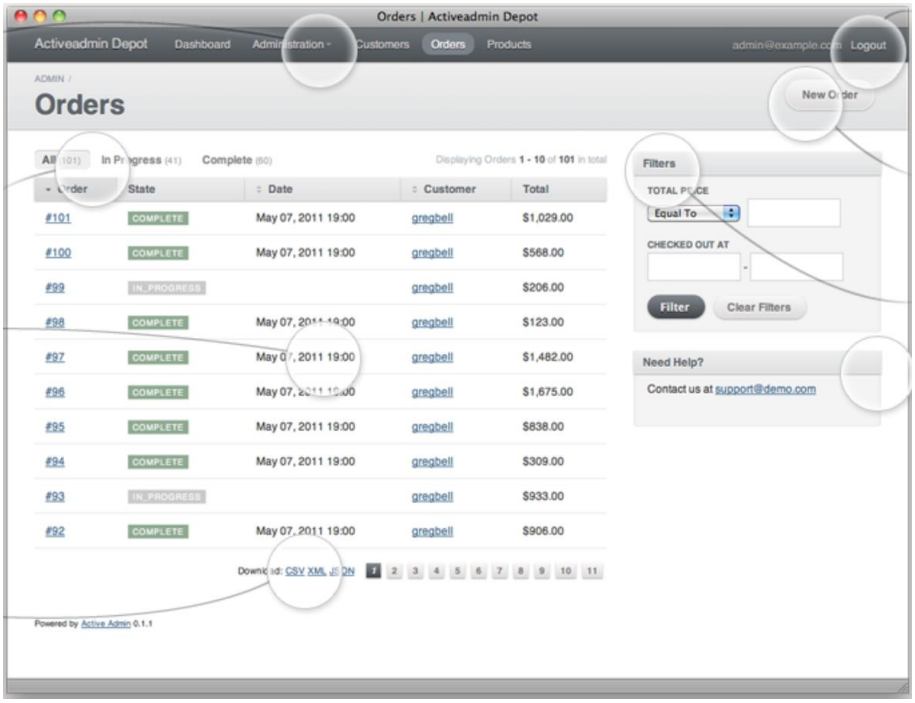



Figure 12 - activeadmin interface [87]

## 2.5.5. Considerations

Based on the framework *scaffolding* examples, we can see the value of *scaffolding* to speed up the process of development as they already have internal or external mechanisms to do that job. It is essential to take into consideration that the *scaffolding* method can work towards specific elements of the projects as it is used to create whole solutions or project structures.

From the project's setups, and in a final user point of view, we can retrieve three core ideas that can be helpful to build our application and run it, which are:

i. The project must be easy to set up, meaning that the project structure, the dependencies, the connection to the database and the migrations need to be straightforward;

ii. It is necessary to provide ways to control and select what we want to *scaffold*, including the control of the *model* itself and the application functionalities;

iii. The platform should include global navigations, content download, user authentication, CRUD, and so on. We must investigate the required functionalities, which will take place in the next chapter.

Following, we will do a more profound review concerning the *Django Scaffolding* solutions. To produce our solution, we intend to use the compiler specifications to generate the code and build all the solution components. That is part of the reason we study what already exists, to understand what we can improve, include, and to retrieve further core design ideas.

## 2.6. Django Scaffolding Solutions

Before talking about other *Django Scaffold* solutions, it is crucial to point out that *Django* offers an admin interface, a powerful tool that reads metadata from the *models* to provide a swift model-centric interface to manage *model* content. Even though is customizable, it does not provide an interface that abstracts away the implementation details of the database tables and fields towards a user perspective. This admin interface is not straightforward for regular users. There is a need to understand the underlying database. As stated in the documentation: "*The admin has many hooks for customization, but beware of trying to use those hooks exclusively. If you need to provide a more process-centric interface that abstracts away the implementation details of database tables and fields, then it's probably time to write your own views."* [88]

There are several admin options already built to improve the official one, like admin interfaces, admin addons, and admin styling.

During this chapter, we will start by analyzing the *Django* admin and its additional admin libraries regarding the interfaces, addons, and styling. Moreover, we will point out the *scaffolding* libraries that help to conceive an admin like solution.

### 2.6.1. Django admin

The admin interface exists by default when starting a project (*url:/../admin/*) and requires superuser credentials to access it. To interact with the objects of the database, first, there is the need to register the *models* (*e.g., admin.site.register(entity)*) [88]. Over the past few years, the interface, built-in *CRUD* operations, and filters have improved. Figure 13 shows one example of a database entity listing and the possibility of using search, add entries, and filter the data.

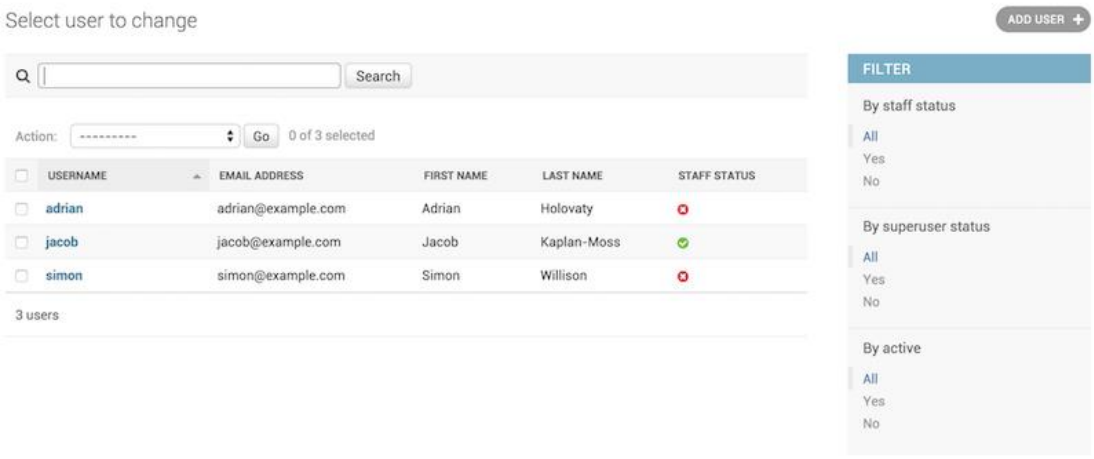



Figure 13 - *Django* Admin Interface List [88]

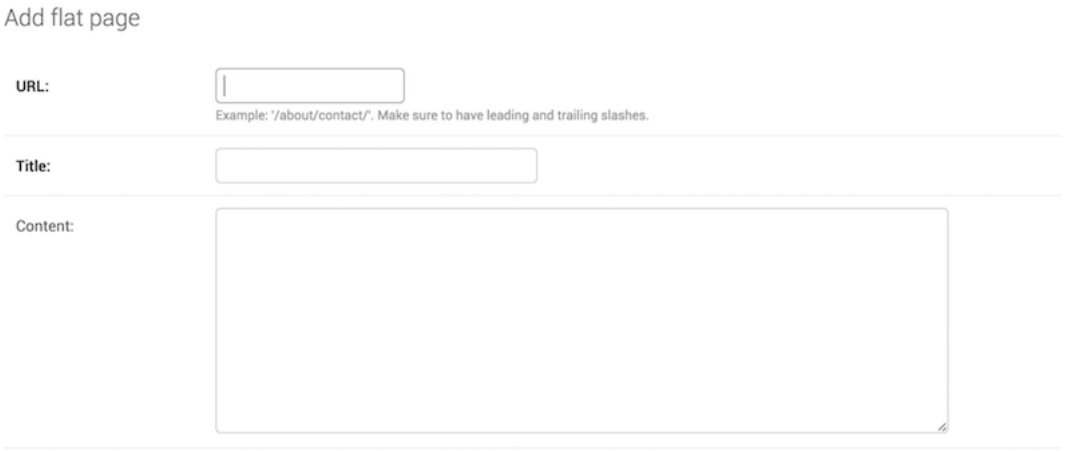



Figure 14 - *Django* Admin Interface Add [88]

When testing *Django* admin, it is possible to check the advantage of this ready-to-use platform. This administration interface is perfect for fast *CRUD* control. However, it has limitations to fulfill our goals, like the little control over the feature set or content. It is necessary to introduce new features when creating a back-end application, there is a need to: change the code to place the components on the *Django* admin; the final user must understand the model structure to perform actions; and there is a lack of satisfying data displays.

Overall, it is hard to beat this improved solution, as it is perfect for programmers who know how to use it. We strive to create a solution that can provide a full manager admin-like platform out-of-the-box, with easier navigation for regular users, proper data displays, and which is straightforward for programmers to apply further embedded business logic.

### *1. Admin improvements*

There are several solutions for *Django* admin improvements, including the interface and styling *Django* packages. They are similar and have the aim to improve the *Django* admin.

The *Django* admin design is not so appealing, but there are many *Django* Interface Packages available to improve it. They focus on exploring and customizing the interface, which includes the logo, the colors, the titles, the pop-ups, and so on. [89]

There are countless alternatives, one hundred and two, so we will point the standard and most used ones. They are the *Django*-Grappelli[3] and *Django*-Autocomplete-Light[4].

The Styling Packages follow the same scheme and include most of the packages found in *Django* Interfaces [90].

[3] *https://readthedocs.org/projects/django-grappelli/*

[4] *https://readthedocs.org/projects/django-autocomplete-light/*

2. ***Admin addons***

There are several Admin Addons, but *django-admin-tools* is the most used among the possibilities. These tools aim to provide improvements, mods, or utilities to the *Django* default administration area. [91]

### 2.6.2. Scaffolding libraries

There are *Django Scaffolding Libraries* to aid in developing projects. The most used are the *django-crudbuilder, django-generic-scaffold, django-popupcrud, django-cruds* related to the quick generation of *CRUD views, django-naqsh* that is a bootstrapping tool to create production-ready web services and *snipty* to track snippets. The *CRUD* libraries also grant another functionalities out-of-the-box and the possibility to set variables that define how to generate and handle the code, so the developers only perform the needed adjustments. [92]

1. ***django-crudbuilder***

*django-crudbuilder* is the most complete library, it has requirements for *Django* 1.10+, *Python* 2.7+ or 3.3+ and *django-tables2*. This package provides several features beyond the class-based *views* for *CRUD*. It applies *django-tables2* to present the objects in the *templates*, defines the several required *URLs*, enables/disables permissions for each *CRUD view*, allows defining *querysets* for each listing *view*, all its *views* are extensible and uses default *bootstrap3*. Besides, there are control parameters related to each *model*, which allows setting the fields for listings, search, login requirements, and content access permissions. The installation is easy to perform, first, there is the need to install the application and add it to the *INSTALLED_APPS*. After there is the need to include the *models*, the *URLs* and to define the parameters for each *view*. [93]

2. ***django-generic-scaffold***

*django-generic-scaffold* has requirements for *Python* 2.7 with *Django* 1.6-1.11 or *Python* 3.5 with *Django* 1.8-2.1. It allows creating *CRUD views* for each *model* only with introducing some lines of code and defines all routes for each *view*. However, it has limitations: there is a need to set the *forms* and *views* associated with the updated content, set the access permissions for each *view*, and create the *templates* where each *view* will render its data. [94]

3. ***django-popcrud***

The *django-popCRUD* has requirements for *Python* 2.7 or 3.4 with a *Django* version higher than 1.8. This library has the particularity of implementing the needed *CRUD* operations through *HTML* pop-ups. In parallel with *django-crudbuilder*, it gives the possibility to define parameters in each *view*, such as the display fields, listing fields and the *URLs* for each type of operation. [95]

*4.* ***django-cruds***

*django-cruds* has requirements for *Python* 2.7+ or *Python* 3.4+ and *Django* greater than 1.10. It provides an easy way to generate *CRUD views* by adding functionality to each *model*, and *URLs* by adding a few lines of code into "*urls.py"*. Unlike the others, it does not have control parameters. [96]

### 2.6.3. Considerations

It was essential to study the existing libraries as they allow us to verify the needed functionalities and procedures to conceive these *scaffolding* methodologies. We intend to implement the main features we have collected, such as the *URL-building* mechanisms for each *CRUD* operation, built-in role-based access control, and various management parameters for each *view*. Concerning the control parameters, there will be a focus on managing the login and the user permissions associated with each *view* per user type, control the listing fields, search fields, queries, several entries per table and the *CSS* of these tables, and control the display fields.

In the next chapter, we perform the system requirements to validate our needs. The next Table 3 translates the main *scaffolding* features of each of the *Django scaffolding* packages available in the market.

Table 3 - Django Scaffolding Comparison

| **Packages** / **Specifications** | **crudbuilder** | **generic-scaffold** | **popcrud** | **cruds** |
|---|---|---|---|---|
| Automatic CRUD creation | ✓ | ✓ | ✓ | ✓ |
| Imbued Login | ✓ | ✓ | | |
| User content Permissions | ✓ | ✓ | | |
| Table fields | ✓ | | ✓ | |
| Table pagination | ✓ | | | |
| Table CSS | ✓ | | ✓ | |
| Search fields | ✓ | | | |
| Display fields | | | ✓ | |
| Automatic URL creation | ✓ | ✓ | ✓ | ✓ |

## 2.7. Summary

After the topics discussed in the literature review, we can now take several ideas to produce our solution.

From the studied *CMSs*, we can conclude that although they offer all the tools to create a great solution, they have constraints that we will not find in a solution built from scratch, creating our

solution empowers flexibility and comprehensibility, which provides us with total control of our result. Furthermore, *CMSs* do not present mechanisms to produce management platforms out-of-the-box, and sometimes, implementing business logic to the solution can be challenging.

First, we start by selecting the right software architectural pattern among a list of the most used. We adopt for *MVC* since it provides the means to manage databases, with system-wide modifications that are easy to maintain and to reuse. And because it is the base architecture for most of today's web frameworks.

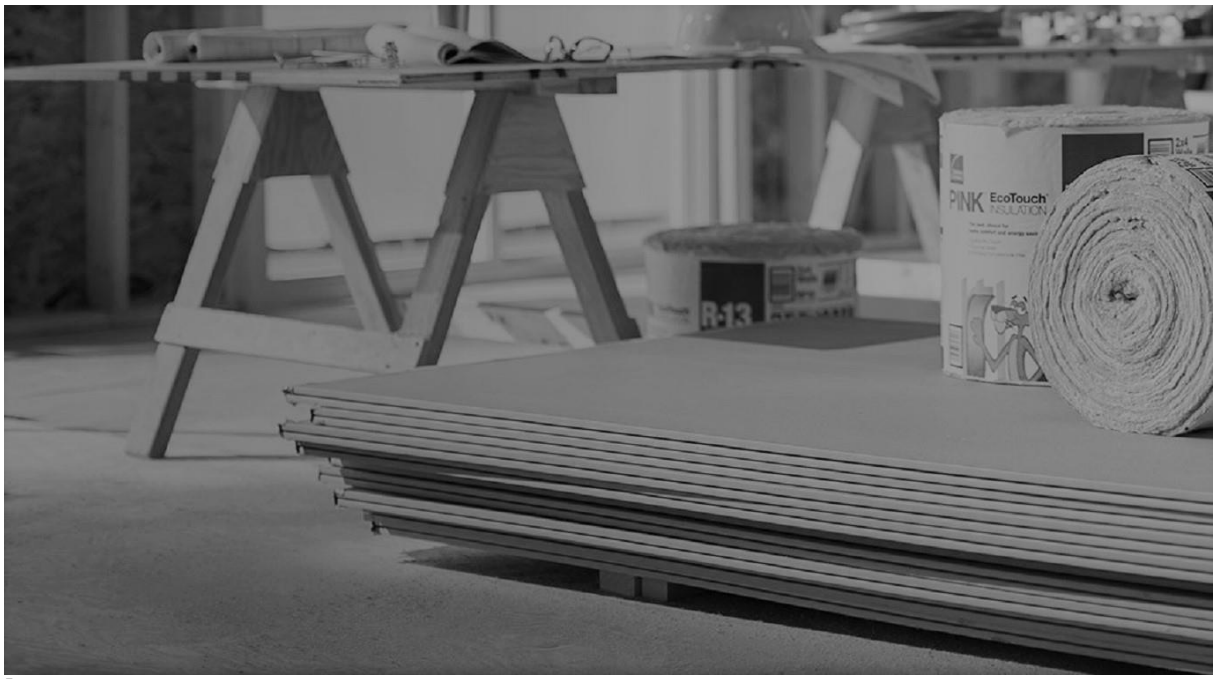
We have done a brief introduction to each of the chosen frameworks and select the one that best fits our needs. Even though *Express* had a better performance overall, we favor *Django,* because it is easier to develop with, and it has more out-of-the-box functionalities, being the dominant point for our settlement.

Based on the *Django* documentation and the *DBMSs* that comply with the framework, we choose *PostgreSQL*. This *DMBS* has support for its up-to-date versions, and in terms of migrations, it was the fittest of all the other possibilities.

We study the value of the *scaffolding* concept towards the framework development. By studying the *scaffolding* methodologies for the addressed frameworks, we understood crucial points and how to implement them. With this, we retrieve core ideas that will be helpful to build our application and run it.

We focus on the chosen framework and study all the possibilities available on the market. There is the *Django* admin, where its interface does not meet our demands. We can use this platform for back-end management but is not intuitive enough for our end users. It would be arduous to implement new sets of business rules and customize its *template,* then developing a new application. Besides, it has other limitations, such as the lack of satisfying data displays and the poor relationship context between data. That context is essential for the end-users of the system, to understand how the entities of the database relate, and to ensure the proper data input. Despite the *scaffolding* library's assemblage, we could verify that these libraries do not meet all the needed functionalities. There is the need to perform extra tasks associated with the *CRUD* operations, build dynamic relationships between different entities, and *scaffolding* for a complete back-end solution. In conclusion, we need a solution with more functionalities than the *Django scaffolding* packages and that it looks like *Django admin,* but with a structure that targets specific end-users.

# 3. SYSTEM REQUIREMENTS ANALYSIS



[5]

*In the previous chapter, we tested similar solutions while gathering and building our system requirements. This chapter describes the solution requirements*.

## 3.1. Challenges

With this project, we intend to build a dynamic solution that follows *Django MVT* architecture, creating complete solutions based on *model* interpretation to fit established business rules. Using *scaffolding* mechanisms to produce code becomes crucial to achieve our expected goal. It is imperative to design a solution that combines the mechanisms of *Django* admin and the existing *Django scaffolding* packages, but regarding our specific solution.

The project has two distinct parts:

1. Regarding the users of the application, we must design an engaging platform for users who might not know the system but who will perform all its operations. We expect them to manage database records and to perform other actions as well;

[5] https://content.ikon.mn/news/2019/4/17/8fcce8_building-materials-heros_x974.jpg

2. We want to automate the creation of different solutions regarding distinct data *models*. It is also essential to facilitate the implementation and adjustments of project modules. We will center our focus on the modularity and dynamism of the solution architecture, so it is always possible to introduce new business rules. Therefore, we expect developers to perform modifications to the solution modules, if needed, without implementing significant changes.

After performing the literature review and the study towards other solutions, we established the specifications. The challenge is to develop a system that meets the two points mentioned.

We select the operations to include in our system. We start by thinking about the requirements that can integrate this platform. Afterward, we decide on the components we want to embed in the solution and how to expose them to the users.

In this chapter, we will discuss the functional requirements, the nonfunctional requirements, and the use cases.

## 3.2. Functional Requirements

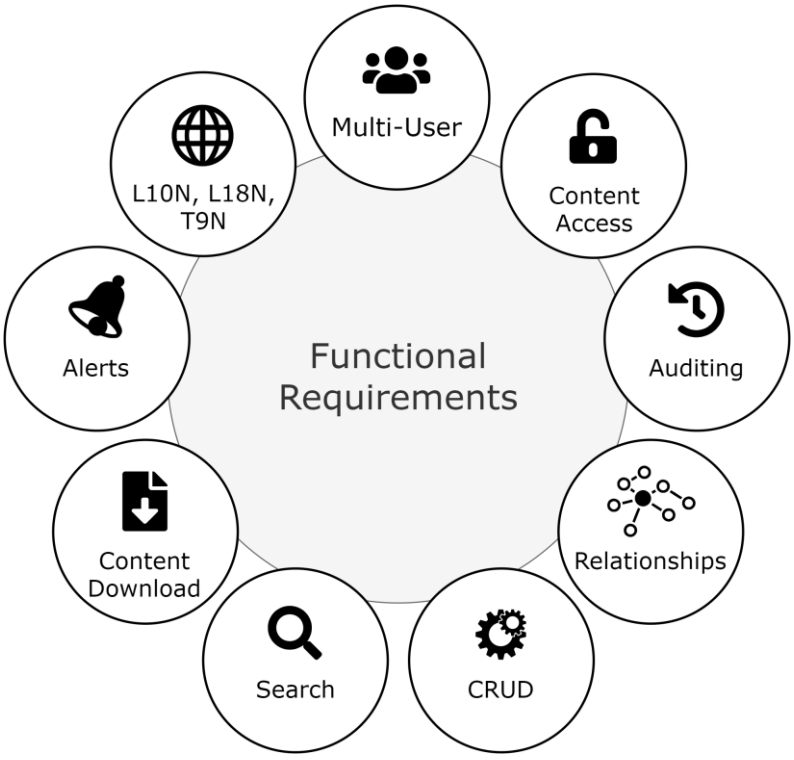



Figure 15 - Functional Requirements

After discussing the requirements with the dissertation advisors, we selected those that we found most interesting to deliver on the back-end platform generator product. The following topics describe the requirements gathered:

1. ***Multi-User***

The system must be multi-user, meaning that, by default, it must allow access by multiple users at the same time. Management systems are usually for various users, with one or more administrators and an end-user community. The administrators have all the control towards what the system offers, while end-users, which we call staff, use the standard functionalities of the system. This system design will take into consideration that staff rarely possess a technical understanding of the product design.

Only registered users can access the system. Users must perform authentication procedures by supplying user identification and password.

The system will have two user roles: staff and administrators. One or more system administrators are entitled to create staff accounts. Staff will only be able to change the password by email confirmation if needed, while administrators can create, change, delete, suspend or deactivate user accounts.

2. ***Content access***

The system must include different permissions, which translate into different access towards application content, separating the administrators from the staff. In the chapter's sequence, we will distinguish the various system interactions performed by the staff and administration members.

All the users will have a status. Only active users can access the system, ensuring that suspended or deactivated users lose access to system functionalities while keeping their user logs intact for their reactivation.

3. ***Auditing***

We address auditing like the process of tracking changes and maintaining their logs in the reach of our platform.

With auditing, for each database entity, we can correlate current records with their previous modifications. This feature adds value to the platform since it enables the control of the actions performed while it allows retrieving and comparing previous database record versions.

Only administrators must have access to this feature.

4. ***Entity Relationships***

Dynamic entity relationship navigation is one of the main functionalities that our platform should address. It is necessary to relate entities so that the system can provide users with mechanisms that allow navigation among different entities while keeping the context between them. If a user is checking for an entity record that relates to others, these relationships should be present as well as their number of occurrences.

Another way to promote content contextualization is by using Breadcrumbs. Their proper use can enhance the user experience, allowing the track of actions performed when browsing through the platform. Websites with large hierarchical structures broadly use this method. They can have some disadvantages, such as the possibility of becoming confusing or clumsy. But they bring more advantages, like facilitating navigation, improving user experience (*UX*), reflecting the hierarchy of the website, and are visually attractive.

5. ***Create and Manage Entities***

This requirement relates to quick and intuitive content handling, such as listing, adding, changing, viewing, deleting, or updating entries. It is essential to contextualize the *CRUD* operations. When a user is performing the add or edit operation, if there are previous associations, like stated at the *Entity Relationships*, all the related inputs must fill accordingly. We will detail this process during the fourth chapter. This functionality leads to fewer errors, as it helps users by filling the fields automatically.

All the operations must be validated, meaning that when adding or editing entities, each field must have associated rules depending on the field type. This ensures that the user respects the database structure, and there are no input inconsistencies.

6. ***Search***

Search inputs must match the parameters listed and viewed by users at the listing interfaces. Therefore, this validation works only toward the displayed parameters.

7. ***Content download***

Each user must be able to download content in the form of *PDF* or *CSV*. The *PDF* contemplates all the displayed data of a record. The *CSV* download enables users to extract a listing from the system. It must be possible to retrieve all information from a database item and expose it to a spreadsheet, if needed. Both functionalities are essential to extract data from the application.

8. ***Alert system***

The alert systems provide contextual feedback with messages toward user actions. These alerts follow proper styling and use different colors as a conveying purpose. Displaying these messages with a fitting color ensures that the content is understandable by itself.

Our system must contemplate alert mechanisms in required areas that will help users understand specific scenarios. This approach is essential in every kind of system for feedback reasons.

9. ***Internationalization and localization***

Internationalization (*I18N*) is the process of preparing software for different locations, and developers usually do it. It passes by marking the right parts of the application to translate or format.

Localization (*L10N*) uses the internationalization so that it is possible to place the current translations (*T9N*) and set the local formats. Subsequently, when a user selects a language, all the marked content changes accordingly.

Both internationalization and localization are necessary for today's era because they provide the means to change content regarding the audience at stake.

The system must incorporate the most used languages and formats, delivering a tool for global use.

## 3.3. Non-Functional Requirements

1. ***Role-base Access Control Policies***

   Each user must have a role that translates into different system privileges.

2. ***Easy to deploy***

   Easy to deploy means that the system and all the activities for it to be available for use must be easy to set up. Deployment comprises several processes that change according to the system design, which leads to specific requirements and characteristics. It also requires several steps, like the release, installation, activation, updates, version tracking, deactivation, and remove.

   Our platform design process must consider all those processes. Therefore, we must design it with a simple and manageable structure to deploy in multiple environments.

3. ***Support modern web***

   This system is designed to be used in desktop environments, as it is a back-end management system. However, the application must also work in mobile scenarios. Therefore, we should include modern web components and practices, having in mind current browsers, such as Chrome and Firefox, that keep on improving and supporting *HTML 5*.

4. ***Software Quality Assurance***

   Quality Assurance (*QA*) is of most importance in today's software development, as it ensures that the application is being tested and is ready for release. Testing the modules against every scenario is arduous and not bulletproof. Automatic testing can be beneficial but also complicated, so conducting tests by the developers of the application is usually the most desirable option.

   In solution development, we will test all the modules to ensure they respect their purpose and work properly.

## 3.4. Use Cases

Use Cases (*UC*), displayed in Figure 16, document how users (actors) interact with the system to achieve a goal.

We defined the actors of the systems based on the system requirements. There will be administrators with full permission to the platform content and that can manage the staff. Staff will have permission to change and retrieve the information marked as non-administrator. Deactivated users can't perform any actions towards the system.

The user stories gather are:

1. **UC1:** Users can restore and redefine their password via mail confirmation;
2. **UC2:** Users can change the system language;
3. **UC3:** Users can manage entities and their *CRUD* operations marked as non-administrator;
4. **UC4:** Only the administrators have the permissions to change marked core system *CRUD* operations. For instance, when an item has administrator privileges, only the administrator has the power to access it and change it;
5. **UC5:** Users can sort the listings by the respective parameters or use the default sorting query;
6. **UC6:** Users can retrieve and search for non-administrator entity records. This functionality is essential when the entity holds a high number of records;
7. **UC7:** Once in a listing, users can extract the entire records in the form of a *CSV* file. It is convenient when exporting data for external usage;
8. **UC8:** Once in a display *template*, users can extract the record information in *PDF*;
9. **UC9:** Administrators can see the underlying database diagram icon. This approach gives the administrators the feasibility to check the actual database and all its structure in the *template* by presenting the database structure;
10. **UC10:** Auditing is useful for administration. Only the administrators should have the possibility to access a list of entity changes;
11. **UC11:** Following the auditing, the comparison comes as a version control that allows the administrators to see the changes made to the selected listing records;
12. **UC12:** Administrators can manage all users inside the system. It is essential as it allows admins to change user records and to create new ones.

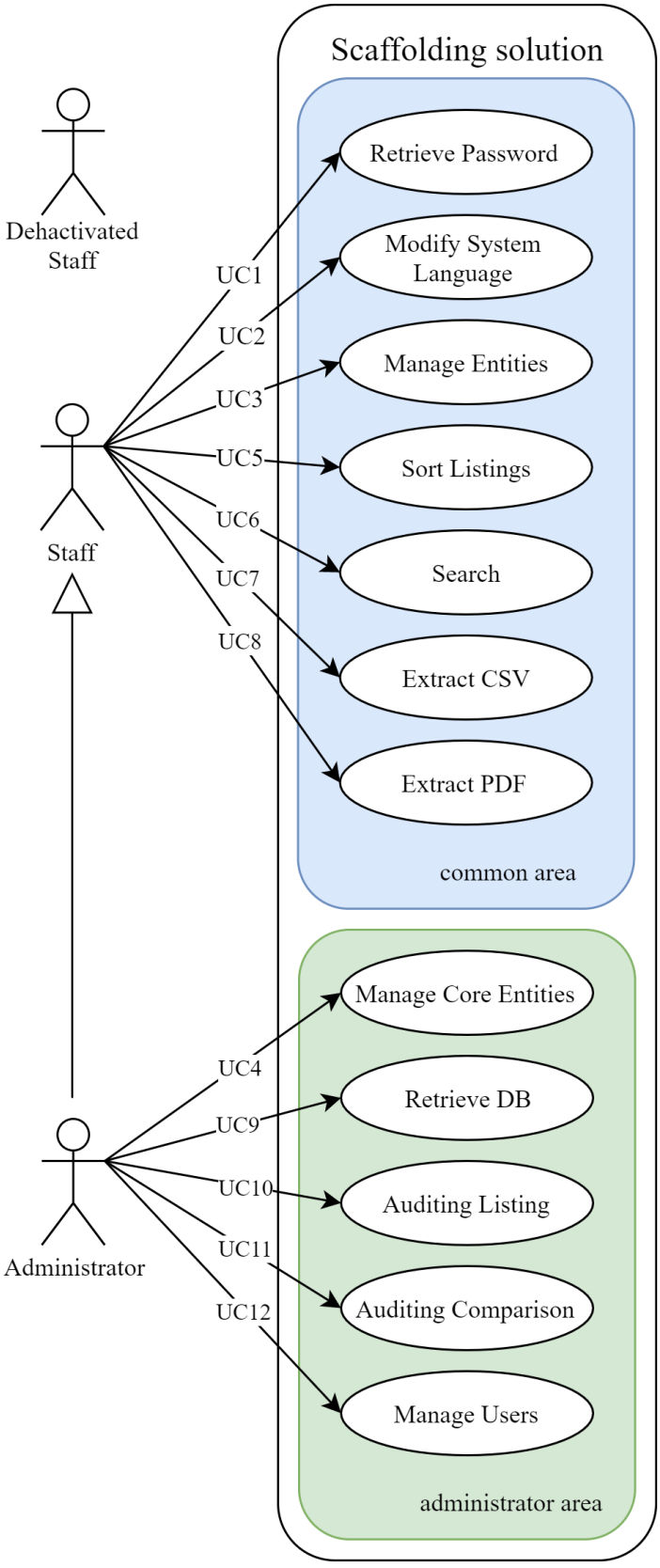



Figure 16 - Use cases

# 4. THE SYSTEM IMPLEMENTATION

[6]

*In this chapter, we present the system implementation. We detail the adopted technologies, perform a system design overview, explain the MVT structure, expose our solution modules, describe the script that generates the missing parts of our back-end solution, and we provide a summary that correlates all chapter components.*

## 4.1. System Technologies Overview

To achieve a desirable solution, we used several technologies throughout the project development. In this subchapter, we will present an overview of those technologies. The *IDE* (Visual Studio+*PTVS*), the version controller (*TFS*), the *DBMS* (*pgAdmin*), and those used by the *Django* framework itself (*PIP*, *Bootstrap*, *CSS*, *HTML*, *JS*, and *FontAwesome*).

[6] *https://agcks.org/images/landing-page/construction-industry-building-up-kansas.jpg*

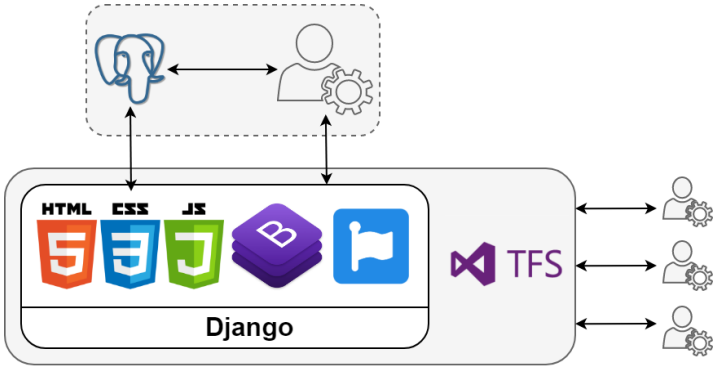



Figure 17 - System Overview

1. ***Visual Studio***

For the development process, we used the Visual Studio *IDE*, and *PTVS*[7]. *PTVS* is an open-source plugin that allows programmers to use Visual Studio as a *Python IDE*. It endorses a lot of functionalities, such as browsing, debugging, IntelliSense, *Django* development, among others. Microsoft and its community ensure the development and support of this plugin.

2. ***TFS***

Team Foundation Server (*TFS*) is a version control product provided by Microsoft. The development team that used this project imposed the use of *TFS*. The *TFS* covers the entire application life cycle, enabling all the development and operation capabilities.

3. ***pgAdmin***

We used *PostgreSQL* as it seemed to be the *RDBMS* best suited for our purpose. Besides, during the development process, we also used pgAdmin.

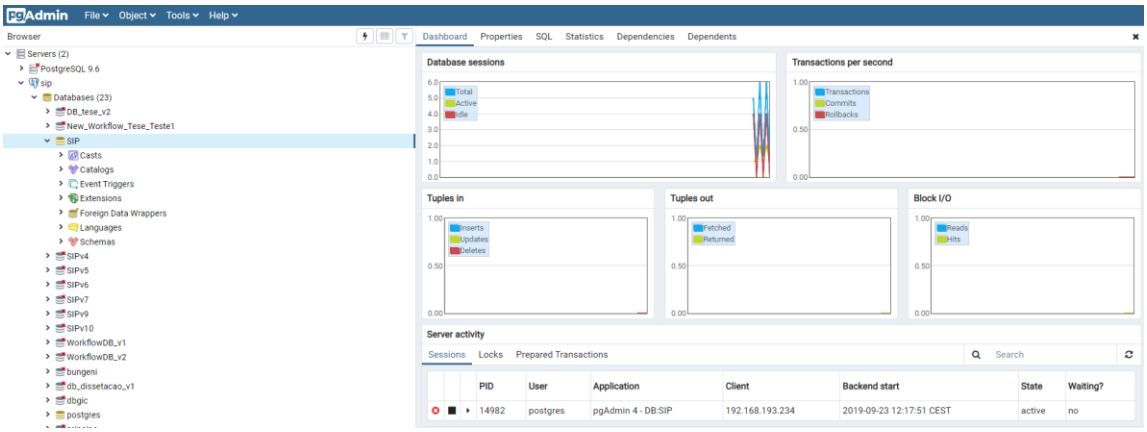



Figure 18 - pgAdmin interface

[7] *https://microsoft.github.io/PTVS/*

PgAdmin is the most used administration and development platform for interacting with *PostgreSQL*. This platform was crucial to perform quick queries and testing the database, as it provided an appealing and intuitive interface.

### *4. Django*

As already mentioned, all the project will be around the *Django* framework, with the version 1.11LTS supported until April of 2020. We did not use *Django* 2.2LTS, considering we started the project before its release, but we will migrate the project as future work. Figure 19 describes the *Django* version and its support.

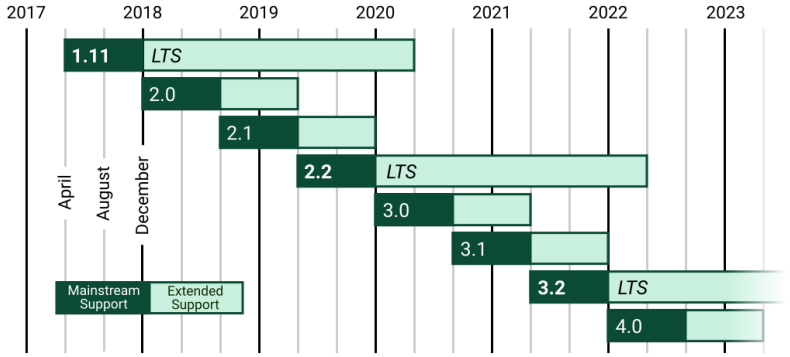



Figure 19 - Django latest versions [97]

### *5. Pip modules*

```
amqp==1.4.9
anyjson==0.3.3
billiard==3.3.0.23
celery==3.1.25
comtypes==1.1.4
coreapi==2.0.9
Django==1.11.18
django-braces==1.11.0
django-celery==3.2.1
django-extensions==1.7.5
djangorestframework==3.5.3
django-rest-swagger==2.1.0
django-simple-history==1.9.0
docx2txt==0.6
graphviz==0.10.1
itypes==1.1.0
kombu==3.0.37
openapi-codec==1.1.7
pdfkit==0.6.1
Pillow==4.0.0
pip==19.1.1
psycopg2==2.6.2
pydotplus==2.0.2
pyparsing==2.4.0
PyPDF2==1.26.0
pytz==2016.10
reportlab==3.4.0
requests==2.12.1
setuptools==39.0.1
simplejson==3.10.0
six==1.10.0
unicodecsv==0.14.1
uritemplate==3.0.0
xmltodict==0.10.2
```

Figure 20 - requirements.txt

*Django* has the imbued Pip Install Packages to include all the needed external modules that we use for the different required functionalities. We create a *requirement* file in which we include each

module and its version (*e.g., Figure 20*). Then, we create our virtual environment on the Visual Studio, which uses the file to import all the needed modules.

We see it as a good strategy since it overcomes future dependency conflicts when deploying the project. Developers create virtual environments that contain a copy of a specific interpreter, so when they activate a virtual environment, the project will only have the packages installed on that sub-folder. [98]

*6.* ***Bootstrap 4, HTML and CSS***

Regarding the interface, we used the latest *Bootstrap 4*, which is an open-source framework for developing responsive projects on the web. The *HTML* is the standard markup language for our web pages, and the *CSS* specifies our styles, like page layouts, colors, fonts, and so on.

*7.* ***JavaScript***

*JS* is a scripting or programming language that enables the creation and control of dynamic content. We use *JS* to create *PDFs* out of the *HTML* content, to implement a different data picker, to edit text area inputs (*TinyMCE*), among others.

*8.* ***Font Awesome***

We use icons as a fundamental approach to help users recognize patterns when applying the same to similar actions. For example, when a user wants to edit a record, the button will always contain the same pencil icon. Font Awesome is one of the most used open-source web libraries. One of the best advantages is the scalability of the icons.

## 4.2. System Design Overview

To give an initial insight, so that the reader understands the processes and how the next chapters and subchapters relate, we will make a brief introduction, considering the final product.

Based on the *Django* architecture (*e.g., Figure 21*), it is possible to understand the structure of the framework. Following, we describe our methodology for relating the *MVT* components (e.g., *Figure 22*). After understanding how to connect the objects to facilitate navigation throughout the application, we developed the features for each of them (*e.g., Figure 23*). Besides the essential operations (e.g., adding, deleting, listing, etc.), we highlight the relationship mechanisms (*e.g.,* Figure 24*, Figure 25*).

After designing the system operations, we created the solution components. We start by defining the structure of the *URLs* (*e.g., Figure 26, Figure 27, Figure 28*), which indirectly translate the features and actions of the system. We created the *views*, with the main *home view*, which is introductory and the starting point to access the other entities of the system, by rendering information regarding each of them (*e.g., Figure 30*). Subsequently, the solution instantiates *views* that contain the

respective control parameters (*e.g., Figure 32*). Each of those *views* (*e.g., Figure 33, Figure 34,* Figure 35, *Figure 36, Figure 37, Figure 38, Figure 39*), call the corresponding *template* that follows dynamic methodologies depending on the context (*e.g., Figure 41, Figure 42, Figure 43*), and where we implement further custom *tags* and *filters* (*e.g., Figure 44)*.

We can conclude that given the relationship mechanisms, and the correct structuring of *URLs*, *views*, and *templates*, it is possible to design the components to work regardless of the imposed data model. That must be supported by the help of a generating script (*e.g., Figure 45*) to produce the missing pieces of the puzzle.

With this overview, it is difficult to understand all the processes, but we think it is crucial to contextualize the reader before talking about each component. This chapter will support the reader through the remaining chapters by promoting a sense of interconnection among them.

## 4.3. App structure

It is necessary to understand the *MVT* architecture, to learn the basics of the *Django* framework and how its components interact.

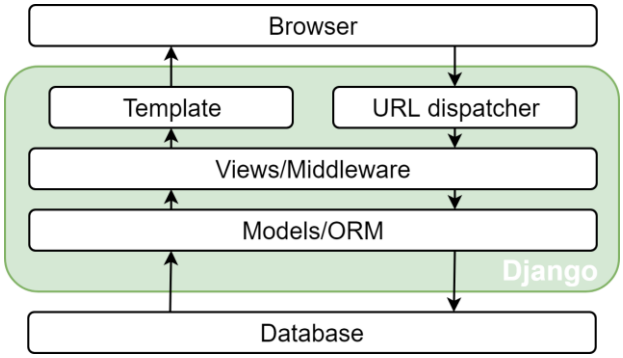



Figure 21 - *Django* MVT architecture

As shown in Figure 21, the *MVT* architecture holds three main components, the *models*, the *views*, and the *templates*. The *model* operates in the data access layer and helps to handle data. The *view* works towards business logic by communicating with the *model* and rendering data to the *template*. Finally, the *template* works like the presentation layer, which handles the user interface.

When a user requests a resource in the browser, *Django* works as a *controller* that checks the *URLs*. The requested *URL* calls a *view* that interacts with the *model* for the data and with the *template* to render the processed data. Finally, *Django* responds to the user sending a *template* as a response.

Throughout this chapter, we will report how we designed each of the elements (*URLs*, *views*, and *templates*), and how we made sure that the final solution was dynamic regardless of the imposed data *model*.

It is important to note that, although this solution has already served as the basis for different systems, this dissertation focuses on how to achieve and build our dynamic back-end generator platform for data control. Therefore, we will get into detail regarding the components of our solution, aiming to show how our *Django* project template, along with the script, can generate such solutions.

### 4.3.1. Implemented solution mechanisms overview

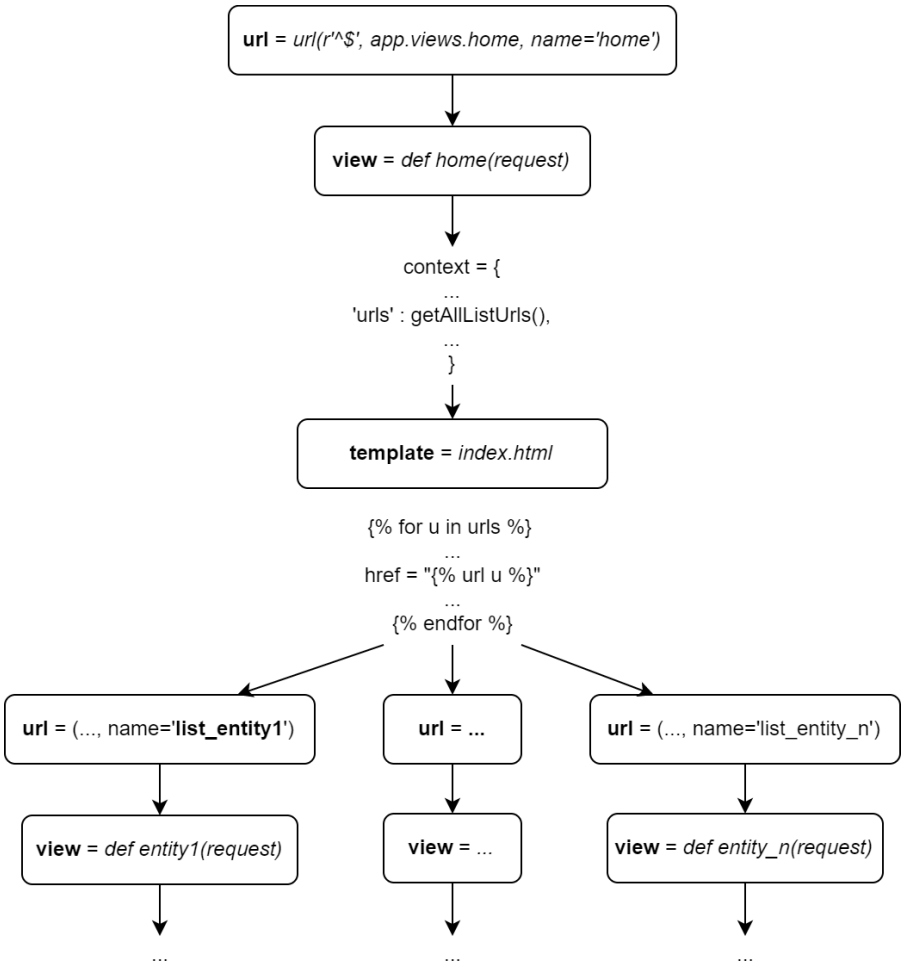



Figure 22 – "*URL->View->Template*" mechanism

It is imperative to understand that the *URLs* call the *views*, and those *views* call *templates*, which will provide links to call other *URLs*.

Our project strives to present the relationships between all system entities for the average users directly and intuitively. Figure 22 shows the first system calls and the first interactions with the system. The following steps translate these relations:

i. When the project starts, the *Django* framework triggers the home *URL* that calls the home *view*;
ii. The home *view* renders all the processed information to the *index.html template*, including all the project listing *URLs* in form of references (*href*);
iii. Selecting those references, triggers the respective *view* which renders its specific template (*URL-> view-> template-> URL-> view-> template > ...*).

We start by structuring and connecting the operations that the system must include. When a user starts our platform, the system displays the main page (*home*), that presents all the entities of the system. By selecting one of those entities, it will start a cycle of interactions. Figure 23 shows what operations are available regarding any chosen entity (*e.g., download, audit, list, add, edit*).

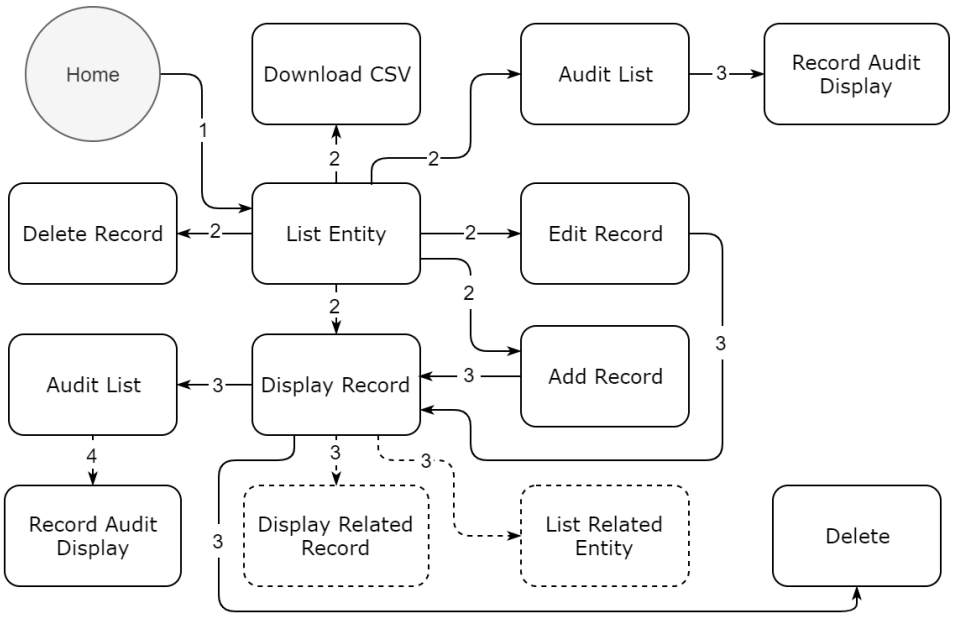



Figure 23 – Available platform operations for a specific model

To make it explicit and as an example, following Figure 23 and based on Figure 25, if a user selects an entity *Author* in the *home* template, it will lead him to a list of *Authors* (*List Entity).* Once in that list, it is possible to select an *Author* record and see its description (*Display Record)*. After, there is an implemented mechanism that renders specific *paths (dotted boxes),* that translate the existing relationships with other entities depending on that *Author* (*e.g., List Related Entity: "Book")* or which he references (*e.g., List Related Records: "Country"*), promoting contextualized navigation. These mechanisms are the basis of our project and what makes it so different from other solutions.

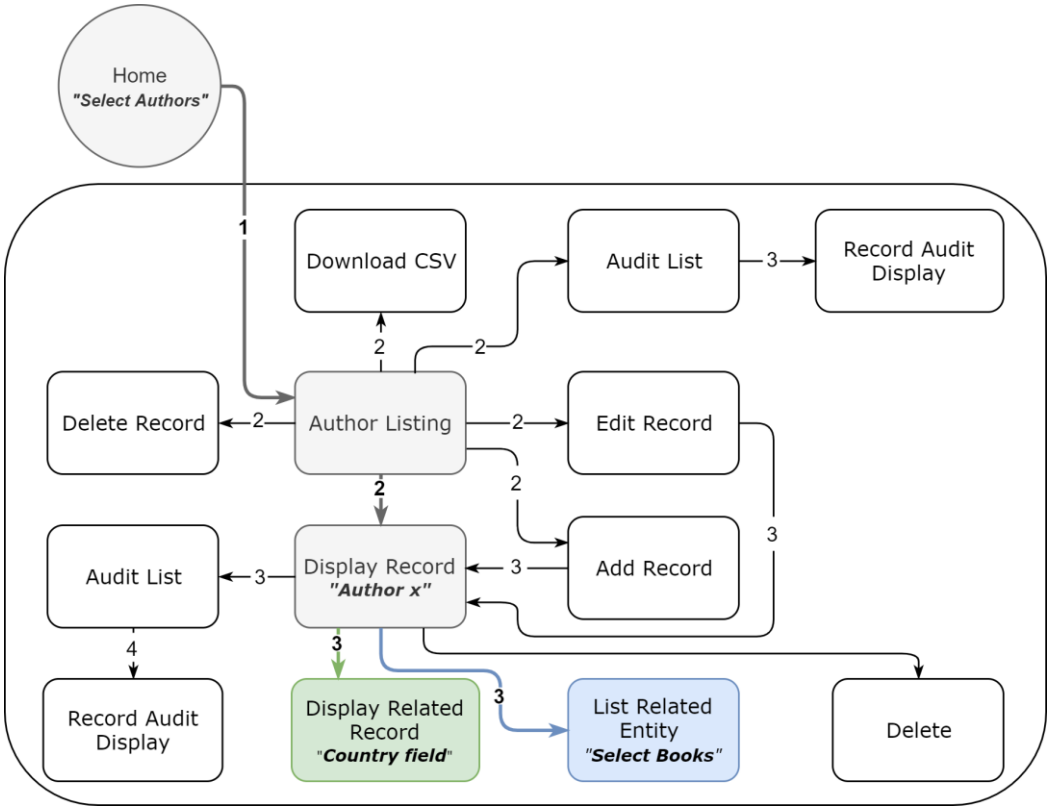



Figure 24 - Contextualized iteration based on a displayed record

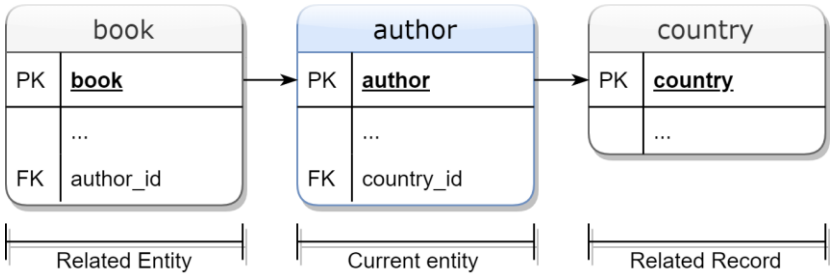



Figure 25 - Database relation example

Related to the *List Related Entities*, if that same *Author x* had published *Books*, the user can access the list of those *Books*, keeping the context and starting an iteration all over. Therefore, in this next iteration, following Figure 24, the system creates the *List of Related Books* regarding the contextualized *book* list, which is a queried list of the *books* of *Author x*.

The navigation is like a representation of the database itself, where the system performs queries as a user progresses through the relationships. Depending on the *model* structure of the problem at hand, it is possible to iterate over lots of contextualized entities by filtering each new iteration depending on the previous ones.

As for the *Display Related Records*, if an *Author x* contains *FKs* regarding other entities (*e.g., Country)*, the platform template will provide links to access those entities display.

We present more elucidative pictures in Appendix B regarding the display related record and list related entity relationships.

When building our templates, we will simplify the display of both relations so that our end-users can intuitively navigate in the platform (*e.g., Figure 60*, point *4* and *5*).We conclude that no matter which entity we access, the platform will always provide ways to access related entities or the records it references. During this chapter, we will present how we build each of the modules (*URLs*, *views* and *templates*).

### 4.3.2. URLs

An elegant and clean *URL* scheme is essential in a high-quality web application. Since *Django* has almost no restrictions when designing *URLs*, it is imperative to create them the best way possible. The *URL Python* module relates to the *URL* path *expressions* and *views*. In resume, when a user requests a page, the framework *Django*: [99]

i. Determinates which *root* module to use;
ii. Loads that module and looks for the variable in the *urlpatterns*;
iii. Runs over each *URL* pattern until it finds the matching request;
iv. Once found, it calls the *view* of the matching *URL*, or if it does not, it requests a proper error-handling *view*.

In this project, we divided the *urlpatterns* into three sections for each *model* of our solution. These *URLs* are inside a directory that contains separate files that relate to each *model* entity (*e.g., app/urls/entity.py*). By applying this directory design, it is easy to replace or change those files if needed. The three separate *urlpatterns* per file are:

*1.* ***CRUD and CSV***

These *urlpatterns* have specific *URLs* that translate six different operations. The *URLs* relate to *views* that allow posting, listing, editing, deleting, displaying, or retrieving the *CSV* for each *model* entity in the database.

The *URLs* are straightforward and divided into two sections. They can start by the entity name, followed by the operation (*entity/operation/*) or by placing the *ID* in-between when there is the need to know its context (*entity/(?P<pk>\d+)/operation/*). Figure 26 presents these *URLs*.

```
urlpatterns += [
    url(r'^entity/', include([
            url(r'^add/$', app.views.ViewEntity().post, name='post_new_entity'),
            url(r'^list/$', app.views.ViewEntity().list, name='list_entity'),
            url(r'^csv/$', app.views.ViewEntity().csv, name='csv_entity'),
        ])),
    url(r'^entity/(?P<pk>\d+)/', include([
            url(r'^$', app.views.ViewEntity().delete, name='delete_entity'),
            url(r'^edit/$', app.views.ViewEntity().edit, name='edit_entity'),
            url(r'^info/$', app.views.ViewEntity().display, name='display_entity'),
        ])),
]
```

Figure 26 - *CRUD* and CSV urlpatterns

### 2. ***Auditing***

There are three possibilities for the auditing *URLs*: when towards all the entries of a specific *model*, regarding all *CRUD* operations; concerning a data record of the *model*, meaning there is a need for the context *ID*; or, concerning the display of any entry of the previous lists, where the user can retrieve information about all the changes made to it.

```
urlpatterns += [
    url(r'^entity/', include([
        url(r'^audit/$',  app.views.ViewEntity().audit, name='audit_entity'),
        url(r'^(?P<pk>\d+)/audit/$',  app.views.ViewEntity().audit, name='audit_entity'),
        url(r'^(?P<pk>\d+)/audit_display/$',  app.views.ViewEntity().display_audit,
           name='audit_display_entity'),
    ])),
]
```

Figure 27 – Auditing urlpatterns

### 3. ***Database Entity Relationships***

While the *urlpatterns* for the *CRUD* and auditing always follow the same structure, the same does not happen with the entity relationships. They vary depending on the *model* structure.

To create these *URLs*, we used the *get_model()*, which allows us to access the *_meta.get_fields()*. Therefore, it is possible to verify the associations of the current entity and create the *urlpatterns* (*e.g., Figure 28*). The project generates them based on their entity *name* and own *view*, as shown in the picture.

```
urlpatterns += [ url(r'^entity/(?P<pk>\d+)/info/', include([
        url(r'^entity2/list/$', app.views.ViewEntity2().dependents),
        url(r'^entity3/list/$', app.views.ViewEntity3().dependents),
    ])),
]
```

Figure 28 – Database entity relationships urlpatterns

The *URL* system always respects our imposed taxonomy. It is possible to verify that the first two sets of *URLs* contain a *name* parameter that changes depending on the *name* of the *entity*. The design of the *URL* names was crucial. Because the *views* can render them, depending on the *model*, and then call them the respective *name* in the *templates* (*e.g.*, *{% url name %}*). So, by defining the *name* (*e.g.*, *entity=example*) in the *view*, it is easy to perform the aggregation between the operation and the name itself (*e.g.*, *operation_ + self.entity*), which defines the *URL* names render to the *templates (e.g., edit_example, delete_example)).*

Besides this mechanism, we also concatenate a processed *query* parameter to the *URL* call (e.g., *{% url name %}?{{query}}*), which contains the logic regarding the breadcrumbs, search, pagination and sorting parameters.

### 4.3.3. Views

After we explain our *URL* structure and how *Django* works as a controller that checks the resources in the *URLs*, it is time to talk about the *views* triggered by those *URLs*.

Our program contains a *home() view* that it is independent of all the others, a *CrudView()* class, responsible for instantiating all the *model* entities, and a *breadcrumbs* module used by all the *views*.

1. ***Home***

The first triggered *view* when the project starts is the *home view*, which has two essential functionalities. It verifies if the whole project structure is correct and builds a data structure to render the template with all information associated with each data model.

Regarding the first functionality, we have a *crawler* that checks if all *models*, *views*, *templates*, and *forms* respect the existing data models. In case something is missing or incorrect, the *view* renders an *info* message that asks the user to run the creation *script* (*e.g.*, Figure 29).

Regarding the other functionality, the *view* renders several parameters to the *template* with a detailed description regarding each data model. For each entity *URL name*, we create our data structure containing the label, the number of objects in the database, how many relationships it has with the other entities, the access restrictions (*e.g., admin or staff*), if it contains auditing, and we generate a color (*e.g., random.seed(key)*) so that the same *model* has a respective associated color.

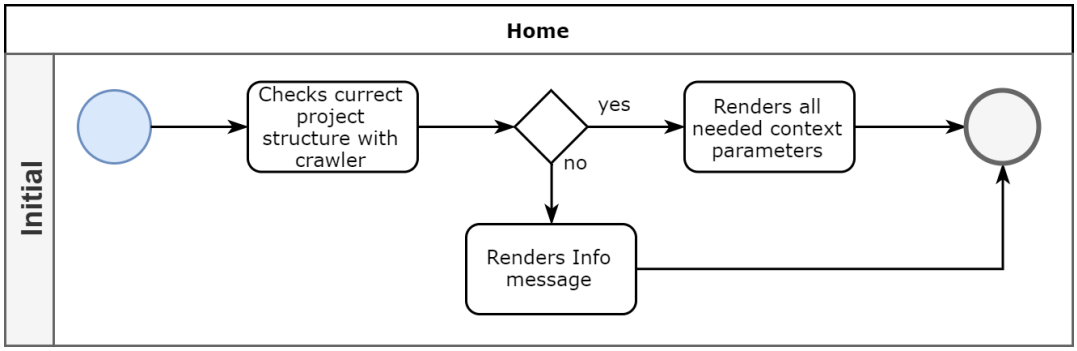



Figure 29 - Home view interaction diagram

```
[('list_country', ['Country', 7, 0, False, True, 'rgb(247,253,203)']),
('list_content', ['Content', 5, 1, True, True, 'rgb(205,214,245)'])]
```

Figure 30 - Rendered data structure

*2.* ***Breadcrumbs***

It is one of the main modules of our solution and provides many contexts to our platform. Because of our dynamic solution design, we built the *breadcrumbs* from scratch, which implied testing the modules to verify their correctness. The *breadcrumbs* use two additional classes, the *BreadcrumbsMediator()* and *Breadcrumbs().*

```
self.type = type                #t
self.entidade = entidade        #e
self.id = id                    #i
self.page = page                #p
self.query = query              #q
self.order_by = order_by        #o
self.dependecies = dependecies  #d
```

Figure 31 - Breadcrumbs parameters (t/e/i/p/q/o/d)

The *BreadcrumbsMediator()* encapsulates how the set of *breadcrumbs* objects interact. It takes the full path, domain, and sub-domain for the object, and performs operations like:

i. *get_previous()*: gets the previous *URL*;
ii. *add_new_bread*(): adds a new *breadcrumb* component based on many parameters;
iii. *get_breadcumbs*(): gets the current *breadcrumbs*;
iv. And several private methods to support the above ones.

*add_new_bread()* uses the *Breadcrumbs()* class on the mediator, and it builds each object based on the previous *breadcrumbs* including the type, entity, ID, page, query, order, and

dependencies. The main idea to keep is that each *breadcrumb* can contain a good deal of information for the application usage.

To resume this entire module, without talking about the programming mechanics, the breadcrumb consists in five parts in the following order:

i. Domain (*e.g., http://ieeta-sip.web.ua.pt/*);
ii. Possible subdomain (*e.g., backoffice/*);
iii. The current *view* that follows the *Django URL* nomenclature (*e.g., entity/list/*);
iv. The current *view* breadcrumb (*e.g., ?&t=list&e=entity*) that increments with each interaction, for each *view*, keeping track of the context (*e.g., ?&t=list&e=entity&t=info&e=entity &i=483*);
v. And the additional parameters, which are the query (*q*), page (*p*) and order (*o*), that can combine in *3!* different ways (*e.g., &q=test&p=2&o=3*).

*3.* ***CrudView***

CrudView is the main class of our project, which contains the main dynamic views. It is possible to set the fields (*e.g., Figure 32*) in the subclasses constructors without the necessity to rewrite the views, or to override some of the constituent views. By doing so, the application can generate everything according to those values. The parameters are:

i. *title*: defines the *title* in the singular, so that the proper *views* use it (*e.g.; title "author"*, when rendered to the edit template, becomes *"edit + author"*);
ii. *title_p*: defines the *title* in the plural (*e.g.; title "authors"*, when rendered to the list template, becomes *"list of + authors"*);
iii. *form*: sets the respective *form* for the *view*;
iv. *model*: sets the respective *model* for the *view*;
v. *entity*: for programming mechanisms, like for example to build the *URL* references;
vi. *fields*: to set the *fields* to present on the listings from the chosen *model*;
vii. *displayHTML*: to set the default display *HTML*, which the user can change for a custom *template*.
viii. *listHTML*: to set the default list *HTML*;
ix. *listPageRange*: to set the default page range of the listings;
x. *query*: to set the default listing *query*, it is possible to place any *query*;
xi. *csvHead*: by default, it uses the same fields of the listings, but it is possible to add or remove fields for the *CSV* output;
xii. *editable*: True, means everyone has the permission to edit the records of this *model*, and False, means that only admins can;
xiii. *addEntry*: permissions for adding records;

xiv. *associations*: permissions towards the display of the entity relationships links;

xv. *permission*: permissions towards all the *model* functionalities and operations.

```
title = _('Entity'),
title_p = _('Entities'),
form = EntityForm,
model = Entity,
entity = _('entity'),
fields =  ['id', 'title'],
displayHTML = 'display/CRUD_mostrar.html',
listHTML = 'list/CRUD_listar.html',
listPageRange = 10,
query =  ['id'],
csvHead = ['id', 'title'],
editable = True,
addEntry = True,
associations = True,
permission = True,
```

Figure 32 - Constructor parameters

The *CrudView* holds several *views* with well-defined functionalities. For each, there are three decorators, which are:

i. *@method_decorator(login_required):* This decorator ensures that users without permission to access the *view* cannot access it and redirects them to the login page.

ii. *@method_decorator(csrf_protect):* This decorator works toward *CSRF* protection.

iii. *@method_decorator(require_http_methods(["POST", "GET"])):* This decorator restricts the access to the *views* based on the HTTP methods chosen (*e.g., POST, GET*).

Besides the decorators, all the *views* start with a condition, based on the constructor *permission* parameter, that checks the user type, and may assign a different *view* content towards staff and administration members.

The *views* are where most of the programming logic lies, and everything comes to how they process the data *models* and render that information to the *templates*. Therefore, understanding the main *views* is essential in this phase. During this chapter, we will explain each of the *views* and display their activity diagram. On the activity diagram, the blue circles represent the start, and the gray circles describe the *template* rendering. The *views* are:

*a. Add and Edit*

Both *views* start by building and checking the *breadcrumbs* component to provide context. After, they verify if there is any previous context associated with the entity at hand and, if so, the right form fields are auto filled and marked as disabled. The difference between add and edit *views* is that edit *view* renders and shows the information present in the database.

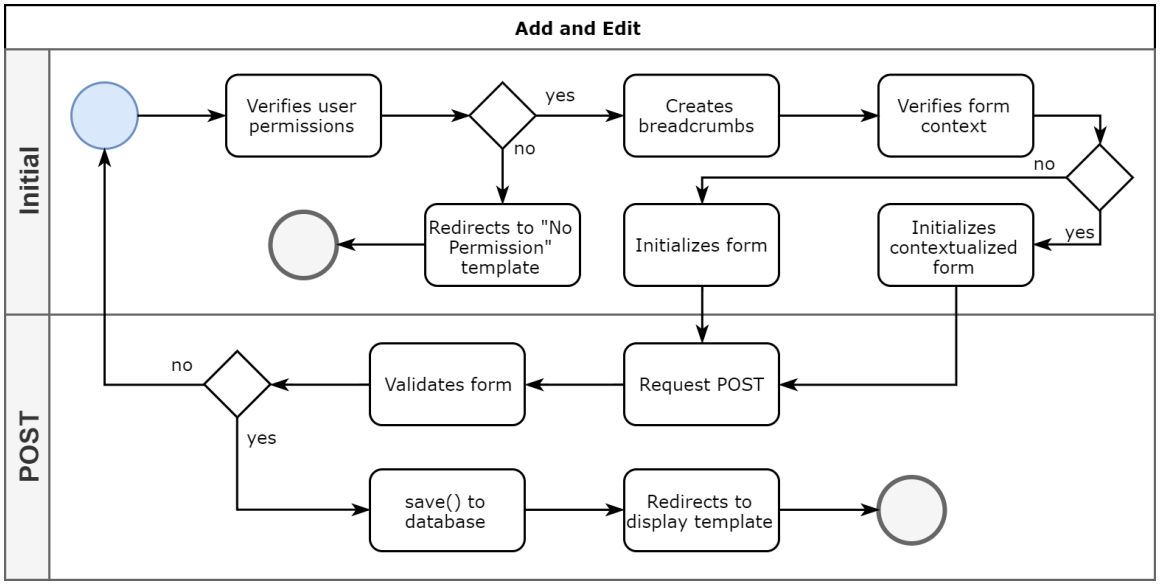



Figure 33 - Add and edit view interaction diagram

So, and before the *POST*, the *views* render the form title, *breadcrumbs*, and a link to the previous page based on the *breadcrumb's* information.

When the user performs the request method *POST*, and if everything is valid, the *views* complete the *form.save()* to the database and redirect the user to the display *view*.

These *views* apply additional methods, such as:

i. *fill_form_using_breadcrumbs():* This method aims to fill the *forms* given a context. We stated that our *breadcrumbs* store all the relevant information, so, by translating them and with the proper programming, we can retrieve the needed data and fill some form inputs. We also make those inputs disabled (*e.g.*, *add_edit_context_form()*), so the user cannot change them, assuring that the user is in context and performs the add/edit operations following a path. This method was necessary in our platform, and it is one of the high points of our platform comparing with *Django* admin, as it abstracts away the need to know the data relationships by promoting context.

ii. *redirect_to_display():* Checks the *post.save()* entity *PK*. This allows the *HttpResponseRedirect()* to the display view, after the successful *POST*.

*b. Delete*

When someone deletes a record, it removes it from the database. In case the record has associations, to make sure the user does not remove necessary information, instead of using *on_delete=CASCADE*, we always use *on_delete=PROTECTED*. By doing this, we inform the user that before deleting the record, he must remove other related entities by displaying a proper message.

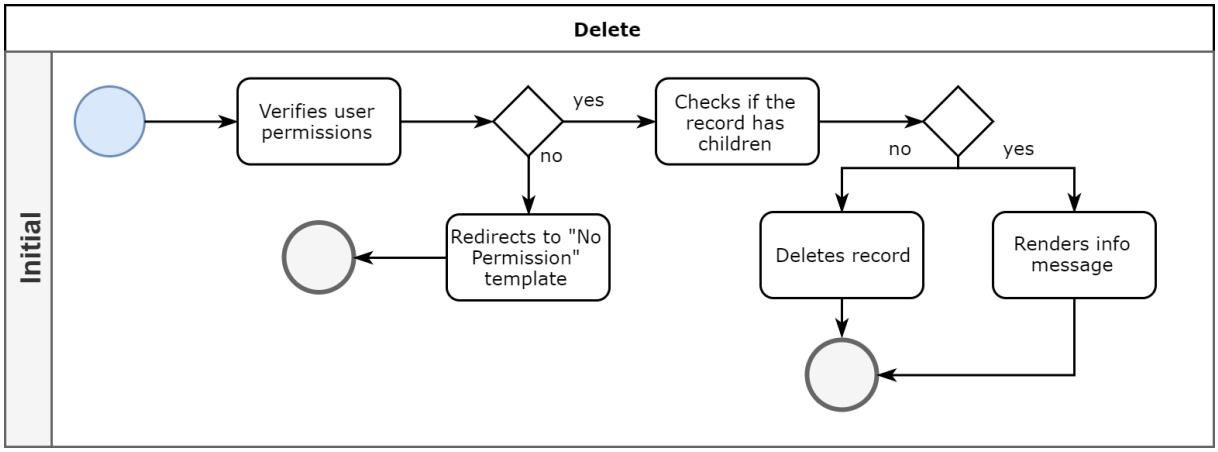



Figure 34 - Delete view interaction diagram

In case a user concludes the delete action, the *view* does a *HTTP_REFERER* redirect, staying on the same refreshed listing page.

*c. CSV*

Downloading content in *CSV* was one of our functional requirements. This module is essential for users to retrieve information from the platform regarding the lists of data.

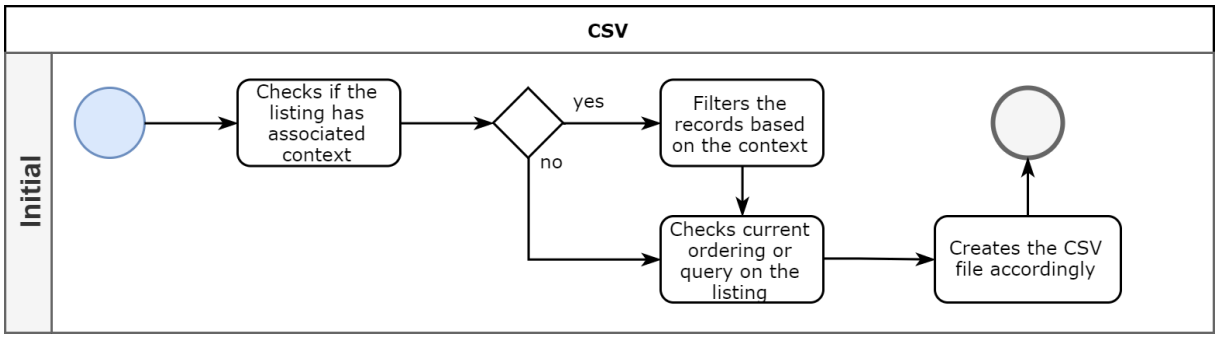



Figure 35 - CSV view interaction diagram

These lists can change depending on the order, query, and context, so it is essential to download the *CSV* file accordingly.

*d. Display*

The display renders essential information towards the *template*. Besides providing all the information on the record, it also presents shortcuts to related information. Meaning that if the display fields or the record itself have relationships with another record or entity, the *view* renders the links to access that same information. There is also information about the auditing, concerning the last user to change the current record, containing the respective date and operation performed.

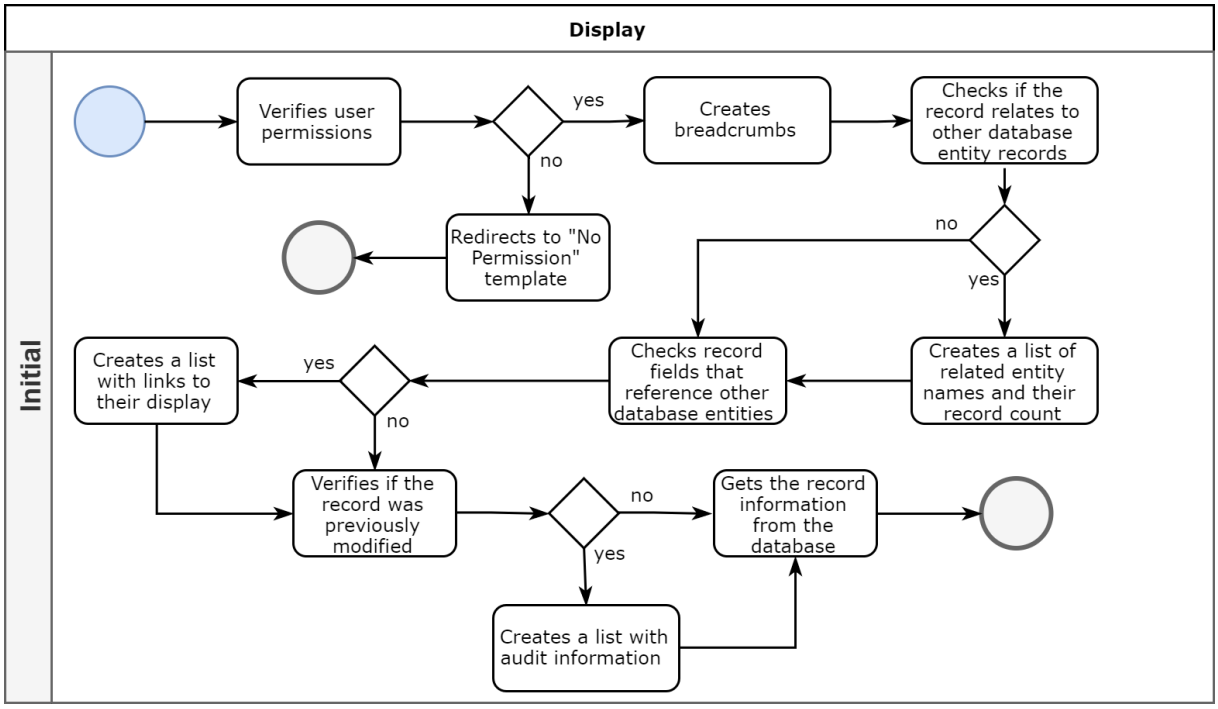



Figure 36 - Display view interaction diagram

The display *view* uses additional methods, such as:

i. *getRelated_tables():* This method aims to create a list of all entity names related to the record in question and the number of entries for each of them. Thus, through the *Model _meta API*, it is possible to know which *models* relate to the current *model*. Then, by using the proper *filter()*.count*()*, it is possible to retrieve the number of records for each of the related entities and build the *URLS*.

ii. *getRelated_record_by_field():* This method aims to verify if any of the record display fields contains relation with others (*FKs*), and if so, the *view* renders those links to the *template*.

iii. *get_last_user_mod():* Builds a list that carries the *id* of the last audit of the record in question, the user who made it, the date, and the operation.

*e. Dependents*

Dependents arise from the entity dependency *URLs* and the view display. *Dependents()* acts as an intermediary *view*, which, depending on the entity and relationship chosen on the *template* (*e.g., getRelated_tables()*), uses the entity's *PK* to call the related entity listing based on its query. If a user is in a rendered display *template* and the user presses one of the relationship entity links, the *dependents()* work as a shortcut to their listing *view*. This *view* works as an intermediary *view*.

*f. List*

The listing *view* contains several components. First, it creates the *breadcrumb*, then creates the list, which may have context with an entity's display, or generic, when accessed from the home page.

Besides the above, the *view* always renders several parameters, which are explicit in the activity diagram. Including the list query, the labels of each column of the listing, the page range, the title, the links to edit/delete/add *view* entities, the auditing link, the *CSV* download link, among others.

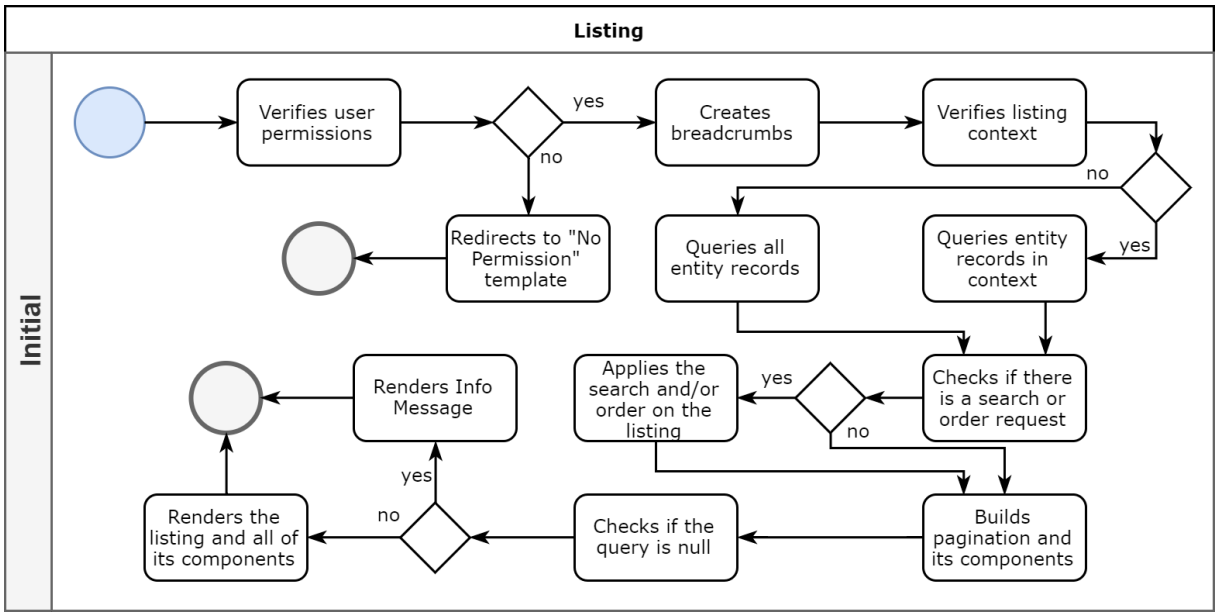



Figure 37 - Listing view interaction diagram

The *view* holds additional methods, which are:

i. *list_query()*: This method uses the *self._fields* and *self._query* constructor values to process the listing, which can be in context or towards all the model records.

ii. *get_query_from_request():* The method disputes when the *template* search field holds an input. It gets the value *(e.g., Get.get ("q"))* that *search()* method uses.

iii. *search()*: It is applied by the *query_search_orderby()*. This method uses the entity, the fields, and the query so it can search anything in the database associated with them. The search relates to the constructor entity parameters field array (*fields = [ 'entityId', 'name', ...]*). Meaning if a *model* has five fields, but there are only two fields in the array, the search is towards those two. This feature is important since it makes the searchable fields easy to change without affecting the search method and working regardless of their order. If the model has *FKs*, it searches based on the queried value of the fields and not their *IDs*, which is preponderant when performing the search. This process requires getting the *model* associated with the field (*ContentType.objects.get(model=f).model_class()*), choosing the proper

searchable attribute (*model.attrName()*), and executing the search towards its *__str__* model definition and not the *ID* itself. This method always returns the filtered list based on the search query value.

iv. *query_search_orderby():* It maintains or changes the listing contexts as it is possible to search or change the query in the template. This method checks for both and returns the *searchQuery* (*search()*) and the query in question. It is important to realize that the listing uses the constructor default *self._query*, but the template allows its modification.

v. *build_pagination():* It is a simple method that uses *Paginator()* to create paginations within a range (*self._listPageRange*).

vi. *message_output():* Uses the *Messages* framework of *Django* to inform when there are no entries to list.

vii. *order_by_param():* Again, and concerning context, this method checks if the *URL* has the ordering context parameter.

*g. Auditing*

To implement the audit mechanism, we use an external library (*e.g., django-simple-history*) that suits our needs. This library allows storage of the actions performed by the users regarding create, update, and delete operations.

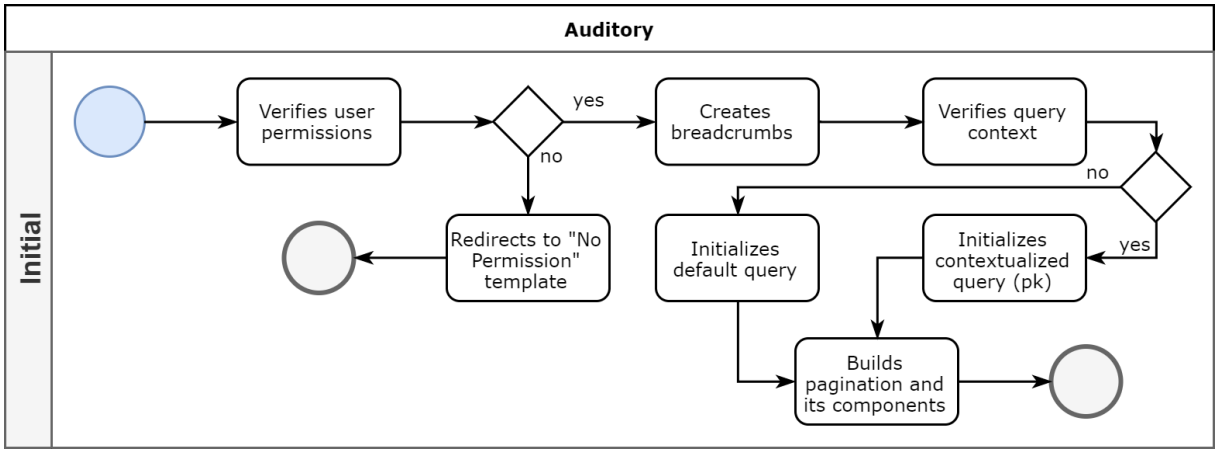



Figure 38 - Auditing view interaction diagram

We built our *view* to render enough information to create a list, like the listing mentioned above. In this listing we include:

i. *'history_user_id'*: to know which user audited the record;

ii. *'record_field'*: to know the record in question, we use the dunder str value of each model.

iii. *'history_date'*: to know the time when someone changes the record;

iv. *'history_type'*: to know the type of operation performed (*e.g., create(+), update(~), delete(-)*)

The audit may refer to all changes of a data model or of a specific entry in that data model. We query the list by chronologic date (*e.g., 'history_date'*), which is helpful for search reasons.

*h. Audit display*

The main goal of the audit display is to operate as a version controller. Thus, it all comes down on being able to check the differences between the selected entry and the current one, when in context for the same entity record (*ID*).

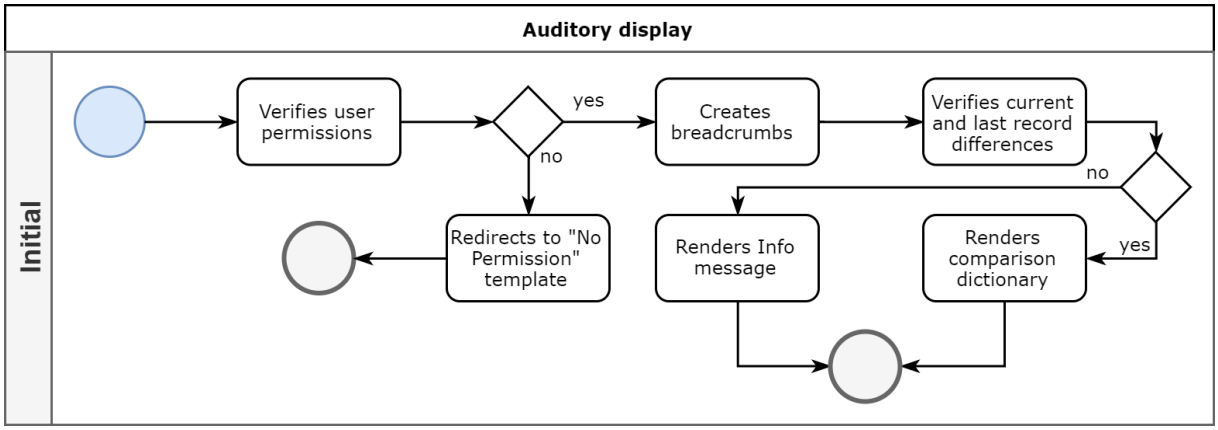



Figure 39 - Interaction diagram of the auditing display view

In the view, we have a *dictionary* that renders all data to the *template*. The *dictionary* possesses the name of the record as the *key* and a list of *values* (*e.g., [current, previous, comparison]*). The last parameter is the *Boolean* relation between the first two, which are the latest and the selected entry, for that record in the database.

Besides, this *dictionary* also contains information related to the changes, the user, and the operation that took place. Through this approach, we can work the *dictionary* (*e.g., Figure 40*) on the *template* side, since the *view* renders all the necessary information.

```
{'id': [1, 1, True], 'title': ['Portugal', 'Portugal', True],
'publisher': ['Porto Editora', 'Porto Editora', True],
'isbn': ['123', '12', False], 'author_id': [1, 1, True],
'history_date': [datetime.datetime(2019, 9, 26, 13, 26, 47, 490572, tzinfo=<UTC>),
datetime.datetime(2019, 8, 1, 13, 41, 26, 805662, tzinfo=<UTC>), False],
'history_user_id': [1, 1, True], 'history_type': ['~', '+', False]}
```

Figure 40 – Home context dictionary example

## 4.3.4. Templates

As mentioned earlier, after the *views* interpret and process the model information, they render that same information to the *templates*. The *templates* are the components that make up the system interface.

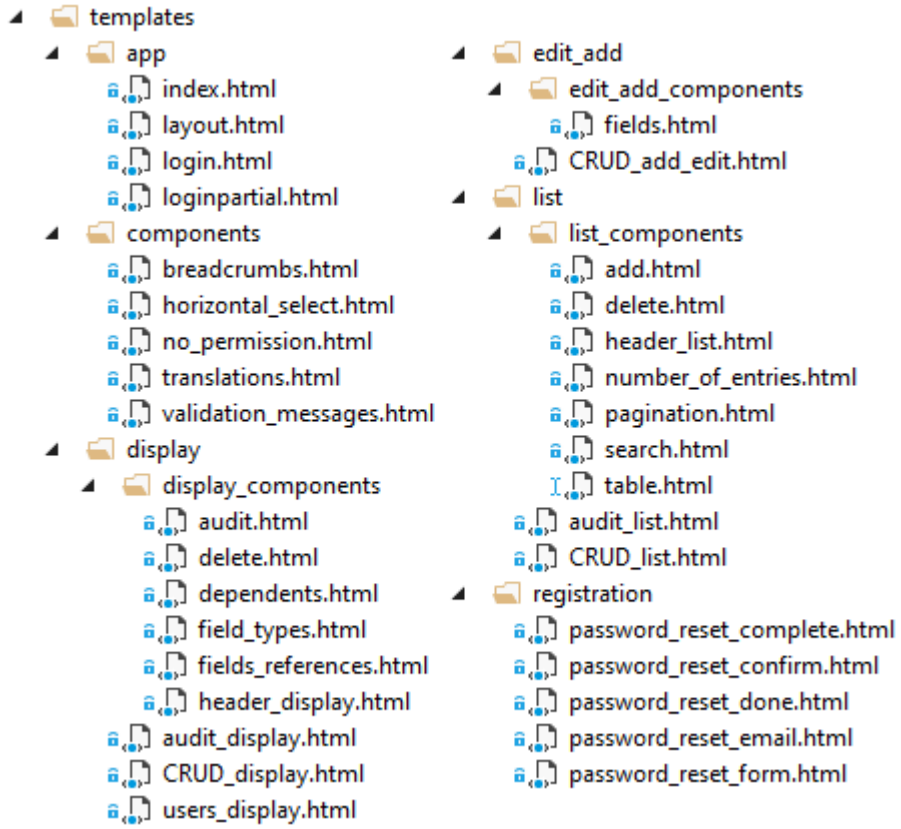



Figure 41 - Templates

We distribute the *templates* into six directories that aggregate the ones with mutual objectives, including the fundamental elements of the application (*app*), the general components (*components*), the display (*display*), editing and adding (*edit_add*), the listings (*lists*), and the registration (*registration*). We describe each template of Figure 41 in Appendix C.

We build the application on top of generic *templates*, meaning that regardless of the number of *models* or *views*, the *templates* remain identical. Nevertheless, within the need for creating new *templates*, they can *{% include %}* the ones in the subdirectories (*sub-templates*) as they hold the base *HTML* components, promoting code reuse. This feature is ideal for aiding in introducing new business rules and speeding up development time.

Following, we will summarize some *templates* that supply our solution with dynamic content to provide a notion about their aim and inherent mechanisms. The solution possesses fifteen primary *templates* and nineteen *sub-templates* (*subdirectories* + *components*). Since we will show their interface in the next chapter, we will only present some code, so the reader can understand a bit of how to achieve part of the dynamism. The examples are:

1. ***validation_messages***

The *template CRUD_list* includes the *sub-template validation_messages*, allowing the corresponding *views* to use the *Messages Framework* of *Django* and set the message type.

**Messages**

View

```
messages.info(request, _('No results!'))
```

Template

```
{% include '../components/validation_messages.html' %}
```

validation_messages.html

```
{% if messages %}
    {% for message in messages %}
        {% if message.tags == "success" %}
            <div class="alert alert-success">   ...
        {% elif message.tags == "info" %}
            <div class="alert alert-info">      ...
        {% elif message.tags == "warning" %}
            <div class="alert alert-warning">   ...
        {% elif message.tags == "error" %}
            <div class="alert alert-danger">    ...
        {% endif %}
    {% endfor %}
{% endif %}
```

Figure 42 – Messages related templates

As a result, and as presents in Figure 42, the *sub-template* uses the *Messages framework* engine and selects the *alert-info* by checking the rendered *message.tags*.

This example shows how it is possible to create dynamism on the *template* side. By including this *sub-template* to handle *messages*, there will never be the need to create a new *HTML* regarding the *alerts*.

2. ***fields***

Another example is present in the fields *sub-template*, which the *CRUD_display template* includes. Since the *view* renders the *form*, which contains the different data types and the *fields.html* sub-template can filter all the field types, with conditions, and apply business rules and/or *HTML* according to them.

As we can see in Figure 43, it is possible to render any *form*, and the *template* will apply rules to that *form* and display the data as intended. This is beneficial when applying global rules to the solution *forms*.

**Forms**

**View**

```
httpForm = self._form(initial={ param : fk})
```

**Template**

```
{% for field in form %}
    ...
    {% include 'display/display_components/field_types.html' %}
    ...
{% endfor %}
```

**fields.html**

```
<!--Select-->
{% if field|fieldtype == "Select" %}
    {% trans field.id_for_label|to_and as myvar %}
    {% if "Id" in myvar  %}
        <p>{{ form.instance|get_attribute:myvar|default_if_none:"----"|default:"----"|safe}}</p>
    {% else %}
        <p>{{ form.instance|get_attributeDisplay:myvar|default_if_none:"----"|default:"----"|safe}}</p>
    {% endif %}
<!--Radio-->
{% elif field|fieldtype == "RadioSelect" %}
    {% trans field.id_for_label|to_and as radio_var %}
    <p>{{form.instance|get_attributeDisplayRadio:radio_var|yesno}}</p>
 <!--Hour-->
{% elif field|fieldtype == "TimeInput" %}
    <p>{{ field.value|slice:":-2" }}</p>
 <!--Date-->
{% elif field|fieldtype == "DateInput" %}
    <p>{{ field.value|date:"d/m/Y"|default_if_none:"----"|default:"----"|safe }}</p>
<!--Textfields-->
{% elif field|fieldtype == "Textarea" %}
    {{ field.value|default_if_none:"----"|default:"----"|safe }}<p></p>
{% else %}
    <p>{{ field.value|default_if_none:"----"|default:"----"|safe }}</p>
{% endif %}
```

Figure 43 – Form fields related templates

Throughout this project, there are many *template* mechanisms preponderant to apply part of the dynamism of our back-end solution.

Even though *Django* comes with several built-in *tags* and *filters*, which are crucial to apply rules in the template, sometimes there is the need to create extra functionalities, by extending the *template* engine and define custom ones.

During our project, we created multiple custom *tags* and *filters*. For example (*e.g.*, *Figure 44*), we registered a filter (*@register.filter*) that we use at the *template* (*{{field|append_ast_if_req}}{{field.label}}*). We needed to identify all the required field labels of the project with an asterisk. It is possible to add the asterisks to the labels on the *forms*, but by using this *filter*, it is possible to add, change or remove them much faster.

We can conclude that although the *views* interpret and manipulates the *models*, the *templates*, by using the *tags* and *filters*, can rearrange this data on the interface side and assign functionalities when necessary.

```
@register.filter
def append_ast_if_req (field):
    return '* ' if field.field.required else ""
```

Figure 44 - Custom filter example

## 4.4. Scripting

From the start, our main aim was to create a dynamic application that can translate all the data *models* for back-end management purposes. Therefore, we focus most of our attention on analyzing and building the previous dynamic modules. Nevertheless, to achieve an adequate solution, there is still the need to create pieces that correlate them. Even though our modules work in all the scenarios, depending on the *model* interpretation, our project template is still not fully functional towards them.

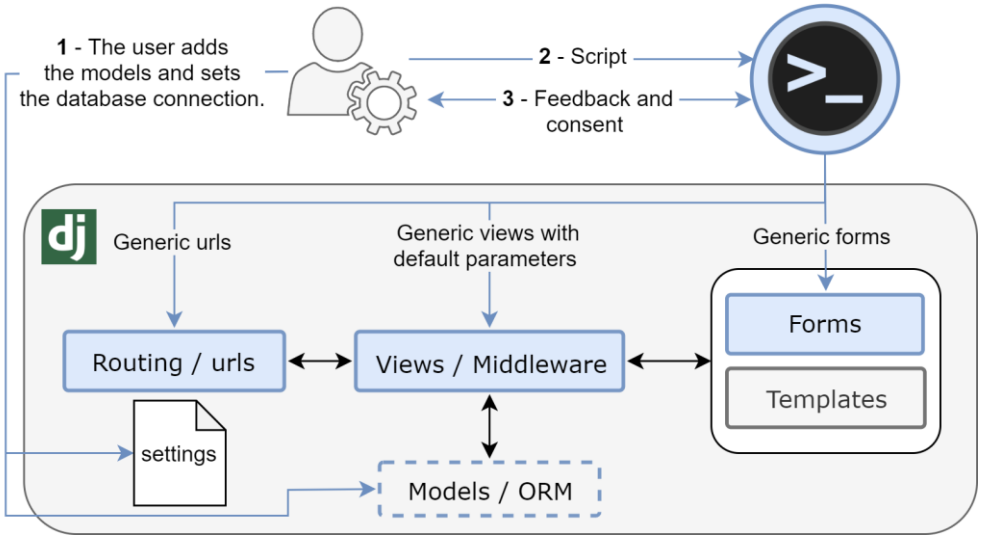



Figure 45 - Script overview

To design the previous *MVT* components, we began by validating the solution for different data models, both in terms of their data types and the relationships between their constituent entities. With this, we improve the programming towards the dynamic modules, redesigning them, so that when interpreting a data model, it is necessary to create the smallest number of interconnecting pieces with this *script*. The project template carries most of the logic to promote this dynamism, while the script creates what is missing.

To use the project template and create a solution, the user needs to run our script, which generates the missing parts of the architecture (*e.g., blue components on the Figure 45*). To create a solution the programmer must follow tree steps, which are:

i. Connect the project to its database in the *settings.py*, by providing information like the engine, name, user, password, host, and port.
ii. Create the *Django model* file, which must include the well-structured *models*;
iii. Run the script and consent to the project changes, so that the script can generate the missing pieces.

*django.core.management.base* was essential to build our *Django* script. When executed, it performs actions like:

1. ***Database connection***

First, the script verifies if the data connection is valid, and if so, it informs the user.

2. ***Data model***

The script will check the *model* existence and its correct structure since the model is essential for the accurate script output. If the user does not set up the *models* or they are not correct, the script will output the corrections and customizations needed.

3. ***Migrations***

The concept of migrations in *Django* is the way this framework propagates the *model* to the database. The *makemigrations* command is used to create new migrations based on *model* variations, and the *migrate* command, applies them.

At this point, and since the project already holds a data *model* and the proper database connection, it is necessary to migrate the data by using the *call_command('makemigrations', interactive=True)* and *call_command('migrate', interactive=True)*.

The script runs both commands interactively, being possible to perform the migrations, by providing a proper output and allowing the user to handle potential conflicts or to let the script do it for itself.

4. ***Forms, views and URLs***

In the previous chapter, we build our solution and place the components in separate directories, respecting a pre-defined structure. The goal of the script is to create the missing pieces, which are the *forms*, *views*, and *URLs*.

The script needs to create the *forms*, so according to each data type, the script verifies the correspondent *model* entity field and creates them accordingly.

It also creates the *views* in separate files (*e.g., app/views/view_example.py*). By interpreting each related *model*, it is possible to attribute the right value to each of the variables (*e.g.; Figure 32).*

As for the URLs, and as stated (e.g., Figure 26, Figure 27, Figure 28), the urlpatterns have three components, for each of the models, also in separate files (e.g.,

app/urls/url_example.py). Additionally, the *script* checks if the *models* have relationships and builds those *URLs* dynamically.

Before creating the components, the script checks if they already exist and if so, a message will appear, questioning if the user wants to keep the former configurations or rewrite them.

*5.* ***Folder tree***

The directory tree is a simplistic way to expose what the script creates. It is crucial for feedback and understanding. Therefore, it is possible to interpret whether the script creates (+), changes (~), or keeps the files.

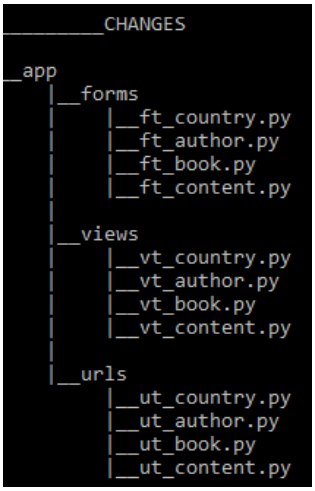



Figure 46 - Created components

*6.* ***Graph***

The command *call_command('graph_models', a=True, I=graph_models, output="app/static/pics/graph.png", Django=True)* creates the graph image, which goes to *a* proper folder of the *Django* solution. It is essential to understand that this script only creates the missing pieces of our project premade structure, and most of the logic is on the project template itself.

## 4.5. Summary

The *MVT* architecture itself has many advantages such as increased productivity, uniformity of software structure, reduces complexity, the application is easier to maintain, allows reuse of modules, reduces development time, among others. Our solution aims to explore these advantages and build dynamic modules to work with the most varied data *models*.

We planned the features and relationships between the system components, making them adaptable, regardless of the imposed data *model*, creating a dynamic and helpful database management platform. With this, we designed a project that requires only the *models*, and which is easy to set,

either through the parameters of the *views* constructors or by overriding or adding new methods. Besides, the structure holds standardized *URLs* and *templates* that link the *views* together and add dynamism to the data they render.

Even though the project template for our *MVT* solution is dynamic regarding the models, the script still needs to create missing pieces.

With both our *MVT* solution and script, we could create a solution that follows the *Django* admin standard for dynamic data handling, but which gives better user contextualization, more functionalities, and delivers a structure easy to change or to add new features.

# 5. INTERFACE

[8]

*In this chapter, we will display two examples of back-end management solutions based on our system. We start by analyzing the central system features, presenting the respective interfaces regarding a data model. Then we display a product ready solution based on a specific scenario, showing how simple it is to integrate new business rules.*

## 5.1. Default Solution

To show one potential product of our application and to give a clear understanding of how it works, we used a simple data *model* of three related tables (*e.g., Figure 25*). During this chapter, we will introduce each of those interfaces and their features that translate the pre-established requirements for our solution. It is important to note that we created and designed these interfaces from scratch. We will present detailed images, which can include red color numbers to mark their elements.

[8] *https://www.planetizen.com/files/images/Architecture%2001%20-%20D.%20Laird_0.jpg*

1. ***Login menu***

As stated, the system supports different user accounts. In the login menu, both administrators and staff members can perform three simple actions like changing the system language, recover their password, or login into the application. Figure 47 translates this interface.

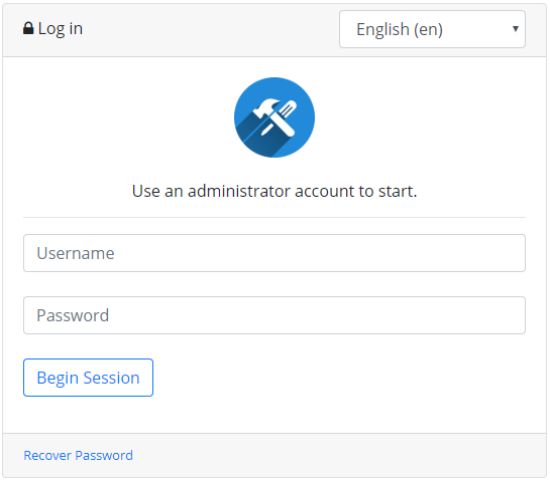



Figure 47 - Login Interface

2. ***Retrieve password***

For a user to recover the password, there is a simple email-confirmation mechanism. By default, the application uses a server that takes care of the process that enables the interaction between the user and the emailing service. When retrieving the password, there is a group of interfaces and steps to complete. They are:

i. The first step is to type the email in the input box and press the *reset password* button (*e.g., Figure 48*).

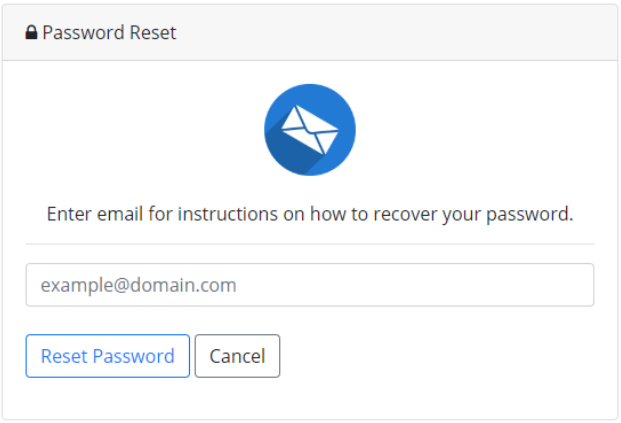



Figure 48 - Password Recovery email

ii. A message will pop to inform the user about the forwarding of the email (*e.g.*, *Figure 49*).

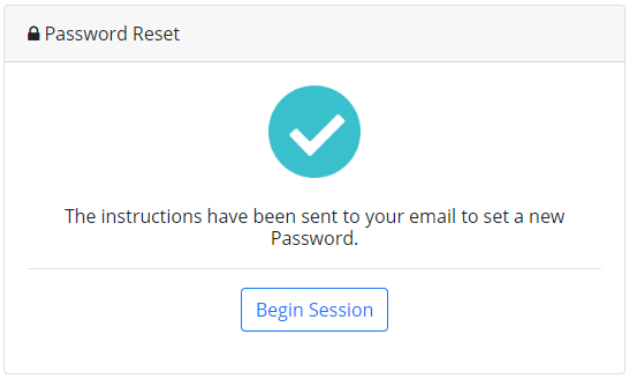



Figure 49 - Password Recovery info message

iii. The user must check the email for a generated link (*e.g.*, *Figure 50*).

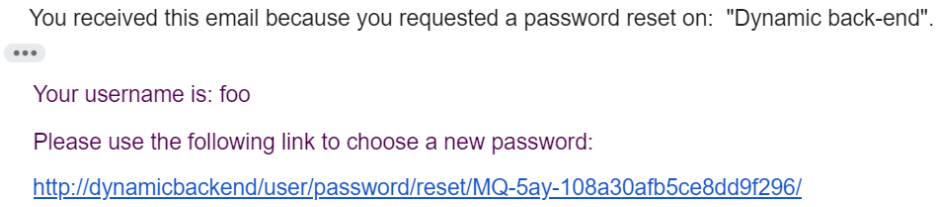



Figure 50 - Email with generated link

iv. The link will redirect the user to another interface, where it is necessary to insert the new password with many validations for security reasons, like the number of letters, digits, and symbols (*e.g.*, *Figure 51*). In case the link is not valid, it shows a message.

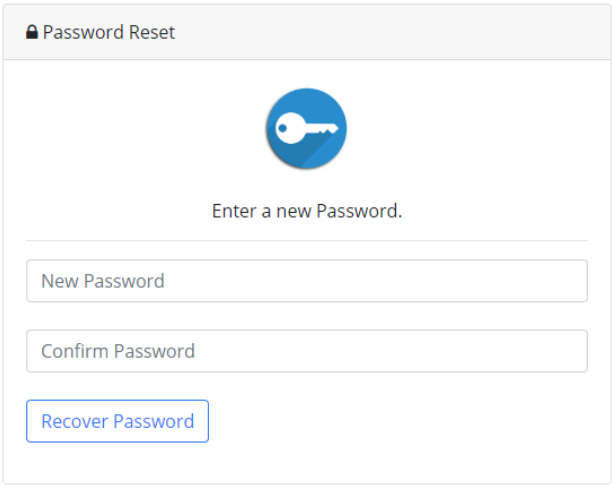



Figure 51 – Enter new password

v. If the user changes the password, another message *template* will pop to inform the user (*e.g.*, *Figure 52*).

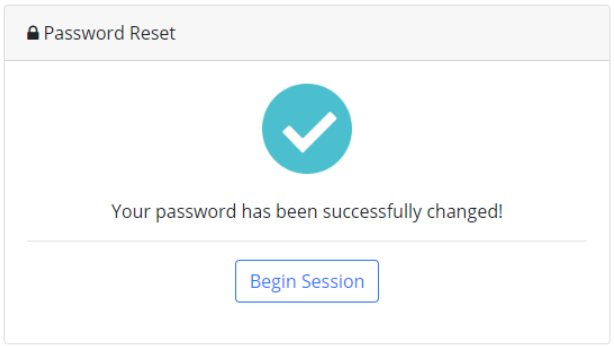



Figure 52 - Password Change info message

*3.* ***Main page***

The main page is the most prominent interface of the platform, as it provides the user with feedback regarding plenty of content.

First and before using the platform, it is necessary to execute the generating script (e.g., Figure 45). If the user neglects to run the script, the platform will show an alert to notify him (*e.g.*, *Figure 53*).

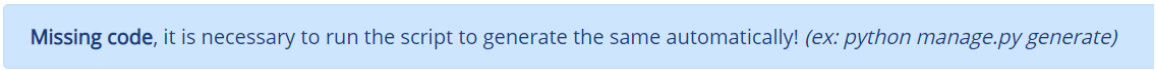



Figure 53 - Missing code info message

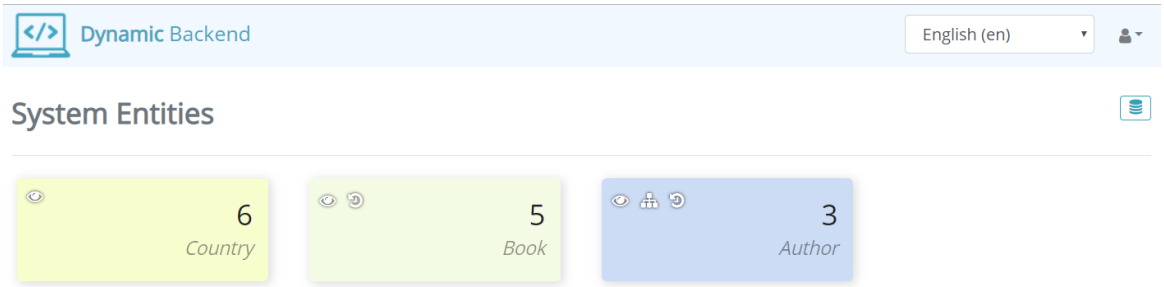



Figure 54 - Main page

After running the script, the main interface and the solution itself are ready to provide feedback, like the entity names and their respective number of records (*e.g.*, *Figure 54*). The contrasting colors are always the same for each entity, so it is easier to interpret the data, not only by

its name but by the color too. These boxes are pressable and linked to a list with their specific entity records.

In the top-right of the navigation bar, there is a user icon that, when pressed (*e.g., Figure 55*), , contains information regarding the logged user and provides a log off button. In case the user is an admin, it gives a link to the administration area of all the users. It also provides an internationalization drop-down, which allows changing the language of the platform. This internationalization refers to all the immutable text of the platform.

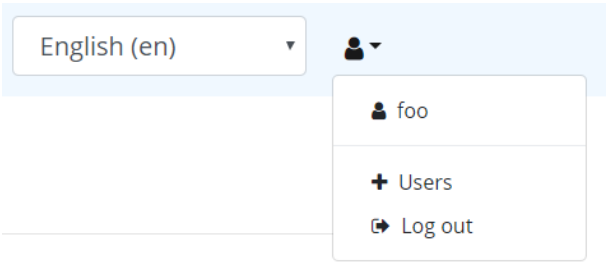



Figure 55 - Internationalization and user icon menu

The database icon (*e.g., Figure 54*) leads to the display of the database diagram of the current *model* structure, which helps to understand the *model* itself. Figure 56 shows that diagram.

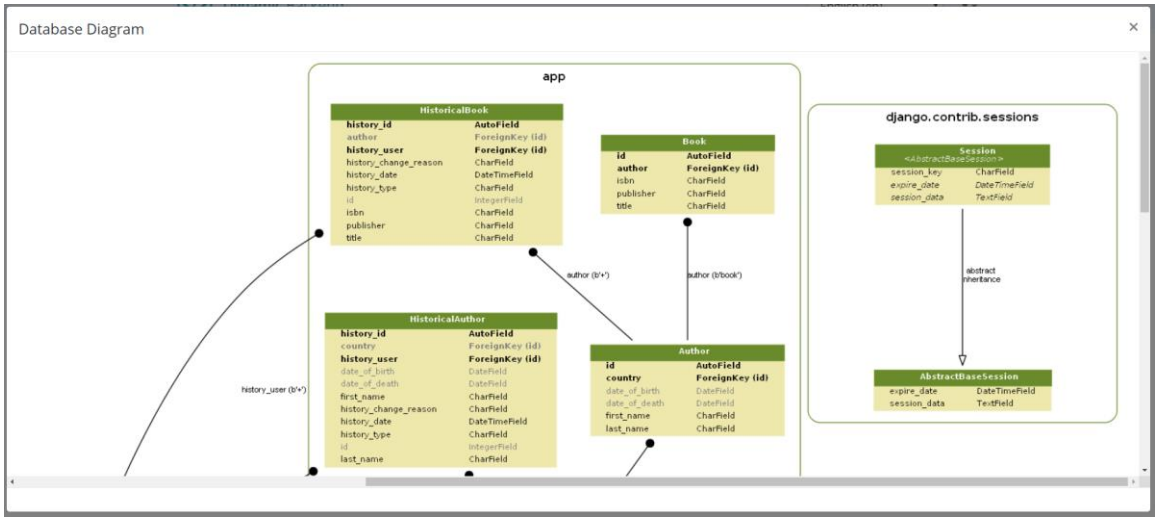



Figure 56 - Model Diagram

Inside each entity rectangle of the main interface (*e.g., Figure 57*) , it is easy to spot the white icons with black borders. These icons hold a special meaning for feedback reasons. The eye icon (*1*) shows that staff can access the entity, and the eye with a bracket means only admins can access it. Items marked with the clockwise icon (*2*) determine that the specific model holds an associated auditing table regarding all the actions performed by the platform users. The relationship icon (*3*)

identifies all the entities that possess relationships with others. All of them include tooltips to facilitate the user's perception. Both the entity name (*4*) and the number of records (*5*) are easy to spot.


Figure 57 - Entity representation

### *4. List entity records*

By selecting one entity from the main page, the platform redirects the user to their corresponding list of records. In this record page, it is possible to see all the entries for that specific entity or an info message if there is none to display (*e.g.*, *Figure 58*).

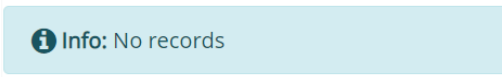



Figure 58 - No records info message

For each database entity list, there are nine main components present, the breadcrumbs (*1*), the entries (*2*), the search input (*3*), the addition button (*4*), the *CSV* file download button (*5*), the audit button (*6*), the edit button (*7*) and delete button (*8*).

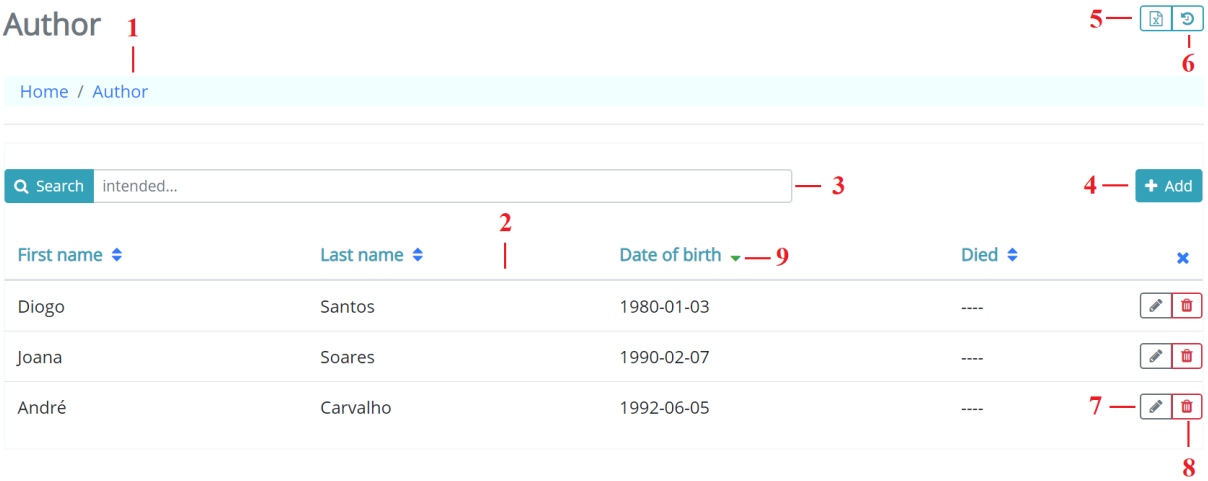



Figure 59 - Listing template

The *CSV* icon will trigger the download of all listing content, including the records not on display at the current pagination. Each table entry also holds a hover clickable property that connects it to the corresponding record display.

There are also sorting buttons that can change the default order (*9*). When pressed, the sorting icon turns green so the user can have feedback.

5. ***Display***

The display *template (e.g.*, Figure 60) presents plenty of functionalities, as mentioned.

It presents all the data of the record in question, a swift access button for editing (*1*), a button to download all the information in *PDF* format (*2*), and another button that connects the user to the history of modifications (*3*). Besides, it contains links to related entities (*4*) or icons that translate relations of fields with other tables (*5*). This interface also provides information about the last user record changes for agile management by the team (*6*).

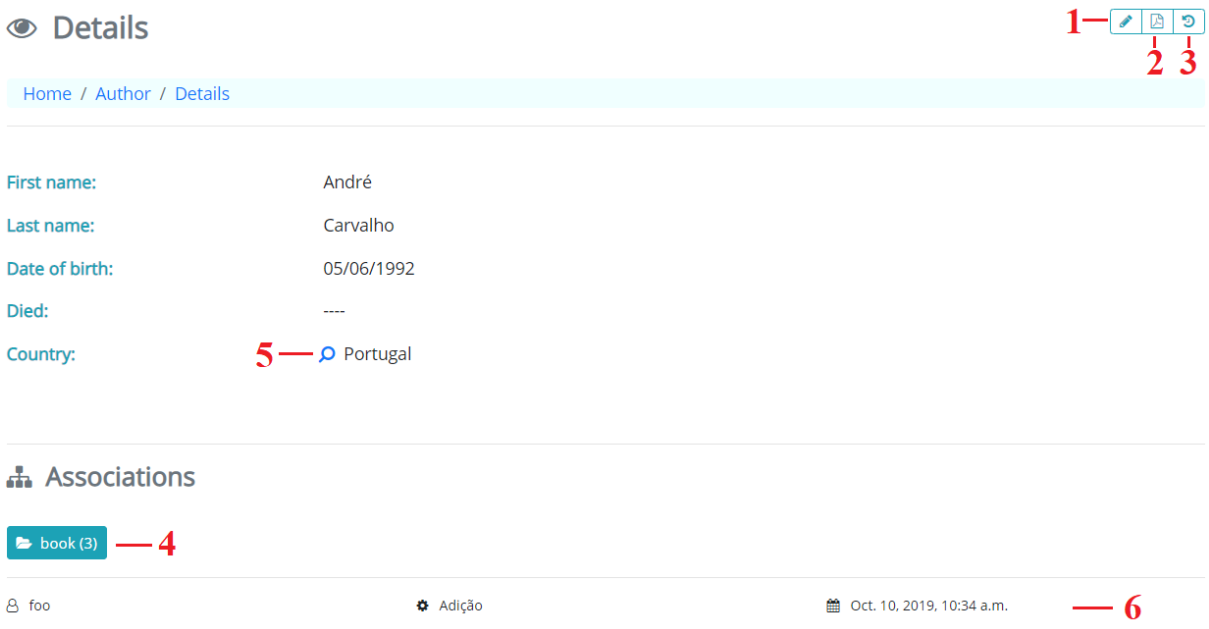



Figure 60 – Display template

6. ***Add and Edit***

These interfaces allow users to add or edit a record and perform the *POST* if the *form* is valid. If the user actions have a previous entity context (*FKs*), the platform will automatically fill those fields. In Figure 61, we can see that by selecting the list of books of the author *Andre* (*e.g.; Figure 60*), and by choosing a book from the following contextualized list, when editing a book, the author's input fills and disables.

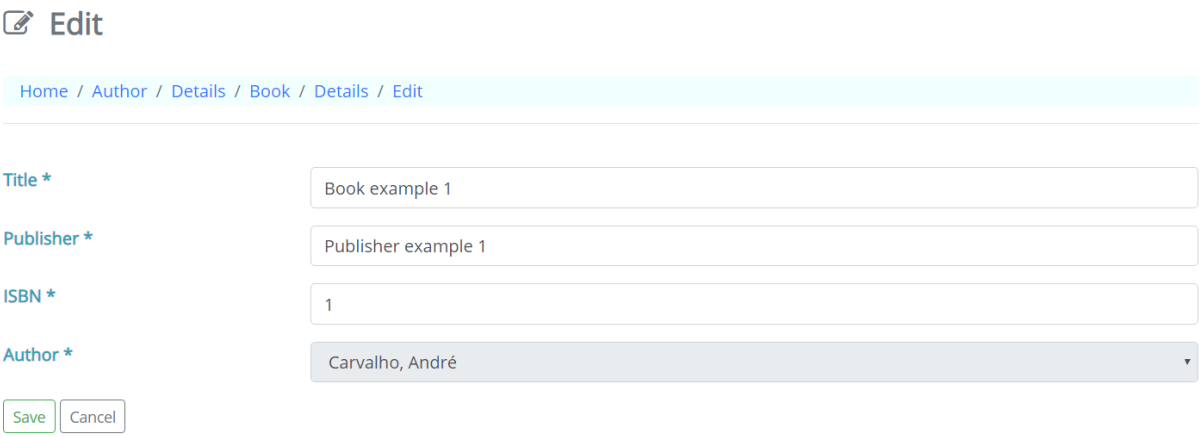



Figure 61 - Edit interface

### 7. *History list*

As for auditing, it can be towards the whole entity or a record of the *model* in question. The *template* is very intuitive and includes a table sorted by date of modification. Each row links to a *template* that displays more detailed information about the auditing.


Figure 62 - Auditing template

### 8. *History comparison*

The history comparison interface is also very intuitive and follows the previous one. This *template* recalls a version controller, as it compares the selected entry to the current one. This interface is essential for administrating purposes, as it is possible to understand the changes and when they took place.

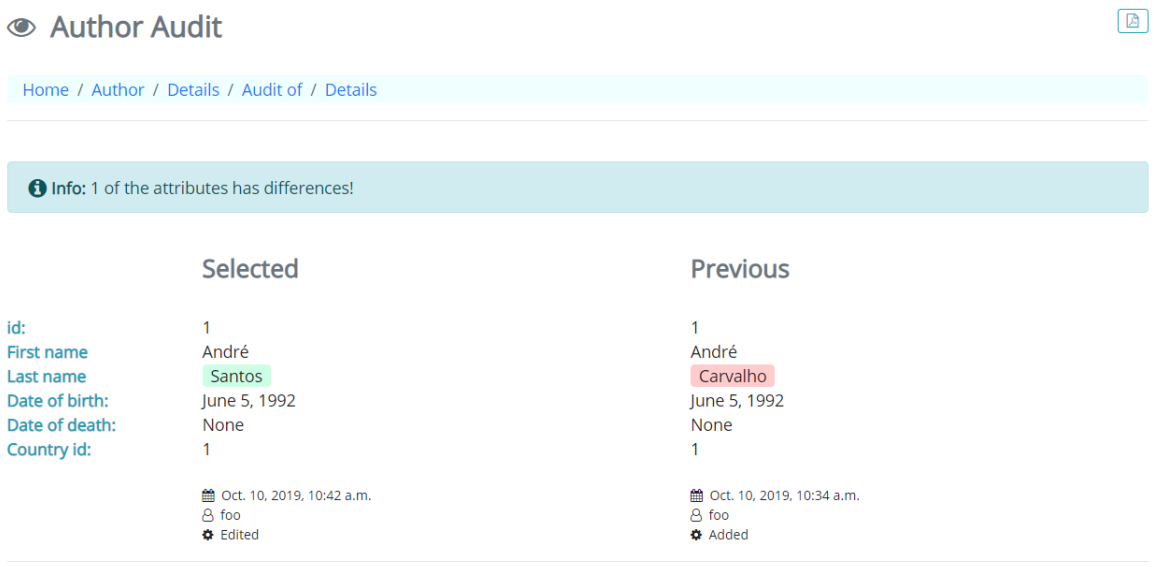



Figure 63 - Auditing comparison template

## 5.2. Proof of concept

While developing our solution, we used different data models, to verify its dynamism, by testing the solution towards different data and relationship types, so we could add complexity and validate its proper functioning. This process was valuable since it enabled us to verify all potential scenarios and to adapt modules if required.

But one of the essential proofs of concept to validate the dynamism of our platform, and to show the ease of adding business rules, was the back-end platform built for East-Timor Parliament, as it included almost all data types and relationships. We gather the requirements and designed a structure according to the stakeholders. When settled, this database contained 36 tables. By using our template, along with the generating script, we produce the management platform for this same data model. For the stakeholders, the result provided full control over all models and ease of navigating through their intuitive relationships. Besides, our stakeholders needed more functionalities, and that is where the control *parameters* (*e.g.*, *Figure 32*) are convenient by allowing us to control some platform functionalities. Furthermore, by overriding the *views* for each of the *models*, it was possible to insert new business rules. The modules design and the way they relate (loose coupling) allowed adding business rules without influencing the rest of the project structure.

This final product, addressed in Appendix D, is in use and is the basis for the back-end of the East-Timor Parliament, where we also integrate a workflow engine. Besides, we created additional services and a front-end platform to access the data entered and validated by our back-end solution.

With this proof of concept, we could verify the speed of implementation of a data management platform and the simplicity and dynamism our solution offers.

We have also found that including new business rules is straightforward for a person who knows the structure of our solution and the way its *components* are connected. By looking at our project structure, it is easy to understand which parameters to change and what to override to implement new business logic.

We tested this by presenting the programming mechanisms to people I work with and letting them perform actions towards the system structure. The procedures were experiences that pass by:

i. Letting the users set a database model and change the settings;
ii. Explaining how they could use the project template, and run the generating script;
iii. Let them verify how the script changes the template to build the solution, based on the chosen models (also presented by the script output, *Figure 46*).
iv. And, by explaining how they could change the *parameters* (*e.g.*, *Figure 32*) to control the views regarding each model.

After these tests, we can say that our solution is easy to set up by executing a script, regardless of the imposed data models. And if well documented, programmers can add further business logic.

# 6. CONCLUSION

[9]

*During this chapter, we will perform a brief conclusion about our dissertation work. We provide an answer to the research in question, summarize and discuss on the research steps, make recommendations, and emphasize our contribution. Afterward, we will address future work, talking about what we propose to add or change in the current solution.*

## 6.1. Conclusion

The aim of this dissertation was to design a back-end generator that could provide dynamic back-end solutions to handle databases regardless of the imposed data models and which facilitates including new business rules.

After our research towards other platforms that perform similar ideas, we gather the main concepts and functionalities, which were crucial for developing our platform. As a result, we could

[9] *https://images.axios.com/5Xnh73CKIegTIA2TSyhbKkt3Cag=/0x0:5760x3240/1600x900/2018/06/28/1530195505707.jpg*

incorporate attractive requirements and build a product that could deliver functionalities that other solutions did not.

We produce what we purpose, a tool for the *Django* framework, that provides mechanisms to control databases and apply business rules. Comparing to other solutions, like the *Django admin*, our platform has the main advantage of contextualizing the relationships between data through its interface. By presenting intuitive model relations and filling the forms based on the platform context, leading to lesser inconsistencies or insertion errors, promoting comprehensibility. The platform works as a tool that programmers can use as a template, which can become the basis for more elaborate solutions. By using it, users do not have to be programming experts. They must only provide a data model, and it is possible to generate whole project structures to control that data.

## 6.2. Future work

We can verify that the system does not contain some features that could be a bonus for users to choose it over other solutions.

By using our platform, the stakeholders realized that they needed more types of users and roles. We need to improve the authentication and authorization mechanisms that we implemented. It is necessary to design a strategy to add different users besides staff and administrators, and we must create access control mechanisms for application content. There are differences between the authentication and authorization mechanisms. Authentication involves verifying if a user logs into a system by credential verification, for which we are using Django's default mechanisms. Authorization requires checking the user and defining what they can do in the application, for which we are using Django's permissions framework along with view-related decorators. Regarding the future work, we must come with new modules that can extend these mechanisms to the problem at hand.

We might have to establish additional control parameters to include new and easy logic control to the default solution (*e.g.*, *Figure 32*).

There are already newer versions of Django, updating the framework, is an advantage concerning the extended support.

We must create an interface concerning the generating script, providing a pleasant graphical interface to enter the data and build the project, instead of adding everything through the terminal.

To finish, we need to create documentation to help when using the template and script. It is always an essential part of the software for developers, as it improves the development time and knowledge transfer.

# REFERENCES


[1] K. M. Wilburn and H. R. Wilburn, “The impact of technology on business and society,” vol. 12, no. 1, pp. 23–39, 2018.

[2] P. K. Nikoloski, “The Role of Information Technology in the Business Sector,” vol. 3, no. 12, pp. 303–309, 2014.

[3] T. Sharp, Helen and Hall, *Agile Processes in Software Engineering and Extreme Programming*. 2016.

[4] Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin, “Speculative Analysis of Integrated Development Environment Recommendations,” *ACM SIGPLAN Not.*, vol. 47, no. 10, pp. 669–682, 2012.

[5] J. Yang, S. Lu, and A. Cheung, “How not to structure your database-backed web applications : a study of performance bugs in the wild ∗,” *2018 IEEE/ACM 40th Int. Conf. Softw. Eng.*, pp. 800–810, 2018.

[6] C. Lee and Y. Zheng, “SQL-to-NoSQL Schema Denormalization and Migration : A Study on Content Management Systems,” *2015 IEEE Int. Conf. Syst. Man, Cybern.*, pp. 2022–2026, 2015.

[7] N. A. Khan and H. Ahangar, “Use of Open Content Management Systems in Government Sector,” *2018 5th Int. Symp. Emerg. Trends Technol. Libr. Inf. Serv.*, pp. 183–187, 2018.

[8] A. Krouska, C. Troussas, and M. Virvou, “Comparing LMS and CMS platforms supporting social e-learning in higher education,” *2017 8th Int. Conf. Information, Intell. Syst. Appl. IISA 2017*, vol. 2018-January, pp. 1–6, 2018.

[9] J. Cabot, “A Content Management System to Democratize Publishing,” *IEEE Softw.*, vol. 35, no. 3, pp. 89–92, 2018.

[10] Wordpress, “Blog Tool, Publishing Platform, and CMS - WordPress.org Portugal,” 2019. [Online]. Available: https://pt.wordpress.org/.

[11] N. F. W. Saunders and B. Kobe, “The Predikin webserver : improved prediction of protein kinase peptide specificity using structural information,” *Nucleic Acids Res.*, vol. 36, no. 2, pp. 286–290, 2008.

[12] J. B. Patel, Savan K and Rathod, VR and Prajapati, “Performance Analysis of Content Management Systems-Joomla, Drupal and WordPress,” *Int. J. Comput. Appl.*, vol. 21, no. 4,

pp. 39–43, 2011.

[13] Joomla, “The Flexible Platform Empowering Website Creators,” 2019. [Online]. Available: https://www.joomla.org/.

[14] A. Garza, “From OPAC to CMS: Drupal as an extensible library platform,” *Libr. Hi Tech*, vol. 33, no. 1, pp. 252–267, 2015.

[15] A. Velios and A. Martin, “Off-the-shelf CRM with Drupal: a case study of documenting decorated papers,” *Int. J. Digit. Libr.*, vol. 18, no. 4, pp. 321–331, 2017.

[16] Drupal, “Drupal,” 2019. [Online]. Available: https://www.drupal.org/.

[17] U. A. CONSUMERS, “Becoming an online business owner,” 2015.

[18] Shopify, “Shopify,” 2019. [Online]. Available: https://www.shopify.com/.

[19] H. and others Aad, Georges and Abbott, B and Abdallah, J and Abdelalim, AA and Abdesselam, A and Abdinov, O and Abi, B and Abolins, M and Abramowicz, H and Abreu, “The ATLAS Simulation Infrastructure,” *Eur. Phys. J. C Part. Fields*, vol. 70, no. 3, pp. 823–874, 2010.

[20] M. Kassab, M. Mazzara, J. Lee, and G. Succi, “Software architectural patterns in practice : an empirical study,” *Innov. Syst. Softw. Eng.*, vol. 14, no. 4, pp. 263–271, 2018.

[21] T. Reenskaug, “The Model-View-Controller ( MVC ) Its Past and Present,” *Univ. Oslo Draft*, pp. 1–16, 2003.

[22] S. Burbeck, “Applications Programming in Smalltalk-80 ( TM ): How to use Model-View-Controller ( MVC ),” *Smalltalk-80 v2*, vol. 80, no. Mvc, pp. 1–11, 1992.

[23] S. Lappalainen and T. Kobayashi, “A Pattern Language for MVC Derivatives,” *Proc. 6th Asian Conf. Pattern Lang. Programs*, 2017.

[24] H. Sulaiman and N. Jamil, “Information Security Governance model to enhance zakat information management in Malaysian Zakat Institutions,” *Proc. 6th Int. Conf. Inf. Technol. Multimed.*, pp. 200–205, 2014.

[25] V. Mallawaarachchi, “10 Common Software Architectural Patterns in a nutshell,” 2019. [Online]. Available: https://towardsdatascience.com/10-common-software-architectural-patterns-in-a-nutshell-a0b47a1e9013.

[26] H. S. Oluwatosin, “Client-Server Model,” *IOSRJ Comput. Eng*, vol. 16, no. 1, pp. 2278--8727, 2014.

[27] C. Kambalyal, “3-Tier Architecture,” vol. 2, pp. 1–19.

[28] P.-Y. Chan, Lee-Kwun and Lau, “Investigating the Impact of System Quality on Service-Oriented Business Intelligence Architecture,” vol. 8, no. 4, 2018.

[29] L. Shashwat, Anurag and Kumar, Deepak and Chanana, “Message Level Security Enhancement For Service Oriented Architecture,” *2018 4th Int. Conf. Comput. Intell. \&*

*Commun. Technol.*, pp. 1–6, 2018.

[30] N. Komoda, “Service Oriented Architecture ( SOA ) in Industrial Systems,” *2006 4th IEEE Int. Conf. Ind. Informatics*, pp. 1–5, 2006.

[31] M. Richards, *Software Architecture Patterns*. 2015.

[32] L. Zheng and B. Wei, “Application of microservice architecture cloud environment project development,” *MATEC Web Conf.*, vol. 189, pp. 3–23, 2018.

[33] C. Fok, Chien-Liang and Roman, Gruia-Catalin and Lu, “Servilla : A flexible service provisioning middleware for heterogeneous,” *Sci. Comput. Program.*, vol. 77, no. 6, pp. 663–684, 2012.

[34] J. Thönes, “Microservices,” *IEEE Softw.*, vol. 32, no. 1, 2015.

[35] Z. Xiao, I. Wijegunaratne, and X. Qiang, “Reflections on SOA and Microservices,” *2016 4th Int. Conf. Enterp. Syst.*, pp. 60–67, 2016.

[36] S. Khwaja and M. Alshayeb, “A Framework for Evaluating Software Design Pattern Specification Languages,” *2013 IEEE/ACIS 12th Int. Conf. Comput. Inf. Sci.*, pp. 41–45, 2013.

[37] GitHub, “Web application frameworks.” .

[38] “Web framework rankings | HotFrameworks.” [Online]. Available: https://hotframeworks.com.

[39] Z. Li, Shenliang and Si, “Information Publishing System Based on the Framework of Django,” *China Acad. Conf. Print. \& Packag. Media Technol.*, pp. 375–381, 2016.

[40] A. Pinkham, *Django Unleashed*. 2015.

[41] T. P. S. Foundation, “Pip install,” 2019. [Online]. Available: https://pip.pypa.io/en/stable/reference/pip_install/.

[42] T. Chen, S. Member, W. Shang, and Z. M. Jiang, “Finding and Evaluating the Performance Impact of Redundant Data Access for Applications that are Developed Using Object-Relational Mapping Frameworks,” *IEEE Trans. Softw. Eng.*, vol. 42, no. 12, pp. 1148–1161, 2016.

[43] and F. Q. Chuanhong Zhou, Chao Dai, Pujia Shuai, “Development on Management System of Automated High-Rise Warehouse for Mid-Small Enterprises Based on Django,” *Int. Work. Adv. Manuf. Autom.*, pp. 71–79, 2017.

[44] Express, “Frameworks built on Express,” 2019. [Online]. Available: https://expressjs.com/en/resources/frameworks.html.

[45] M. and individual Colaborators, “Introdução a Express/Node.” [Online]. Available: https://developer.mozilla.org/pt-PT/docs/Learn/No-servidor/Express_Nodejs/Introduction.

[46] S. Tian, Y and Chen, ZC and Zhou, HF and Wu, LJ and Long, C and Lin, PJ and Cheng, “An online health monitoring system for photovoltaic arrays based on the B / S architecture An online health monitoring system for photovoltaic arrays based on the B / S architecture,” *IOP Conf. Ser. Earth Environ. Sci.*, vol. 188, no. 1, pp. 12–64, 2018.

[47] R. Patil, Suprita M and Vijayalashmi, M and Tapaskar, “IoT based solar energy monitoring system,” *2017 Int. Conf. Energy, Commun. Data Anal. Soft Comput.*, pp. 1574–1579, 2017.

[48] Armin Ronacher, “Welcome, Flask (A Python Microframework),” 2019. [Online]. Available: http://flask.pocoo.org/.

[49] J. Jimenez, Andres and Garcia-Diaz, Vicente and Anzola, “Design of a System for Vehicle Tra ic Estimation for Applications on IoT,” *Proc. 4th Multidiscip. Int. Soc. Networks Conf.*, p. 15, 2017.

[50] N. Solanki, D. Shah, and A. Shah, “A Survey on different Framework of PHP,” *Int. J. Latest Technol. Eng. Manag. \& Appl. Sci.*, vol. VI, no. Vi, pp. 155–158, 2017.

[51] Y. Chen, Xianjun and Ji, Zhoupeng and Fan, Yu and Zhan, “Restful API Architecture Based on Laravel Framework,” vol. 910, no. 1, pp. 12–16, 2017.

[52] V. V Parkar, P. P. Shinde, S. C. Gadade, and P. M. Shinde, “Utilization of Laravel Framework for Development of Web Based Recruitment Tool,” *IOSR J. Comput. Eng.*, pp. 36–41, 2016.

[53] and J. V. McCord, Chris, Bruce Tate, *Programming Phoenix: Productive|> Reliable|> Fast*. 2016.

[54] Phoenix, “Phoenix Overview,” 2019. [Online]. Available: https://hexdocs.pm/phoenix/overview.html#a-note-about-these-guides.

[55] H. Smidt and M. Thornton, “Smart Application Development for IoT Asset Management Using Graph Database Modeling and High-Availability Web Services,” *Proc. 51st Hawaii Int. Conf. Syst. Sci.*, vol. 9, pp. 5787–5796, 2018.

[56] F. P. Conter, “UTILIZAÇÃO DO FRAMEWORK " RUBY ON RAILS " NO DESENVOLVIMENTO DE UM MÓDULO WEB PARA UM SISTEMA DE BIBLIOTECA Objetivo Metodologia Resultado Bibliografia,” p. 20020101, 2007.

[57] R. on Rails, “Rails has everything you need,” 2019.

[58] C. and others Pandey, Himani and Rastogi, Himani and Gupta, “Web-Based Network Management System implemented using Hibernate , JBoss and Spring Framework,” pp. 1–5, 2017.

[59] A. Srai, F. Guerouate, N. Berbiche, H. D. Lahsini, C. Prince, and S. B. P. Sallee, “Applying MDA approach for Spring MVC Framework,” vol. 12, no. 14, pp. 4372–4381, 2017.

[60] StackShare, “What are some alternatives to trending open source and SaaS tools in 2019?” [Online]. Available: https://stackshare.io/stackups/trending.

[61] Owasp.org, “Category:OWASP Top Ten Project - OWASP.” [Online]. Available: https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project.

[62] Techempower, “TechEmpower Framework Benchmarks.” [Online]. Available: https://www.techempower.com/benchmarks/.

[63] V. Bhagat and A. Gopal, “Roll of Relational Algebra and Query Optimizer in Different Types of DBMS,” vol. 5, no. 3, pp. 6–16, 2016.

[64] J. Letkowski, “Doing database design with MySQL,” vol. 6, p. 1, 2015.

[65] A. Boicea, F. Radulescu, and L. I. Agapin, “MongoDB vs Oracle - database comparison,” *2012 Third Int. Conf. Emerg. Intell. Data Web Technol.*, pp. 330–335, 2012.

[66] G. Gallas, EJ and Canali, L and Vasileva, P and Dumitru, Andrei and Baranowski, Z and Formica, A and Dimitrov, “An Oracle-based event index for ATLAS,” vol. 898, pp. 42–33, 2017.

[67] T. Masunaga, Yoshifumi and Nagata, Yugo and Ishii, “Extending the View Updatability of Relational Databases from Set Semantics to Bag Semantics and Its Implementation on PostgreSQL,” *Proc. 12th Int. Conf. Ubiquitous Inf. Manag. Commun.*, p. 19, 2018.

[68] S. Sultana and S. Dixit, “Indexes in PostgreSQL,” *2017 Int. Conf. Innov. Mech. Ind. Appl.*, pp. 512–515, 2017.

[69] Sqlite, “About SQLite,” 2019. [Online]. Available: https://www.sqlite.org/about.html.

[70] Django Software Foundation, “Django Databases,” 2019. [Online]. Available: https://docs.djangoproject.com/en/1.11/ref/databases/.

[71] Django Software Foundation, “Django Migrations,” 2019. [Online]. Available: https://docs.djangoproject.com/en/1.11/topics/migrations/#backend-support.

[72] Wood and G. David and Bruner, Jerome S and Ross, “The role of tutoring in problem solving,” *J. child Psychol. psychiatry*, vol. 17, no. 2, pp. 89–100, 1976.

[73] P. Tonkovikj and J. Evans, “Product Licenses Database Application,” *Eur. Organ. Nucl. Res.*, 2016.

[74] Yeoman, “Generators | Yeoman.” [Online]. Available: https://yeoman.io/generators/.

[75] Expressjs, “Express application generator.” [Online]. Available: https://expressjs.com/en/starter/generator.html.

[76] S. (simov), “Express Admin.” [Online]. Available: https://simov.github.io/express-admin/.

[77] Npm, “express-admin.” [Online]. Available: https://www.npmjs.com/package/express-admin.

[78] A. Verma, “MVC ARCHITECTURE : A COMPARITIVE STUDY BETWEEN RUBY ON RAILS AND LARAVEL,” vol. 5, no. 5, pp. 196–198, 2014.

[79] GitHub, “JeffreyWay/Laravel-4-Generators.” [Online]. Available: https://github.com/JeffreyWay/Laravel-4-Generators.

[80] Laravel, “Artisan Console - Laravel - The PHP Framework For Web Artisans.” [Online]. Available: https://laravel.com/docs/5.8/artisan.

[81] GitHub, “laracasts/Laravel-5-Generators-Extended.” .

[82] “Laravel Generator.” [Online]. Available: http://labs.infyom.com/Laravelgenerator/.

[83] “Laravel Voyager.” [Online]. Available: https://laravelvoyager.com/.

[84] Spring, “Spring Initializr.” [Online]. Available: https://start.spring.io/.

[85] Spring, “Spring Projects.” [Online]. Available: https://spring.io/projects/Spring-boot.

[86] C. C. A.-S. A. 3.0, “Ruby on Rails Guides: Getting Started with Rails,” 2019. [Online]. Available: https://guides.rubyonrails.org/v3.2.13/getting_started.html#getting-up-and-running-quickly-with-scaffolding.

[87] “Active Admin | The administration framework for Ruby on Rails.” [Online]. Available: https://activeadmin.info/.

[88] D. S. Foundation, “The Django admin site,” 2019. [Online]. Available: https://docs.djangoproject.com/en/2.2/ref/contrib/admin/.

[89] “Django Packages : Admin Interface.” [Online]. Available: https://djangopackages.org/grids/g/admin-interface/.

[90] “Django Packages : Admin Styling.” [Online]. Available: https://djangopackages.org/grids/g/admin-styling/.

[91] “Django Packages : Admin Addons.” [Online]. Available: https://djangopackages.org/grids/g/admin-addons/.

[92] “Django Packages : Scaffolding.” [Online]. Available: https://djangopackages.org/grids/g/scaffolding/.

[93] “django-crudbuilder : Generic way to create model CRUD — django-crudbuilder 0.1.5 documentation.” [Online]. Available: https://django-crudbuilder.readthedocs.io/en/latest/index.html.

[94] “spapas/django-generic-scaffold.” [Online]. Available: https://github.com/spapas/django-generic-scaffold/.

[95] “Welcome to django-popupcrud’s documentation! — django-popupcrud 0.7.1 documentation.” [Online]. Available: https://django-popupcrud.readthedocs.io/en/latest/.

[96] “bmihelac/django-cruds.” [Online]. Available: https://github.com/bmihelac/django-cruds.

[97] “Static.djangoproject.com.” [Online]. Available: https://static.djangoproject.com/img/release-roadmap.688d8d65db0b.png.

[98] Microsoft, “Manage Python environments and interpreters - Visual Studio,” 2019. [Online]. Available: https://docs.microsoft.com/en-us/visualstudio/python/managing-python-environments-in-visual-studio?view=vs-2019.

[99] Django Software Foundation, “URL dispatcher | Django documentation | Django,” 2019. [Online]. Available: https://docs.djangoproject.com/en/1.11/topics/http/urls/.

# APPENDIX

## A. Databases comparison

To complement the database comparison, we built a table that summarizes information about each of them taking. Table 4 is informative, it translates the information gathered from the *SGBDs* official pages, and the *Django* documentation regarding the supported *SGBDs*.

Table 4 - SGBD comparisons

| **RDBSM**<br>**Info** | **MySQL** | **Oracle** | **PostgreSQL** | **SQLite** |
|---|---|---|---|---|
| Developer | Oracle | Oracle | PostgreSQL | D. Richard Hipp |
| First Release | 1995 | 1980 | 1996 | 2000 |
| Use terms | Free and Commercial | Free and Commercial | Free | Free |
| *Django* Support versions | 5.5.x - 5.7.x. *MySQL* 8 and later aren't supported. | Oracle Database Server 11.2+ and cx_Oracle *Python* driver from 5.2 through 6.4.1 | 9.3+ | SQLite 3.6.21+ |
| *Django* extensions needed | MySQLdb version 1.2.3 or later | Privileges to run a list of commands | psycopg2 from version 2.5.4 through 2.7.7 | pysqlite2 or sqlite3 |
| *Django* schema support | No support for transactions around schema alteration operations | Not address on documentation | Default value columns need full table rewrites | Must be emulated by *Django* |

# B. Implemented solution mechanism overview

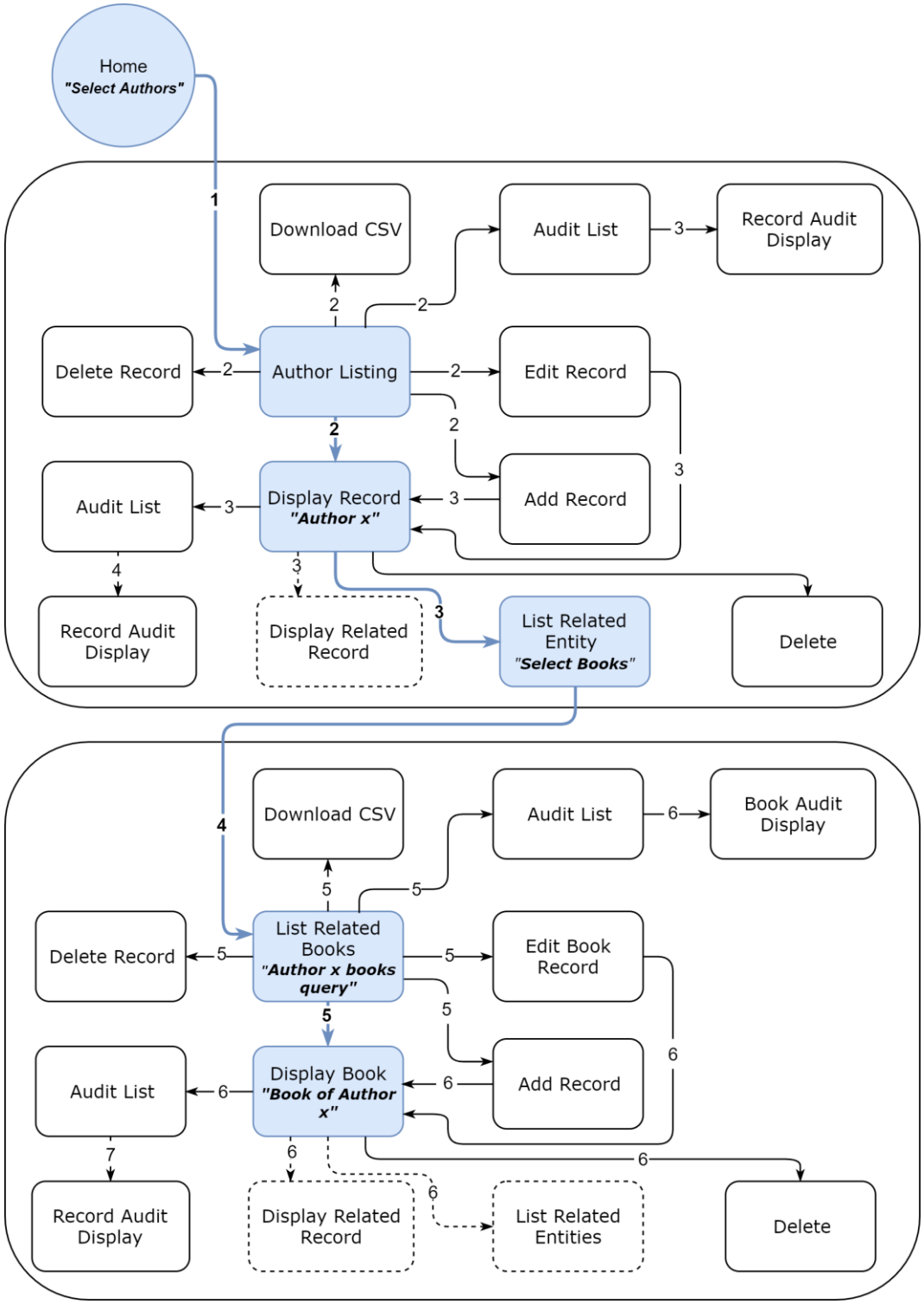



Figure 64 - Relationships for list of related entities

This appendix intends to display the interaction mechanisms already described in the dissertation structure. Figure 64 and Figure 65 relate to Figure 25. Meaning that, for each colored circle, there is a *URL* that calls a *view*, which uses and processes the *model* to get information, so then it can render variables to the *template*. Figure 64 describes the related entity lists, which are all the entities that reference the entity in question. Figure 65 describes the display-related record mechanisms, which are the entities referenced by the entity in question. This is the way we relate the data in our system platform.

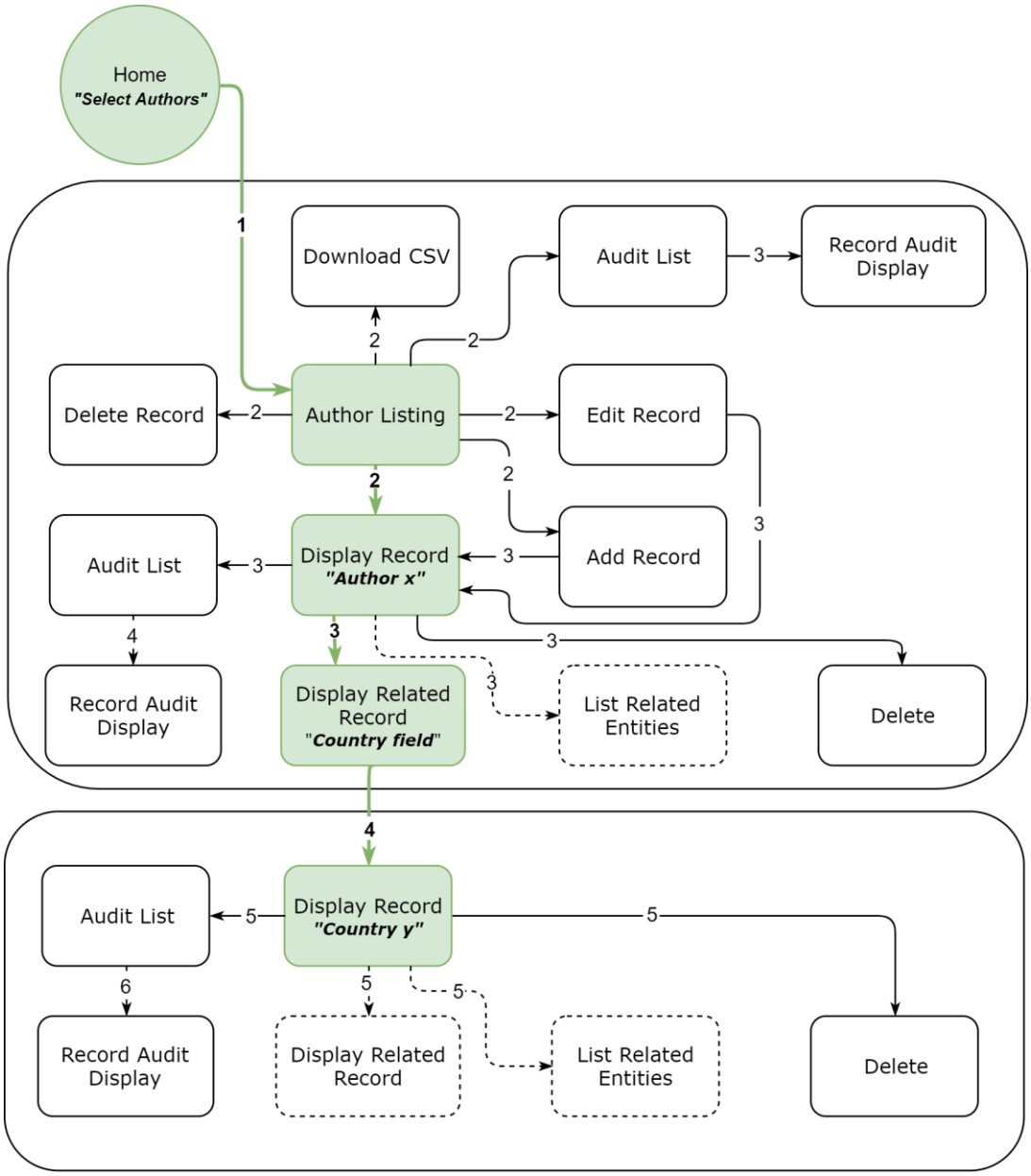



Figure 65 - Relationships for related records

# C. Templates

We will present an overview of all the templates of our solution.

1. ***App***

   The app contains the initial templates of our application.

Table 5 - App templates

| template | description |
|---|---|
| layout | Where we load the static files and define the links and scrips. Then we have the body, which contains the navigation bar and where we define a container with the *{% block content %}*, so that all the other *HTML* can extend this layout. By doing so, all the other templates can use the header and footer. |
| index | Presents the entities of the system and their info, as already presented (*e.g., Figure 54*). |
| login | Is regarding the first page that appears when a user logs into our platform (*e.g., Figure 47*). |
| *loginpartial* | Is a component of the header present in the *layout.html*, that provides information about the user, a link to the user administration area, a link to the translation's drop-down template, and the button to log off (*e.g., Figure 55*). |

2. ***Components***

   The components aggregate all the application-wide sub-templates.

Table 6 - Components templates

| template | description |
|---|---|
| *breadcrumbs* | Trails all the steps a user performed. |
| *permission* | Triggers every time a user does not have permission to carry an action or access content. |
| *translations* | Creates the languages drop-down of the application, used by the *loginpartial* template. |
| *validation_ messages* | Presented in the dissertation structure (*e.g., Figure 42*). |

3. ***Display***

   For the display, we have three main templates that use the sub-templates.

Table 7 - Display templates

| template | description |
|---|---|
| *CRUD_display* | Template (*e.g., Figure 60*) which aggregates most of the components to |

| | |
|---|---|
| | build the interface. |
| *audit_display* | Like *CRUD_display*, but it applies different rules about the audit itself. |
| *users_display* | Like *CRUD_display*, but with specific changes regarding the users. |

Table 8 - Display sub-templates

| template | description |
|---|---|
| *audit* | Component that contains audit information of a specific entity changes (user, operation, date). |
| *delete* | Modal form to perform the delete operation. |
| *dependents* | Presents the entities which relate to the specific displayed record. |
| *field_types* | Already addressed during the dissertation structure (*e.g.*, *Figure 43)*. |
| *header_display* | Holds icons that link the user to the edit template, the audit template, and the retrieval of the *PDF*. |
| *fields_references* | Generates links to the display fields that contain relations with other records. |

4. ***Edit_add***

It is the simplest of the templates that include a generic edit/add template and one sub-template related to all the field specifications.

5. ***List***

It includes two main templates and seven sub-templates.

Table 9 - List templates

| template | description |
|---|---|
| *CRUD_list* | Aggregates all the sub-templates in the desired way. |
| *audit_list* | Like *CRUD_list*, but build towards the audit. |

Table 10 - List sub-templates

| template | description |
|---|---|
| *add* | It links the user to the addition template, which can have context or not, depending on the listing. |
| *delete* | Modal that allows deleting list records. |
| *header_list* | Aggregates the header links, including the *CSV* file, the audit, and the title. |
| *number_of_entries* | Counts the number of listing entries and the current page. |
| *pagination* | To build the paginations. |
| *search* | The input to perform search towards the listing entries. |
| *table* | To create the tables. |

6. ***Registration***

Includes all the templates regarding the password reset:

Table 11 - Registration templates

| template | description |
|---|---|
| *reset_form* | To choose the email where the user wants to proceed with the password reset (*e.g., Figure 48*). |
| *reset_done* | Informs the user know about the email (*e.g., Figure 49*). |
| *reset_email* | Builds the email that the user will receive (*e.g., Figure 50*) |
| *reset_confirm* | Where the user introduces the new password (*e.g., Figure 51*). |
| *reset_complete* | To inform the user about the successful password change (*e.g., Figure 52*). |

# D. East Timor Solution

This solution follows our template and script, but we changed it to be in line with what the stakeholders needed. During this section, we will present some changes and explain how we added further business logic.

This solution had 36 tables, so it was imperative that our design work flawlessly, which required testing and validating the output. Figure 67 presents the tables to provide a perspective on the number of relations between the entities. We adapt the default solution header to exhibit the more important ones from which we can navigate to all others, changing the default *index.html*.


Figure 66 - Header of East Timor platform

To transform this solution and adapt to the stakeholders, we follow two steps:

i. Control the functionalities and templates by using the view parameters (*e.g.*, *Figure 32*), like the titles, permissions, and templates;
ii. Override or introduce new views without destroying the system relationships.

Concluding that because of the well-defined structure, we could make immediate changes, and introduce new business logic. Another example of the dynamism is the ease to add functionalities, that we prove by integrating an external workflow engine.

Figure 67 - East Timor database test case