Politecnico di Torino

Master's degree in Computer Science

Realization of a SaaS Web application for the engineering and automation of management control phases of a company

Curriculum Software

Candidate
Xhoi Kerbizi

Supervisor
Prof. Maurizio Morisio

ACCADEMIC YEAR 2019/2020

Abstract

This thesis is dedicated to the study, the design, the implementation and the deployment of a Software as a Service (SaaS) web application commissioned by the company where the thesis work has been carried out. The final aim of the application is to configure a centralized system in which the administrators are allowed to perform management operations and evaluate the financial health of the company, and the employees can have a clear working plan. Among the required functionalities there are the management of orders, order types, revenues, comments, revenue objectives and behavioural reports. The required functionalities are common to any company, the application is therefore very general and can be shared through a subscription to more customers, exploiting in this way the multi-tenancy feature.

In the application, two user roles are defined, namely the Operator and the Collaborator. According to the role and its assigned privileges, the user can access to all or some of these functionalities. The Operator corresponds to the administrator of the platform and is responsible to register the orders received from the customers. For each order, the administrator defines a revenue plan and assigns the order to a team of collaborators, who is responsible to achieve its billing, namely all the imports of the revenue plan of that order are paid by the customers. Each collaborator follows the orders assigned to him updating the advancement state of their revenues, until the payment of each of them is completed and marks them as terminated. In addition, chat-based conversations between team of collaborators and operator are possible, where the revenue state of an order is clearly communicated, so that the administrators can take further measurements in case the deadlines are not respected. Moreover, the Operator can access to a dashboard, where pie-chart graphs show the company revenues grouped by different data (order type, assigned collaborator user), and a behavioural report illustrates the company revenues per month and per year, allowing to make comparisons between monthly revenues objectives of the current year and monthly revenues of any year (current or past). The revenue objectives and the comparison year can be set manually by the Operator.

Concerning the technology adopted to achieve the thesis goal, the ASP.NET Zero solution has been used. In particular the frameworks .NET Core, ASP.NET Core and ASP.NET Boilerplate have been used for the backend, whereas the Angular framework has been used for the frontend. A detailed study of these frameworks was necessary to achieve the thesis goal.

The thesis is organized as follows. The SaaS applications are introduced in Chapter. 1. The main concepts of the SaaS software model are described and examples of the most popular applications based on it are reported. The advantages and disadvantages of the SaaS model, fundamental to understand how convenient it is for the companies to adopt this approach and for the customers to start subscriptions, are discussed. The available technologies for SaaS web development are listed, commented and compared to the ASP.NET Zero solution. The goal of the developed application is introduced, describing briefly the required functionalities and the technologies used to implement it.

Chapter 2 is dedicated to the design of the developed application. The data model and all the entities and their relationships are described in detail. Also, the user roles granted by this application are presented as well as the allowed functionalities for each user role.

Chapter 3 focuses on the architecture and on the structure of the application. The concepts of Domain Data Driven architectural pattern and the SOLID principles are introduced and the way how they are implemented by the ASP.NET Zero solution is explained. Afterwards, the overall application structure and its building sub-projects are described in detail, showing, for each of them, the own functionality and the exploited technology. Elements like application services, entities, DTOs, repositories, dependency injection, EntityFrameworkCore, angular modules and components are the main topic of this chapter. Also, all the steps needed in order to implement a feature in all the full stack are summarized.

Chapter 4 introduces the main concepts and techniques of the Agile process management model, concentrating on the Scrum methodology and its implementation in the present work.

The testing and the validation of the developed software are explained in Chapter. 5. The Agile testing is introduced and its possible implementations is followed by the current approach adopted in the present work. The type of tests done and how they are validated are also described.

In Chapter. 6 the deployment of the application on Microsoft Azure is presented and the procedures to achieve it are described in detail. Also some general concepts of the DevOps toolchain are described as well as the DevOps toolchain adopted in this work.

Finally, the thesis ends with the conclusions and the outlook.

Contents

To my family

Chapter 1

Introduction

The cloud-based *Software As A Service* (SaaS) model is now widespread all around the world. From the point of view of the customers, more and more client companies are looking for applications based on it, because it does not have installations problems or the signature of strict contracts, but it only requires an internet connection and a subscription (paid over the time and not all in once) in order to be used. Instead, regarding the software vendors, since the market of SaaS applications is in constant growth, they see a good opportunity on this software model, and therefore push for the realization of the SaaS version of their most important applications.

It all began in 1961, after a speech to MIT students given by John McCarthy, a famous computer scientist that won the Turing award for his work in artificial intelligence (AI), who *suggested* that computation may someday be organized as a public utility. In other words, the concept of cloud computing as a shared resource of computing power was born. After that occasion, the idea was around for some time but only in the late 1990s (time of internet expansion) the web-based technology needed to support SaaS became mature. At that time different companies, as for instance Salesforce, which was founded specifically to create cloud software, began offering traditional enterprise solutions, such as customer relationship management (CRM), through a SaaS model. The enterprise applications (EA) are applications designed specifically to meet the needs of an organization. Initially, the community delivering enterprise software did not take the SaaS paradigm seriously. Only in the following decade SaaS experienced a rapid growth and got an important placement in the software market.

Nowadays SaaS represents one of the main cloud technology models, together with *platform as a service* (PaaS) and *infrastructure as a service* (IaaS). Briefly, the first one provides a platform that allows customer companies to develop, execute, and manage their applications removing the complexity of creating and keeping the infrastructure, which is normally related to developing and launching an application. While, the second provides to the customers a virtual IT infrastructure, comprising both hardware and software, with more advantages than a traditional one.

After this short historical introduction, we proceed in the following sections focusing on the characteristics of SaaS web applications, on their differences with the standard web applications and finally introducing the goal of this work.

1.1 SaaS Web Applications

Software as A Service (SaaS) is a method of delivering software that allows users to access data from any device (i.e. computer, smartphone and tablet) using an internet connection and a web browser. Differently from the traditional web applications, the SaaS ones are hosted in a cloud environment, which is developed and maintained directly by the software producers or third-party ones are used (i.e. Microsoft Azure).

Moreover, in this model, software producers are responsible to manage the hosting and maintenance of the servers, databases and application code. While, the customers need to start a subscription in order to use the software, so they don't pay to own it, but to use it for as much time as needed. For this reason, SaaS is called alternatively "pay as you go".

Furthermore, SaaS applications are typically based on a *multi-tenant architecture* where a single instance of the application, with a single configuration and a single version, is shared to all customers, keeping their personal data and settings isolated. The configuration regards the choice of the hardware, the network and the operating system. The customers are identified in this architecture as *tenants*. A tenant could be seen as a group of users who access with the same privileges to the instance of the software in execution. Each of the tenants has its own data and configurations (beyond the configurations common to all tenants). In addition to the customer tenants, the host tenant is typically configured. This represents the owner of the SaaS application and it can be used to create and manage customer tenants. Moreover, the multi-tenant architecture helps to make easier and more efficient the maintenance of the software, allowing to provide quicker upgrades and bug fixes. The SaaS provider has only to apply changes on the single instance of the application and these are then propagated to all the customers, since they use the same instance.

There exist basically two approaches in order to implement the multi-tenant architecture in an application. The first one is by separating the tenants data logically, that is only one database for all tenants is used. Their data is separated by using a unique identifier for each tenant. While, alternatively there is the physical separation of data, where different databases for each tenant are provided. This helps to scale the SaaS application as the number of clients grows and also scale the database as per the clients need.

In order to ensure scalability, the SaaS application can be installed on different hosting servers, implementing the so called *horizontal scaling*. Indeed, a pre-realease version of the application is usually deployed in a staging (pre-production) hosting server, accessible to only a limited number of customers who are required to do some testing. The multi-tenant architecture is the opposite of that of a traditional software, where several physical copies of the software, each of a different version and with a different configuration, are installed in different customer establishments. As an exception, some SaaS applications do not implement a multi-tenancy architecture, but use other mechanisms, such as virtualization, to manage a large number of customers. Whether multi-tenancy is a necessary component for a SaaS application is a much discussed topic nowadays.

A wide range of providers and products are available in the SaaS market. The SaaS applications can be designed specifically to meet the needs of single users or of an whole organization. The most common examples of SaaS application are those related with customer relationship management (Salesforce, Oracle CX Cloud Suite), enterprise resource planning (Oracle ERP Cloud, Acumatica Cloud ERP), video conferences (Zoom), video streaming (Netflix), email management (Google Gmail), sales management (Groove) and many others. While among the most known SaaS providers there is Azure, Dropbox and Oracle. Azure is a Microsoft cloud computing service that allows developers to design, test, deploy, and manage applications and services using data centers managed by Microsoft. Dropbox, instead, is a file hosting service that offers services, like cloud storage, file synchronization, personal cloud and client applications. Finally, Oracle is a computer technology company specialized in development of cloud-based systems, enterprise software products and database software and technology.

However, before deciding to start a subscription for a SaaS software, the costumer must consider all the advantages and disadvantages this model presents.

1.2 Advantages and disadvantages of SaaS

Comparing to the traditional software, the Saas applications have several advantages. For instance as a first benefit, SaaS eliminates the need for client companies to install and execute applications on their own machines or in their own data centers. This is not at all a negligible feature since this allows them to remove from the *Total Cost Of Ownership* the expenses related to the IT infrastructure. The information technology (IT) infrastructure of an organization is the collection of IT components which provide an IT service. These can be computers, network components and even software. Therefore, the IT costs regard the acquisition and maintenance of the hardware and the software, as well as software installation and support.

Secondly, the SaaS model provides the possibility of more flexible payments since the customer is allowed to subscribe to a licence without the need of acquiring software to install or extra hardware components to aid the software. For instance, customers can use a *pay-as-you-go* model, namely they pay depending on the usage of the application. Alternatively, they can pay for a SaaS service on a fixed monthly (or annually) basis. Customers can also terminate SaaS licenses any time. Moreover, there are many other type of subscriptions with their particular pricing model which can be activated, depending on the customer's needs.

An other advantage of SaaS applications is the scalable usage, namely the software producers provide customers different and flexible type of subscriptions. For instance, the customers can start a subscription to use only some services offered by the SaaS application, or even can demand for new features to be added.

In addition, SaaS model provides also automatic updates, meaning that SaaS providers automatically perform updates and patches on the software, reducing the customer's IT costs.

The SaaS model enables accessibility and persistence, namely since SaaS applications are distributed over the Internet, users can access them from any device that can connect to Internet and from any location. In this case, only the strict polices of some countries can prevent the delivering of a SaaS application.

Finally, SaaS model allows also customization, meaning that SaaS applications are often scalable and can be combined with other business applications, mainly when using applications of the same SaaS producer.

Saas, of course, has also some disadvantages which must be mentioned. A relevant problem of this model is represented by the internet connection which both lies at the basis of the simplicity of using a SaaS application, but also at the same time it can be an unpleasant factor for the customers when it losses. It is fundamental for the company, which uses a SaaS application, to be able to adapt to eventual interruptions of the service, since it could, as the worst effect, produce money losses.

Other disadvantage is the fact that client companies must trust in third-party vendors who provide the software, maintain the software, monitor and report billing and provide a safe environment for the company data. In this situation, issues can occur when the software providers can not guarantee a constant delivery of the service or they change the service offerings without warning the customers first. But, the worst problem encountered by a software producer is the security breach, where the customer's data can be stolen. All these issues can prevent the customers' from using smoothly a SaaS application. Therefore, to proactively reduce the possibility of encountering them, customers should understand in depth the service-level agreement of the SaaS application provider and make sure it is enforced.

One other problem of the SaaS model is the fact the customers do not have any control on the version of the SaaS software they are using. For instance, it happens that a SaaS producer changes the version of software, propagating it to all the customers, also to those who are not interested on having the new version. This unwanted and unexpected upgrade may be a problem because the organization needs some time to train the employees in using the new version of the software.

Switching SaaS providers is really difficult and challenging for customers, because this comports to transfer massive quantities of data between cloud providers. In addition, the data transfer is further complicated by the fact that some vendors use proprietary technology and data formats. The situation in which a customer can not quickly switch between cloud providers is known as *vendor lock-in*.

Finally, security and privacy represent one of the most significant challenge for a SaaS application. In a traditional software, the provider is accountable for removing all the code vulnerabilities of the software, whereas the user is responsible to use a safe infrastructure and network in which to run the software. Instead, in a SaaS model, those who are responsible to provide security are the software vendors and third-party cloud providers.

The following section focuses on the different existing technologies for SaaS web application development and on the one which has been adopted for this project.

1.3 Available technologies for the SaaS development

Since a SaaS web application is basically a traditional web application with some more features such as multi-tenancy, any framework used for web development could be exploited in order to build it. All of them, indeed, are potentially able to implement the concepts of the multi-tenant architecture by using their own language, data structures and data access management. Nowadays, the most popular technologies, used on the backend side, for web development are *ASP.NET*, *ExpressJS*, *Django* and many others.

Starting from the first, ASP.NET [1] is a collection of libraries that form a huge and well-known framework for building modern web applications. It could be used only in Windows operating system, so this could represent an unpleasant limitation for many software producers, who could use alternatively *ASP.NET Core* [2] which is its multi-platform version. Both versions, however, use the C# language. Instead, ExpressJS [3] is one of the frameworks built on *Node JS* used for web development. It is multi-platform and uses Javascript as language. Moreover, it is supplied by several plugins which provide different useful features. Concerning the Django framework [4], it is based on Python language and supports the MVC architecture. It provides some security features, like SQL injection, click-jacking and request forgery.

However, in order to speed up the initial efforts for building the base of the applications, many solutions with built-in functionalities and based on some of these frameworks are provided. They require to start a subscription in order to be used and allow the developers to concentrate only in the new business logic, since all the other functionalities have already been implemented.

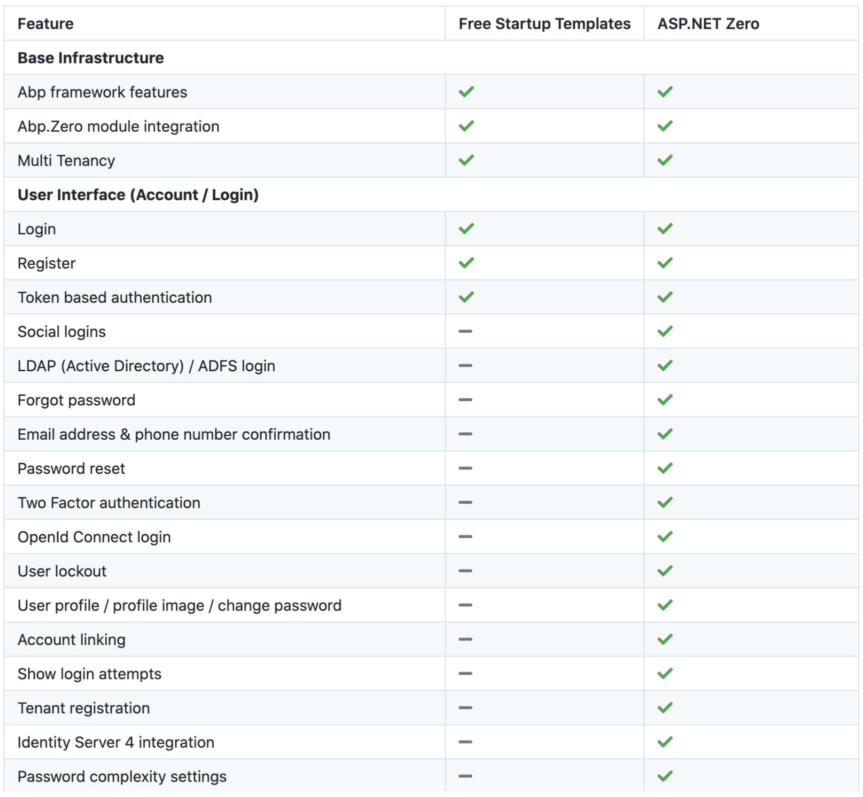
In this thesis work, the **ASP.NET Zero** [5] solution has been adopted, which, among all the solutions based on the ASP.NET Core framework, is the one with the major number of pre-implemented functionalities. Depending on the developer needs, the Zero solution provides different configurations for the backend and the frontend, for instance ASP.NET Core with Angular and ASP.NET Core MVC with JQuery. In the present work the first configuration has been adopted, where an Angular client application is used to query an ASP.NET Core application to get and visualize data and to allow user interactions.

The real utility of this solution is represented by the different built-in functionalities already implemented, so that the developers can only concentrate on developing their business logic, namely the customer's requirements. An extremely important built-in functionality is the multi-tenancy, fundamental for SaaS applications as explained in the previous section, which is managed from the solution in a clear and simple way. Regarding the tenant data separation approach, the configurations with a single database, one database per tenant and hybrid database are supported. An other functionality of the Zero solution is the localization service which allows to support various languages and therefore to extend the applications usage to customers of different nationalities. Indeed, it is possible to translate from different languages directly from the user-interface just by configuring simple .xml files in the backend. Moreover, the Zero solution offers a built-in mechanism for authentication and authorization. The former is composed of login, registration, password-reset and the tenant choice, whereas the latter is based on roles and permissions through the *aspect-oriented programming* (AOP). Moreover, also some cross-cutting concerns like logging, automatic audit/security logging, caching, exception handling and validation have already been implemented. Finally, the Zero solution provides an already implemented chat system and an internal mechanism for user notifications.

An important aspect of the Zero solution is that it is based on the *ASP.NET boilerplate* (ABP) framework [6], which, unlike the ASP.NET Zero, is open source (no subscriptions needed). Among the available configurations, ASP.NET boilerplate can be setup to use ASP.NET Core in the backend. Therefore it could be considered as an alternative to ASP.NET Zero, concerning the technologies for SaaS development based on ASP.NET Core. Moreover, as the Zero solution (see Fig. 1.1), it provides some already implemented features like multi-tenancy, login, registration, token-based authentication. Whereas functionalities like "forgot password", user registration confirmation by email or phone number, password reset, two factor authentication and many others are absent here. Therefore, if the software producers want also these functionalities being already implemented, they could adopt the Zero solution paying a subscription.

Let us now analyze the differences between the ASP.NET Zero and the other popular frameworks used for web development and the reasons why it has been adopted by the company, where this thesis work was carried out.

As it was previously explained, ASP.NET Zero is based on the ASP.NET Boilerplate framework. This framework has already all the structure needed to implement the multi-tenancy feature, which is fundamental for a SaaS web application. Moreover, the way ASP.NET Boilerplate provides the multi-tenancy feature is very simple, because some little configurations are needed when creating the entity classes (this will be explained in detail in Chap. 3). Instead, the other frameworks like Django, ExpressJS, ASP.NET Core need some initial configuration in order to build the structure for the multi-tenancy. Actually, it seems that ASP.NET Boilerplate is already enough to be used for SaaS web development, because it implements the requirements for multi-tenancy and, in addition, provides also built-in mechanism like authentication, authorizations and many others, which can speed up the initial effort required to build the base structure of the application. And one other positive fact is that this framework is open source and free. However, since the company wanted some other



Feature	Free Startup Templates	ASP.NET Zero
Base Infrastructure		
Abp framework features	✓	✓
Abp.Zero module integration	✓	✓
Multi Tenancy	✓	✓
User Interface (Account / Login)		
Login	✓	✓
Register	✓	✓
Token based authentication	✓	✓
Social logins	–	✓
LDAP (Active Directory) / ADFS login	–	✓
Forgot password	–	✓
Email address & phone number confirmation	–	✓
Password reset	–	✓
Two Factor authentication	–	✓
OpenId Connect login	–	✓
User lockout	–	✓
User profile / profile image / change password	–	✓
Account linking	–	✓
Show login attempts	–	✓
Tenant registration	–	✓
Identity Server 4 integration	–	✓
Password complexity settings	–	✓

Figure 1.1: ASP.NET Boilerplate vs ASP.NET Zero (Ref. [7]).

additional features, which are not already implemented by ASP.NET Boilerplate, it opted for beginning a subscription and use the ASP.NET Zero solution.

In conclusion of this section, let us analyse some other possible alternatives of the ASP.NET Zero solution such as SpreadsheetWEB and Servoy (but there are others available). Starting from SpreadsheetWEB, it is a *NoCode* development platform that helps client companies build custom web applications just by simply using Microsoft Excel spreadsheets. It allows users to add elements such as list boxes, drop-downs, check-boxes and sliders in the application by the means of a drag-and-drop interface. Unlike the ASP.NET Zero, SpreadsheetWEB does not support the development of mobile applications. While the ASP.NET Zero allows to build these latter using the Xamarin framework. Moreover, both of them support the user access control and activity dashboard functionalities (namely the management of a dashboard in which different kind of widgets can be positioned), but only ASP.NET Zero provides auditing, user authentication and automatic notifications. On the contrary, only SpreadsheetWEB supports the analytics functionality.

Servoy, instead, is an application development platform that provides client companies particular tools to build custom SaaS applications. For instance, the administrators can exploit the *WYSIWYG* editor to build custom user interfaces for their applications, adding components such as calendars, maps, Gantt charts and others. In computing, the WYSIWYG ("What You See Is What You Get") is a content editing tool. The WYSIWYG editors can be used to create and modify the design elements of a product (i.e. a website or a computer graphics model), without writing any code and allowing to see the final result. For instance, the WYSIWYG HTML editor makes available some graphical elements (i.e. a table, lists and others) that developers can choose and place inside the editor user interface. Then, the editor automatically shows to developers the resulting HTML page containing these elements.

Like ASP.NET Zero, Servoy allows also companies to implement some custom security configurations which comprise user creation and deletion, reset password functionality and many others. Unlike ASP.NET zero, it does not support mobile integration, role management, role-based permissions and other functionalities.

Having introduced the SaaS web applications and the available technologies for their implementation, let us now come back to the present work and discuss the goal of this thesis.

1.4 The goal of the present work

The purpose of the present work is the study, the design, the development and the deployment of a SaaS web application for the engineering and the automation of some management control phases of the company where the internship has been carried out. This company will be referred as the customer of the application from now on. The final aim of the application is to have a centralized system which allows the administrators to evaluate the financial health of the company and the collaborators, namely employees, to have a clear working plan.

The desired application is required to implement different functionalities like the management of orders, order types, revenues, tasks (i.e. comments), revenue objectives and behavioural reports. Since these functionalities are common to any company, the SaaS application will be shared by the means of subscriptions to more customer companies, exploiting in this way the multi-tenancy feature. According to the role and its assigned privileges, the user must be able to access to all or some of these functionalities. In this regard, the definition of two type of user roles is required in this application, the Operator and the Collaborator. The Operator corresponds to the administrator of the platform and is responsible to register the orders received from the customers. For each order, the administrator can define a revenue plan and assign the order to a team of collaborators, who is responsible to achieve its billing, namely all the imports of the revenue plan of that order are paid by the customers. Each collaborator is allowed to follow the orders assigned to him updating the advancement state of their revenues, until the payment of each of them is completed and marks them as terminated. In addition, chat-based conversations between team of collaborators and operators must be possible, where the revenue state of an order can be clearly communicated, so that the administrators can take further measurements in case the deadlines are not respected. Finally, a dashboard with multiple graphics, accessible only by the Operator, must be implemented. Pie-chart graphs are required here to show the company revenues grouped by different data (order type, assigned collaborator user), and a behavioural report to illustrate the company revenues per month and per year, comparing it to monthly revenues objectives of the current year and of a previous year. The revenue objectives and the comparison year can be set manually by the Operator. This dashboard will be used to generate business trends in a future evolution of the project, for instance, by performing machine learning on accumulated data.

From the technical point of view, the application has been built up making use of the ASP.NET Zero solution with the Angular template configuration. More precisely, the application is composed of a backend part and of a frontend part which have been created by exploiting respectively the *ASP.NET Core* and the *Angular* 10 frameworks. The former is an open source multi-platform framework used to build web native applications, whereas the latter is one of the most popular framework for client web development. Concerning the data management, the relational database of *Microsoft*

SQL Server is used in collaboration with *Entity Framework Core* (EFCore). The EFCore is an Object/Relational Mapping (ORM) framework used for entity-table mapping and database versioning through migrations. Finally, the *Microsoft Azure* platform has been adopted for the deployment of the application.

The actual design of the application that has been created is presented in the following chapter, where the data model, and the functional and non functional requirements are described in detail.

The actual implementation of the application, the software development methodology, the testing approach and the deployment are covered in Chap. 3, in Chap. 4, Chap. 5 and Chap. 6 respectively. Finally, the thesis ends with the conclusions and with an outlook of the future work.

Chapter 2

Design of the application

The application developed in the present work is one of the modules which constitute a large web application created to provide different types of services to the customers such as classrooms management system, courses management system and many others. Each module is designed according to a specific business logic, uses its own data and can also access further information which is shared with the other modules. The customer has the possibility to use one or more modules by starting a subscription for each module separately or, with the premium subscription, for a number of them. To ensure the coexistence of many customers using the same application at the same time, the multi-tenancy configuration has been enabled. Each customer therefore represents a specific tenant and has access only to his data and his configurations. The access to data belonging to other costumers is forbidden. Besides the customers tenants, there is also the host tenant which represents and can be used only by the owner of the software and provides configurations and functionalities common to all the other tenants. As an example, the host is allowed to set the UTC time zone for all tenants whereas the single tenant can choose the preferred one, i.e. Western Europe.

The module developed in this work implements the logic and the requirements at the basis of some management control phases common to any company. In this regard the work is quite general and it is not restricted to any specific company. Here the users of the application are interfaced with different types of entities such as *orders*, *order types*, *revenues*, *tasks*, *objectives revenues* and *behavioral reports*. Depending on the role and on the privileges of the specific user, some of the implemented requirements are forbidden. In the next section the types of data managed by the application which constitute the data model is described in detail.

2.1 The data model

It is always a good practice to start from the data model definition when designing a new application, since it represents its core part. A data model of a system, such as a web application, is an abstract scheme which represents the system and describes it using a certain type of formalism. In the present work the *Entity-Relationship* data model has been considered, which is composed of entities, their attributes and relationships. The entities are basically abstract objects which describe the real objects of the system, the attributes are the characteristics defining each entity and the relationships are used to associate different entities through the correspondence of one or more of their attributes. Since in this work a relational database has been adopted, the entities correspond to tables in the database, and their attributes to the columns of these tables. Instead, the relationships between the entities built on their attributes are represented by foreign keys built on the primary key columns of the tables. Let us now start presenting the key entities and their connections.

2.1.1 Entities

The entities used in the present work contain some specific data and in addition they store also some common information like the creation, the modification and the deletion times, the user creator identifier and the tenant identifier used for the multi-tenancy feature. Moreover, each entity is identified uniquely by an incremental integer number and a soft-deletion behavior is applied, namely the rows of the corresponding table in the database are marked as deleted but they are never effectively deleted from the database.

Let us now describe the entities used in the module which is developed in the present work, while their actual implementation in classes will instead be covered in sec. 3.3.3.

Companies. The companies represent the customers who can request none or several order services. Each company is uniquely identified by an incremental number, some specific information as name, vat number, the representative user, the phone number and the ateco code.

Orders. An order is a service offered to the customers. It is uniquely identified by an incremental number, by the customer company for which it is delivered, a value in euro, an offer (a PDF file with the contract between the customer company and the delivering company), an offer number (unique identifier for the order, used internally by the company), the registration date (namely the acceptance date) and the assigned priority. The latter represents the relevance of the order, as compared to an other order, and the possible values are: *Standard*, *High*, *Urgent*. Moreover, an order refers to a particular type (or activity) and is carried out by different users: one *referrer*, namely the person who brings the client in the company, and one or more *operatives*, which compose a team performing the order billing. The referrer could also be part of an operative team. Finally, an order could have a revenue plan (see below), by means of which its total value could be paid by the customers in one or more dates.

Order types. An order type is the kind of activity performed by an order service. It is uniquely identified by an incremental number, a unique name and an explanatory description. More order types with the same name in the system are forbidden.

Users. Users are the subjects who can access the application. A user is uniquely identified by an incremental number and by some personal information (name, surname, email, password and others). Depending on the role, some operations may be forbidden for the them. As an example, only some type of users can create orders. The type of user and their permissions will be discussed in detail in sec. 2.2.1.

Revenues. A revenue corresponds to the payment of an order, or part of it, at a certain date. It is characterized by a unique incremental number identifier, a value in euro, a submission date and the order identifier to which is associated. A revenue is uniquely identified from the others by the pair composed of the order identifier and the submission date. Therefore, there cannot exist two revenues of the same order in the same date. A revenue has an advancement state which describes its degree of completion and a timing, namely the indicator of the completion speed. The timing is calculated automatically by the system, performing a difference between the revenue termination date and the registration date of the order the revenue refers to. Furthermore the algorithm provides the following options:

- flash delivery (up to one week)
- standard delivery (up to one month)
- slow delivery (up to two months)

- ultra-slow delivery (more than two months).

Moreover, for a revenue there could be zero or more tasks associated.

Revenue tasks. They are used to keep a history of the operations achieved by the operative users on a particular revenue. As required by the customer, in the actual implementation of the application, tasks must be used like **comments** exchanged by the users in order to show the status of their works on a certain revenue, but in the future they will be probably transformed into tasks that can be assigned by the referrer to the operative team members. Moreover, a task is characterized by a unique incremental number identifier, a creation date, a descriptive text and the creator user. It is specific to a particular revenue.

Revenue states. They describe the phase of the current revenue. A revenue state is characterized by a unique incremental number identifier, a name, description and an ordering, used to keep an incremental order between the states. In order for the system to work properly, the existence of at least two states and the ordering constriction are expected. An example of revenue states is:

- to be started (at 0%)
- started (up to 30%)
- in progress (up to 50%)
- advanced (up to 100%)
- terminated (at 100%).

Revenues Objectives. These entities are in charge of collecting all revenues per year and per month of the year. They are characterized by a unique incremental identifier, a year, a month and a *IsObjective* field, which is used to differentiate internally the entities representing the objectives from the others. For instance, there could be two entities with the same year and the same month, but only one of them could be the objective. This entity is used to generate the revenues trend chart in the dashboard page, which will be described in detail in sec. 2.2.1.

2.1.2 Entity-Relationship scheme

All the entities described till now are represented in the Entity-Relationship scheme, shown in Fig. 2.1. As it can be seen, they are represented as tables in the database, whereas their attributes as columns of these tables (the not nullable columns are specified in bold). Instead, the relationships between the entities built on their attributes are represented by foreign keys built on the primary key columns of the tables. The list of the different type of relationships with their connecting attributes is the following:

- orders and revenues: one to many (on revenueId attribute). An order can have zero or more revenues associated, whereas a revenue is specific to only one order.
- orders and users:
  - many to many between orders and operative users, which is represented by a further entity connecting them (called *OrderUsers*). An order is managed by zero or more operatives and an operative can handle zero or more orders
  - one to many between referrer users and orders (on userId attribute). There is only one referrer for an order, whereas a referrer can be associated to zero or more orders

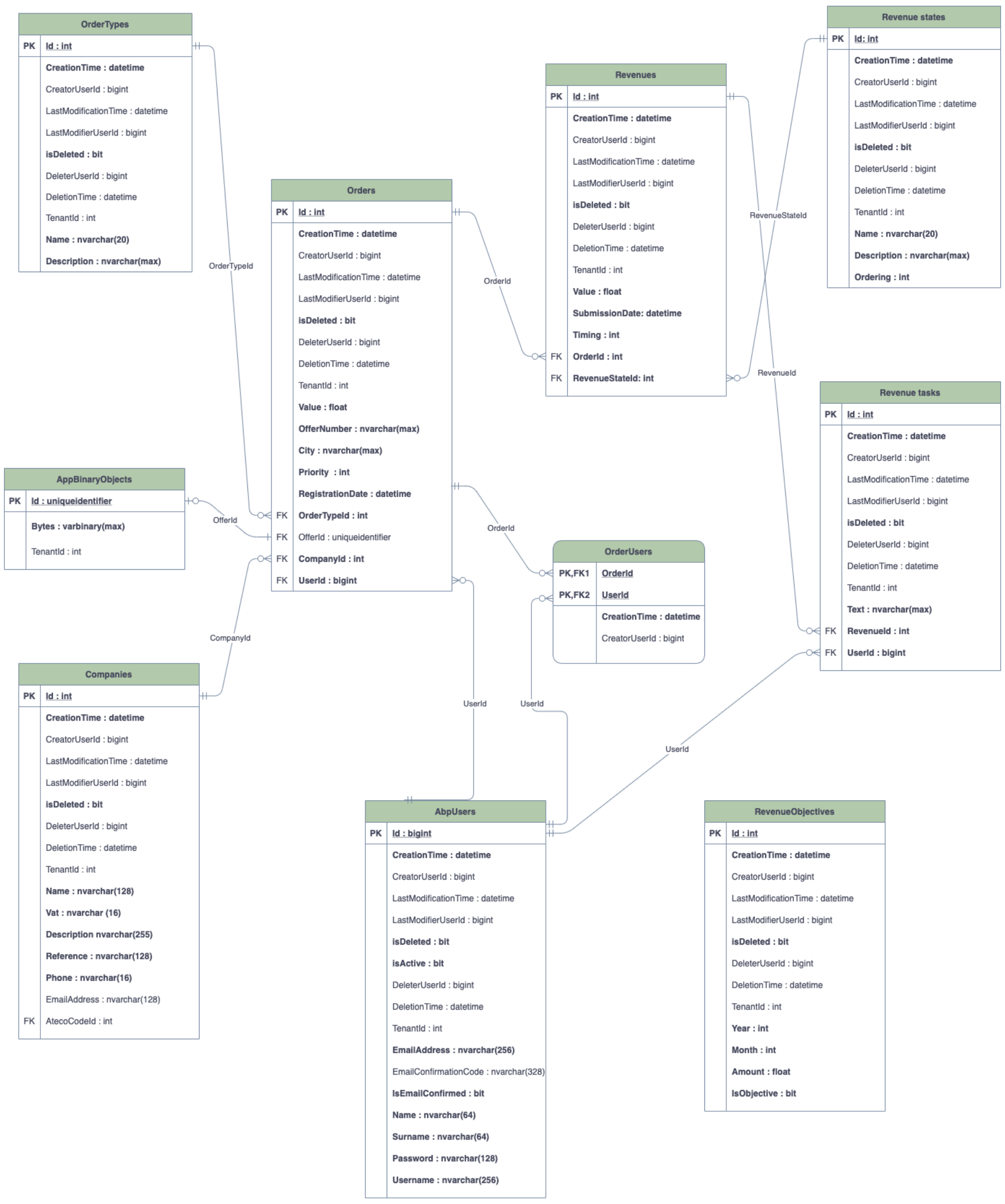


Figure 2.1: The Entity-Relationship scheme.

- order types and orders: one to many (on orderTypeId attribute). An order can have only one order type, whereas an order type can be assigned to zero or more orders
- revenue states and revenues: one to many (on revenueStateId attribute). A revenue can have only one state, while the same state can be associated to zero or more revenues
- revenues and revenue tasks: one to many (on revenueId attribute). A revenue can have zero or more revenue tasks, while a revenue task is specific to only one revenue

- users and revenue tasks: one to many (on revenueId attribute). A user can create zero or more revenue tasks, while a revenue task is specific to only one user creator
- companies and orders: one to many (on companyId). A company can request for zero or more order services, while a specific order is delivered for only one requesting company.

Moreover, ASP.NET Zero provides an already defined entity, called *AppBinaryObjects*, which can be used to store binary files. For instance, in this table the pdf files, containing the offers of orders, can be saved. In this regard, in the order entity the *OfferId* property is needed to store the identifier of the row of the AppBinaryObjects table, where the offer for an order is saved. Then, using a particular implemented service, it is possible to download the file from the database. The relationship between AppbinaryObjects and Orders table is a zero or one to one, meaning that for an order none or only one binary file can be stored, whereas a binary file is specific to only one order, when it is uploaded in the table.

Furthermore, in Fig. 2.1, it can also be seen that all these entities have some common attributes used for auditing. These are the creation, last modification and deletion times, the creator, last modifier and deleter user identifiers and the isDeleted field, used in order to allow the soft-deletion operations. All of these are provided automatically by the ASP.NET Zero solution just by following some simple procedures while defining the entity classes. This will be explained further in sec. 3.3.3.

2.1.3 System integration

In this section, it will be explained how the module developed in the thesis work is integrated with the rest of the larger application.

As introduced in the initial part of this Chapter, the application developed in this work constitutes an independent module of a lager web application, which is composed of multiple other modules, each implementing a specific functionality. The developed module can access in read-only mode to data shared with the other modules. This data is composed of information contained in Companies and Users tables. In addition, the module manages in read-write mode its own data, which instead can not be accessed by the other modules. This personal data comprises information stored in orders, order types, revenues, revenue tasks, revenue states and revenue objectives tables. In this way, the developed module is logically separated by the other modules of the system and does not affect their data. However, in order to fully work, it needs that Users and Companies information are always accessible, otherwise errors occur when using it.

2.2 Requirements for the application

The ASP.NET Zero solution provides already implemented functionalities like authentication, authorization and validation. Therefore, in the following sections only the functional and non-functional requirements that regard the new business logic to be implemented are considered. The already implemented requirements as for instance “authentication of a user when tries to log into the system” or “emails should be sent with a latency of less than 12 hours” will not be put in the following lists.

2.2.1 Functional requirements

The ASP.NET Zero solution provides the possibility to define different types of users and roles. Concerning the module developed here, two types of user roles are needed: the **Operator** and the **Collaborator**. The Operator corresponds to the administration of the platform and it has no limits and constraints. The Collaborator corresponds to all those users who work on the orders billing, for instance both the referrers and the operative team members have typically this role. Unlike the Operator, the role of the Collaborator has limitations. A list of allowed operations is associated to each role.

Starting from the operator role, the following lists of operations is allowed:

- create, edit, delete and view (all) orders
- create, edit, delete and view order types
- create, edit, delete and view revenue states
- create, edit, delete and view order revenues
- create and view revenue tasks
- view and edit the advancements of (all) orders
- view orders advancements reports
- view and edit dashboard statistics.

Whereas the collaborator is allowed to:

- view orders assigned to him (even if it is a referrer or a team member)
- view order types
- view revenue states
- view and edit the advancement of orders assigned to him (even if it is a referrer or a team member)
- create and view revenue tasks
- receive a notification when a new order has been commissioned to him
- receive a notification when being removed from an order commitment.

Finally all the tasks done exclusively by the system are:

- send notification to interested users after assignment to an order by the Operator
- send notification to interested users after remove from an order commitment
- send notification to interested users after a termination of a revenue
- calculate timing of a revenue after its termination
- recalculate reports when a new revenue has been terminated
- recalculate dashboard statistics when a new revenue has been terminated.

Let us now analyze in more detail each operation by using also figures taken from the final application in order to facilitate the discussion. The Fig. 2.2 shows the menu of items available in the module developed in this thesis work (named "Coge"), which are used by the users to navigate all the views.

Visualization of orders. By clicking the "Orders" menu item (Fig. 2.2), the user can visualize all orders present in the system which are displayed in a table format as shown in Fig. 2.3. If there are not any, the user visualizes a "no data" message. The

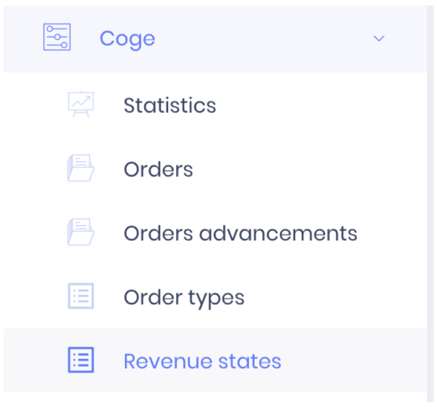
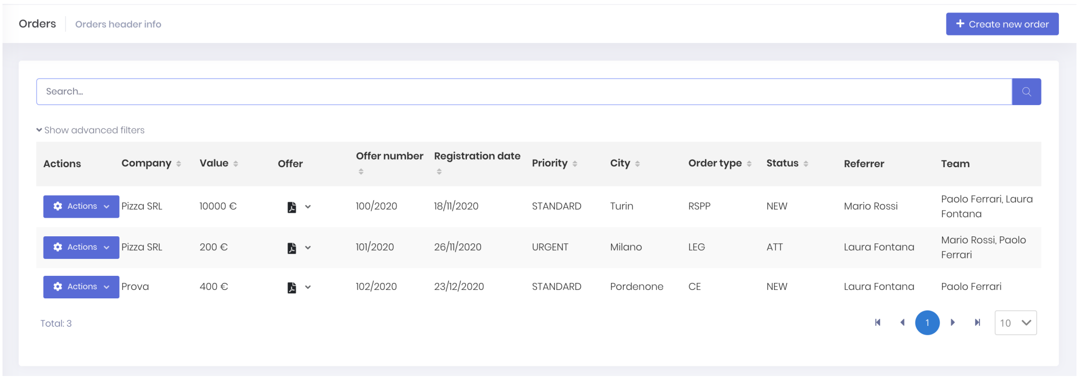


Figure 2.2: The menu items.

status column of an order refers to the activation status of the company for which that order service is delivered. This is computed automatically by the application and it can be *ATT*, namely the company has already been registered in the system in a previous year, or *NEW*, namely the company has been registered in the current year.


Orders | Orders header info | + Create new order

Search...

Show advanced filters

Actions	Company	Value	Offer	Offer number	Registration date	Priority	City	Order type	Status	Referrer	Team
Actions	Pizza SRL	10000 €		100/2020	18/11/2020	STANDARD	Turin	RSPP	NEW	Mario Rossi	Paolo Ferrari, Laura Fontana
Actions	Pizza SRL	200 €		101/2020	26/11/2020	URGENT	Milano	LEG	ATT	Laura Fontana	Mario Rossi, Paolo Ferrari
Actions	Prova	400 €		102/2020	23/12/2020	STANDARD	Pordenone	CE	NEW	Laura Fontana	Paolo Ferrari

Total: 3 | 1 | 10

Figure 2.3: The visualization of orders.

In addition, the table implements the row pagination and ordering for each column. On the top of the page, there are some fields that can be used to filter the rows of the table. As first, there is a generic search field that filters the rows by the inserted text looking for matches inside all the table columns. For instance, the user could search for a certain company name or for a team member's name. Furthermore, the user could use the more specific filtering fields, by clicking on "show advanced filters". This is shown Fig. 2.4. Each of them will search for the inserted text by looking only

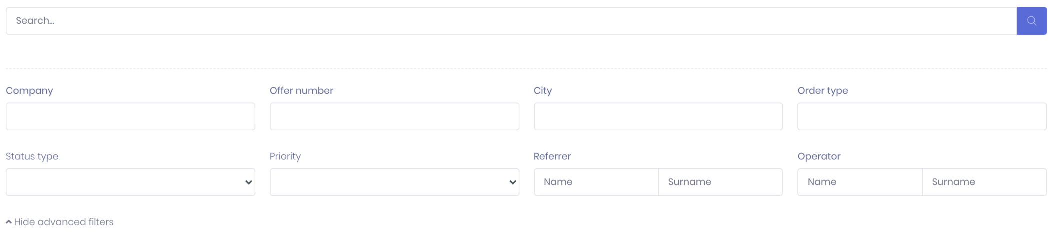


Figure 2.4: The advanced filters for orders.

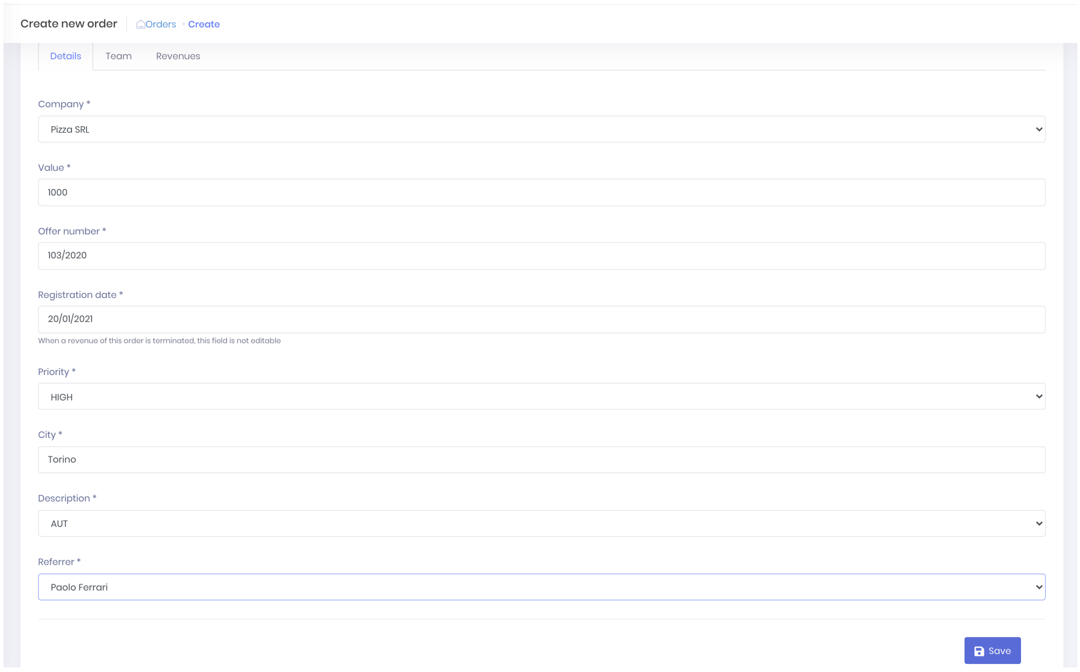
in the specific column it refers to. More precisely, as it can been deduced from Fig.

2.4, it is possible to filter the data by company name, offer number, city, order type name, status, priority, referrer and team member.

Moreover, directly in the table rows, it is possible to upload or download the offer of an order (if there is any), by clicking the drop-down menu in the "Offer" column. Both the front-end and the back-end validation allows the user to insert only PDF files.

By the means of "Actions" column (see Fig. 2.3), the user can perform operations on single rows, that are details visualization (a simple view with all the table information put together), deletion and edition of an order. Instead, by pressing "Create new order" button, the creation of a new order is activated.

Creation of an order. After the pressure of "Create new order" button, the user is redirected to the order creation page, in which a form must be filled with some general information. The form is composed of the company name, for which this


Create new order | Orders • Create

Details | Team | Revenues

Company *
Pizza SRL

Value *
1000

Offer number *
103/2020

Registration date *
20/01/2021
When a revenue of this order is terminated, this field is not editable

Priority *
HIGH

City *
Torino

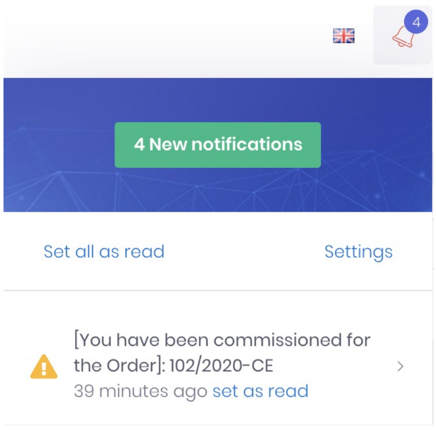
Description *
AUT

Referrer *
Paolo Ferrari

Figure 2.5: Order creation

order is delivered, a value, an offer number, a registration date, a priority, the city, a description (order type name) and a referrer. The implemented validation enables the pressing of the save button, only when all of these fields have been filled, then the new order is created and the user is redirected in the orders visualization page. Otherwise, an error is shown when trying to create an order using an already existing offer number. Besides, the user, corresponding to the referrer chosen before, receives a notification for the new order assignment, as shown in Fig. 2.6. The notification has been implemented adapting a particular service which is provided by the ASP.NET Zero solution. Furthermore, the informative text closed to the registration date field is used to warn the user that it is not possible to modify this field when there is at least one terminated revenue associated to this order. Therefore, since the termination of a revenue is irreversible and the timing is performed with the difference between this date and the revenue termination date, if the user is allowed to change this field the timing values of the already terminated revenues would be wrong.

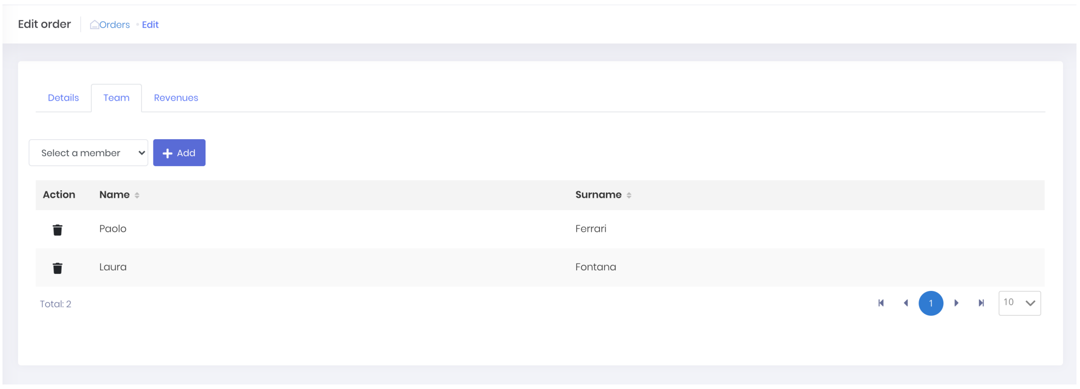
Edition of an order. By choosing the edit action in the orders table (see Fig. 2.3), the same page of order creation is open, with the exception that two more tabs



[You have been commissioned for the Order]: 102/2020-CE
39 minutes ago set as read

Figure 2.6: An example of user notification.

are now accessible, namely "Team" and "Revenues", as it can be seen in Fig. 2.7. In the "Details" tab, the user can change the order general information filled in the creation phase, as explained for the previous functional requirement. In addition, by clicking the "Team" tab, it is possible to build the working team, adding or removing new members who can be chosen from a list of options. This is shown in Fig. 2.7.



Action	Name	Surname
🗑	Paolo	Ferrari
🗑	Laura	Fontana

Total: 2

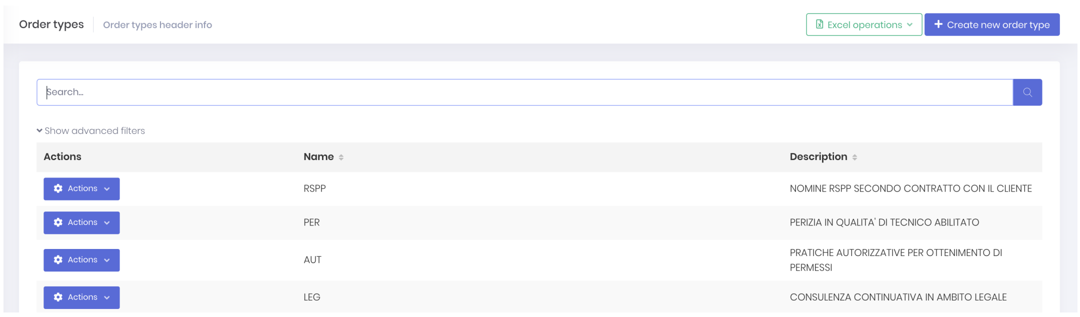
Figure 2.7: Team formation in an order.

In the same way as in the order creation, the assignment of a new referrer or a new team member sends a notification to the corresponding users, warning them about the new commitments. Also, the removal from an order commitment notifies with an appropriate message the relative users.

Finally, in the "Revenues" tab the user can create the revenue plan, so the CRUD operations on revenues are possible here. This will be explained below.

Deletion of an order. By choosing the delete action in the orders table (see Fig. 2.3), a confirm dialog opens and if the "Yes" button is chosen, the order is deleted and no longer viewed in the table. An warning message shows the performing of the operation.

Management of order types. By clicking the "Order types" menu item in Fig. 2.2, the user can **visualize** the table with all the order types (if any) as shown in Fig. 2.8. Similarly to what seen for orders, the table allows pagination, sorting, filtering, new row **creation**, **edition** and **deletion**. Furthermore, in order to simplify the initial migration to our platform, the order types could also be uploaded from an excel file. Lastly, there is a validation which prevents users from deleting a type used

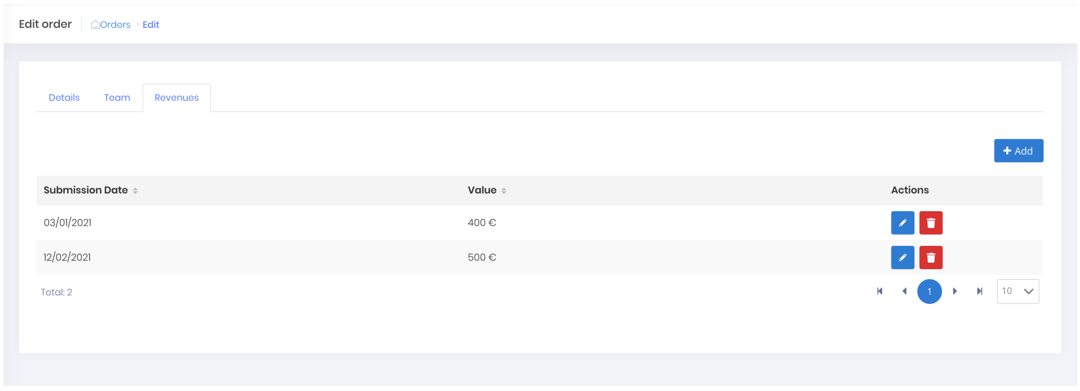


Actions	Name	Description
Actions	RSPP	NOMINE RSPP SECONDO CONTRATTO CON IL CLIENTE
Actions	PER	PERIZIA IN QUALITA' DI TECNICO ABILITATO
Actions	AUT	PRATICHE AUTORIZZATIVE PER OTTENIMENTO DI PERMESSI
Actions	LEG	CONSULENZA CONTINUATIVA IN AMBITO LEGALE

Figure 2.8: Order types.

by at least one order, and from inserting an already existing name in creation and edition. In both cases an error alert is displayed in the page.

Visualization of revenues. Through the "Revenues" tab in edition of an order, the users can visualize all the revenues (if any), represented by the means of a table as shown in Fig. 2.9. By default, the rows are kept sorted by submission date in ascending order, but it is also possible to sort them by each column. In addition, the table implements also the pagination.



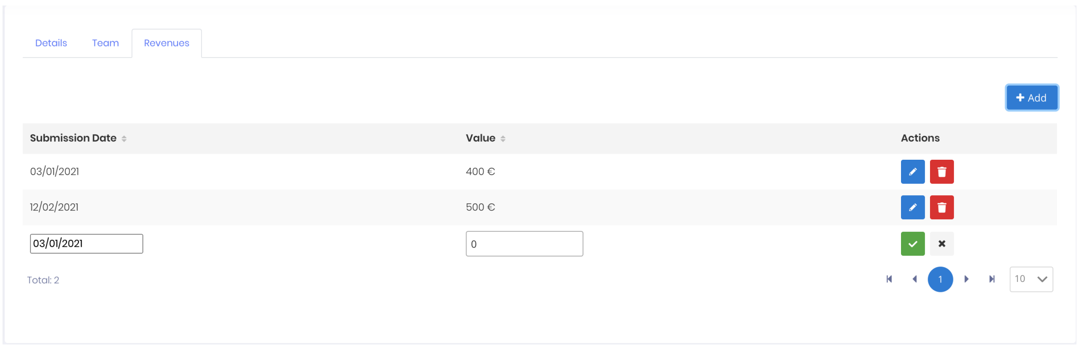
Submission Date	Value	Actions
03/01/2021	400 €	
12/02/2021	500 €	

Figure 2.9: Order revenues.

Creation of a revenue. In the revenues tab of an order shown in Fig. 2.9, by pressing the "Add" button, a new row is added in the table as shown in Fig. 2.10. This row is composed of editable fields in which the user has to insert the revenue data (the submission date and a value, the current date and zero by default respectively) and then he can accept or cancel the operation by clicking one of the two corresponding buttons. When, trying to insert a new revenue with the same submission date of an already existing one, an error message is displayed showing the violation of the constraint and the new row is removed from the table. When creating new revenue items, the application automatically assign them the initial revenue state, which is calculated through an internal logic.

Edition of a revenue. In the revenues tab of an order in Fig. 2.9, by pressing the edit button on one of the row (the first one in the example), the revenue edition is enabled and it is possible to change its data, as shown in Fig. 2.11. Then, the user can accept or reset the changes by clicking the corresponding button. Similarly as for creation, it is not possible to choose an already existing submission date, when changing the submission date of a revenue.

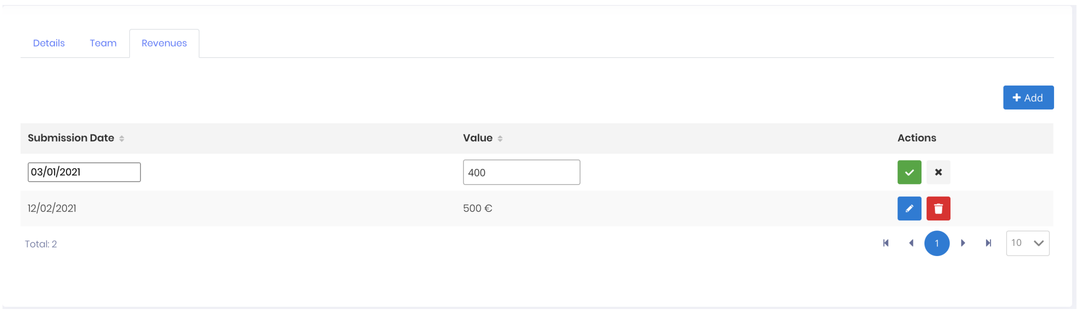
Deletion of a revenue. In the revenues tab of an order (Fig. 2.9), a row can be removed clicking the specific delete button in the "Actions" column. As seen in the



Submission Date	Value	Actions
03/01/2021	400 €	
12/02/2021	500 €	
03/01/2021	0	

Total: 2

Figure 2.10: Revenue creation.



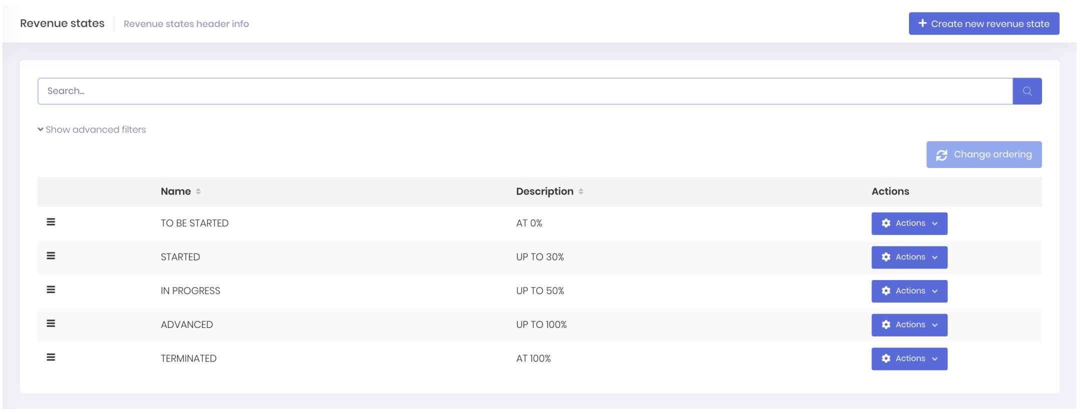
Submission Date	Value	Actions
03/01/2021	400	
12/02/2021	500 €	

Total: 2

Figure 2.11: Revenue edition.

orders table, before removing the row, a dialog is opened asking for confirmation or rejection.

Management of revenue states. By clicking the "Revenue states" menu item, the user can **visualize** the table with all the revenue states (if any) as shown in Fig. 2.12. Similarly to what seen for orders and order types, the table allows sorting,



	Name	Description	Actions
≡	TO BE STARTED	AT 0%	Actions
≡	STARTED	UP TO 30%	Actions
≡	IN PROGRESS	UP TO 50%	Actions
≡	ADVANCED	UP TO 100%	Actions
≡	TERMINATED	AT 100%	Actions

Figure 2.12: Revenue states

filtering, state **creation**, **edition** and **deletion**. Moreover, the application does not allow users to choose already existing name in edition and creation, and to delete a state which is already used by at least one revenue. In both cases, an error message is displayed explaining why the operation can not be done. Besides, since the states are used to show the completion degree of a certain revenue, they must keep an internal ordering. Therefore, it is possible to sort the rows with a drag and drop mechanism. Then, after the user has chosen the ordering, she confirms it by the "Change ordering" button, which is activated only when some reordering is done. The management of revenue states requirement is extremely important since it blocks the other requirements, such as the revenues creation, dashboard statistics visualization, orders advancements visualization and edition. The user is allowed to perform these operations only if she has firstly set at least two revenue states (one initial and one final), since it would not be possible to calculate correctly whether they are initial or final. Indeed, as previously explained, to create a revenue an initial state is needed, whereas a last state is needed for the other requirements (i.e. calculate timing of an order revenue). If there is only one revenue state, it is not possible to distinguish whether it is the initial state or the last one (it could be both).

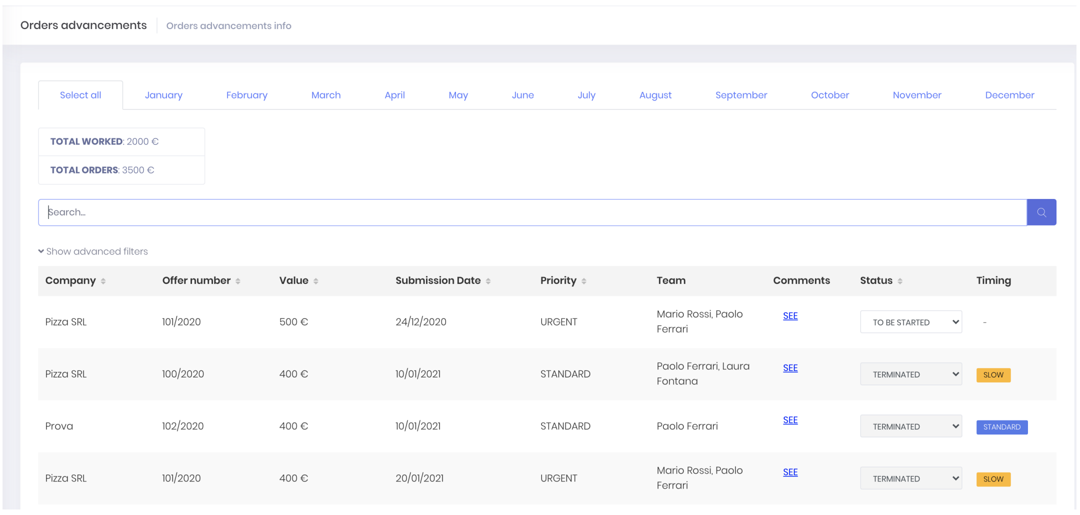
Moreover, after the revenue states are setup and confirmed, if order revenues are created by the user, it is not possible anymore to create new states or to change their ordering. In all of these cases, an error message is shown explaining to the user these constraints.

Management of orders advancements. This functionality consists of both visualization and edition of orders advancements. In the orders advancements page (Fig. 2.13), the user can visualize the list of order revenues with their states, filtered by a selected month or by all months (the whole year). It can be said that this information is an aggregation of data coming from orders, revenues, revenue states and revenue tasks tables. The resulting table, as the preceding ones, allows pagination and sorting by each column. While, by default the rows are sorted by the submission date and then by the priority in ascending order, meaning that if there are two rows with same submission date, the one with the higher priority is viewed as first. In addition, in order to filter per month, a list of tabs has been placed on top of the page. Moreover, like in the orders page, also in this one a filter area, composed of a generic search bar and some specialized input fields, has been implemented.

The user can also set the state of an order revenue by choosing between a list of options. The list of options can be created by the user as seen in the revenue states management page, and, for instance, it could be composed of the following elements (with ascending ordering): "to be started", "started", "advanced" and "terminated". When she chooses the "terminated" one, a confirm dialog is showed because after this operation the state can not be changed anymore. In case of confirmation, the timing is calculated and the advancements reports are being updated. The termination of a revenue means that the user can not modify the registration date of the referring order anymore (as explained previously in the order creation page), since the timing is calculated using this information (as explained in the data model).

Visualization of orders advancement reports. On the top of the "orders advancements" page one can see the reports data (shown in Fig. 2.14), which is composed of the **Total orders**, meaning the total value of registered (accepted) order revenues, and the **Total worked**, that is the total value of the terminated order revenues. These values are referred to all revenues for the selected month or for all months (if "select all" tab is chosen) and they are recalculated by the application every time a new revenue is registered or terminated by the user.

Management of revenue tasks (comments). This functionality is composed both of creation and visualization. By clicking the "SEE" button in the "Comments" column in the orders advancements page (Fig. 2.13), a popup dialog is opened showing the historical of comments exchanged by the working users on a certain order revenue. Here, they are represented with a user interface similar to a chat conversation as shown in Fig. 2.15. Indeed, the current user's comments and avatar are placed on the right part of the dialog, while the other ones on the left. It is also possible to create a

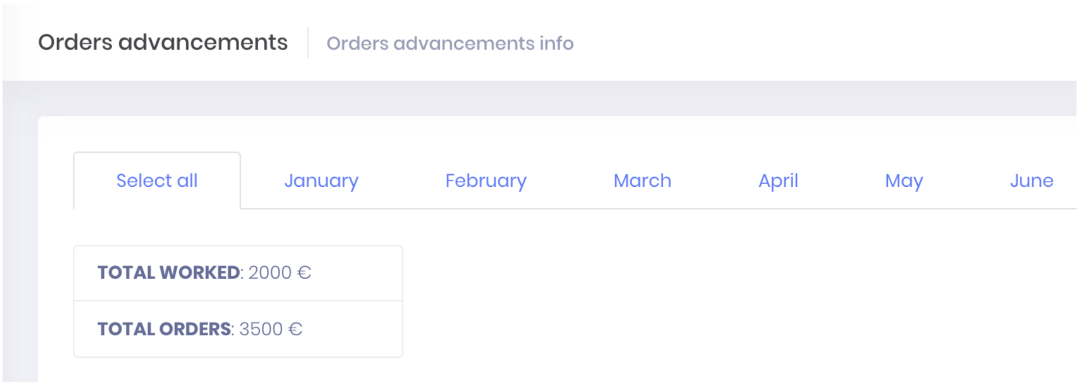


TOTAL WORKED: 2000 €

TOTAL ORDERS: 3500 €

Company	Offer number	Value	Submission Date	Priority	Team	Comments	Status	Timing
Pizza SRL	101/2020	500 €	24/12/2020	URGENT	Mario Rossi, Paolo Ferrari	SEE	TO BE STARTED	-
Pizza SRL	100/2020	400 €	10/01/2021	STANDARD	Paolo Ferrari, Laura Fontana	SEE	TERMINATED	SLOW
Prova	102/2020	400 €	10/01/2021	STANDARD	Paolo Ferrari	SEE	TERMINATED	STANDARD
Pizza SRL	101/2020	400 €	20/01/2021	URGENT	Mario Rossi, Paolo Ferrari	SEE	TERMINATED	SLOW

Figure 2.13: Orders advancement.



TOTAL WORKED: 2000 €

TOTAL ORDERS: 3500 €

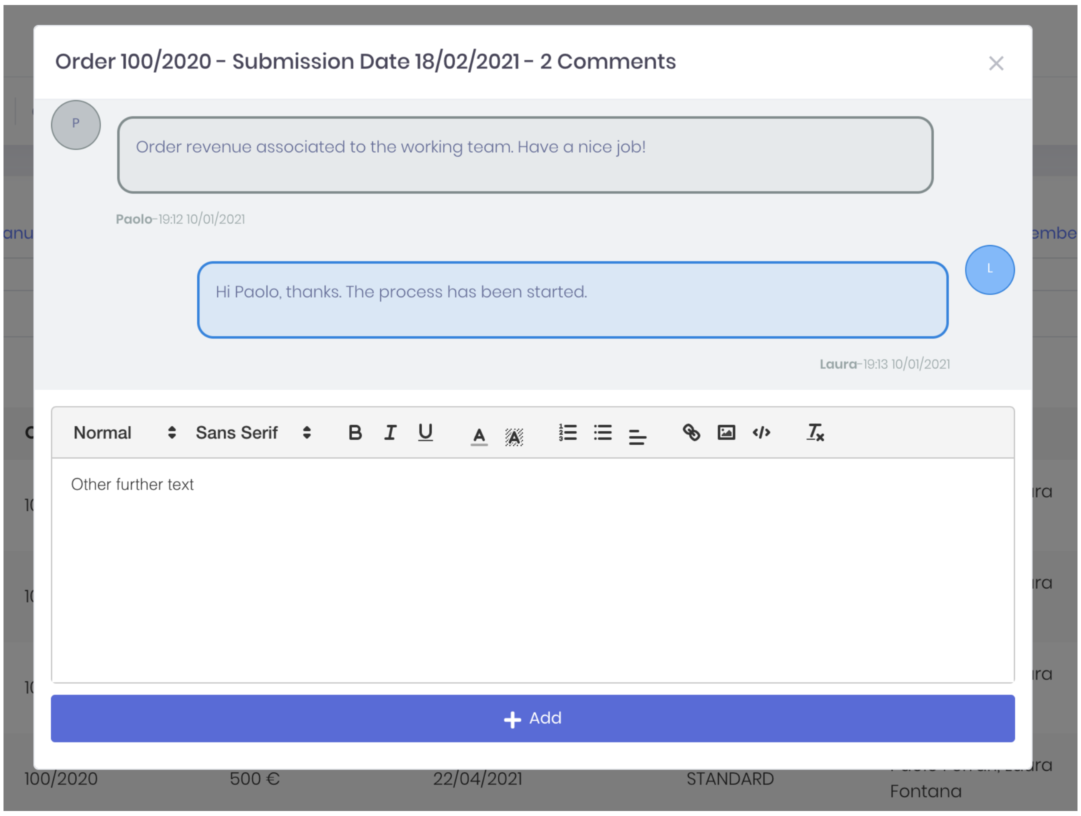
Figure 2.14: Orders advancements reports.

new comment using the text area below and, since it is not possible to edit or delete a comment, the dialog asks the user for a confirmation. The comments are made unchangeable, because they are used to track the activities done on a order revenue, and therefore in this way the working team members can not cheat on the actual state of their work, changing the text.

Dashboard management. In the "statistics" page, the user can visualize a list of charts, which describe the company income grouped by different entities such as

- the percentage of revenue values by each referrer, shown in Fig. 2.17
- the percentage of revenue values by each order type, shown in Fig. 2.18.

Moreover, the dashboard contains the revenues trend chart, in which different values are compared: the *income objective* for each month in the current year, *the actual income* for each month in the current year, and the incomes for each month in an optional past year (which can be chosen by the user). In addition, near to the chart there is a button that can be used to set the income objectives for each month in the current year.


Order 100/2020 - Submission Date 18/02/2021 - 2 Comments

Order revenue associated to the working team. Have a nice job!

Paolo-19:12 10/01/2021

Hi Paolo, thanks. The process has been started.

Laura-19:13 10/01/2021

Other further text

Figure 2.15: Revenue comments.

2.2.2 Non-Functional Requirements

We now turn on the the non-functional requirements implemented in the developed module, without considering those already implemented by the ASP.NET Zero solution.

Concerning the **Usability**, in order to guide the users in the navigation and usage of the application, the following general non-functional requirements have been considered:

- the application must adopt a user interface which is clear and easy to navigate, according to the most recent standards
- the application must show clearly the processing of the actions performed by the users
- the application must show specific validation errors, by opening pop-up alert windows (for instance alert dialogs)
- the application must show the performing of the user actions, by opening pop-up windows (for instance snack bars).

Concerning the first requirement, ASP.NET Zero provides the possibility to use, for the user interface, the *Metronic* theme, which guides the users in the navigation and is quite intuitive. The developers could use for their views the CSS style properties of this theme, which is imported by default in the ASP.NET Zero solution.

Concerning the second requirement, the users know always what is happening when using the application, since in the present work loading indicators are shown when they access pages where there are tables containing data that is retrieved from

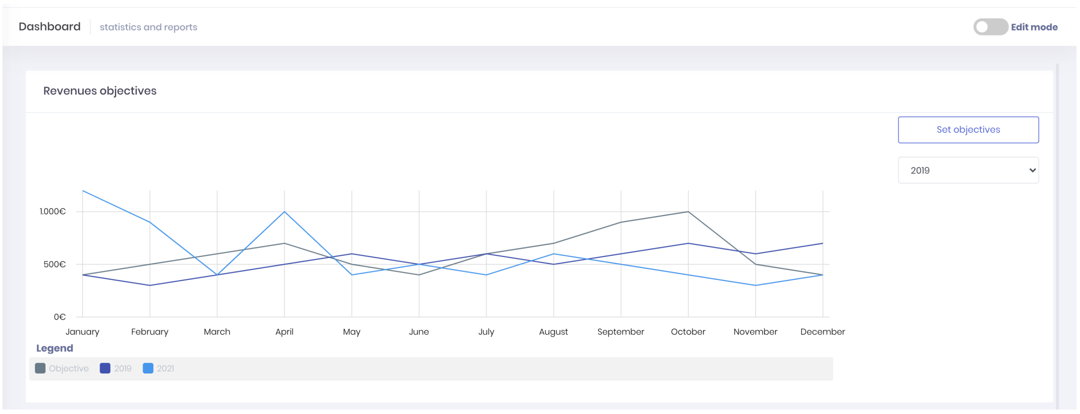


Figure 2.16: Revenues objectives.

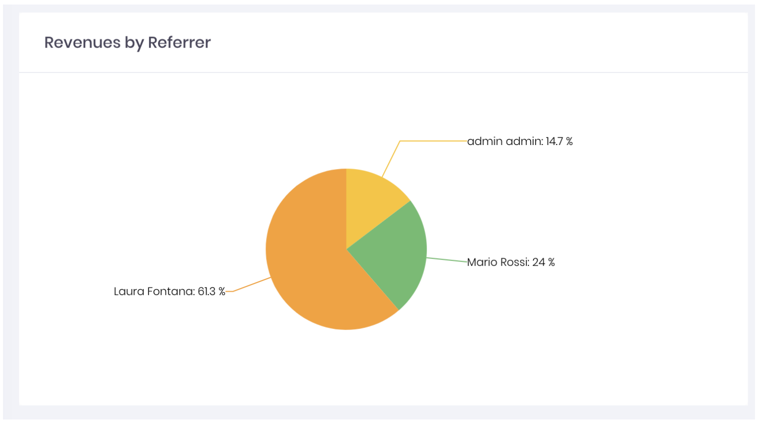


Figure 2.17: Revenue by referrer.

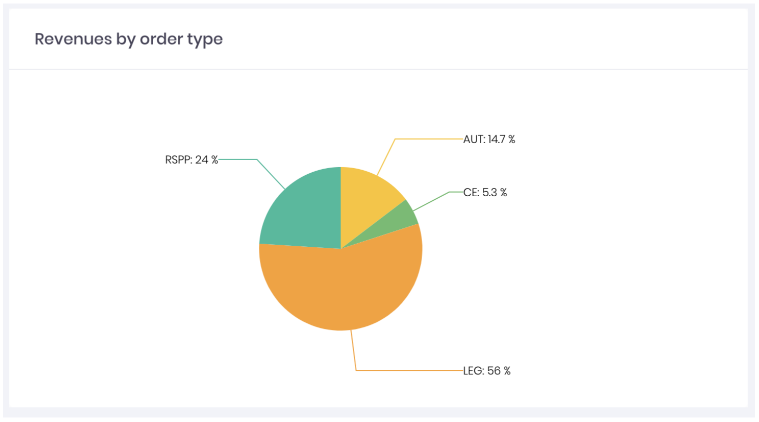


Figure 2.18: Revenue by order type.

the backend. Therefore, the loading indicator is shown until the data becomes available or an error occurs. Furthermore, when submitting a form (for instance in create or edit order page, Fig. 2.5), the save button text is changed with "Saving" and a loading spinner icon, instead of the default "Save" text.

Concerning the validation requirement, when the user tries to send to the server requests which are not valid, an alert dialog explaining the reasons shows up. For instance, this happens while trying to insert a revenue with the same submission date of an existing one for the same order, or when trying to violate the uniqueness of the name and the offer number when creating a new order type and a new order respectively.

Finally, the effects of all the operations performed by the users are clearly visible. In this way, the user is able to see whether the application is responding to his requests or not. For instance, after addition, edition and deletion of table rows, a successful snackbar message is showed, or otherwise an error dialog is displayed, explaining the causes. As an example, after the deletion of an order the user sees the message "Order deleted successfully".

Regarding the **Localization** non-functional requirement, the application must support different languages, in particular Italian and American English. For this purpose, the localization strings for these languages have been configured. Therefore, the user can translate the whole application choosing between one of them.

Also the requirements concerning the **Scalability** have been considered. The Azure platform, where the application is hosted, gives the possibility to extend the resources (i.e storage memory) used by the application when these are running out. It is quite easy to do this, because only few steps of configuration using the Azure dashboard are required.

Finally, regarding the non-functional requirements of **Processes**, we have used Agile techniques (Scrum version) for the project management. Each of the sprints lasted two weeks and was concluded with a release of the application in a pre-production environment, where it was tested by the customer, who gave feed-backs for the next sprint.

In the next chapter, the actual structure of the application and the main modules by which it is composed will be covered.

Chapter 3

Structure of the application

As anticipated in the previous chapter, the application realized here uses the ASP.NET Zero solution. This solution is composed of the ASP.NET core solution and of the Angular solution. They have been used for the backend part and for the frontend part of the application respectively. More precisely they allow to build an Angular application which retrieves data from an ASP.NET Core backend application and allows the user interactions.

These projects are quite complex and intricate and need a thorough description. Since the ASP.NET Core application follows the principles of Domain Driven Design, we start from these principles in Sec. 3.1, which is the starting point of this chapter. Afterwards, the overall structure of the application is described in Sec. 3.2. Then Sec. 3.3 and Sec. 3.4 are dedicated to the detailed description of the ASP.NET Core and Angular solutions. Finally the chapter ends with an example of a full stack implementation of a feature in the application.

3.1 Layering of the application

Since the ASP.NET Zero solution is based on ASP.NET boilerplate, it follows the principles of the **Domain Driven Design** (DDD) for the architecture of a software.

According to the DDD approach, the development of the application must start from the definition of the application domain model. A domain model is a representation of the application data, independent of the way data is stored in the database. Therefore, in order to specify how the data must be stored in a database, an other model is needed. A possible candidate is the data model, which is commonly used for relational databases. Then, since in this work a relational database is used, the data model must be defined from the domain model, namely the entities (corresponding to tables) and their relationships (corresponding to foreign keys between tables).

Furthermore, The DDD approach allows to develop applications conform with **SOLID** principles. These are five fundamental principles used in Object Oriented Programming (OOP), introduced in a publication, titled "Design Principles and Design Patterns" and written by the famous American software engineer Robert Martin in 2000. The SOLID principles aim to develop software that is clearer to understand, easier to keep and to extend, and they are:

- Single responsibility (S)
- Open/closed (O)
- Liskov substitution (L)
- Interface segregation (I)
- Dependency inversion (D)

The Single responsibility principle means that each module or class of an application should be responsible for only one functionality provided by the application.

The open/closed principle refers to the fact that existing and already working software entities, such as classes, modules and methods, should be open to extension and closed to modifications. In other words, modifying an already implemented code can be done only if there are internal bugs that need to be solved. In all the other cases, it should be extended. A typical example of following this principle is the creation of new classes which extend, through the inheritance pattern, an already working class and add new functionalities, without modifying the base class.

Liskov substitution principle claims that class objects of an application should be replaceable only with objects of their child classes. In this way, there are no risk of spoiling the correct behaviour of the application.

According to interface segregation principle, larger interfaces should be divided into smaller ones. In this way, when creating a new class we can concentrate only on creating the methods that are needed for that class, implementing only the interfaces where those methods are defined. With larger interfaces there is the risk of implementing also unnecessary methods.

The dependency inversion principle provides the dependency injection pattern as a way of solving the common problem of OOP related to classes strictly dependent on each other. The dependencies are substituted with abstractions, avoiding in this way compilation errors when strictly coupled classes are instantiated. The dependency injection pattern is explained in detail in Sec. 3.3.10.

In order to implement the DDD principles, ASP.NET Boilerplate proposes a layering of the application, the layer corresponding to the data model being separated from the rest. In particular it identifies the following layers:

- Presentation Layer / Distributed service Layer,
- Application Layer,
- Domain Layer,
- Infrastructure Layer,

shown also in Fig. 3.1.

In the present work, the user through a user agent (i.e. a common browser) access to a client application (the angular application). The client application interacts with the first layer of the backend application, the Distributed Service Layer, by the means of API calls (HTTP requests) and receives HTTP responses transporting JSON content. In general, the client applications tasks consist in retrieving data from the backend server and showing it to the users. However, there are client applications, like angular application, which provide also mechanisms to allow the user interactions.

The Distributed Service Layer is used to serve the backend functionalities by the means of remote APIs. This layer does not contain any business logic but it is composed of elements, such as the *API Controllers*, used to transform the HTTP requests coming from the angular application into operations on the application domain. Namely, the controllers call application services methods passing data, extrapolated from the HTTP requests and stored in DTOs, and return to the angular application HTTP responses, containing the result of operations processed by the service methods. Moreover, as it will be explained in the Web.Core project, described in Sec. 3.3.7, in most of the cases, the Distributed Service Layer is bypassed, meaning that the application services are exposed directly to the client applications, and only in some rare occasions the manual definition of API controllers is needed.

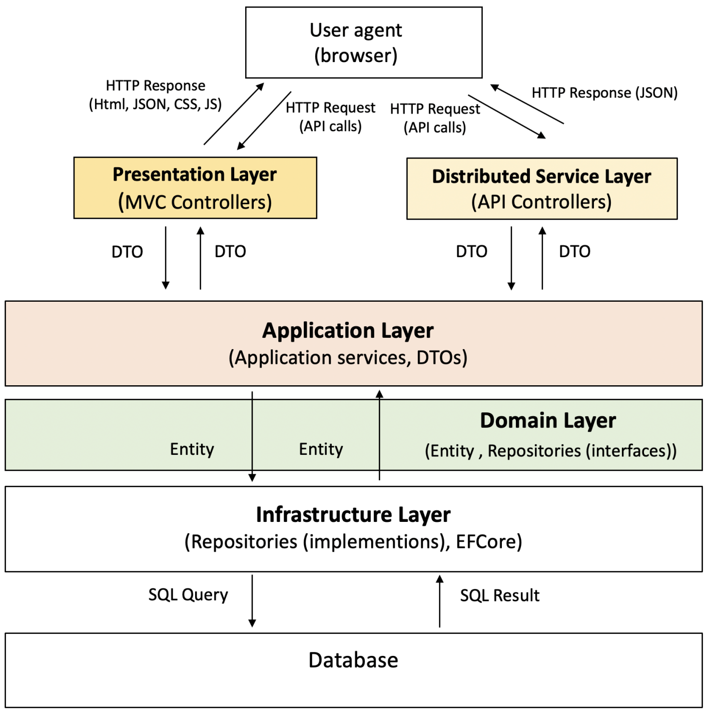

User agent (browser)
HTTP Response (Html, JSON, CSS, JS)
HTTP Request (API calls)
HTTP Request (API calls)
HTTP Response (JSON)
Presentation Layer (MVC Controllers)
Distributed Service Layer (API Controllers)
DTO
DTO
DTO
DTO
Application Layer (Application services, DTOs)
Entity
Entity
Domain Layer (Entity , Repositories (interfaces))
Infrastructure Layer (Repositories (implementions), EFCore)
SQL Query
SQL Result
Database

Figure 3.1: ASP.NET Boilerplate Application Architecture Model.

Alternatively, also MVC controllers can be defined, though in this work there was no need for them. In this case a new layer is defined, called Presentation Layer, which unlike the Distributed service layer, provides to users also an interface that is composed of web pages (namely, html files with css and javascript code). As the previous layer, the same operations are performed, with the difference that MVC controllers send back to the client applications HTTP responses, carrying also web pages. It is in these latter that user interactions take place. Furthermore, MVC controllers are used by ASP.NET Zero solution itself to expose the application services implementing some of the built-in features discussed in the previous chapters.

The Application Layer defines the contract between the user and the application, namely the business logic. The business logic is composed of the set of methods corresponding to the logic actions that a user can perform in the application. These methods are the so called *Application Services* that use the domain layer objects (for instance, entity objects and/or repository interface objects) to perform requested application functionalities. More precisely, the presentation layer or distributed service layer calls application services methods passing them Data Transfer Objects (DTOs). The application service methods map these objects in entity objects and pass them to repository methods in order to perform database queries. The repository methods return back to application service methods the results of the queries, which are mapped by the service methods into DTO objects and returned to the presentation layer or the distributed service layer. Thus, the application layer is a real mediator between the presentation or distributed services and domain layers, isolating them from each other.

The Domain Layer is the core of the application. It represents the data model to which the business logic of the application must be applied. The data model is composed of the entities and their relationships which abstract the data handled by the application. This layer includes entity classes used to perform business logic and defines repository interfaces to read and save entities from the data source (which is typically a database).

The Infrastructure Layer makes the other layers work. For instance, it implements the repository interfaces (for example, using the *Entity Framework Core* ORM) which transfer the entity objects from the database to the domain layer and vice-versa. The repository classes offer methods to find, create, update and delete entity objects by the means of their unique identifiers. Moreover, they offer also methods which can be used to find all the entities connected through relationships to a given entity. In addition, this layer may also include an integration to external third-part libraries used to send emails or to implement other helpful functionalities.

This approach of layering of an application code-base is a widespread technique which helps to reduce its complexity and to improve its re-usability. In the ASP.NET Zero solution the building blocks for Domain layer used in this work are the Entities, the Repositories and the Unit of Works. Instead, the building blocks for the Application layer are the Application Services and the DTOs. Finally, the Infrastructure layer is built using EFCore and the Distributed Service layer is composed of API controllers.

The features of the DDD building blocks elements and their location (as suggested by ASP.NET Zero) inside the Zero solution are covered in detail in the following, after explaining first the overall structure of the Zero solution, which is the topic of the next section.

3.2 Overall structure of the project

As shown in the diagram of Fig 3.2, The ASP.NET Zero solution is composed of the **Angular** and the **ASP.NET Core** solutions.

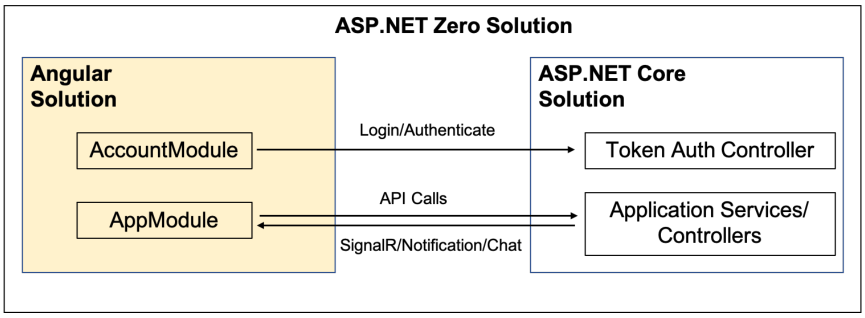


Figure 3.2: The overall project structure (taken from [5]).

The Angular solution corresponds to a client application used to fetch data from the backend application and to allow the user interaction. It corresponds to the frontend part of the whole SaaS web application and it is composed of two main modules: the AccountModule and the AppModule. The AccountModule manages the user authentication, while the AppModule contains the application logic of the frontend part of the application.

On the other hand, the ASP.NET Core solution corresponds to the backend part of the application. It builds an application composed of several components, like the Token Authentication Controller, Application Services and Controllers, which are used to expose REST API end-points that enable users to use the application features, namely to perform its business logic.

In addition, some interactions happen between the components of these two solutions. The AccoutModule dialogues with TokenAuthController, in order to implement the user authentication functionality, whereas, the AppModule calls APIs exposed by the Application Services and Controllers, in order to allow user interactions and visualization of data fetched from backend. Furthermore, the application services can send notifications or chat messages to the frontend part.

The angular solution can be deployed independently from the ASP.NET Core solution. Indeed, they can be served inside the same web server but to different port numbers or inside different web servers and to any port number. When the angular solution is deployed, it is basically an application consisting of a code written in html, javascript and css, therefore it can be served on any web server running on any operating system. By contrary, the ASP.NET Core solution is not composed of html, javascript and css code, but provides API end-points, through which the token-based authentication is made possible and the application service methods are made accessible to the client applications.

In the following section, the backend solution will be explored in detail, keeping more focus on the most important features and functions it offers.

3.3 Backend: ASP.NET Core solution

Depending on the type of the application that someone wants to build, the ASP.NET Zero solution provides different kind of base solutions, such as Web and Mobile. In the present work the **Web** solution is used, since the goal is to realize a SaaS Web application. As shown in Fig. 3.3, the Web solution is composed of different sub-projects, each with its specific functionality and technology. According to DDD principles, Application and Application.Shared compose the Application layer, Core and Core.Shared the Domain layer, EntityFrameworkCore and Migrator the Infrastructure layer and finally Web.Core and Web.Host the Distributed service layer. The presentation layer is built by Web.Public project, which is a separated web application that can be used to create a public web site or a home page for the web application. In the present work no files have been added in this project. Only its login page has been used to access the backend application, as it will be explained in Sec. 3.3.6. Finally, the Tests project contain all the components used to test the application code.

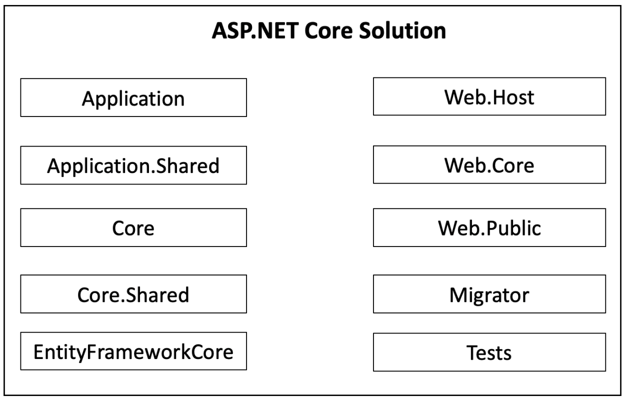


Figure 3.3: The ASP.NET Core solution.

Furthermore, the ASP.NET Boilerplate framework provides the support to build and combine modules together in order to create an application. Different modules can depend on each others. Technically, a module is defined with a class that inherits the *AbpModule* base class, which is contained in the *Abp* package. A module can contain entities, application services, logic for database integration and user-interface elements. Generally, it is a good practice to develop re-usable modules following the principles of DDD pattern and its division in layers. Indeed, in the ASP.NET Zero solution, a module for each sub-project of the ASP.NET Core solution has been already created.

Let us now describe in the next sections all the sub-projects of the ASP.NET Core solution, describing the DDD elements implemented by each of them. The organization of these latter in different sub-projects is not mandatory, but only recommended by the solution, since it helps to make clearer and easier the working environment.

3.3.1 Application

This project contains all the business logic of the application, since the *Application Services* classes are placed here. The latter are simple C# classes used to expose the backend APIs to the angular application. Even though it is not strictly required, in the ASP.NET Zero solution, an application service class should implement an interface, which extends the *IApplicationService* interface, made available by the Zero solution. The interface contains the definition of methods that represent the business logic. It must be as small as possible in order to be consistent with the interface segregation principle of SOLID.

Moreover, as it will be explained further on, this approach of creating application service classes simplifies the management of the dependency injection pattern.

3.3.2 Application.Shared

In this project there are the *interfaces* implemented by the application services classes, whose implementations are contained in the previous project, and the definition of *DTO* classes.

In ASP.NET Boilerplate, a DTO is a simple C# class that has no dependency with other classes and can be designed in any way. Typically, the DTOs are created in order to match entity classes. This approach is used to facilitate the translation of DTOs in entities and vice-versa, using the *AutoMapping* functionality provided by the Zero solution. As previously explained, these translations are needed to allow the communication between application and infrastructure layers.

Furthermore, the DTO classes can extend, through the inheritance pattern, some base classes provided by ASP.NET Boilerplate, from which receive useful additional properties. This approach simplifies the writing of code for the definition of DTO classes, since it allows to not repeat manually the declarations of some properties.

In the present work, the following base classes have been largely used:

- EntityDto<TypeKey>: provides to the child classes a property named "Id" with type corresponding to TypeKey (i.e int, double, float). This could be useful when creating DTOs matching entity classes, avoiding from repeating the manual definition of the Id property in each of them
- PagedAndSortedResultRequestDto: provides the properties MaxResultCount (int), SkipCount (int) and Sorting (string, i.e. "NAME ASC"), which are used to specify additional options for querying the backend data. They are used to request a limited result, paged result and a sorting result to the database respectively. For instance, this DTO is convenient in those pages of the user interface containing tables with pagination and sorting functionalities. For instance, in the present work, this base class is inherited by the DTOs which are passed to the application service method used to retrieve the orders data paged and filtered

- PagedAndSortedDto<T>: provides MaxResult and Items properties, which are very helpful while populating tables with pagination.

3.3.3 Core

This project contains domain layer classes, such as entities. Essentially, entities have ids and are stored in a database. In a relational database, the entities are mapped to tables.

In ASP.NET Boilerplate, an Entity is a C# class which contains a property representing the primary key of the mapping table in the database. An entity class can extend the *Entity* base class, from which it automatically gets the Id property, without defining it manually. By default, the Id type is an incremental int32, but it can be changed by extending the class *Entity<type>* (where "type" can be a long, a string, a guid and many others). By default, the Id property has "Id" as name. This can be changed declaring a new field with the desired name and the [Key] annotation placed before it. All the approaches used to realize the mapping between entities and tables are explained further on in the EntityFrameworkCore project, described in Sec. 3.3.5.

Furthermore, many software producers implement auditing functionalities when concerning the customer's data, managed by their SaaS applications. This functionality is also requested by the customers, since they would like to record the changes of their data over the time and use this information for analysis and reporting purposes. Therefore, when creating entity classes, properties like creation time, creator identifier, last modification time may need to be present in different entities. In this regard, the ASP.NET Boilerplate framework provides some interfaces and base classes which automatically supply these properties in a standard way and without manually defining them in the definition of entity classes. In order to get these properties, the entity classes can implement the auditing interfaces, such as *IHasCreationTime*, *ICreationAudited*, or, as it is suggested, can extend the auditing base classes, such as *CreationAuditedEntity*, *FullAuditedEntity* and others.

In the present work, it was thought reasonable that all entities extend the *FullAuditedEntity* base class, since it provides all the auditing properties, which are CreationTime, CreatorUserId, LastModificationTime, DeletionTime, DeleterUserId and IsDeleted. Regarding the latter, when a row is removed from a table, it is only marked as deleted (namely, IsDelete property is set to true), without being deleted definitively from the database.

As concerning the multi-tenancy functionality, in the present work the single database approach has been followed. Therefore, in order to filter entities per tenant, each entity class implements also the *IMustHaveTenant* interface, which requires the implementation of an extra property, called TenantId (of type int). ABP framework provides the possibility to define filters which automatically filter the entity data when SQL queries are performed in a database. The framework provides pre-defined filters, for instance, to filter deleted entities and entities per tenant, just by making these entities to implement the interfaces IsDeleted and IMustHaveTenant respectively. In addition also custom filters can be defined following procedures which are explained in detail in the official documentation, but in the case of the present work are not necessary.

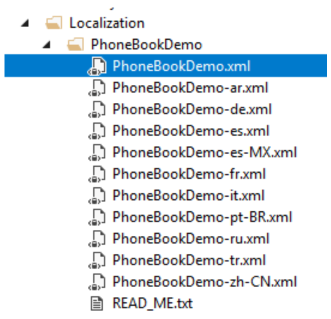
Regarding the multi-tenancy, ASP.NET Boilerplate filters automatically the entities by the TenantId property, returning only those with TenantId equal to the TenantId of the logged user, who performs the SQL query. For instance, when a logged user wants to read the orders data, the application first retrieves his information from the ABP system object that manages the session data (the *IAbpSession* object). Then, it controls to which tenant the user belongs to and returns only the information of the entities (rows) belonging to that tenant. But, all this is possible because the Users entity, as well as all the entities created in this work, contains also the information of the tenant they belong to (the TenantId).

The Core project is also fundamental for the implementation of the Localization feature. The user interface provided by ASP.NET Zero is completely localized, namely the text contents of all its components can be translated into a desired language, choosing from a list of configured languages. ASP.NET Zero supports the dynamic localization, database based localization and per-tenant localization. The Localization feature is implemented here by the means of some XML files, used to translate into the desired language. Each of this file is specific to a different language and contains a set of localization texts. A localization text is a <text> xml element, which is identified uniquely inside the xml file by the value of the name property. Therefore, each localization text must have a unique name, otherwise an exception occurs. An example of a localization text for the Italian language is the following:

```
<text name="Orders">Ordini</text>
```

For instance, when the angular application requests the translation for the localization text "Orders", using the approach that will be explained in Sec. 3.4, the backend application answers with the string stored in the body of the corresponding XML text element ("Ordini").

Moreover, for each new language, that we want to define, a XML file must be created. Also, an additional XML file must be created and used as default in case a requested localization text does not exist for the language currently selected in the user interface. So, when developing a new application feature, if the definition of a new localizable text is needed, it must be put both in the XML files of the languages, for which that text is needed to be translated, and to that of the default language.



- Localization
  - PhoneBookDemo
    - PhoneBookDemo.xml
    - PhoneBookDemo-ar.xml
    - PhoneBookDemo-de.xml
    - PhoneBookDemo-es.xml
    - PhoneBookDemo-es-MX.xml
    - PhoneBookDemo-fr.xml
    - PhoneBookDemo-it.xml
    - PhoneBookDemo-pt-BR.xml
    - PhoneBookDemo-ru.xml
    - PhoneBookDemo-tr.xml
    - PhoneBookDemo-zh-CN.xml
    - READ_ME.txt

Figure 3.4: An example of localization files (taken from Ref. [5]).

An example of XML files is shown in Fig. 3.4, where "PhoneBook" is the ASP.NET Zero project name. As it can be seen, there is an XML file for each desired language (PhoneBookDemo-it.xml, PhoneBookDemo-es.xml) and a default one (PhoneBookDemo.xml).

3.3.4 Core.Shared

This project contains constants, enums and other general classes which can be used across all the backend sub-projects. For instance, the constants used for the validations of DTOs and for configuration of entity objects can be put here. In the present work, the constants storing the minimum length and maximum length values for validation of string fields of DTOs and for configuration of entity objects have been put here.

Moreover, the definition of the constant used to enable (setting to true) or disable (setting to false) the multi-tenancy feature is also here.

3.3.5 EntityFrameworkCore

This project is one of the most important of the solution, since it contains the Entity Framework Core implementation (EFCore) [8], used by default by ASP.NET Zero. This is an Object/Relational Mapping (ORM) framework used for entity-table mapping and database versioning. It provides developers an automated system to access and store the data in a database.

Moreover, ASP.NET Zero solution uses EFCore with **Code-First** approach that consists on focusing first on the domain of the application and starting to create entity classes, instead of designing the database schema first and then creating the entity classes which match the database schema. EFCore creates and modifies the database basing on the entity classes and their configurations.

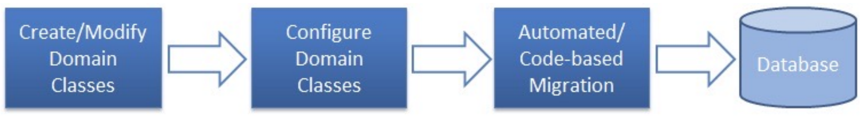


Figure 3.5: The EFCore Code-First approach (Ref. [9]).

The Fig. 3.5 shows the development workflow, followed also in this work, used by EFCore in the code-first approach. As it can be seen, first the entity classes are created or modified (if already existing), then they are configured using some naming conventions or data annotation attributes (which are explained in detail later). Finally, the database schema is created or updated using code-based migrations. About the latter, EFCore provides CLI commands used to generate new migrations. These are auto-generated C# classes containing all the changes in the application data model (namely changes on the entities and their relationships), and apply them to the database. These commands can be executed opening a terminal inside this project folder and, for instance in Unix OS, they are:

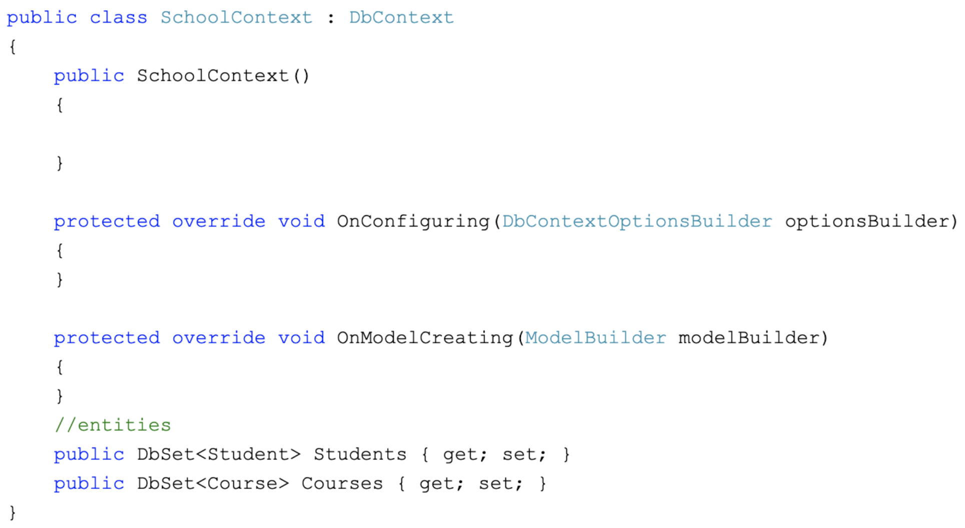
```
dotnet-ef migrations add "Migration_Name"
dotnet-ef database update
```

The first command creates a new migration, specifying the name, whereas the second applies this migration to the underlying database. The migrations classes are all located in the "Migration" folder inside this project. The changes of a migration class remain local until they are applied to a database. To achieve this, a connection with a database must be started. For this purpose, EFCore uses a configuration file, named *appsettings.json* and located in the *Web.Host* project. This file contains a connection string which specifies the information needed to connect to the database (server name, port number and database name). In this way, EFCore can apply the new migrations to the database.

In order to understand in detail how code-first approach works, it is necessary to discuss about some key elements present in this project and used by EFCore, such as the **DbContext** class, the implementations of **repositories**, database **migrations** and other specific concepts of EFCore.

An instance of the DbContext class corresponds to a session with a database in which it is possible to query and store instances of entity classes to the database. It is additionally used for caching, transaction management and to configure the data model. Technically, DbContext is a C# class, which inherits from DbContext base class (defined in the *System.Data.Entity* namespace of EFCore), and contains properties of type DbSet<TEntity> for each entity type in the data model.

The Fig. 3.6 shows a simple example of a DbContext class. As it can be seen, the SchoolContext class extends the DbContext base class and contains the DbSet<Student> and DbSet<Course> properties. The creation of a SchoolContext instance is necessary to start connections with the database, where it is possible to query, store, delete and update the data of Student and Course tables in the database.



```
public class SchoolContext : DbContext
{
    public SchoolContext()
    {

    }

    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
    }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
    }
    //entities
    public DbSet<Student> Students { get; set; }
    public DbSet<Course> Courses { get; set; }
}
```

Figure 3.6: An example of the DbContext class (Ref. [10]).

EFCore automatically configures the data model (namely, creates the mapping between entities and database) by searching for a set of **conventions** and/or **annotations** inside the entity classes, which are found in the DbSet<TEntity> properties of the DbContext class (i.e. Students and Courses properties, in the previous example).

The conventions used by EFCore (code-first approach) are defined in the *System.Data.Entity.ModelConfiguration.Conventions* namespace and some of them are summarized in the following list:

- table names are defined using the pluralized form of the entity classes name. In other words, if we have an entity Order, this creates a table in the database with name Orders
- the primary key is configured based on the property name Id or ClassNameId. In other words, if we have an Order entity and it has a property named Id or OrderId, this corresponds to the primary key in the generated Orders table
- the name of columns of a table are derived from the name of properties of the corresponding entity. In addition, column names can be changed using the code-first data annotation [Column], which is explained in the following.

On the contrary, the data model can be configured based on code-first **data annotations**, which are briefly the followings:

- [Table]: can be put to an entity class to setup the name of the corresponding table in the database
- [Column]: can be put to an entity property to setup the corresponding column name and data type of the table in the database
- [Key]: can be put to an entity property to identify a key property and set as primary key the corresponding column of the table in the database
- [Required]: can be put to an entity property to indicate that the corresponding column of the table in the database has a NotNull type
- [MaxLength]: can be put to an entity property to setup the maximum string length allowed in the corresponding column of table in the database
- [MinLength]: can be put to a en entity property to configure the minimum string length allowed in the corresponding column of the table in the database
- [StringLength]: can be put to a en entity property to configure the maximum string length allowed in the corresponding column of the table in the database.

Instead, one other way to configure the data model is by overriding the *OnModelCreating(ModelBuilder)* method of the DbContext base class and specifying there the configurations that are needed.

Moreover, the configuration of the data model is really important, since it is used in the migrations to the database. When a new migration is created (by applying the command previously seen), EFCore controls if the current data model has changed by comparing it with the one reconstructed using, for instance, the conventions approach. If the two model differ, for instance, a property of an entity has a different type, EFCore adds the code to apply the new changes inside the migration class. Then, the new migration is applied and the database scheme is updated.

Furthermore, DbContext is a combination of the *Repository* and *Unit Of Work* patterns, namely it can be used to perform more SQL operations to a database and, if no errors occur, to apply their results together as a unit to the database. The DbSet<TEntity> properties of the DbContext class represent the repositories, whereas the DbContext class the unit of work system. When a repository method is called, a new DbContext class is instantiated, and therefore a new unit of work starts. The new DbContext instance receives from the repository method the SQL command to apply to the database. Then, if there aren't any exception, the result of the operations are committed, otherwise rolled back.

Let us now describe these key elements of EFCore project, starting from the *Repositories*. A repository is a software design pattern and practice used to create an additional layer between the application layer and the data access layer (namely the database system). A repository is represented as a class object used to manage how the data is manipulated from/to the database. All the code in the application that fetches or pushes a data into the database use these objects, instead of directly accessing the database.

In ABP, the repositories are interfaces used to perform operations on a database using entities. Generally, a separate repository is used for each entity class. The EFCore framework automatically creates default repositories for each entity class defined in DbSet properties of the DbContext class. A default repository is composed of default operations used to access the database, in order to visualize, insert, update and delete its data. An example of the list of the methods used to perform the default operations is the following:

- For visualization: Get, GetAsync, FirstOrDefault, FirstOrDefaultAsync, GetAll

- For insertion: Insert, InsertAsync, InsertAndGetId, InsertAndGetIdAsync
- For edition: Update, UpdateAsync
- For deletion: Delete, DeleteAsync.

As it can be seen, according to the particular case, the database operations can be performed by the means of asynchronous methods or synchronous methods.

Furthermore, there is also the possibility to create *custom repositories* for an entity when custom repository methods are needed for that entity. In order to achieve this, it is necessary to define an interface, that extends IRepository<TEntity> (or IRepository<TEntity, TPrimaryKey>). Then, the new methods for database operations can be defined in this interface using some particular conventions and configurations.

ABP framework allows to inject objects of type IRepository<TEntity> (or IRepository<TEntity, TPrimaryKey>) in the constructor of any application service classes by the means of dependency injection. In addition, all repository instances are *transient*, namely they are instantiated per-usage through the Dependency Injection pattern.

Moreover, it is really important to know the way how ASP.NET Boilerplate manages database connections and transactions. For this purpose, the framework uses its internal **Unit of Work** (UOW) system. Each time ABP executes a unit of work method it opens a connection and begins a transaction with the database. A transaction is composed of all the operations, defined inside the unit of work method, which must be performed to the database. At the end of the execution of the method, the transaction is committed, namely the results of the operations are saved in the database, and the connection is closed. If the method throws an exception, the transaction is rolled back, namely the result of the operations are discarded, and the connection is closed. The UOW system of ABP works by the means of some conventions, automating the process for the configuration of unit of work methods. UOW does not depend on the database provider (i.e. SQL Server, MariaDB, PostGreSQL) and it can be implemented both in web applications and web services.

By the means of particular conventions, the following methods are automatically registered by the ABP framework as unit of work methods:

- all application service methods
- all repository methods
- all MVC, API controller methods

Example of conventions are the fact that application service classes must implement the IApplicationService interface, the repository classes must implement the IRepository interface and so on for the controllers. Alternatively, it is also possible to create custom unit of work methods or by registering them in the PreInitialize() method of the ASP.NET Core solution, or by using the [UnitOfWork] attribute placed before the methods which are wanted to be registered as unit of works.

Furthermore, if there are two unit of work methods and the first calls the second, both use the same database connection and transaction. The first method opens and manages the connection and transaction, while the second keeps using them. For instance, if a repository method is called and there is no unit of work started yet, ABP automatically creates a new unit of work which starts a new transaction, which contains all the operations performed in that repository method. Then, if no exceptions are thrown during the execution of the repository method, it commits the transaction.

Instead, if an application service method is called and it contains repositories methods, as shown in the example of Fig. 3.7, these latter do not begin a new unit of work, but they share the same unit of work, which is initiated by ABP when executing the application service method.

```
public class PersonAppService : IPersonAppService
{
    private readonly IPersonRepository _personRepository;
    private readonly IStatisticsRepository _statisticsRepository;

    public PersonAppService(IPersonRepository personRepository, IStatisticsRepository statisticsRepository)
    {
        _personRepository = personRepository;
        _statisticsRepository = statisticsRepository;
    }

    public void CreatePerson(CreatePersonInput input)
    {
        var person = new Person { Name = input.Name, EmailAddress = input.EmailAddress };
        _personRepository.Insert(person);
        _statisticsRepository.IncrementPeopleCount();
    }
}
```

Figure 3.7: An example of unit of work in ASP.NET Boilerplate (Ref. [6]).

Not all unit of works start transactions. For instance, the HTTP GET requests start new unit of works, since they call application service methods (or controller methods), but they do not start transactions with the database. Instead, all the other types of HTTP request (POST, PUT, DELETE) start a unit of works with also a database transaction, if the transactions are supported by the used database provider. This is because an HTTP GET request do not make any change in the database, whereas the other requests do. Therefore, there is no need to start a transaction when only GET requests are performed.

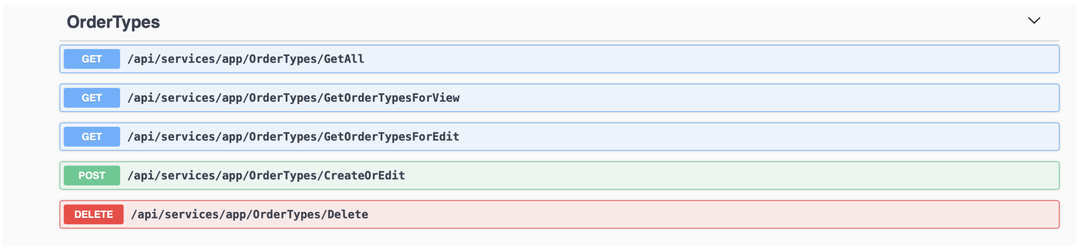
3.3.6 Web.Host

This project contains an application whose execution serves the whole backend application as a remote REST API which can be consumed by any client application. This project does not provide an user interface (i.e. web related files like html, css or javascript). However, if the user interface is needed, the Zero solution makes available also the *Web.MVC* project, which, unlike the Web.Host project, provides also a view, implementing the *Model-View-Controller* pattern.

Moreover, the Web.Host project provides only token based (JWT) authentication and, since there is no user interface, no form-based authentication is implemented. In addition, it does not offer the *CSRF* protection since it is not a security concern in the token-based authentication approach. By contrary, it enables the *CORS* functionality, allowing in this way cross origin requests.

By default, the *Swagger UI* tool [11] is enabled and configured by this project. Swagger UI is an open-source project which allows development team to visualize and interact with the backend APIs, without implementing additional code needed to achieve this. Indeed, Swagger generates automatically an intuitive and interactive API documentation based on the application services or controllers which are created in the ASP.NET Core solution.

So, in this thesis work, when the Web.Host project is executed, first a login page is automatically open in the default browser. Here, the user must insert some information such as a username, password and the tenant name. Then, since Swagger UI in enabled, she can use it to execute and test the APIs, as shown in Fig. 3.8 and Fig. 3.9.



OrderTypes

GET /api/services/app/OrderTypes/GetAll

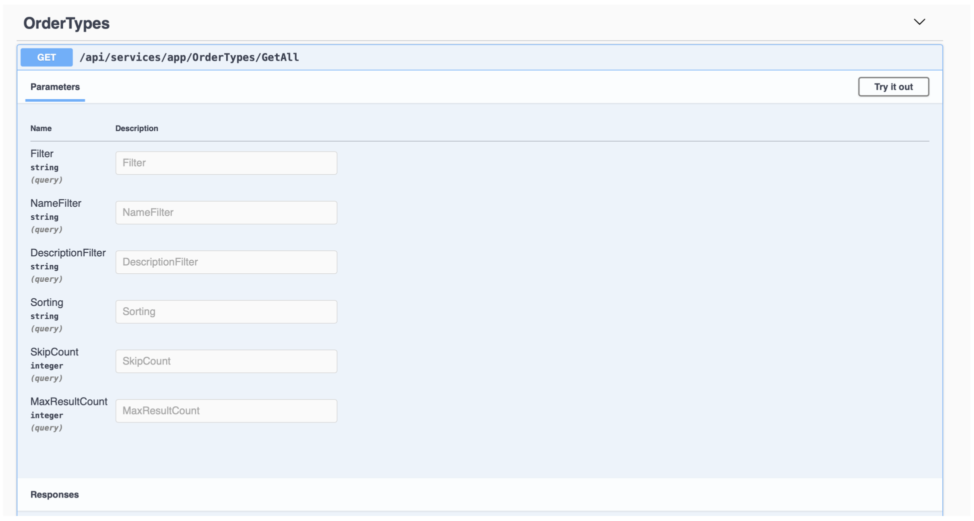
GET /api/services/app/OrderTypes/GetOrderTypesForView

GET /api/services/app/OrderTypes/GetOrderTypesForEdit

POST /api/services/app/OrderTypes/CreateOrEdit

DELETE /api/services/app/OrderTypes/Delete

Figure 3.8: The swagger UI interface.



OrderTypes

GET /api/services/app/OrderTypes/GetAll

Parameters — Try it out

Name	Description
Filter string (query)	Filter
NameFilter string (query)	NameFilter
DescriptionFilter string (query)	DescriptionFilter
Sorting string (query)	Sorting
SkipCount integer (query)	SkipCount
MaxResultCount integer (query)	MaxResultCount

Responses

Figure 3.9: An example of API testing in the swagger UI interface.

3.3.7 Web.Core

This project contains some useful classes that are used in the Web.Host project, for instance, the *API Controllers* and *MVC Controllers* classes.

The MVC Controllers are created using the MVC (Model-View-Controller) programming pattern. This divides an application (typically, a backend application) in three parts: the Model, the View and the Controller. The Controller receives user inputs (i.e. http requests or operations on the View), updates the Model and selects and returns a View. The Controller behaves as mediator between the Model and the View. Instead, the View contains all the components used to implement the application user interface. It allows users to perform operations which are handled by the Controller. Moreover, it retrieves the data to render from the Model, but does not perform any operations on the Model. Typically a View can be a html page, enriched with some code used to retrieve data from the Model. The Model is composed of the objects used to retrieve and store data from/to a database. It is updated by the Controller as result of user interactions.

Instead, the API Controllers are similar to the MVC Controllers, with the difference that no view components are managed by them. They receive HTTP requests from the client applications and send back HTTP responses transporting data serialized in json and/or XML format (no html pages are transported). With this approach, the view components (if needed) are managed directly by the client applications. This is common for Single Page Applications (SPA, i.e. Angular), since they interact with a backend server only sending request for getting or saving data and the received data is autonomously rendered by them. In addition, the SPAs internally implement the routing mechanism (as it will be explained in Sec. 3.4). No html pages corresponding to different requested URLs are returned by the server, as it is done for traditional web applications (i.e. ASP.NET Core MVC web applications).

An important fact is that the ASP.NET Zero solution quite adopts AJAX requests to provide a better user experience. An AJAX request is typically a HTTP request made by a browser-resident javascript code (i.e. an angular application) that uses XML to encode the request data and the response data (if present).

Moreover, for what concerns the feature implemented in the present work, it is necessary to create API controllers working as mediators between the angular application and the application services. What happens is that a user, interacting with the angular application, sends a HTTP request to the Web.Host API application. The latter calls the controller method that has been registered as the handler for that specific http request. Then, this method calls in turn an application service method, passing it the data extrapolated from the http request. Next, according to the business logic, the application service method processes the request and produces a result, which is then returned back serialized to the client by the controller method.

However, the ABP framework allows to expose directly all the application services to the remote clients, making unnecessary the manual implementation of API controllers. In order to achieve this functionality, it is required to insert the following configuration in the *Pre-Initialize* method of the module *<ProjectName>WebCoreModule* (which is found in the current project):

```
Configuration.Modules.AbpAspNetCore()
.CreateControllersForAppServices (typeof(MyAppModule).Assembly,
moduleName: 'app', useConventionalHttpVerbs: true);
```

This configuration is set by default by the framework and allows to automatically create API Controllers for all application services which have been defined with the procedure described in the *Application* project. When this conversion is achieved, the default route of an application service has this form:

```
/api/services/<module-name>/<service-name>/<method-name>
```

For example, if we have an application service named *OrdersAppService*, which defines a method called *CreateOrEdit*, the latter is accessible from the clients through the following URL:

```
/api/services/app/orders/createOrEdit
```

where "app" is the module name chosen in the previous configuration (its default value is "app"). Moreover, if the parameter *useConventionalHttpVerbs* of the configuration code is set to true (which is the default value), ASP.NET Boilerplate framework determines the HTTP request types (called also verbs) for the application service methods by using a set of *naming conventions*. A list of them is the following:

- Get: used if the method name starts with "Get".
- Put: used if the method name starts with "Put" or "Update".
- Delete: used if the method name starts with "Delete" or "Remove".
- Post: used if the method name starts with "Post", "Create" or "Insert".

- Patch: used if the method name starts with "Patch".
- Otherwise, if none of the previous case verifies, post is used by default as HTTP verb.

Alternatively, the application services methods can be labelled by attributes of *Microsoft.AspNetCore.Mvc.Core* package used to specify the HTTP action or routes of those methods, bypassing the naming conventions. For instance, "[HttPost]" and "[Route("routePath")]" could be exploited in order to indicate a POST method and the route through which that method is accessible respectively.

In conclusion, in the present work, for the majority of the functional requirements, the application services have been implemented using the default approach described previously. However, in some cases it was needed to implement manually the controllers classes. For instance, for the requests regarding the upload of the excel files for the order types and the PDF files for the orders offers, a controller class with a handler method have been created. This because it was not possible to manage this kind of request directly in the corresponding application services.

3.3.8 Migrator

This project contains a console application called *Migrator.exe* and used to create and/or migrate host and tenant databases. First the application gets from a configuration file, located in this project and named appsettings.json, the connection string for connections to the host database. Then, it creates the database (if not already existing) and applies the migrations. Next, it gets the connection strings of the tenant databases and runs migrations to those databases. It skips a tenant if it has not a dedicated database or its database has already been migrated for one other tenant (this happens for databases shared between several tenants). Moreover, the developers can use this tool while working in the development environment to migrate the databases in the production environment, instead of using the Migrate.exe application of EntityFrameworkCore (which requires some further configurations and can only work with a single database). However, in the present work the migrator of EFCore has been adopted, since the management of a single database was required.

3.3.9 Tests

This sub-project contains all the classes used for the unit and integration testing. In ASP.NET Zero, a test is a C# class that extends the *AppTestBase* (provided by the solution), which initializes all the testing system, creates an in-memory fake database, inserts the seeds with initial data in the database and access the application as admin and as default tenant.

Typically, unit testing is applied to the services classes (as well as in this thesis work). In ASP.NET Zero, the unit tests are run on a copy of the development database, which is recreated each time a test is executed and dropped when the test terminates. In other words, the database copy leaves in ram memory. In this way, the unit tests data and the customer's data are kept separated, avoiding eventual data pollution.

Moreover, it is a good practise to populate the testing database with some initial data. This can be achieved by creating seed classes, that contain methods called by ASP.NET Zero before the execution of each test method. The methods of the seed classes produce some data, which is then inserted in the testing database every time a test method is executed and dropped when it terminates. The seed data can be created inside the seed classes or by the means of in-memory data structures (i.e.

arrays) or by parsing external csv files. The seed data approach helps to create lighter and smarter test code, avoiding to repeat in test methods code used to prepare the testing environment. However, this requires more attention, because the initial data can change and the tests could fail, even though the application services logic remains correct. Therefore, it is better, as much as possible, to write tests that do not depend from initial data.

Technically, in order to perform unit testing on the methods of an application service class, it is necessary to create an instance of its implemented interface and initialize it with:

```
Resolve<InterfaceTypeName>();
```

This instruction performs the dependency injection by constructor operation (explained further on), which creates and inject an instance of the application service class implementing that interface. In this way, all the methods of this class can be called, passing input parameters, and their result can be tested.

An example of a test class is:

```
public class OrderType_Tests : AppTestBase
{
    private readonly IOrderTypesAppService _orderTypeAppService;

    public OrderType_Tests() =>
    _orderTypeAppService = Resolve<IOrderTypesAppService>();

    [Fact]
    public void Should_Get_All_OrderTypes_Without_Any_Filter()
    {
        //Act
        var orderTypes =
        _orderTypeAppService.GetAll(new GetOrderTypesInput());

        //Assert
        orderTypes.Result.Items.Count().ShouldBe(10);
    }
```

Since ASP.Net Boilerplate uses ASP.NET Core, which uses the *XUnit* library for unit testing, the *[Fact]* annotation must be put before each test method. Typically, as it can been seen in this code example, the body of a test method could logically be split into a "Act" section, where the tested application service method is called, and a "Assert" section where the test logic is implemented using the application service returned data. In this case, the test should verify if the GetAll method of the application service returns all the order types elements currently present in the database (which should be ten).

Lastly, before concluding this section, it may be interesting to know more about the way the ASP.NET Zero solution implements common pattern and functionalities like **Dependency Injection**, **Authorization** and **Validation**. These are fundamental to understand the overall functioning of the solution.

3.3.10 Common functionalities

Similarly to the other frameworks based on the object-oriented programming, also in ASP.NET Zero there is the problem of classes which depend on each other. A class

A is dependent from a class B, if A defines a class variable with type B. Therefore, the class A cannot be instantiated if class B is not created first. As a result of this, there is an error in the compilation phase of class A.

In the case of ASP.NET Zero solution, the classes which may depend on each other are application services, repositories and controllers. For instance, when a controller class must be instantiated, since one of its methods may contain a call for a method of an application service class, an instance of the latter class must be provided first. The same problem occurs between application services and repositories, since the former may need to call methods of the latter.

In order to solve the problem of classes dependent on each other, ASP.NET Zero uses the *Inversion Of Control* (IoC) and the *Dependency Injection* (DI) patterns. The IoC pattern removes the dependencies between classes during the compilation time, keeping them only in execution time. Following the previous example, the class A must not define a class variable of type B, but a class variable of type an interface that abstracts the behavior of B. Then, an external component, the IoC container (known also as Dependency Injection framework), takes care of substituting the interface with an instance of B, when A is created.

The IoC container first registers the association between interfaces and their implementing classes. Then, when required by the application during the execution time, it resolves the interfaces by providing, through the Dependency Injection, an instance of the classes implementing them. By default, ASP.NET Boilerplate is configured to use *Castle Windsor*, which, among the other IoC containers produced by Microsoft, is the most mature one. The other known IoC containers are Unity, Ninject and StructureMap.

In ASP.NET Boilerplate, there are different ways of registering the associations between interfaces and their implementing classes in the IoC container. Most of the time, the *conventional registration* is sufficient. This consists on Boilerplate automatically registering, inside its default IoC container, all repositories, application services and controllers, by exploiting some convention in naming and the implementation of particular interfaces. Alternatively, for additional operations, such as custom registrations, injection hooks and interceptors, the functionalities offered by Castle Windsor could be used.

An example of conventional registration performed by the ASP.NET Boilerplate is the following. Suppose we have the IPersonAppService interface and a PersonAppService class that implements it, ABP automatically registers their association, only if IPersonAppService implements the IApplicationService interface. In this approach also the naming conventions are very important. For instance, it is possible to change the name of PersonAppService to MyPersonAppService or another name which contains the "PersonAppService" suffix. This registers it to IPersonAppService because it has the same suffix. However, if the application service is named without the suffix, such as "PeopleService", it is not automatically registered to the IPersonAppService, but it can be registered to the IoC container using self-registration, namely the association between the interface and the class is manually registered inside the Castle Windsor container. The same approach is followed for repositories classes which implement the *IRepository* interface and for controllers implementing *IController*.

In all these cases the associations between interfaces and implementing classes are registered as *transient*, namely the implementing classes are instantiated each time they are needed during the execution of the application.

For what concerns the dependency injection, ASP.NET boilerplate supports the dependency injection by *constructor* and by *property*. The Fig. 3.10 shows an example of the two approaches used together. In dependency injection by constructor, an

```
public class PersonAppService
{
    public ILogger Logger { get; set; }

    private IPersonRepository _personRepository;

    public PersonAppService(IPersonRepository personRepository)
    {
        _personRepository = personRepository;
        Logger = NullLogger.Instance;
    }

    public void CreatePerson(string name, int age)
    {
        Logger.Debug("Inserting a new person to database with name = " + name);
        var person = new Person { Name = name, Age = age };
        _personRepository.Insert(person);
        Logger.Debug("Successfully inserted!");
    }
}
```

Figure 3.10: Example of the dependency injection pattern (taken from Ref. [6]).

interface object is passed in the constructor of a class, which needs an instance of the class implementing this interface. The IoC container automatically creates an object of this class and pass it into the constructor. In the example of Fig. 3.10, an object of PersonRepository is created for IPersonRepository interface an passed to the constructor of the class PersonAppService. The constructor injection pattern is a good way of providing the dependencies of a class, because it does not allow to create an instance of the class without supplying first all its dependencies. It is also a strict way of explicitly declaring which are the requirements of a class, so that it can be instantiated and work properly.

However, in some occasions a class may depend on other classes, but can work without them. This behaviour is true for cross-cutting functionalities such as logging. A class can work without logging functionality, but it can write logs only if a logger instance is supplied to it. In this case, the dependencies can be resolved using class variable properties (i.e. Logger in the code), instead of recovering them inside the class constructor, as it is done in dependency injection by constructor.

Furthermore, one other important functionality offered by the ASP.NET Zero solution is the **Authorization**. Basically, Authorization is a technique used to check if a user is allowed to perform some specific operation in an application. In ASP.NET Zero, it can be implemented both at application services and controllers levels. The ASP.NET Boilerplate framework defines a permission based infrastructure in order to implement the authorization functionality. This infrastructure exploits the IPermissionChecker service to check permissions. In ASP.NET Boilerplate, this service can be implemented as preferred, whereas in the Zero solution it has already been implemented. At the bases of this permission based system lays the fact that a unique permission is defined for each operation that needs to be authorized. In the Zero solution, there already exists a class derived from *AuthorizationProvider* base class, which is used to define the permissions. A possible example of this class could be the following:

```
public class MyAuthorizationProvider : AuthorizationProvider
{
    public override void SetPermissions(IPermissionDefinitionContext context)
    {
        var administration = context.CreatePermission("Administration");

        var userManagement = administration.CreateChildPermission("Administration.UserManagement");
        userManagement.CreateChildPermission("Administration.UserManagement.CreateUser");

        var roleManagement = administration.CreateChildPermission("Administration.RoleManagement");
    }
}
```

Figure 3.11: An example of an AuthorizationProvider class (Ref. [6]).

As it can be seen, new permissions are created by the means of "CreatePermission" and "CreateChildPermission" methods. These methods need a series of parameters, among which also the permission string. Once the permissions are created, it is possible to use the AbpAuthorize attribute to check them before executing an application service method. For instance, considering the application service method in Fig.3.12, a user can not call it if he has not the permission "Administration.UserManagement.CreateUser", which has been previously defined in the AuthorizationProvider class.

```
[AbpAuthorize("Administration.UserManagement.CreateUser")]
public void CreateUser(CreateUserInput input)
{
    //A user can not execute this method if he is not granted the "Administration.UserManagement.CreateUser" permission.
}
```

Figure 3.12: An example of an authorize attribute (Ref. [6]).

Furthermore, there are also the attributes AbpMvcAuthorize and AbpApiAuthorize to check permissions in methods of MVC Controllers and Web API Controllers respectively.

This approach of authorization management has been followed also in the present work. All the application service methods are annotated with the AbpAuthorize attribute, since the customer required to restrict some operations only to particular user's roles, as discussed in Sec. 2.2.

Very important to know is also how ASP.NET Zero manages the user input **Validation**. The ASP.NET Boilerplate framework provides a system to automatically validate input data (DTOs) passed on all application service methods, MVC controller and API controller methods. ABP can automatically validate inputs to these methods by the means of data annotation attributes or custom validations.

Concerning the first approach, attributes like Required, MaxLength, MinLength, RegularExpression and many others can be used to check the validity of the properties of DTO objects, throwing an *AbpValidationException* if any of them is invalid. For instance, the Required attribute controls if a DTO field is not null, whereas MaxLength and MinLength attributes specify the maximum and the minimum length respectively of a DTO string property. In addition, ASP.NET Boilerplate checks also whether an input DTO is null or not, throwing an AbpValidationException in the second case. In this way, the writing of null-check code, inside the previously cited methods, is not necessary.

Morevoer, if data annotation attributes are not sufficient and a more precise validation is needed, the custom validations could be a good choice. In this regard, a DTO class can implement the *ICustomValidate* interface, which declares the*AddValidationErrors* method to be implemented. In this method, the validation errors for each DTO property are decided by following some standard procedures.

Finally, as previously introduced, ABP registers as automatically validated all the application services, MVC controllers and API controllers. However, the framework provides also some attributes used to specify that a certain class must be automatically validated. Attributes like *DisableValidation* and *EnableValidation* can be used to enable or disable validation on classes or their inner elements (methods, properties). The DisableValidation attribute can be used to disable the validation on classes, methods or properties of DTOs, in which they are applied. Whereas, the EnableValidation attribute can only be used to enable the validation for a method, if it is disabled in the containing class.

This concludes the description of the backend part. The next section introduces instead the frontend part.

3.4 Frontend: Angular solution

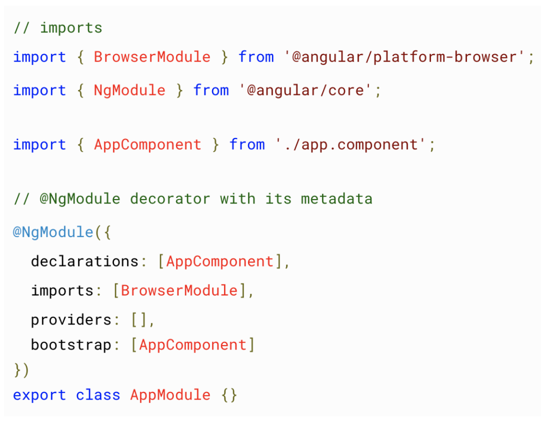
As previously explained, the frontend part of the SaaS application is implemented by the angular solution. In this section, the angular solution is explored, concentrating on its building elements and on some of the useful features offered by ASP.NET Zero in order to facilitate the development of the business logic. Let us start by introducing the key concepts of the Angular framework.

3.4.1 Basic elements of Angular

The main elements of an angular application are the modules, the components and the services. The **modules** (NgModules [12]) allow to organize better the application code and extend it using external libraries. The modules could be thought as containers, in which the source code, composed of components, directives, pipes, services, is collected and organized into coherent blocks of functionality, each based on a feature, application business domain, or common set of utilities. Technically, a module is a Typescript class marked with the *@NgModule* decorator, that takes in input a configuration object, containing all the information needed by Angular in order to create the module. The Fig. 3.13 shows an example of a NgModule class and its configuration object. As it can be seen, the latter is composed of four fields:

- Declarations: specifies the components defined in this module, i.e. AppComponent
- Imports: specifies the dependencies of this module, i.e. FormsModule, BrowserModule. These dependencies are typically angular libraries
- Providers: specifies the list of all services which can be injected in all the components defined in this module (i.e., in AppComponent). For instance, the service classes used to query the backend APIs must be put here, in order to be injected and used in the components
- Bootstrap: specifies the root component used for bootstrapping the app (only the root module must have this field)

The **Components** [13] are the real building blocks of an Angular application. They are used to present the application views and to allow the user interactions. They are composed of three files. The first is a typescript file that contains the definition of a class marked with the *@Component* decorator. This is used to retrieve the data from a backend server by the means of services. Then, there is a html file



```
// imports
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';

import { AppComponent } from './app.component';

// @NgModule decorator with its metadata
@NgModule({
  declarations: [AppComponent],
  imports: [BrowserModule],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule {}
```

Figure 3.13: An example of NgModule class (Ref. [12]).

used to present the component view, which gets the data to be visualized from the typescript class. Finally, a css file is used to configure the style of the component view. The association of these three files together is performed in the configuration object passed to the @Component decorator, which tells Angular how to create the component and how to render its view.

Services [14], instead, are typescript classes marked with the *@Injectable* decorator. They are not instantiated directly by the Components, but can be injected by the Angular *Injector* inside the constructor of the components, exploiting the Dependency Injection pattern. In order to perform an injection into a component, the service class must be registered in the "Providers" section of the module, which imports that component, as in this example:

```
@NgModule({ providers: [ <service-class> ]})
```

Instead, the component that needs to use it, must declare the dependency in its constructor, for instance as:

```
constructor (private myService: <service-class>){}
```

Furthermore, every Angular application has at least one module, the root module, which can be bootstrapped in order to launch the application. In case of a simple application with few components, the presence of only the root module is enough. By contrary, as the application grows, the root module should be refactored into more feature modules that represent collections of related functionalities, which must then be imported in the root module. This approach is followed also by the Angular solution of the present work, as it is explained in the next section.

3.4.2 Structure of Angular solution

The Fig. 3.14 shows that the angular project is essentially composed of three main modules.

The RootModule is the one responsible to bootstrap the application. It is created by the file *main.ts* (located inside the src folder), which is the entry-point of the Angular solution. When the angular application bootstraps, it calls a method inside main.ts which loads the root module with all its components.

The AccountModule contains the components and services which implement functionalities, such as login, two factor authentication, registration, password forget/reset, email activation and others.

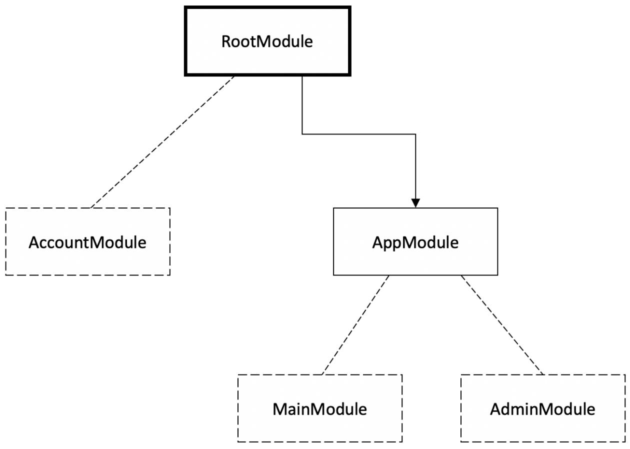


Figure 3.14: The frontend structure (Ref. [5]).

The AppModule is used to group together the application modules and provide a base layout. It contains in turn two other sub modules: AdminModule and MainModule. The AdminModule contains the components and services for user management, role management, tenant management, language management and settings. By contrary, the MainModule is used to develop the application. Here, all the Angular components and services used to implement the application business logic are imported. For instance, in the "declaration" section of this module, the components for order visualization, creation and edition have been imported and the same for all the other functionalities implemented in the present work. While, in the "entry-point" section, the pop-up dialogs have been imported, such as the one used for comments creation and visualization. However, ASP.NET Zero suggests to divide this module into smaller modules, instead of adding all the functionalities inside it. This approach allows better organization of the application structure and allows the re-usability of single modules in other parts of the project or in other projects.

Moreover, the ASP.NET Zero angular solution provides also a *routing module* for each of the main modules previously described. For example, for the RootModule there is the RootRoutingModule, for the AccountModule the AccountRoutingModule and so on for the others.

The routing modules are simple modules used to register a set of routes. A route is an association between an URL and the module or component that must be loaded by the Angular **Router** when that URL is requested by the browser. The Router is a core part of the Angular framework. It listens to the changes of URL in the browser's address line. Then, when a new URL is requested, it checks for matching, looking in the set of all registered URLs (inside all routing modules) and loads the module or the component corresponding to that URL.

Therefore, for each of these fundamental modules a set of routes have already been registered by ASP.NET Zero. For example, for AccountModule the set of routes starting with "/account" (like "/account/login") have been registered, while for AdminModule those starting with "/app/admin" (like "/app/admin/users").

In addition, the loading of modules is of type *lazy*, namely a module and all its components are loaded only if at least one of its routes has been activated. For instance, when an URL starting with "app/admin" is requested, only the AdminModule and its components are loaded. This approach brings better startup time and a better development time.

Besides the main modules previously describe, ASP.NET Zero provide some others which are shared between the main modules. A list of them is the following:

- app/shared/common/app-common.module: contains functionalities used by MainModule and AdminModule
- shared/common/common.module: contains functionalities used by AccountModule and AppModule, and their sub modules
- shared/utils/utils.module: contains functionalities used by all modules, and their sub modules. General code which must be used also in different applications is placed here
- shared/service-proxies/service-proxy.module: contains code auto generated by nswag. In the nswag folder of the solution root folder there is a .bat file, whose execution automatically generates all the services classes used to query the backend APIs. These services classes are all defined inside the main class defined in the *service.proxy.ts* file, which is placed inside the shared/service-proxies folder. Then, in order to be injected and used in the components, these services must be imported in this module.

3.4.3 Functionalities offered by Zero

Concerning the functionalities offered by ASP.NET Zero for the angular solution, very useful is the component super-class, named *AppComponentBase*, that can be inherited when creating new components. This class provides additional pre-injected services used for localization, permission checker, user interface notification, app navigation, settings and many others. The example bellow shows a child component inheriting the aforesaid services objects from the AppComponentBase super-class.

```
export class MyComponentName extends AppComponentBase
{
    constructor(
        injector: Injector
    ) {
        super(injector);
    }
}
```

With the "super(injector)" instruction, the child component calls the constructor of the base component, where all the instances of these service classes are injected and saved into global variables, which are then inherited by the child component.

Let us now furnish some details on how these service instances can be used inside the child components. Concerning the localization service, a function is made available, defined as follows:

```
this.l('<LocalizationString>');
```

This function could be used in the child component class to get a given word translated into the language currently selected by the logged user. For instance, this.l('Orders') will return the string "Ordini", if the Italian language has been chosen. To achieve this, the "Orders" localizable string must be put in the XML file for the Italian language, as previously described in Core project (Sec. 3.3.3). The same functionality can be obtained in a component view, as shown in the following example:

```
<span>{{l('Orders')}}</span>.
```

An important and already implemented service offered by ASP.NET Zero is the *app-navigation.service.ts*, located in the "app/shared/layout/nav" folder of the Angular solution. This defines a service class which allows to create menu item elements in the application layout. The following code shows, for instance, how a new menu item can be added in the application layout:

```
new AppMenuItem("Orders", "<authorization_string>", "<icon_name>",
    "/app/main/orders");
```

AppMenuItem is a typescript class, containing a constructor which requires some parameters in input. The first parameter is the name of the new menu item, the second the authorization string that the user must have in order to view this item, the third represents the icon name and the last one corresponds to the route which must be activated when this menu item is clicked. In this example, the orders component will be rendered. In addition, the menu item name is localized, meaning that the localize function (previously explained) is applied to the name parameter when creating the menu item. Therefore, the menu item name is automatically translated into the language currently set by the user.

Regarding the spinner service, ASP.NET Zero uses *ngx-spinner* (that is an element of Angular Bootstrap that has been imported in the Angular solution) to temporary block UI elements and show loading indicators. A component inheriting the AppComponentBase receives an instance of this spinner service through injection. To show and hide the loading spinner, the *show()* and *hide()* methods can be called on the injected service object. The following code shows an example of how to use the inherited service object in order to show and hide the indicator:

```
this.spinnerService.show();

setTimeout(() => {
    this.spinnerService.hide();
}, 1000);
```

Furthermore, the components inheriting AppComponentBase, receive a pre-injected service, called *PermissionCheckerService*, used to check user permissions. This service makes available a function, named *isGranted()*, which receives as input an authorization string and checks if the current user has the permissions corresponding to that authorization string, returning true or false. This method can be typically used in a component view to hide buttons or other elements to the users who do not have some given permissions. For instance, this has been done for the "Create new order" button, because only the Operator user can create a new order, whereas the Collaborator is not allowed to do that. A simple code example is:

```
<button *ngIf="isGranted('Pages.Administration.Orders.Create')"
    (click)="createOrder()">
    {{l("CreateNewOrder")}}
</button>
```

Finally, AppComponentBase injects also an object of the NotificationService class, which provides four possible methods (success, info, warn, error), each with its specific functionality and layout. These service methods output pop-up windows in the bottom of the page where the result of operations done by the users are shown. In this work, for instance, when a user deletes an item from the orders table, a message shows the success of the operation. The same approach has been followed for the operations done by users in all the table structures implemented in the present work. An example of a notification service method could be the following:

```
this.notify.success('a message text', 'optional title');
```

As it can be seen, this.notify is an object of the NotificationService service class inherited by AppComponentBase, and it opens a pop-up window showing a success text message. Moreover, each of the four methods, previously cited, expects a first parameter for the message text and a second optional for the title.

Having explained also the frontend part, let us now put everything together and present an example of a full stack implementation of a feature.

3.5 Full stack implementation of a feature

This section summarizes all the steps required to implement a particular feature of the application, from the backend to the frontend. With some little variations, also the other features can be implemented. For instance, it could happen that some of these steps are not needed for a feature, because they have already been executed for one other.

We focus on the feature "User can visualize all the orders types", assumpting that no entity has already been created to represent the order types data.

Starting from backend, the first thing to do is to create the "OrderType" entity. The procedure, described for the creation of entity classes in Core and EntityFrameworkCore sub-projects (Sec. 3.3.3 and Sec. 3.3.5 respectively), could be followed. Next, since the OrderTypes table does not exist yet in the database, first the DbSet<OrderType> property must be declared in the DbContext class inside the EntityFrameworkCore project. Then a new migration, containing the code for the creation of the OrderTypes table, must be generated. Finally, the new migration must be applied to the database using the correspondent CLI command. The details of this sequence of procedures has been described in the EntityFramework sub-project (see Sec. 3.3.5).

Once the new table has been added in the database, a new application service class and the required DTO classes must be created, following the procedures explained in the Application and Application.Shared sub-projects, described in Sec. 3.3.1 and Sec. 3.3.2 respectively. Concerning the authorization, a new authorization string for the order types visualization must be created, then this must be assigned to the user's role who is allowed to visualize all the orders (in this case, both Operator and Collaborator roles). Next, in the application service class, the method which returns all the order types must be created and marked with the authorization string, previously created. To do this, the procedure, regarding the management of authorization described in Sec. 3.3.10, can be followed. Then, before moving to the frontend implementation, the unit tests for the new application service methods can now be added in the Tests project. Optionally, it is advisable to test them also by the means of Swagger UI to find and solve possible errors. If the new functionality works correctly in the backend side, the tests in the frontend are easier to be performed.

Concerning the frontend part, the first required step is to execute the refresh.bat file, that is inside the nswag folder of the Angular solution. This generates a new proxy class, named OrderTypesAppServiceProxy and added inside the service.proxies.ts class. The generated proxy class contains the method which can be used to call the new backend API created for the visualization of order types. Since the *Application Services as Controllers* approach is configured, this API corresponds to the new implemented application service method.

Then, a new component composed of the .ts, .html and .css files must be created to allow the orders types visualization. Next, a new routing item needs to be registered in

AppRoutingModule class (in app-routing-module.ts), specifying the path that renders the new component, when that path is inserted in the browser address line. Also, in this routing item, the authorization string required to access that path must be specified. This allows to prevent the access of unauthorized users, showing an error message. Consequently, a new menu-item labelled "Order types" must be added in the app-navigation-service.ts class, so that the order types component is loadable from the menu. The menu item must specify also the authorization string, which makes it invisible to unauthorized users. Finally, the procedures for the addition of the application feature are concluded with a final end-to-end testing which checks if the overall full stack implementation, frontend and backend, works properly.

Regarding the implementation of other features, the steps concerning the entity creation and the database migration, as well as the addition of a new menu item, may not be necessary, because done previously for other features. For instance, for the creation and edition of order types, these latter steps are not required, since already done for order types visualization.

Finally, the sequence of steps just described can be followed like an approximate guidance for the implementation of all the other features, taking in account, though, some exceptional cases. For instance, for the requirement concerning the loading of order types data through an excel file, it was necessary the manual implementation of a new controller class and a method which handles the serialized excel file data coming from the angular client. This was not possible to do directly using an application service method.

Let us now describe a possible interaction between the client angular and the ASP.NET Core backend server (namely, the Web.Host application).

Let us assume that a user wants to visualize all the orders actually present in the system. Inside the MainModule of the angular solution, a component containing the logic for orders visualization has be created. This component is composed of a table, showing the orders data, and a search button. When the user clicks the search button, a service method is called. The latter performs a call of a backend API, namely an HTTP request for a certain URL, to retrieve all the orders data. This service method receives in input a DTO that can use to specify filters in the request for the orders data.Then, if the *Application services as Controllers* configuration is enabled, this request is handled by the application service method which has been registered as the handler of that kind of request. By contrary, if that configuration is disabled, the request is handled by a method of a controller class, which however calls the same application service method. Next, the application service method receives in input the DTO with the filters and processes the request by calling the repository method, which has been created by EFCore for Order entity. This repository method performs a SQL query to retrieve the data from the database. In addition, the application service method can use the DTO data received in input to specify the filters to use, when it calls the repository method. The repository method returns to the application service method a list of Order entity objects. Afterwards, the application service method transform this list into a list of DTOs, which is then returned back serialized to the angular application.

Finally, the returned data arrives to the component, first introduced, which renders them in its view. The user can now visualize the orders in the table.

This section concludes the description of the application structure. In the next chapter, the methodology used for the development of the application will be discussed. Therefore general concepts of Agile software development process will be introduced, and particular attention will be directed to the Scrum framework, which has been adopted in the present work.

Chapter 4

The application development methodology

This chapter is dedicated to Agile process for project management. The essential idea of the Agile process is the iteration of the work chain from the discussion of the costumers requirements to their practical implementation as a team work. This allows on the one hand to meet the costumers requirements in an efficient and precise way, and on the other hand to facilitate the organization of the work to be carried by the team of developers. Among the practical realizations of the Agile process there is the Scrum framework. This is the framework which has been adopted in the present thesis work. And because of its importance it is described here in detail.

4.1 Agile methodologies

Agile is a process in which the management of a project (i.e. software project) is divided into several cyclic stages, where the development team implements the requested features keeping a constant collaboration with the stakeholders. The stakeholders of a project are the set of persons, companies or systems that are directly or indirectly interested in the project and who may influence or be influenced by the result of the project. In the case of the present work, the stakeholders correspond to the customer company that has requested the application, and in the future, also to other companies which may be interested in using this application, by starting new subscriptions. These entities coincide also with the end-users of the developed application. In general, an end-user is a person who uses the final application. He stays in contact with the software producer to which sends notifications in case of errors. Typically, an end-user can be the customers who pay for the software or a person who works for the customers, behaving as a mediator between these latter and the software producer.

The Agile process starts with customers describing to the project team the functionalities of the desired product and which kind of problems it must solve. In this way, the team knows exactly which are the customer's needs. Then, the development process can start and the team performs, in cyclic iterations, activities like planning, developing and testing in order to implement the product requirements. The key of Agile development methodology is the continuous collaboration between team members and between teams and stakeholders in order to perform the right decisions for the achievement of the product goal. Moreover, short iterations (typically from one to four weeks) of work are preferred, instead of long term ones, to allow fast production and constant reviews of the product.

The core of the Agile methodology was developed by a group of seventeen people in 2001 in a written form. Their Agile Manifesto of Software Development [15] shows the importance of delivering high-quality product and collaborating with customers. The Agile methodology is strongly based on the following main points:

- **individuals and interactions over processes and tools**: self organization and encouragement of team members are fundamental in Agile, as well as the interactions due to working in same location and pair programming.
- **working software over comprehensive documentation**: instead of relying on a complete documentation, showing a demo of the working software is considered as the best way to communicate with the customers and understand their needs.
- **customer collaboration over contract negotiation**: as it is not possible to collect all the requirements of a product before its implementation and because they may change with time, a continuous interaction with the costumers is fundamental to understand and implement the right requirements, instead of deciding initial strict contracts with all the needed requirements.
- **responding to change over following a plan**: because the customers tend to change their needs quite often, the Agile development is focused on quick responses to changes and therefore on a continuous development of the product.

Nowadays, there are several frameworks implementing the Agile principles, such as Scrum, Kanban, Extreme Programming (XP), and Adaptive Project Framework (APF).

To summarize, the Agile methodology provides a way of managing the development process of a product by the means of short iterations and frequent collaborations with the customers, in order to fulfill their needs and increment the product quality. Cross-functional and self-organized teams are established, whose job is to select, implement and test the product features in the next agile iterations. No comprehensive documentations containing the requirements and design specifications are written by the team, but only quick and clear information needed to start working as soon as possible.

In this thesis, the Agile methodology has been used for the realization of the web application and in particular the Scrum framework has been adopted. Such framework is the topic of the next section.

4.2 The Scrum framework

Following the definition of Scrum given by the its creators in the The Scrum Guide [16], Scrum is a framework that encourages individuals, teams and businesses to build value creating adaptive solutions for complex problems.

Scrum requires a Scrum Master to set up an environment where

- a Product Owner organizes into a Product Backlog all the features to be implemented for a product,
- the Development Team selects the set of features and implement them during a Sprint,
- the Scrum Team and the Stakeholders inspect the results and take decisions for the next Sprint.

These points are then iterated.

The Scrum framework has been intentionally made incomplete, it only describes the basic elements necessary to implement the Scrum theory. The scrum rules have been thought to guide the relationship and interactions between the people and not to provide them detailed instructions about how to implement Scrum. Indeed, there are several possible implementations of Scrum, but each of them, in order to be Scrum conform, must follow its basic principles.

The founding pillars of the Scrum framework are essentially three, namely

- **transparency**: the software development process and the work must be clear to those doing the work as well as to those receiving the work.
- **inspection**: it has to be done frequently and attentively on artifacts and product goal to detect undesirable changes or issues which may bring to a deviation with respect to the product goal.
- **adaptation**: performed to meet the goals of the costumer. The process applied to a product, as well as the materials produced by it, must be adjusted any time the process deviates outside some acceptable limits or the resulting product is not acceptable. In this way the risk for further deviations is reduced.

Transparency enables inspection, which is performed on Scrum artifacts. These correspond to the work or the value of a product and are produced by the Scrum events. The inspection brings to light possible issues. Inspection then leads to adaptation, in order to meet the customer's needs. These three points are clearly strongly connected, transparency being the most important since without transparency inspection is misleading and without inspection there is no adaptation. Low transparency on artifacts, instead, leads to decisions which decreases the value of the work (i.e. lower priorities features are delivered before higher ones) and at the same time increases the risk of not being aligned to the customer's needs.

The key elements of Scrum are the **Scrum team**, the **Scrum events** and the **Scrum artifacts**. The scrum team is explained in the next section whereas the events and their artifacts are described in Sec. 4.2.2.

4.2.1 The Scrum team

The very fundamental unit of Scrum is the **Scrum Team**, composed of a Product Owner, a Scrum Master and the team members (namely the developers). Within a Scrum Team, there are no sub-teams or hierarchies, but all people involved are collaborators in order to achieve the customer's product goal.

The **Product Owner** controls the priority order of items in the *Product backlog*, directing team toward most valuable work and working closely with the stakeholders in order to deliver the highest business value. The Product backlog is one of the Scrum artifacts, property of the Product owner, which is composed of a set of items that represent the software product requirements. The Product backlog is in continuous changing and can be managed using planning software, spreadsheets, or even a wall of post-its. The technique used to manage the Product backlog is a free choice, as well as the format of its items. The product backlog items are usually defined using the *user stories* format. A user story is an essential description of a functional requirements. A typical example of its format is

```
As a <actor type>
I want <to do something>
So that some value is created.
```

For instance, in practice a story could be

```
As a unregistered user
I want to create an account
So that I can create orders.
```

For each story, the product owner writes some acceptance criteria, which can be easily transformed in automated tests. It is also important to define when a story can be considered as done. Typically, the definition of done for a user story consists on the code being added, then tested with unit tests and finally tested with end-to-end testing.

Moreover, the Product owner makes sure that the needs of the customer and end-users are understood by the team, by showing the requirements for a software product and being always available to answer team members' questions. He can be considered as the protector of the product vision, in other words he represents the customer. So, compared to the other Scrum team members, he better knows who is the product build for, why the customer needs it and how the customer will use it.

The **Scrum Master** is the member of the scrum team who is responsible for helping everyone (Scrum team and organization) understand Scrum theory and practice. His main objective is to produce a self-organizing and cross-functional team. He adapts to the team experience and, as the team becomes self-managing, his work is completed. When it is necessary, he removes the obstacles for the team which can be external (helps to solve hardware issues) or internal (helps the team to understand the problem and encourages to find a solution).

The **Team members** are the developers who are responsible for the implementation of the customer's requirement for a software product. They form a self-organizing team for the decision of the tools and techniques to be used, and the assignment of tasks. The team members estimate together the effort needed for the implementation of all the required features and the time needed to complete them. Team sizes typically span between five to nine members with an adequate variety of skills, and each team member collaborates to achieve the product goal.

4.2.2 The Scrum process

The scrum formal events making the working process are sprint planning, daily scrum, sprint review and sprint retrospective. They are represented by the diagram in Fig. 4.1.

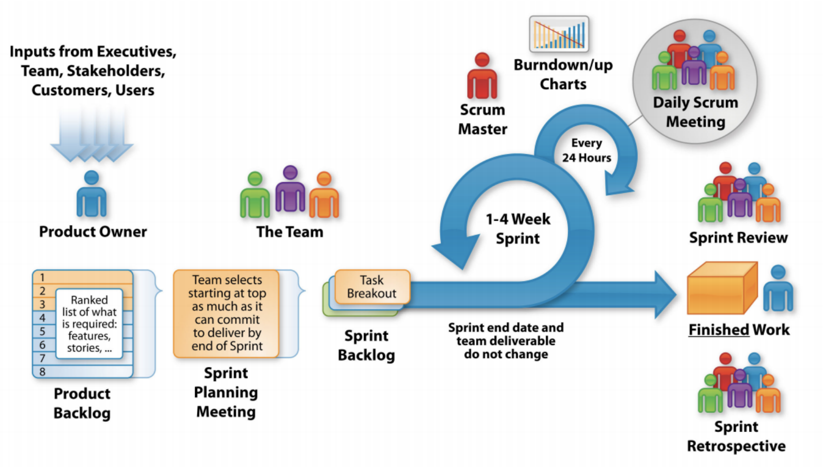

Inputs from Executives, Team, Stakeholders, Customers, Users
Product Owner
The Team
Scrum Master
Burndown/up Charts
Daily Scrum Meeting
Every 24 Hours
1-4 Week Sprint
Sprint Review
1
2
3
4
5
6
7
8
Ranked list of what is required: features, stories, ...
Product Backlog
Team selects starting at top as much as it can commit to deliver by end of Sprint
Sprint Planning Meeting
Task Breakout
Sprint Backlog
Sprint end date and team deliverable do not change
Finished Work
Sprint Retrospective

Figure 4.1: Representation of the scrum process (Ref: "Analysis of User Stories and Effort Estimations in Agile Software Development", written by Rupert Dürre).

As it can be seen in the diagram, the scrum iteration begins with the product owner receiving inputs from the team, the stakeholders, the customers and the application users. Then he updates the Product backlog, ranking all the user stories in descending order, namely places on the top the ones with higher priorities.

In the **Sprint Planning** the team decides the set of stories to commit for the next sprint, choosing from the product backlog. The team then splits up each of these stories into smaller tasks, each task requiring as maximum effort about half a day. The definition of done of a task can be the same of that seen for a user story. The **Sprint** is the basic iteration in the Scrum process, which produces a part of working software to be showed and examined at its end. Typically the length of the Scrum sprints should be from one to four weeks.

During the Sprint Planning event, the **Sprint Backlog** occurs, where the team creates the Sprint Backlog artifact for the next sprint. The Sprint Backlog artifact is property of the team and is composed of the list of committed stories and related tasks, needed to implement them. The list of tasks may change during the sprint, while modifications on the list of committed stories are blocked until the end of the sprint. In the Spring Backlog also some additional tasks are put by the team such as those for team improvement, research work, performance and security requirements and bug fixing. In addition, the team performs some estimations regarding the overall number of tasks, the tasks hours and the tasks difficulty points. These estimations could be done by the means of numerous techniques and they are used basically for organizing the work. After the creation of the Sprint Backlog, the sprint iteration could start and each team member can begin to work on the assigned tasks. The tasks are assigned internally by the team members, neither the product owner or scrum master are allowed to assign tasks.

During the sprint iteration, the team members must organize **Daily Scrum** meetings. These are hold every day and are brief (maximum 15 minutes long), where only the team members participate. Each team member shows which tasks have been done, what remains and the obstacles possibly encountered.

At the end of the sprint iteration the **Spring Review** event occurs where there is the demonstration of some parts of working software to the stakeholders. The team members receive some feedback on the implemented features and must write reports on the incomplete stories and record the reactions of the stakeholders. The team must be total transparent in this phase, since this meeting is also the base for future decisions made by the Product Owner. According to the work done by the team and customer's needs, the Product Owner adjusts the items of the Product Backlog for the next sprint. No sprint planning is performed in this phase.

After the Sprint review event, the team must write a **Sprint Retrospective**. This is considered the last event of the sprint. The team, the product owner and the scrum master participate in this phase. The team focuses on the lessons learned in the sprint, collecting some data in order to understand what happened and generates insights, trying to find the causes of the problems encountered. Then, for the next sprint, it identifies one or two things to improve and the correspondent actions to be performed.

Then the scrum process restarts again with the Sprint Planning phase for the next sprint. The team selects the other user stories to commit for the next sprint and so on. It could happen that, at the beginning of each sprint, the product owner adds new user stories based on the results of the previous sprints or the changing of the customer needs.

4.3 The organization of the present work

In the present thesis experience, I worked in a team where there was one product owner (who behaved sometimes also as a scrum master) and a team of developers. The product owner was in charge of managing the product backlog which contained the software products requirements and issues. No particular formats, such as the user stories described above, were used in order to indicate the requirements, but rather a free representation was adopted. For the sprint planning we used the "Boards" and "Issues" section of the GitLab platform which will be explained in detail in Sec. 6.2.3.

Inside the scrum team my role varied from that of a Product Owner to that of a full stack developer. I participated to several meetings with the customer as it was allowed for students working on their thesis. Therefore I experimented directly how a Product Owner works in order to understand the customer's product goal and to direct the work of the team. In each of these meetings, the new requirements and some eventual changes for the old requirements were identified and put in the product backlog ordered by the priority.

As it was explained in Chap. 2, the application developed in this thesis is one of the modules of a bigger application. I was responsible to work in my module only, while the other members of my team worked in the other modules. Often we discussed on common problems and design choices, in order to improve our knowledge and skills, increasing also the code quality.

Then, in accordance with the other team members, the features to deliver for the sprint were decided, being guided also by the product owner to deliver first the ones with higher priorities. I split the requirements into self-assigned tasks. For each task, I studied the knowledge and techniques necessary to complete it. The definition of done used for the requirements consisted in code developed and published in Git, unit testing and end-to-end testing.

Each sprint lasted two weeks more or less and daily scrum meetings were setup every day in the morning, where I had to update the other team members about the state of my work, notifying eventual issues or impediments. Each sprint was concluded with the final sprint review where the application with the new features was presented in a meeting, where the team members, product owner and customer participated. Then, the new sprint was planned adding the eventual new issues to solve in the product backlog and choosing the other requirements to be delivered in the next sprint.

In the next chapter, general concepts on software testing and the testing approach adopted in the present work will be covered in detail.

Chapter 5

Testing the application

The realization of a business application comprises also the testing procedures. In the software development process, the testing activities are very important and fundamental to check if the developed software meets the customer's needs and also to identify all the software issues earlier in order to produce a quality product.

An issue can be a defect or a bug. A defect in a software occurs when the software works but it does not implement the customer's requirements. By contrary, a bug is a result of a code fault, which happens when a software feature generates a failure. The latter represents an expected behaviour in the execution of the software and may occur either if the software presents errors or the requirements are incomplete or wrong.

In the next section, some basic concepts of the software testing are discussed in order to introduce the agile testing approach, which has been adopted in this work.

5.1 Software testing

The software testing is the process used to check if the developed software satisfies the defined business requirements and to identify its possible issues. The software testing aims to produce quality products aligned to the customer's needs. Moreover, according to the adopted software development methodology, this process can be done either during the development phase or at the end of it.

In general, there exist two types of software testing: the *Static Testing* and the *Dynamic Testing*.

The Static Testing, also known as *Verification*, is the process used to check if the work products of a software satisfy the defined requirements. A work product is any important deliverable (i.e. document, source code) created during a phase of the software development process (planning, design, development and testing). Some examples of work products are requirements and design specifications, architectural models, user interface prototypes, user documentation, source code, test cases and test scripts. The static testing guarantees that a developed software has been implemented according to the requirements and design specifications, defined through the collaboration with the stakeholders. In Static Testing there is no execution of code, but only analysis on the work products are conducted in order to find any kind of defects. The static testing process consists of activities such as reviews, walk-throughs and inspection.

Briefly, a review (or technical review) is an event, where a work product is inspected for defects by a team of colleagues of the person who produced it. For instance, in a review, the team tries to find any discrepancies between the requirements of an application and the design specifications decided for that application.

The inspection event is a group review with more formality. Indeed, well specified roles are assigned to the group members. Inside a group there are producers, moderators, reviewers and recorders. The inspection process is ordered and systematic. For example, the reviewers first participate to an orientation meeting, after which they individually inspect the work product searching for defects. Then, a moderator holds an inspection meeting where all the team members are present. During the meeting, participants first decide together a checklist, which then use to review the work product, by examining one section at a time. Defects are recorded here and the final version of the product work is established. If a rework for the product is needed, other inspections can be carried on to check the new changes.

In the walk-through event, a meeting is held by the producer of a work product, who explains the product to his team. The participants of the meeting can ask questions (if there are any) and a team member makes note of the review comments.

Dynamic Testing, also known as *Validation*, is the process through which it is possible to verify if a developed software meets all the customer's needs. In addition, this testing can be used to check if a software satisfies all the customer's requirements when deployed on a particular environment (for instance, the customer's environment). The validation process comprises the activities like unit testing, integration testing, system testing and user acceptance testing.

The unit tests are performed by developers on the units of source code (i.e. modules, classes), developed by them. This testing checks if single units of source code work correctly and, therefore, are ready to be used. In addition, it helps to reduce the cost of bug fixing, since bugs are identified during the early phases of the software development life-cycle. However, not all defects and bugs can be discovered by the unit testing.

In the integration tests, different and already tested modules are combined and tested together. For instance, integration testing verifies the correctness of the data flow between different modules of a software. In the traditional testing methodology (i.e. Waterfall), this type of testing is performed by the team of testers, whereas, as it will be explained in the next section, in Agile testing this is done by all the agile team. Furthermore, successful integration testing ensures the possibility to perform the following system testing.

The system testing is performed on all the integrated software system. It allows to check if the software system satisfies all the customer's requirements. It tests the overall interaction of the software modules and it allows the testing of non-functional requirements such as load, performance, reliability and security.

The user acceptance testing is performed by the customers to verify and accept the software before deploying it to the customer's environment (called also production). This test is done at the end of the testing process, after unit testing, integration testing and system testing. The user acceptance testing is carried on by deploying the software in a separate testing environment (called also pre-production or staging), which has the same data configuration of the production environment. The user acceptance tests help to discover compatibility issues with other systems present in the customer's environment and to uncover load and performance defects which may occur during the execution of the application in that environment. The user acceptance tests can also be automated.

Moreover, there is also *end-to-end* testing, that can be performed once the system testing has terminated successfully. This type of testing is done by the development team or tester team (in case of traditional software development process) and verifies the correctness of the whole application workflow. For instance, with end-to-end testing it is possible to check if the interactions between the frontend and backend part of an application happen smoothly. In this way, the whole data flow from the graphical user interface until the access to the database can be tested, uncovering possible issues. End-to-end testing can be carried on either in a development environment, every time a new feature is implemented, or in a pre-production environment. It is advisable to perform end-to-end testing in both of these cases. Moreover, successful end-to-end testing ease the user acceptance testing. Well done and continuous end-to-end testing makes the user acceptance testing just as a double check to control if everything works properly. The end-to-end testing can be automated or manually. It is preferred the second option, because it is easier to be realized.

Many approaches are available for software testing, depending on the software development methodology adopted. In the next section, the Agile testing and how it has been implemented in the present work are presented.

5.2 The Agile testing

Agile testing is a software testing practice that follows the principles of Agile software development methodology. As explained in the Chap. 4, Agile is an iterative development methodology in which software requirements are decided by the means of cooperation between self-organizing teams and customers, aligning the development of the software to the needs of the customers.

To understand how agile testing works, it is fundamental to know which are its basic principles and benefits.

In agile testing, the team performs *continuous testing* in each iteration, which is the only way to guarantee the continuous development of a product. Since the agile iterations are really short (typical from one to four weeks), *continuous feedback* is provided from the customers, helping to solve problems and to fulfill all their needs. The combination of continuous testing and continuous feedback saves time and money, because all the problems of the software are discovered in early stages. In addition, they make shorter the time of customer's feedback response.

With continuous feedbacks from the customers, agile testing is flexible and highly adaptable to every changes on the software. As soon as the customer tests the software and provides feedbacks, the improvements and testing on these improvements could be performed.

Agile methodology bases on *test driven* concept, namely testing is done at the same time of development. While, in the traditional software development process (i.e. Waterfall), the testing is performed after the development phase. Furthermore, unlike the traditional software process, where only the test team is responsible for testing, in agile also the developers and business analysts (i.e. the Product Owner in Scrum) must test the application. This allows to keep a simplified and clean code. All defects which are uncovered by the agile development team are fixed inside the same agile iteration.

Then, the agile testing allows the presence of *less documentation*, since the team should concentrate more on developing and testing without losing too much time on writing comprehensive documentations about the tests it wants to perform. The development team focuses only on checklists which need to be tested, not on writing in detail the possible issues that can be detected.

Lastly, through the daily meetings, the Agile testing allows to have a better determination of issues.

Furthermore, there exist different Agile testing methods such as *Behavior Driven Development* (BDD), *Acceptance Test Driven Development* (ATDD) and *Exploratory*

Testing. Starting from Behavior Driven Development, it improves communication between agile team and stakeholders so that all features are clear to everybody before the development process starts. Moreover, a continuous communication based on examples is kept between developers, testers, and business analysts. The examples are called *Scenarios*, which are written in a special format and contains some information used to describe how a software feature should act in different situations with different input parameters. These information are called *executable specifications*, since they are composed of both specifications and inputs passed to these automated tests.

Acceptance Test Driven Development (ATDD) bases on meetings between the team members who work impersonating different point of views, such as the customer, the developer and the tester. In these meetings acceptance tests including the different perspectives are created. The customer is focused on the problem that has to be solved, the developer is focused on how the problem will be solved and the tester is focused on the possible issues that can be encountered. The acceptance tests represent the user's point of view and describe how the software system will work, checking if it works as supposed. In some cases acceptance tests are built automated.

Lastly in Exploratory Testing, the design and execution phases of testing are done together. This testing accentuates the importance of agile principles, such as working on software over comprehensive documentation, individuals and interactions over processes and tools, and customer collaboration over contract negotiation. These principles, described in detail in Chap. 4, make the exploratory testing more adaptable to changes. The explanatory testing allows to identify the functionalities of an application by exploring the application. More precisely, the testers try to first learn the application, then proceed with designing and executing the tests according to their discoveries.

Concerning the software verification and validation, Agile does not provide an official and standard way to implement them. If Scrum version is used, the software verification can be done by the means of the scrum events and their artifacts (Product backlog and Sprint backlog). For instance, during each sprint iteration, the team members can inspect the Sprint backlog or the Sprint review documents, in order to check if the implemented requirements correspond to the defined ones. While, the Product owner can perform verification activities either on products produced in the sprint reviews or through the meetings with the customer, which occur before the beginning of each sprint iteration.

Instead, the software validation can be referred, in the Scrum approach, to the *definition of done* (DoD) concept, whose implementation is a free choice of the development team. A Product backlog item is considered as done when it meets some requirements which, for instance, can be: code added in Version control source (i.e. Git), unit tests passing and end-to-end tests passing. Further tests, which can be done, are integration tests, system tests and user-acceptance tests. Concerning these latter, it is a free choice of the development team to decide whether and how to do them.

5.3 The applied testing procedure

In the present work the Scrum implementation of Agile has been adopted, where cross-functional teams were composed and all team members participated in testing. Particular attention have been put on the unit testing, on the end-to-end testing and on the user acceptance testing. The unit-testing was performed on the application services methods (backend testing), which as explained are those implementing the business logic. Instead, end-to-end testing was done each time a new feature was implemented and on the whole implemented application features, including both frontend and backend.

In particular, as concerns the implementation of the unit tests, the ASP.NET Core solution of ASP.NET Zero provides an entire sub-project (called Tests) dedicated to unit testing and integration testing, which has been explained in detail in Sec. 3.3.9. As introduced in that section, the *AppTestBase* class, provided by ASP.NET Zero solution, allows also to add initial seed data in the testing database. In the case of the present work, the seed data has been both loaded using Excel files and in-memory data structures, while implementing the unit tests. This approach of using seed data helped to write lighter and smarter unit tests code. For instance, let us consider the unit test method which tests the service method implementing the creation of a revenue functionality. In the test method, in order to check the creation of a revenue, an order object is needed to be created first, because a revenue needs to be referred to an order identifier, otherwise it can not be created. Therefore, instead of creating the order inside this test method, an already created one can be used. Indeed, an efficient way to achieve this is by inserting this order data in the initial seed data, which is applied to the testing database when the test method is run.

Regarding the user acceptance tests, they are not automated in the present work. In order to check if the application worked properly also in the production environment, at the end of each Scrum sprint, the application was deployed in a pre-production environment, for which less hardware resources and a copy of the production database (containing the customer's data) were exploited. In this phase, the customer could test the application manually, using his real data. After that, the team organized a meeting with the customer who provided some feedbacks about the features delivered in the sprint. In that phase the customer could accept or not the new features, asking sometimes for further improvements and rising bugs which had not been detected in the development phase. The possible bug fixes and features improvements were inserted in the product backlog and performed in the next sprint iterations. One must say, however, that this approach of deploying the application in a pre-production environment is not mandatory but it is a good practice. Also, the development team wanted to assure the software quality and fulfillment of the software requirements before bringing it into production.

Furthermore, it was noticed that the procedure of performing both development and testing in the same sprint iteration helped to solve issues as soon as they were discovered. Instead, the frequent software deliveries resulted quite useful to fully understand the customer's product goal. Sometimes, more meetings with the customer were organized both at the beginning and at the end of the sprint iterations.

In conclusion, all the features requested by the customers have been developed and accepted. The next step is the deployment of the application in the production environment. This is explained in the next chapter where the DevOps toolchain general concepts and the one used in this thesis work are covered as well.

Chapter 6

Deployment of the application

The final step of the realization of a web application is the deployment process, namely the final application is made public and available to the user for usage. This chapter is dedicated to the deployment process.

As already explained in Chap. 3, the application developed in this work is composed of two solutions, namely the Angular solution and the ASP.NET Core solution. The angular solution contains the angular client application which retrieves the data to be rendered by interacting with the Web.Host API application, that is the application inside the ASP.NET Core solution used to expose the APIs to the angular client application.

These two applications have been used only locally till now. In order to make them being used also through the Internet it is necessary to deploy them somewhere. There are many web hosting services available online that can be used for this purpose. Among them, the Microsoft Azure service has been chosen. The Microsoft Azure platform [17] provides the activation of different type of app services such as web apps, mobile apps and many others.

Section 6.1 is dedicated to the different stages needed for the deployment of the application on the Microsoft Azure platform.

Afterwards, the DevOps toolchain general concepts and the toolchain adopted in this project are explained in Sec. 6.2.

6.1 Deployment on the Azure platform

It is possible to publish the angular client and the Web.Host API applications together or separately. In the present work, they have been deployed separately. For this purpose the following Azure services have been created:

- a Web App used to host the Web.Host API application,
- a Web App used to host the Angular client application,
- a SQL Database, which provides a server with a database to handle the customer's data.

An Azure web app service is a service based on HTTP used for hosting web applications and mobile applications. Publishing a web application into an Azure web app resource means that the application is hosted inside a web server. A web server is a container which hosts web applications and makes them accessible to the users from a base address (URL). The type of web server related to an Azure web app resource depends on the operating system chosen in the creation process of that web app resource, which will be described later. The web server contains each web application inside a folder, and all these folders are contained in a root folder. In the case of the IIS web server (used by Windows OS) the root folder is called "wwwroot".

In general, the deployment of a web application consists in compressing its code in a special format (i.e. .zip, .jar and .war) and uploading it in the root folder of a web server container. The latter extracts the content of the uploaded file and places it in a new folder. Finally, it serves the application to a particular URL, which depends on the type of web server being used.

Instead, the Azure SQL Database is a platform as a service (PaaS) database engine that handles database management operations such as monitoring, patches and backups, without the need of human intervention. It is based on the latest version of the Microsoft SQL Server database.

Moreover, since it was thought reasonable and convenient to have two deployment environments, one for staging and one for production, these resources have been created for both of the environments. Therefore, the application is first deployed in the staging environment, where it is used by both customer and developers for testing purposes. Then, if everything works correctly, the application is deployed in the production environment, where the customer uses it and the IT operators team monitor its performance.

Let us now explain the steps required to deploy the application in the production environment. In the next sections the creation of the needed Azure resources is described, followed by the steps which must be followed to deploy the backend and frontend applications in these resources. The same steps can be followed for the staging environment too, with some little variations (i.e. the database connection string is different for the staging environment).

6.1.1 Creation of the Azure resources

In order to create an Azure Website for the Web.Host API application, the "Web App" app service of Azure has been created, which is accessible from the Azure dashboard, shown in Fig. 6.1.

After clicking the "Web App", a new panel opens in which some information are required, as shown in Fig. 6.2. The required information are the application name, the adopted azure subscription, the resource group and the app service plan.

The application name is simply the name of the web app service where the Web.Host API application is deployed. In this work the name "compliance-in-cloud-api" has been chosen. In addition Azure assigns the suffix "azure.websites.net" to the new web app names.

The resource group, instead, is a kind of container that contains more Azure services. In the present work, a new resource group has been created and associated to the three created resources introduced in Sec. 6.1.

The Azure subscription is a logical container used to deploy and consume resources in Azure. It holds the details of all resources like web apps, databases, virtual machines and so on. Depending on the type, these can be free subscriptions, Pay-As-You-Go subscription and others.

An App Service plan configures a collection of measured resources which can be used by a web app to run. One or more web apps can be configured to run in the same App Service plan. For the web app resource of this work, the subscription and app service plan, used by the company where the internship was carried on, have been chosen. In addition, as shown in Fig. 6.2, it is mandatory to set also the runtime stack and operating system. The first refers to the technology stack used to develop the web app, while the operating system corresponds to the web app hosting platform. In this regard, the framework .NET Core 3.1 (LTS) and Windows OS have been chosen.

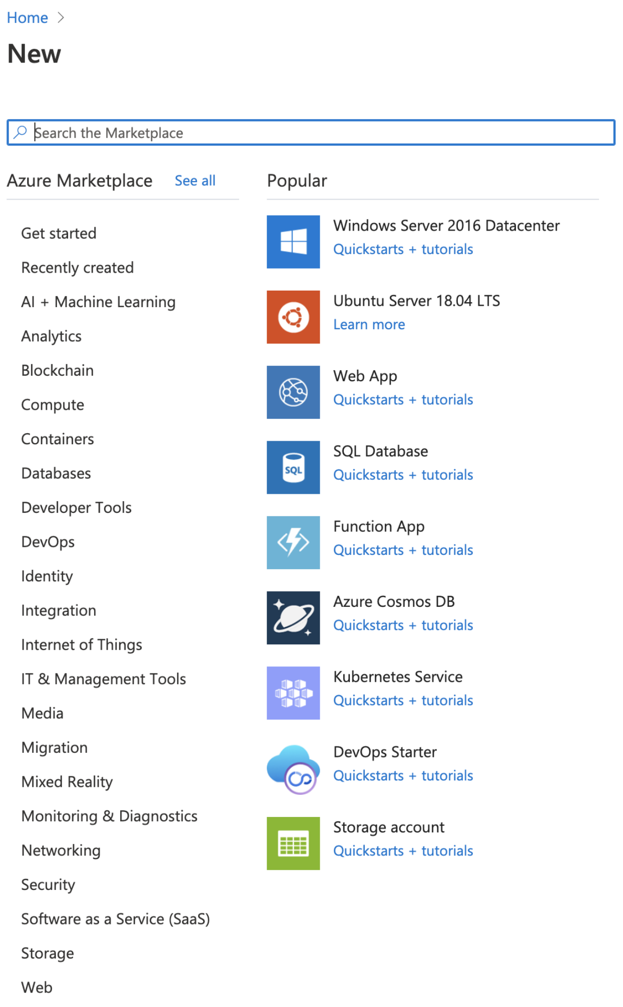
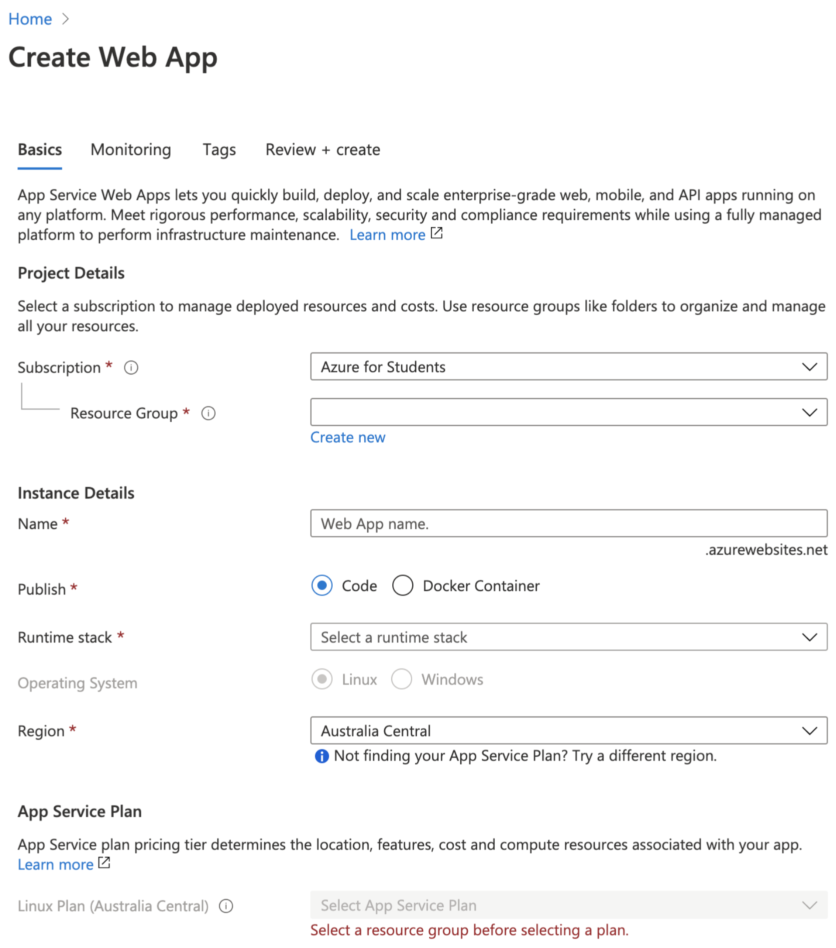


Figure 6.1: The Azure resources.

In order to store the customers data, a SQL database is needed. Therefore, from the same window shown in Fig. 6.1 it is possible to create a SQL Database resource. After clicking the button "SQL Database", a new panel opens (see Fig. 6.3) asking to insert some information like the database name ("ComplianceInCloudDB" has been chosen) and information for the creation of a new server used to allow connections to the database. The latter regards the server name ("complianceincloud.database.windows.net" has been chosen), credentials for login and localization information (i.e. Western Europe). Also here the information concerning the app service plan, the resource group and the azure subscription are requested. These can be the same of those chosen for the web app resource created for the deployment of Web.Host API application.

Furthermore, in order to create an Azure Website for Angular, the procedure is the same of the one followed for the Web.Host API application. Therefore, from the page shown in Fig. 6.1, a new web app service must be created specifying the application name (i.e. "compliance-in-cloud"), the resource group, the azure subscription and the app service plan, which can be the same of those used for the other previous resources.


Home >

Create Web App

Basics Monitoring Tags Review + create

App Service Web Apps lets you quickly build, deploy, and scale enterprise-grade web, mobile, and API apps running on any platform. Meet rigorous performance, scalability, security and compliance requirements while using a fully managed platform to perform infrastructure maintenance. Learn more

Project Details

Select a subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

Subscription *	Azure for Students
Resource Group *	Create new

Instance Details

Name *	Web App name. .azurewebsites.net
Publish *	[✓ Code] Docker Container
Runtime stack *	Select a runtime stack
Operating System	[✓ Linux] Windows
Region *	Australia Central

Not finding your App Service Plan? Try a different region.

App Service Plan

App Service plan pricing tier determines the location, features, cost and compute resources associated with your app. Learn more

Linux Plan (Australia Central)	Select App Service Plan

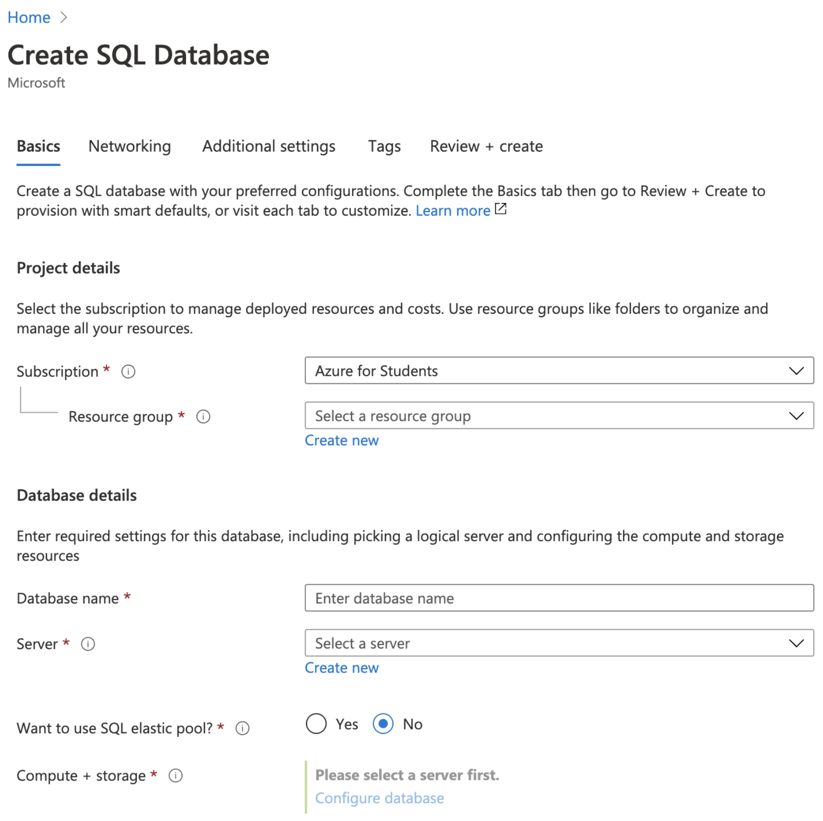
Select a resource group before selecting a plan.

Figure 6.2: Web App resource creation.

6.1.2 Publication of the Web.Host application

To publish the Web.Host application in the azure platform, the first thing to do is to apply the local database migrations to the new SQL database resource created in the Azure platform. In order to achieve this, some initial configurations are needed. First, the connection string in the *appsettings.json* file of Web.Host project must be set as shown in the Fig. 6.4. With this connection string, the EntityFrameworkCore client application can connect to the new SQL database on Azure. Then, the IP of the EFCore client must be inserted in the range of allowed IPs of the SQL database server. For this purpose, it is possible to configure the firewall for client access or using the management studio program (in Windows OS) or setting the range of allowed client IPs directly in the Azure platform (inside a section of the SQL database resource). Finally, the database migrations can be applied to the SQL database resource, following the procedures explained in Sec. 3.3.5.

Afterwards, in order to publish the Web.Host API application into the new web app resource created on Azure, it is necessary to perform some changes to the *appsettings.production.json* file, located inside the Web.Host project. As it can be seen in Fig. 6.5, the connection string is substituted with the new connection string to connect to the new SQL database server. In addition, in the "App" section, the root address for the backend server and the angular client are specified. These correspond to the URLs where the two Azure web apps, created before, are accessible.



Home >

Create SQL Database

Microsoft

Basics | Networking | Additional settings | Tags | Review + create

Create a SQL database with your preferred configurations. Complete the Basics tab then go to Review + Create to provision with smart defaults, or visit each tab to customize. Learn more

Project details

Select the subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

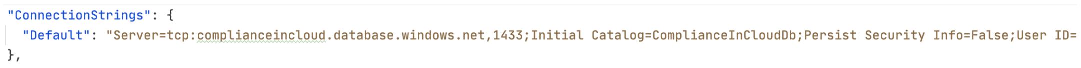
Subscription *	Azure for Students
Resource group *	Select a resource group Create new

Database details

Enter required settings for this database, including picking a logical server and configuring the compute and storage resources

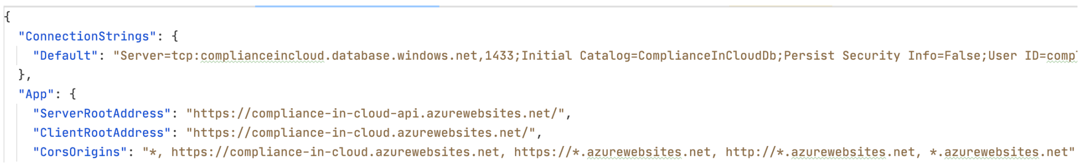
Database name *	Enter database name
Server *	Select a server Create new
Want to use SQL elastic pool? *	Yes [✓ No]
Compute + storage *	Please select a server first. Configure database

Figure 6.3: SQL database resource creation.



```
"ConnectionStrings": {
  "Default": "Server=tcp:complianceincloud.database.windows.net,1433;Initial Catalog=ComplianceInCloudDb;Persist Security Info=False;User ID=
},
```

Figure 6.4: The app-settings json file.



```
{
  "ConnectionStrings": {
    "Default": "Server=tcp:complianceincloud.database.windows.net,1433;Initial Catalog=ComplianceInCloudDb;Persist Security Info=False;User ID=comp
  },
  "App": {
    "ServerRootAddress": "https://compliance-in-cloud-api.azurewebsites.net/",
    "ClientRootAddress": "https://compliance-in-cloud.azurewebsites.net/",
    "CorsOrigins": "*, https://compliance-in-cloud.azurewebsites.net, https://*.azurewebsites.net, http://*.azurewebsites.net, *.azurewebsites.net"
```

Figure 6.5: Configuration of the appsettings.production.json file.

Finally, the Web.Host application is published on Azure. If using Visual Studio IDE, by right clicking the Web.Host project and then "Publish", a new panel is opened asking where to publish the application. Therefore, after a first configuration, the "compliance-in-cloud-api" option must be chosen, which is the name of the web app previously created in Azure, to host the Web.Host application. Finally after clicking the "Publish" button and waiting some minutes, the Web.Host application is deployed on the root address:

`https://compliance-in-cloud-api.azurewebsites.net/`

as configured in the "ServerRootAddress" section of *appsettings.production.json*, shown in Fig. 6.5.

6.1.3 Publication of the Angular application

At this point it only remains to publish the Angular application to the corresponding Azure web app resource. The procedure to achieve this is the following.

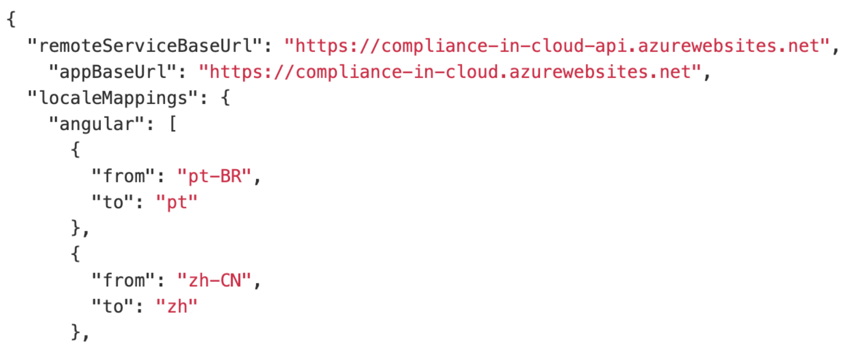
The process starts preparing the publish folder inside the angular solution. First, the yarn command is executed to restore all the angular packages. Then, the publish folder, called dist, is created by running the command (in a terminal open inside the angular solution folder):

```
npm run publish
```

Afterwards it is necessary to copy the *web.config* file, positioned in the angular solution folder, into the new generated "angular/dist" folder. This is necessary, because the angular application, being a Single Page Application, uses the client-side routing mechanism and does not need a backend server to dynamically compose application pages. Therefore, what happens is that if the Azure web server, where the angular application is hosted, receives requests for files (URLs) that does not have, it returns an error message.

Since this is not the desired behavior when using a Single Page Application, the Azure web server must be configured to return the application host page (index.html), when receives requests for not existing files. The web server in question is a IIS (Internet Services Information) one, since, as previously explained, the angular application is hosted in an environment comprising the Windows OS and .NET Core 3.1 (LTS) framework installed. This type of web server needs a configuration file, like the one proposed in web.config, to add the rule for redirection to the application homepage and to not throw error messages when receiving requests for not existing files. And this is the reason why the web.config file must be copied inside angular/dist folder, which, as it is explained later, must be deployed in the corresponding Azure web server.

Next, the file *appconfig.production.json*, located inside the "angular/dist/assets/" folder, must be configured as shown in Fig. 6.6. This is needed to specify to the client angular essentially two things: its root address, from which it is accessible by the users, and the root address for connections with backend APIs. The latter is the root address of the Web.Host API application.

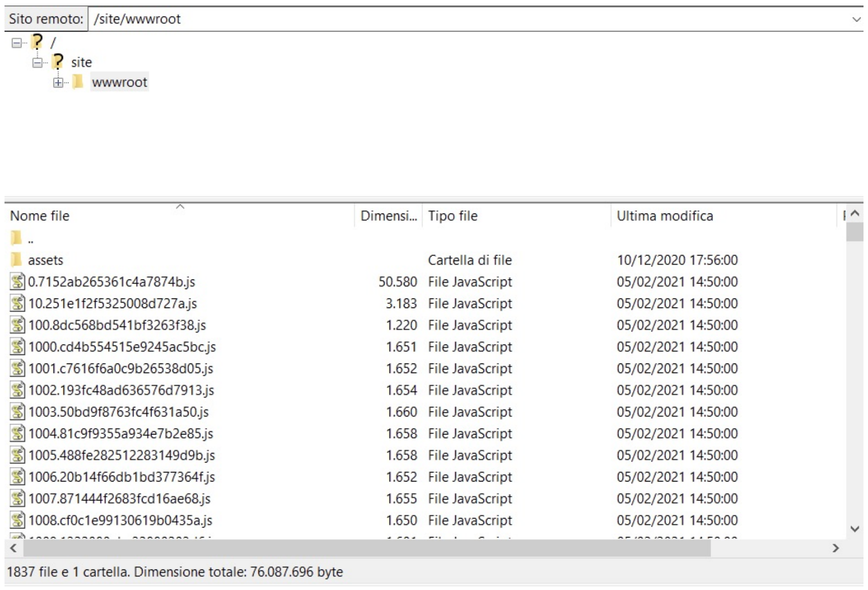


```
{
  "remoteServiceBaseUrl": "https://compliance-in-cloud-api.azurewebsites.net",
    "appBaseUrl": "https://compliance-in-cloud.azurewebsites.net",
  "localeMappings": {
    "angular": [
      {
        "from": "pt-BR",
        "to": "pt"
      },
      {
        "from": "zh-CN",
        "to": "zh"
      },
```

Figure 6.6: Angular appconfig.production.json file configuration.

Finally, the dist folder must be uploaded into the Azure web app resource created for the angular application. This has been done sending FTP requests through the FTP client FileZilla. These FTP requests transfer files from the dist folder to the *wwwroot* folder, which is the folder of the IIS web server where all the web applications are hosted. The Fig. 6.7 shows the structure of the wwwroot folder. In this way, the angular application is hosted inside the IIS web server and is accessible by the users from the address:

`https://compliance-in-cloud.azurewebsites.net`



Sito remoto: /site/wwwroot

```
/
  site
    wwwroot
```

Nome file	Dimensi...	Tipo file	Ultima modifica
..			
assets		Cartella di file	10/12/2020 17:56:00
0.7152ab265361c4a7874b.js	50.580	File JavaScript	05/02/2021 14:50:00
10.251e1f2f5325008d727a.js	3.183	File JavaScript	05/02/2021 14:50:00
100.8dc568bd541bf3263f38.js	1.220	File JavaScript	05/02/2021 14:50:00
1000.cd4b554515e9245ac5bc.js	1.651	File JavaScript	05/02/2021 14:50:00
1001.c7616f6a0c9b26538d05.js	1.652	File JavaScript	05/02/2021 14:50:00
1002.193fc48ad636576d7913.js	1.654	File JavaScript	05/02/2021 14:50:00
1003.50bd9f8763fc4f631a50.js	1.660	File JavaScript	05/02/2021 14:50:00
1004.81c9f9355a934e7b2e85.js	1.658	File JavaScript	05/02/2021 14:50:00
1005.488fe282512283149d9b.js	1.658	File JavaScript	05/02/2021 14:50:00
1006.20b14f66db1bd377364f.js	1.652	File JavaScript	05/02/2021 14:50:00
1007.871444f2683fcd16ae68.js	1.655	File JavaScript	05/02/2021 14:50:00
1008.cf0c1e99130619b0435a.js	1.650	File JavaScript	05/02/2021 14:50:00

1837 file e 1 cartella. Dimensione totale: 76.087.696 byte

Figure 6.7: The wwwroot folder structure in FileZilla.

This completes the section on the deployment of the web application in the Azure platform. The following section is dedicated to the DevOps toolchain where the main concepts and the description of the one adopted in this work are presented.

6.2 DevOps Toolchain

In general a toolchain is a digital set of tools that supports in the achievement of a specific objective. A **DevOps toolchain**, known also as DevOps pipeline, is a combination of the most efficient tools used to implement the key practises of *DevOps*, according to agile principles.

The word "DevOps" is a combination of the terms "development" and "operations", thought to represent a collaboration between the development teams and IT operations teams of a company. In its more general meaning, DevOps is a kind of philosophy that encourages a better communication and collaboration between these two teams and other teams inside an organization. While, in its strictest meaning, DevOps describes the usage of techniques, like the iterative software development, automation of the processes, and deployment and maintenance of software-defined infrastructure. The term also covers the concepts of culture changes, such as building trust and unity between developers and system administrators and aligning the software products to customers requirements. Following the agile principles, DevOps aims to reduce the length of the software development life-cycle (SDLC) and to provide continuous and fast delivery of software with high quality and reliability.

6.2.1 The DevOps life-cycle

The DevOps process can be visualized as an infinite loop, composed of the stages plan, code, build, test, release, deploy, operate and monitor. Then, after feedbacks from the customer or IT operators teams, the loop starts again with the plan stage. These steps together form the *DevOps lifecycle*. The figure Fig. 6.8 shows an example of a DevOps toolchain, with all the building stages and some example of tools.

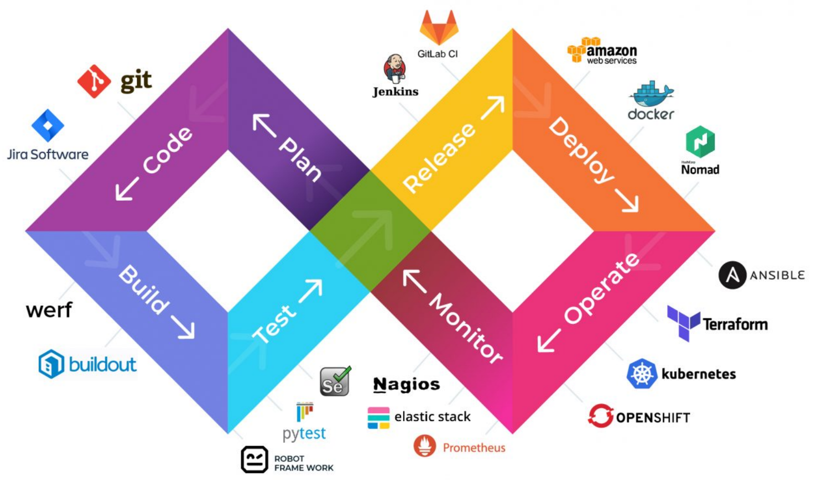


Figure 6.8: An example of DevOps toolchain (Ref. [18]).

The DevOps lifecycle starts with the **plan** phase, where development and IT operations teams, business analysts and other stakeholders define the application requirements and set its goals. This phase can be assisted by a set of tools which provide transparency among all these participants, helping, for instance, in the agile iteration planning. A couple of examples of the planning tools are Asana and Clarizen.

The next phase is **code**, which consists on developers working on the application design and on the creation of its code. This can be done either by the means of a simple text editor or an integrated software development environment (i.e. Visual Studio). The application code can be written in any language, but it is maintained in a repository by using *Version Control* tools. The code maintenance is referred to as source code management. The most popular Version Control tools are Git, CVS, and JIRA.

The **build** phase handles the application builds and versions by the means of automated tools used to compile and package the application code for a future deployment to a production environment or for applying automated testing on it. The code written in the DevOps code phase is pulled from the repository, where it is stored, and then compiled into a *binary artifact* (namely an executable file, such as a zip, a war or a jar) which is then published again in the repository. Tools like Ant, Maven, Gradle can be used in this phase for building or packaging the code into an executable file that can be sent to any of the next DevOps phases.

The artifacts produced in the previous phase can be used by the **test** phase for continuous testing (manual or automated). This is performed in order to ensure the highest code quality, detecting and resolving all the occurring issues. The tests can be performed in a local development environment (directly in the IDE) or in a staging environment, which has the same configuration of data of the production environment. The application can be deployed in a staging environment using, for instance, the docker containers. Some examples of tools used in this phase are JUnit, Selenium and TestNG. These tools allow to perform automated testing, by the means of written QAs ("Question and Answers") used to test several application code-bases at the same time to ensure that there are no issues in the functionalities.

In the **release** phase, a new build of the application code is released into a staging or pre-production environment. Here a limited group of users must validate it against the build of the application currently deployed in that environment. Example of tools suitable for this phase are Docker, Ansible, Jenkins and Kubernetes.

Once the new application build is tested in the release phase and everything works correctly, it can be deployed directly in the production environment, where it can used by the customers. This is done by the **deploy** phase, which involves tools that assist in managing, coordinating, scheduling, and automating software releases into production. Also for this phase, the tools Docker, Ansible, Jenkins and Kubernetes can be used.

With the **operate** (or configure) phase, the DevOps IT operations start. Here the IT operations admin manages applications and the resources used by them in production. Sample tools include Ansible, Puppet and Chef.

Finally, in the **monitor** phase, IT operations teams continuously monitor the application performance and health to ensure that everything runs without problems in production. The data produced by monitoring can be passed back to the development team and other stakeholders to provide instructions and improvements for the next DevOps cycle. Monitoring operations can be assisted by tools like Datadog, Grafana and Nagios. These tools help to resolve errors like low memory, server not reachable and network related problems. In addition, they guarantee also the security and availability of the services.

Monitoring the infrastructure and application performance as well as the end-users experience brings to the development of new software features and therefore to the starting of a new DevOps cycle.

6.2.2 DevOps common practices

DevOps environments generally use common practises, such as continuous development, continuous delivery, continuous integration, continuous testing, continuous deployment and continuous monitoring. Each of these may include one or more DevOps life-cycle stages. Let us go through these practices focusing on the particular DevOps life-cycle stages used by each of them.

Continuous development. This practice involves the "plan" and "code" phases of the DevOps life-cycle. At the beginning of each agile iteration, the application requirements and goals are defined. These decisions can be made also taking in considerations the feedback produced by the continuous monitoring practice of the previous DevOps cycle. Then, the development team decides the features to be released in the next iteration and starts working on them.

Continuous integration (CI). This is the core of the entire DevOps life-cycle. This is a software development practice in which the developers require to commit changes to the source code more frequently (on a daily or a weekly basis). After every changes on the application code, a new build of this code is done. This procedure not only detects early compilation errors, but it includes also activities like code review, unit testing, integration testing and code packaging.

The continuous development of the application implies that the new application code changes need to be integrated continuously and without errors with the existing code to reflect changes to the end-users. In the case of the above mentioned Jenkins tool, every time there is a change in a git repository, Jenkins pulls the updated code and creates a new build of that code. This build is then deployed on a staging server, for testing purposes, or on a production server, where it is used by the customers.

Continuous testing. This practice involves the "Build" and "Test" phases of DevOps life-cycle. Every time the application code is written or updated, it executes pre-scheduled and automated tests. These tests allows to speed the delivery of application code to production. Executing automated tests saves a lot of time and effort. After that all the tests pass successfully, the application code is continuously integrated with the existing code.

Continuous delivery. This practice automates the delivery of application code changes, after passing the previous testing phase, to a pre-production environment. Then, a team member can decide whether to publish these changes into production or not. This practice does not publish the new changes automatically in production, because it could be useful to do some end-to-end testing before delivering them to customers.

Continuous deployment (CD). It is similar to continuous delivery, with the difference that this practice automates the release of new or updated application code into production. A company doing continuous deployment may want to release code or feature changes more times per day. The usage of container technologies, such as Docker, Kubernetes and many others, can enable continuous deployment by helping to maintain consistency of the code across different environments (i.e. development, test, staging, production). Moreover, by the means of *configuration management* tools, a consistency in terms of system configurations and updates is kept across all the servers (staging server and production server).

Continuous monitoring. This practice correspond to "Monitor" phase, where the performance of the application in production is continuously monitored by the IT operations team. The possible issues found in this phase are signaled to the development team so that they can fix them in the next continuous development phase. This approach is very important because speed ups the resolution of problems.

Really important are also the DevOps tools regarding the **Collaboration** between different teams inside an organization. These tools help teams work together also if they do not share the same location. Faster and clearer communication brings to faster software releases, avoiding time losses. Some examples of collaboration tools are Slack, Campfire and Skype.

Moreover, the common objective of organizations is to create a DevOps toolchain composed of one tool for each building stages. This is commonly known as "healthy" toolchain and allows to delivery the best software products more quickly and more efficiently.

In a DevOps toolchain standardization and consistency principles are fundamental, namely the team members must follow the same procedure implemented by the toolchain every time, leaving nothing to personal interpretations. The toolchain practically models the processes through which the DevOps practices are applied, so using the right one is important.

Furthermore, there are two ways to build a toolchain: *in-the-box* and *custom*. An in-the-box toolchain is a solution which has been built by third producers and offers several subscriptions which can be chosen. This approach allows to have a higher standardization and integration in tools used, requiring less personnel to implement the toolchain. By contrary, the custom toolchain consists on selecting personally the single tools and organize them to function together. This approach avoids from depending in tools built by other producers, but it can result to be more restrictive in the budget.

An important benefit of the DevOps toolchain is the fact that allows a **faster time to innovation**, namely it helps the businesses to innovate quicker and better, by standardizing a pipeline where continuous development of software is performed. The continuous development, together with automated testing and monitoring, provides fast and high quality deliveries of software products. The organizations which are able to innovate faster than others gain a competitive advantage.

The DevOps toolchain provides also a **fine-tuning incident control**, which means that using a structured pipeline and infrastructure enables teams to respond to occurring problems more quickly and efficiently.

Finally, the DevOps toolchain offers the **quality assurance**, namely it solves software defects quickly and accurately guaranteeing software releases of high quality. The automated reports coming from monitoring done by the IT operations team help the development team get faster feedbacks on software defects. This helps to speed up the resolution of problems and increase the end-user satisfaction, delivering in this way software products of higher quality.

Having introduced the DevOps toolchain in general, let us now analyze the DevOps toolchain adopted in this work.

6.2.3 The adopted DevOps toolchain

The toolchain implemented in this work does not have for each stage a tool. For the collaboration and communication between the team members the calls and conversations on Google Meets have been adopted. Daily scrum meetings and demonstrations of the new features to the stakeholders took place using this platform.

For planning and issue tracking operations, the "Issues" and "Boards" sections of GitLab platform have been exploited. A team member, after an initial discussion with the other team members, creates a new issue which may refer to the addition of new features, edition of old features or bug-fixing. In each issue one or more labels can be put to specify the type of activity it represents (i.e. feature, low fix, medium fix and high fix) and the state of completeness (i.e. to do, done, doing, blocked). The labels can be created in the "Labels" section of GitLab, which is located inside the "Issues" section. Moreover, always inside the "Issues" section, there is the "Boards" section, where new GitLab boards can be created. A GitLab board is a collector of issues presenting a particular label. For instance, the "Done" board contains all the issues having the "Done" label. Moreover, when creating a new repository, GitLab provides default boards, such as "Open" and "Close", but it is possible to create new boards. Then, when a new issue is created, it is inserted into the default board named "Open". While, when it is closed it is put automatically in "Close" board. In addition, the issues could also be associated to a milestone (deadline). All the issue with a certain milestone are grouped in the "Issues/Milestones" section.

Concerning the Version control management, the Git with GitLab platform have been adopted. When a new issue is assigned to a team member, she creates a new branch from the master branch. Then, she works locally on that issue and pushes the new changes to the created branch. Afterwards, she waits to get a code review done by one other team member. If her changes are approved, she can merge her branch to the master branch, otherwise she has to solve the possible errors or add improvements and repeat the procedure. The corrections done by the other team member can be reported by using the code review section of Gitlab, where comments on lines of code are possible.

Instead, regarding the configuration management and monitoring operations, no tools have been used. However, Microsoft Azure provides some services which assist in these operations. For instance, the *Azure Monitor* tool helps to monitor deployed applications, generating insights (Azure *Application insights*) used to understand their performance. Then, in order to solve issues related to deployed applications or to resources used by them, Azure monitor provides two possible approaches. Automatic alerts can be sent to the IT operations administrator, demanding him the task to understand and solve them, or automated scripts can be executed, which try to solve the problems by their own. In addition, Azure allows to manage the settings of the Azure services (i.e. Web app) where the applications are deployed. If extra resources are needed (i.e. disk, ram), these can be extended whenever necessary without affecting the customers' accessibility. In this regard, also the usage of the Azure *App configuration* service could be useful. This creates a central configuration storage, which can be used to share settings across different Azure app services which have been created in the platform.

Moreover, for the build stage the .zip artifacts have been exploited in the deployment process of the application.

Finally, for continuous integration and contiguous delivery/deployment, the GitLab CI/CD DevOps was thought to be a valid technique. In the following section the main characteristics of this technique are explained, proposing at the end the implementation for the present work.

6.3 GitLab CI/CD DevOps

GitLab CI/CD [19] is a tool used to develop software by the means of continuous integration (CI), continuous delivery (CD) and continuous deployment (CD) methodologies. These are defined as "continuous methodologies" and thought to automate the execution of scripts to reduce the risk of producing errors when creating applications. The processes starting from the development of new application code until its deployment in an environment is automated by these methodologies, requiring less human involvement or, in some cases, no involvement. GitLab CI/CD includes continuously activities like building, testing, and deploying new code changes of an application in small iterations, minimizing the possibility of implementing new code based on previous versions containing bugs or failures.

Let us now concentrate on these three continuous methodologies and explain in detail how they work.

6.3.1 The continuous methodologies

The **continuous integration** methodology works by pushing small code changes to the application code stored in a Git repository. Every time new code changes are pushed, a pipeline of scripts is automatically and continuously executed in order to build, test and validate these code changes, before merging them into the master (main) branch. This happens both when pushing to the master branch and to development (or feature) branches and decreases the risk of introducing errors in the application. The same GitLab could be considered as an example of using the continuous integration technique as a software development method. For every push to a project stored in a GitLab repository, a set of scripts can be configured in order to be executed and validate the new code changes.

Continuous delivery is the natural consequence of continuous integration. This methodology allows to continuously deploy new code changes of the application, every time they are pushed to the git repository where the application code is stored. However, the deployment of the new code is activated manually by the developers.

Finally, **continuous deployment** is similar to continuous delivery, with the difference that the application is deployed automatically by GitLab, without any human intervention.

These methodologies are really important because allow to find bugs and errors early in the development life-cycle, guaranteeing that all the code deployed on production environment meets all the code standards and requirements decided for the final application.

Let us now describe how the GitLab CI/CD is used in the development process and which tools it offers.

6.3.2 The CI/CD basic workflow

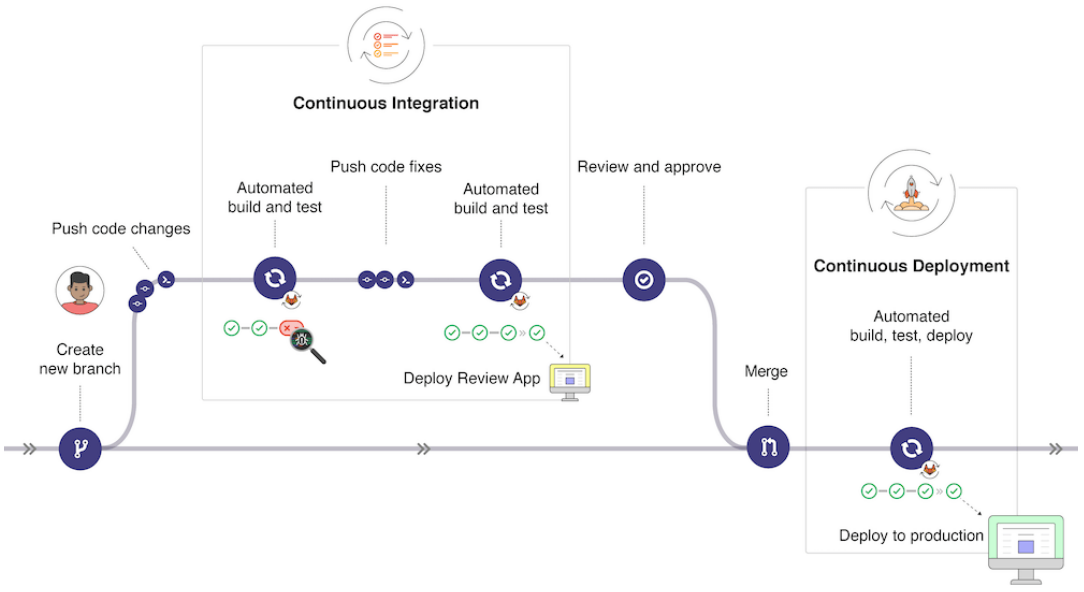


Figure 6.9: The GitLab CI/CD basic workflow (Ref. [19]).

The Fig. 6.9 represents an example of how GitLab CI/CD operates in a common development process. Assume that a developer has discussed with her team about a feature implementation, opening an issue in GitLab, and worked locally on it. After she pushes a new commit with the new code changes to the feature branch (created from the master branch) in a remote repository in GitLab, the CI/CD pipeline is activated. This runs, sequentially or in parallel, some automated scripts in order to build and test the application and preview the changes per merge request through the GitLab review app. The latter is a type of collaboration tool offered by GitLab DevOps, that provides an interface where the changes on the application code are shown. Afterwards, when she is satisfied with her implementation, she waits her code to be reviewed and approved by one other team member. After getting the approval, she merges the feature branch into the master branch. Finally, GitLab CI/CD automatically deploys the new changes to production. If errors occur the new changes can be easily rolled back by her or any member of the team.

Figure 6.10 shows the features available at each stage of the GitLab CI/CD practices. These stages are called with different names with respect to what seen in the previous section, but they represent the same concepts. These are *Verify*, *Package* and *Release*. Let us focus on the characteristics of each of them.

In **Verify** (or Test) stage it is possible to automatically build and test the application with the continuous integration practice, examine the quality of the source code

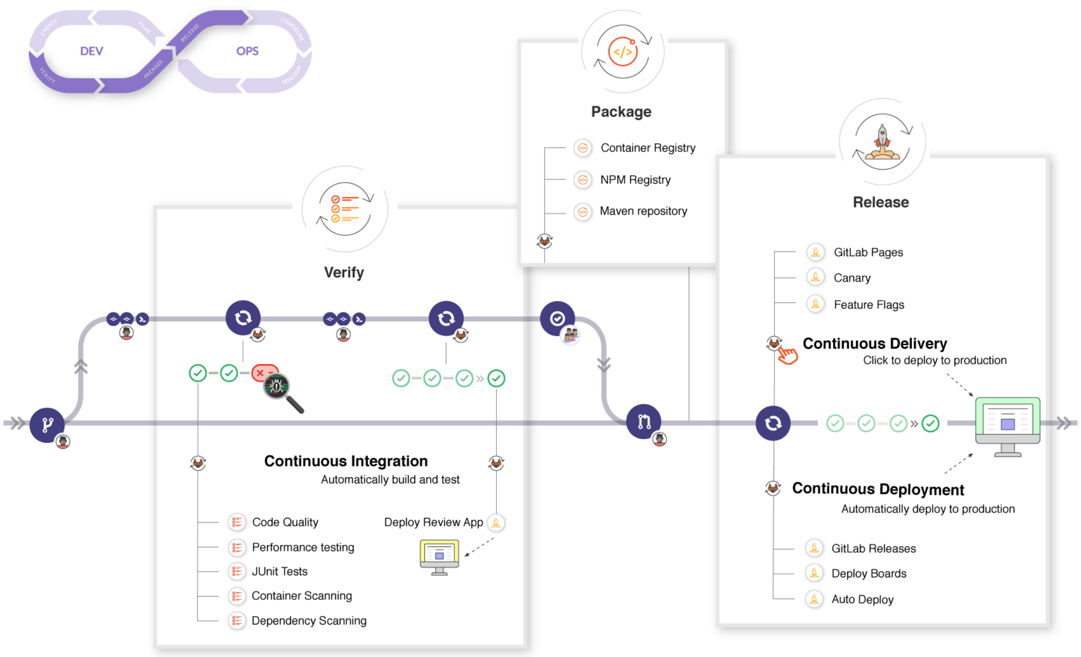


Figure 6.10: The GitLab CI/CD basic workflow extended (Ref. [19]).

with the GitLab *Code Quality*, perform unit tests and deploy the code changes with GitLab *Review Apps*. In **Package** (or Build) stage GitLab allows to store docker images with the *Container Registry* and store packages with the *Package Registry*. Finally, the **Release** stage provides server features, such as continuous deployment (automatically deploying the application to production), continuous delivery (manually click to deploy the application to production) and many others.

6.3.3 The CI/CD configuration file

Technically a GitLab CI/CD is configured by a file called **.gitlab-ci.yml** and located in the root folder of a git repository. This file creates a **pipeline**, which is triggered by GitLab every time new changes of the application code are pushed in any branch of the repository.

Pipelines are the top-level elements of continuous integration, delivery and deployment practices. A pipeline is composed of one or more **stages**, that are run in order and can each contain one or more **jobs**, which are run in parallel. The stages and their jobs are specified in the .yml configuration file. The jobs elements define the operations to perform, typically they consist in execution of scripts. Example of jobs are those used to compile or test the application code. Whereas, the stages elements define when to run the jobs. For instance, a pipeline could be composed of a stage that runs scripts to test the application, which is executed after a stage that compiles the application code.

Moreover, the jobs are executed independently from each other by **runners** (i.e. GitLab Runner agent), which are agents that run the code defined in the .git-lab-ci.yml file. More precisely, a runner takes a job of a pipeline, runs it, and returns back its result to the GitLab instance. Runners can be created by administrators and are visible in the GitLab user interface, when they can be configured. They can be specific to certain projects or available to all projects. Briefly, through the GitLab user interface three different type of runners can be created. The *shared runners*, available to all groups and projects in a GitLab instance, the *group runners*, available to all projects and subgroups in a group, and the *specific runners*, associated to specific projects (typically, specific runners are used for one project at a time).

Furthermore, if there are sufficient runners, different jobs of the same stage can be executed in parallel. If all jobs of a stage succeed, the next stage of the pipeline is executed. By contrary, if any of the jobs of a stage fails, the next stage of the pipeline is not executed and the pipeline terminates without completing.

In general, pipelines are executed automatically and require no intervention after they have been created. However, sometimes manual interaction with pipelines is possible.

A typical implementation of a pipeline is composed of four stages, executed in the following order:

- a build stage with a job used to compile the application code
- a test stage with two jobs used to perform automated tests on the application code. For instance, one job is used to test the frontend part and the other the backend part of a web application
- a staging stage with a job used to deploy the application in a staging environment
- production stage with a job used to deploy the application in a production environment.

Moreover, the pipelines can be configured in many ways. For instance, there are the *basic* pipelines, in which the stages are executed in the order they are specified in .yml configuration file. The jobs of each stage are executed in parallel. As an alternative, there are the *Directed Acyclic Graph* (DAG) pipelines, which are based on relationships between jobs and are executed faster than the basic pipelines.

As previously explained, to use GitLab CI/CD it is needed to have the application code stored in a Git repository (i.e. a GitLab repository), and the file .gitlab-ci.yml located in the root repository of the GitLab repository, which contains the configuration of the CI/CD pipeline. In general, in this file it is possible to define:

- the scripts which needed to be run
- the way to run the scripts (automatically or manually).
- where to deploy the application
- project dependencies and caching policies

An example of a gitlab-ci.yml configuration file is shown in Fig. 6.11. In this example, *build-code-job* of the build stage is executed first. It prints the Ruby version of the operating system in which the job is run, then executes the "rake" command to build the project files. If this job completes successfully, the two test-code-job jobs of the test stage are run in parallel and execute the tests on the files. The pipeline in the example is composed of three jobs, grouped into two stages, and is triggered every time new changes are pushed to any branch in the project.

Briefly, taking as example the Fig. 6.11, let us introduce some example of syntax and keywords that can be used in the .gitlab-ci.yml file. However, for more information, it is possible to examine the ".gitlab-ci.yml" section of the GitLab CI/CD documentation [19].

The stage elements are defined as the key word *stages*, then a column followed by the dashed list of stage names (i.e. build and test, in the example). While, the jobs are defined with an arbitrary name and must contain at least the *script* clause. The

```
stages:
  - build
  - test

build-code-job:
  stage: build
  script:
    - echo "Check the ruby version, then build some Ruby project files:"
    - ruby -v
    - rake

test-code-job1:
  stage: test
  script:
    - echo "If the files are built successfully, test some files with one command:"
    - rake test1

test-code-job2:
  stage: test
  script:
    - echo "If the files are built successfully, test other files with a different command:"
    - rake test2
```

Figure 6.11: An example of the gitlab-ci.yml file (Ref. [19]).

job's name must be unique among all jobs defined in the .yml file, while a stage can have also the same name of one of its composing jobs. Inside the script clause of a job it is possible to specify the list of commands which must be executed. A command can execute code directly ("npm install") or run a script ("./script.sh") in the root repository of the GitLab project. In addition to the script section, it is possible to define the *before_script* one, which contains the set of commands which must be executed before the commands of the script clause. Typically, these commands are needed to update packages and installation files.

Furthermore, inside each job it is possible to specify the stage where it must be executed, using the key word *stage* followed by the stage name (i.e. stage: build, in the previous example). In addition, inside a job it is possible to specify the docker image where the GitLab runner executes the scripts of that job. If this is not specified, the default image of Gitlab runner is used, which is the latest version of Ubuntu. Moreover, there are no limits on the number of jobs which can be defined.

Let us analyze now the CI/CD pipeline adopted in the present work, describing in detail all its building blocks.

6.3.4 Implementation of the CI/CD pipeline

The CI/CD pipeline is composed of three stages, named *build_be*, *build_fe* and *deploy*. These are executed sequentially and the next one can start if the previous one has finished. Therefore, first build_be is executed, then build_fe and lastly deploy. Each of these stages is composed of a single job. The build_be and the build_fe stages contain the jobs used to build the ASP.NET Core Web.Host application (backend) and the Angular application (frontend) respectively, producing two artifacts (zip files). Instead, the deploy stage is used to deploy these two applications on the two Azure web app resources, which have been described in detail in the previous section. Since this is done by the means of the two artifacts which are produced by the previous stages, for this reason the deploy stage can start only if the execution of the build_fe stage terminates successfully.

The build_be stage is activated when new changes on the master branch are pushed. The job of this stage is executed inside a docker image environment. This is based on Linux Debian operating system, above which .NET Core 3.1 SDK (Software development kit) has been installed. The image URL in the docker registry is "mcr.microsoft.com/dotnet/core/sdk:3.1".

In this job a *before_script* section has been defined, where some instructions are executed in order to update all the installed packages, install the zip package, install the dotnet-ef tool (necessary to apply EFCore migrations) and restore the Nuget packages of the ASP.NET Core solution. This section is followed by a *script* section containing the list of instructions which must be executed by the job. A first instruction builds the ASP.NET Core solution for the release. This instruction creates a new folder in the ASP.NET Core solution root folder, named "build". Then, the content of this directory is zipped using the zip command of the zip package installed in before_script section.

In addition, build_be job executes a cli command necessary to create the migrations to be applied to the database and save them in a file (with .sql extension). Then, it runs an external Windows Powershell script, which receives as parameters this sql migrations file and the environment to which the application will be deployed (in this case, Production). The script obtains the connection string, which is the same seen also in Sec. 6.1.2, and applies the migrations to the production database, executing the dotnet cli command already seen in Sec. 3.3.5 ("dotnet-ef database update"). This time, the command is run specifying also as parameter the connection string to the production database, which is optional by default. This parameter is not needed when applying the migrations inside an Integrated Development Environment (IDE), because it can recover automatically the database connection string by the means of the appsettings.json file.

Then, in a final *artifacts* section, an artifact item is defined, whose "path" field points to the "build" folder (containing the zip file created before). In addition, this artifact is stored in the GitLab cache for one day.

The build_fe job contains all the scripts needed to build the Angular application so that it is ready to be published on Azure. This job uses a different image, which is Linux Alpine with Node installed on it (its name in docker hub is "node:12.16.3-alpine"). As done for build_be, in this job there is a before_script section where some CLI commands are executed in order to install the zip package, gulp package and the angular cli. Instead, a script section contains the list of all commands required to build the angular solution. In order, first the command "npm run build-production" creates the "dist" folder inside the angular solution folder. The "dist" folder contains all the files needed for deploying the angular application in Azure. An other command creates a zip file zipping the content of this folder. A terminal artifacts section defines a new artifact with expiration time of one day and path url pointing to the zip file contained in the "dist" folder.

Finally, the deploy job is executed. This uses the default image of GitLab, namely the last version of Ubuntu. Also here there is a script section which contains the list of operations which must be performed. First, the Azure command line interface is installed (AzureCLIDeb). Then, the azure cli command used to login in the azure platform is performed. This command uses some credential which are stored in proper global variables inside the .yml configuration file (typically they are put on the top). Then, the ASP.NET Core Web.Host application is published on the corresponding web app resource created in Azure by the means of an azure cli command. This command receives as parameters the web app name, group resource name and the zip artifact produced in the build_be stage. Finally, also the angular application is published using the same azure command, but this time passing as parameter the web app name used for the angular application in Azure and the artifact produced by build_fe stage. These web app names and the resource group name are saved in global variables inside the gitlab-ci.yml.

This discussion concludes the chapter and completes the different steps which have been followed from the design to the implementation and deployment of the web application. The next pages are dedicated to the conclusions of the thesis.

Conclusions

In this thesis work a SaaS web application has been developed. It was a practical problem to solve for a company who commissioned the task. The application purpose is the engineering and automation of the management control phases typical of any company. A user who can be an operator or a collaborator can interact via a simple and intuitive user interface with different entities. According to his role and therefore on the assigned privileges, a user can for instance create and modify orders and revenues. He can visualize and write comments and manage revenues objectives or even visualize behavioural reports.

These operations are very common to any company and therefore the developed application can be regarded as very general. Moreover, the multi-tenancy feature has been implemented in the application, an important feature which allows the application to be used by different customers at the same time once they have started a subscription.

All the stages of the application development process, from the definition of the requirements, to the design of the application and its final implementation have been worked out carefully in order to build a quality product. In particular the design of the application data model as well as the business requirements definition have been conducted understanding and following the customer's needs. To achieve this, several and frequent meetings, according to the Scrum software development methodology, have been arranged. Moreover, in order to adapt to the evolving customer's needs, in some cases modifications on the data model and business logic were necessary.

To realize the requested application, a constant study was conducted on all the technologies needed, concerning the ASP.NET Zero solution and the frameworks ASP.NET Boilerplate, ASP.NET Core and Angular. The first three required the study of C# language, while Angular required the study of typescript, javascript, html and css languages.

Finally, the application has been tested in all the implemented functionalities, accepted by the customer and then deployed in production, where the customer can use it.

However, when starting to work on the development of the application, there were some initial problems in configuring the work environment. These issues involved the connection to the database used for development which were solved by setting some configurations on the startup class of the ASP.NET Core solution.

As explained in the thesis, the ASP.NET Zero solution is used in this work. This offers many already implemented functionalities which can be very convenient to speed up the initial time and effort needed to configure the base structure of an application. In this way, the developers can only concentrate on implementing only the customer requirements for the application, increasing their productivity time. However, these comforts present also some drawbacks. Since these functionalities are implemented by others, it may be a problem if errors occur when using them. Thus, before developers start using these functionalities, they should have a general idea on how they have been implemented by consulting the official documentation of

ASP.NET Zero. Furthermore, it may be useful to reuse the code of these functionalities for the implementation of other features.

Moreover, the application base structure followed by ASP.NET Zero helps the developers in keeping the working environment clear and efficient. As explained in Sec. 3.3, each sub-project of ASP.NET Core solution should be reserved for particular classes or elements. However, this organization of these files in the corresponding sub-project is not strictly required, but only advisable. Therefore, the non respect of this structure may produce sometimes issues which can be avoided by following simple procedures and conventions.

As explained in Sec. 3.4, ASP.NET Zero provides the possibility to have a typescript file (named "service.proxies.ts") which can be auto-generated by the solution executing a .bat file in the "nswag" folder. The typescript file contains all the angular service classes and methods, which can be used by the angular components to perform API calls in order to retrieve data from the backend application. This approach avoids from implementing manually the services which allows to save much time. However, the problem is that the service.proxies.ts file can be very large and difficult to manage. Indeed, sometimes resolving merge conflicts, interested this file, was really complex and implies a huge and useless waste of time. A good compromise can be achieved by continuing to auto-generate these service methods, but reorganizing their code in different service classes, considering one class for each backend endpoint. Alternatively, the auto-generating functionality is not even used and all the services classes and methods are implemented manually. This approach requires more time and it is more error-prone.

Concerning the Scrum approach used for the software process management, it is noticeable how this helped in delivering quickly the requested features. Doing the sprint planning activities by the means of the Gitlab "Issues" and "Boards" sections considerably helped to trace the status of issues over the time, and even to understand clearly which functionalities had higher priority. However, there occurred sometimes some miss-understandings with the customer, but solved quite quickly with clarification meetings. With these experiences, I could see the importance of understanding the customer's needs, especially when I had to build the application data model. Making an error in this phase means to create a new database migration and change lot of files in the backend and, often, also in the frontend. This problem happened one time only at the beginning of the internship, due to a misunderstanding with the customer, which involved a refactoring of some files.

Furthermore, fundamental were also the sprint reviews, where the customer, after testing the application, could give some feedbacks. In these meetings, I could understand if the product really satisfied the customers and which improvements were necessary to do. Besides the meetings with customer held in the sprint reviews, I had the possibility to participate also to those where a normal developer should not be allowed. This is a clear example of the fact that Scrum framework is general and everyone could implement it as desires. For instance, I could assist to meetings between the product owner and the customer, where the requirements, concerning the application module I was in charge to develop, were decided. Then, together with the product owner I could choose the most valuable requirements to deliver in each scrum iteration. Also, the daily scrum meetings resulted to be very useful, because allowed to have a clear idea of the progress of work and to solve some issues common to different team members.

This thesis experience enriched and extended the knowledge acquired in the master's degree courses. I worked in all the stack of software development. I could operate on data model design, data access management with EFCore and MSSQL

Database, requirements definition and implementation, testing and deployment on the final hosting server. All these different knowledge put together produced a final and complete product.

The study of the key elements of ASP.NET Core framework was partly eased by the fact I had already seen the same concepts, but in other frameworks used for web development (i.e. Spring Boot). Concepts like services, DTOs, controllers, entities and repositories are quite general, the only difference stays in how they are implemented in the different frameworks. Thus the study of ASP.NET Core was done making always comparisons with Spring Boot. The same approach was done by confronting the EFCore ORM (used in the present work) with the Hibernate ORM (studied together with Spring Boot in a master's degree course). These comparisons helped to make stronger the concepts about the importance and the utility of ORMs, but also to be aware about their possible drawbacks. As concerning the present work, EFCore results to be really useful. No writing of complex queries was needed, the default repositories with CRUD methods provided by EFCore were enough. However, custom repositories are possible too when the requested SQL queries are not conventional and articulated. The ORM is a powerful tool, but only if we configure it correctly, we can exploit its full potential.

To conclude, the work has been completed and the goal has been reached. The final product met all the costumer's requirements. Improvements of the application are however possible but not essential as for instance, some UI elements can be made more attractive for the customer. As an example, the revenues report on the top of the "Orders Advancements" section can be improved by using some user-friendly icons and assigning a different CSS style. Also, more unit testing can be performed. It would allow to find issues that are detectable only under some particular circumstances where neither end-to-end testing can easily reach.

Acknowledgements

I thank my parents who gave me the opportunity to study in a prestigious university and my brother of being my academic and life model. I thank the company for giving me the possibility to work on this project with them and my tutors for the help provided. I thank also my university colleagues with whom I had the possibility to collaborate in several academic projects which contributed to enrich my computer engineering knowledge.

Bibliography

[1] ASP.NET. *Link to aspnet documentation page,* URL: https://docs.microsoft.com/it-it/dotnet/.

[2] ASP.NET Core. *Link to aspnet core documentation page,* URL: https://docs.microsoft.com/en-us/aspnet/core/?view=aspnetcore-5.0.

[3] Express JS. *Link to expressJs documentation page,* URL: https://expressjs.com/.

[4] Django. *Link to django documentation page,* URL: https://docs.djangoproject.com/en/3.1/.

[5] ASP.NET Zero. *Link to the asp.net zero documentation page,* URL: https://docs.aspnetzero.com/en/aspnet-core-angular/latest/.

[6] ASP.NET Boilerplate. *Link to the asp.net boilerplate documentation page,* URL: https://aspnetboilerplate.com/Pages/Documents.

[7] ASP.NET Boilerplate vs ASP.NET Zero. *Link to comparison between ABP and Zero,* URL: https://docs.aspnetzero.com/en/common/latest/Abp-Template-vs-AspNet-Zero.

[8] Integration to EntityFrameworkCore in ASP.NET Boilerplate. *Link to EFCore integration documentation page,* URL: https://aspnetboilerplate.com/Pages/Documents/EntityFramework-Integration.

[9] EFCore code-first reference documentation. *Link to the EFCore code-first reference documentation,* URL: https://www.entityframeworktutorial.net/code-first/what-is-code-first.aspx.

[10] EFCore reference documentation. *Link to the EFCore reference documentation,* URL: https://www.entityframeworktutorial.net/efcore/entity-framework-core.aspx.

[11] Swagger UI. *Link to Swagger UI documentation page,* URL: https://swagger.io/tools/swagger-ui/.

[12] Angular NgModules. *Link to Angular NgModules documentation page,* URL: https://angular.io/guide/ngmodules.

[13] Angular Components. *Link to Angular Components documentation page,* URL: https://angular.io/guide/components.

[14] Angular Services. *Link to Angular Services documentation page,* URL: https://angular.io/guide/services.

[15] Agile Manifest. *Link to the Agile manifest document,* URL: http://agilemanifesto.org/iso/en/manifesto.html.

[16] Scrum official guide. *Link to the scrum official guide document,* URL: https://www.scrumguides.org/scrum-guide.html.

[17] Microsoft Azure. *Link to Microsoft Azure documentation page,* URL: https://docs.microsoft.com/en-us/azure/?product=featured.

[18] DevOps toolchain image. *Link to DevOps toolchain image,* URL: https://quintagroup.com/services/devops.

[19] GitLab CI/CD reference documentation. *Link to the GitLab CI/CD reference documentation,* URL: https://docs.gitlab.com/ee/ci/.