**Master Degree Thesis**

# Design and development of the real-time tournament feature in the MAK07 game platform


## Francesco Pipitò

## Supervised by

Prof. Giovanni Malnati


Master Degree course in Computer Engineering



Department of Control and Computer Engineering
Turin Polytechnic
Italy, Turin

July 2018

**Design and development of the real-time tournament feature in the MAK07 game platform**

**Francesco Pipitò**

Supervised by:

**Prof. Giovanni Malnati**

Department of Control and Computer Engineering

# Abstract

The *Mobile Gaming* is a new interesting phenomenon, capable of providing different entertainment possibilities for a wide range of people. The reasons of its success seem to be the free and easy accessibility from whoever and wherever, thanks to a technology that radically changed several aspects of everyday life, the smartphone. Mobile gaming has allowed the growth of videogames world, increasing its audience and global turnover.
In this context, the *Mak07* game was developed. This is a puzzle game, based on the simple idea of combining together seven numbers, exploiting the arithmetic operations, sum, subtraction, multiplication and division, in order to obtain zero as final result. The group of seven numbers is called *schema*, and each time a schema is solved, a new one appears on the screen. A player has two minutes to solve the highest number of schemata. At the end of the game, a final score is evaluated summing up all the partial scores obtained in each solved schema. A smarter solution and a lower execution time generate an higher partial score. The game underwent several development phases, each of them has introduced new features that improved the game experience. Initially, the game offered, only, the possibility to solve schemata, one after the other within two minutes. Then, in order to introduce player-to-player interactions, the functionality of having challenges among players and the possibility of creating a network of friends were added. These features introduced a new social and multi-player experience that made Mak07 a complete game, that can be, however, improved. That's why it was thought to develop a new feature that improves the competitiveness and increases the game experience: the tournament functionality.

This final elaborate has the purpose of designing and implementing both the client and server of the tournament feature for the Mak07 game. Until now, the game experience is suited for short but intense interactions, thanks to challenge and training mode. The tournament game modality instead is suited for longer interactions, in which players can improve their skills and acquire

more experience. A tournament has the aim of gathering and managing a limited amount of players that want to compete against each other until only one winner remains. A tournament is based on time intervals, in fact it is composed of several phases of different duration time. After a tournament is created, the registration phase starts, in which the system users are able to sign up to the tournament or delete their registration. This phase lasts 20 minutes, after which the tournament begins. So, a sequence of round phases, each of which lasts 10 minutes, is performed. The number of rounds is related to the maximum number of tournament participants, and each round has a number of challenges according to the players taking part to the round. At the end, the tournament winner is proclaimed. Due to this structure, a tournament has an execution time longer than a single challenge. Since that, the tournament feature is designed to allow the users to have longer and satisfying game sessions, unlike those, short and intense, offered by the challenges. During the algorithm development, several situations are tackled in order to guarantee the correctness of information. Suited synchronization mechanisms are developed to guarantee consistency and integrity during the registration and round phases, and a big effort is spent to create a working fusion among the mechanism of challenges and the tournament one. A challenge is based on a state transition mechanism, it has to cross several intermediate states before reaching a final situation. So, its execution time is not fixed, due to, also the fact that the transition from a state to another depends on asynchronous user actions. As a consequence, the challenge mechanism is not suited for the tournament modality, in which a challenge must be played within the 10 minutes scheduled for each round. Efforts have been made to write as much as possible modular and independent code and to not modify the challenge algorithm, that is, instead, exploited. In conclusion, the developed algorithm makes the tournament experience rewarding as much as possible, but, at the same time, it is capable of managing all the synchronization problems that could arise in a multi-user environment.

The technologies used to develop the Mak07 back end software ensure an easy and continuous evolution of the system. The Spring framework, thanks to the *dependency injection* and the *aspect-oriented programming* features, makes it easier and more satisfying developing Java code, ensuring a lighter and linear programming model, prone to future improvements. Spring offers low code coupling, which is, also, supported by the REST principles, that are essential in a web based system. Finally, the storing solution adopted for the Mak07 system is MongoDB. It is a non-relational database, offering an high scalability quality, ensured by the *replication* and *sharding* techniques, and a schema less storing model, based on *documents* and *collections*. This NoSQL solution is suited for a dynamic environment as that of Mak07.

The backbone architecture that supports the Mak07 system is a client-server model, in which the clients are the mobile devices, and the server is the machine hosting the back end software. The server exploits the *Docker* tool to create, on top of the operating system, a virtualized architecture com-

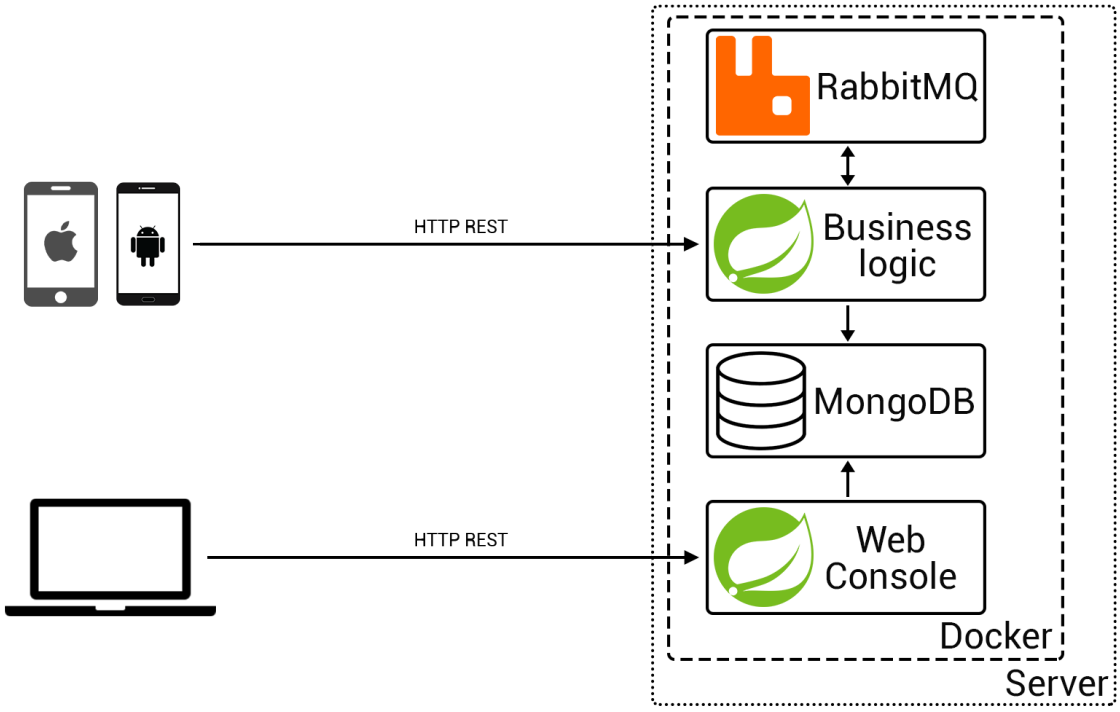


Figure 1: Mak07 architecture.

posed by four main components, called *containers*. Each container hosts a service of the Mak07 system, like the business logic or the database.

In conclusion, all the starting objectives, that led to the tournaments feature development, have been reached, and, after a final phase of testing, the tournament functionality was released as update of the Mak07 mobile application.

# Acknowledgements

I would like to thank all the people who supported me not only in this last months, but also during all the university years.

I would first thank my thesis advisor Prof. Giovanni Malnati, that gave me the possibility of making a professional experience, working on an interesting project. He mentored me with his helpful comments and advices, pushing me to do my best. I would like also to express my thanks to all the staff of TonicMinds that supported me during my final work.

I thank all my friends, and my flatmates, Andrea and Salvo, with which I have shared a lot of beautiful moments.

Finally, I must express my very profound gratitude to all my family, in particular to my parents and to my girlfriend, Valeria, for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of developing and writing this thesis. This accomplishment would not have been possible without them. Thank you.

# Contents

# List of Figures

# Listings

# Chapter 1

# Introduction, Motivations and Goals

Since the birth of the first videogame, in the January of 1947, the videogames world has had a constant evolution. The first videogames were developed by big companies, only for research purposes. Soon, however, they have become a phenomenon that would have involved a growing number of curious people. Originally, the videogames were developed by university students, during their free time, and were run on the first personal computers. With the increase of personal computer sales, also the number of people interested in deepening the videogames world is grown. An important invention that has increased the videogames popularity was the *console*. A console is a custom computer created to execute only videogames. Over the years, the videogames and console market is grown so much that companies, with the only purpose of developing videogames and dedicated hardware, are born. The advent of internet has signed, another, deep change in the videogames world. Internet has not only minimized the distance among people spread all over the world, but also, between gamers playing the same videogames.

Today, videogames have a strong impact on people of all ages. It is a so widespread phenomenon that competition events are born, in which players, coming from all over the world, compete against each other. Videogames offer new entertainment possibilities, thanks to their variety. Like movies, also videogames can be categorized into several genres, according to the covered topics and the games modality. Some of the main categories are: adventure, action, role playing game, first person shooter, etc.

In the last years, the videogames world gets closed to a technology used in the everyday life, the *smartphone*. So, the phenomenon of *Mobile gaming* is born and it spread in a little time, actively contributing to increase the videogames global income. A mobile game is a videogame developed to be played on mobile devices, like smartphones and tablets. The mobile gaming has evolved together with the technology at the base of this phenomenon. In fact, the smartphones evolution has generated more powerful and high-performing devices, able to run fancier and more satisfying mobile games. The success of this phenomenon is due to several reasons. A mobile game

can be played from every one and in every moment, due to the fact that every person has a smartphone.

In a context in which the mobile game is sharply increasing, the *Mak07* project is born. Mak07 is a mobile game available for smartphone and tablets, addressed to players of all ages that want to have fun but, also, to stimulate their brain. It is based on the simple idea of combining together seven numbers exploiting the four arithmetic operation: sum, subtraction, multiplication and division. A group of seven numbers is called *schema*, and the purpose of combining them together is to obtain zero as final result. In order to not make the game too difficult or too easy, the game engine presents some constraints, like the impossibility of exploiting negative numbers or using zero in multiplication. Each Mak07 game lasts two minutes, in which each player has to solve one schema after the other. In conclusion, the final score obtained depends on the number of schemata solved, the time spent on each schema, and how smart is the solution proposed for each schema. As mentioned, Mak07 has the purpose of entertaining and amusing the players, indeed, the game crossed several development phases in which new features have been included in order to improve the game experience. In a first time there was only the possibility to play one game after the other, but, soon, was introduced the first important feature of the game, the challenges. This functionality has opened the doors to competitive gaming. A challenge couples together two players that want to compete against each other. It is composed by a starting phase in which a challenge invitation is exchanged among the two players, and by a game phase in which two game sessions are played by both players. So each player obtains a score that is compared with the one achieved by the other player in order to proclaim the winner. A challenge ends when both players have played their game. In fact an user can play its game whenever he wants, supporting the concept of "play whenever you want" of mobile gaming. Each player can challenge not only a random player, but also a friend. In fact the Mak07 game has, also, a social aspect, ensured by a player profile page, in which there are collected some statistics about challenge played, maximum score obtained, etc., and a list of friends that can be directly challenged or chatted. In conclusion the challenges feature, supported by the social aspect, has introduced an important improvement to the game, which can be exploited for creating a new game experience. That’s why it was thought to develop a new interesting game possibility, the *Tournament* feature. The tournament functionality is introduced in order to not, only, increase the game experience, but also to extend the time spent on game by players. In fact, the challenge mode, as well as the training mode, offers a game experience short but intense, suited to fill the little free times of everyday life. Instead, the tournament one offers longer game sessions, that make Mak07 appealing, also, for who has more time to be spent on the game. The tournaments have the purpose of increasing the competitiveness, initially offered only by the challenges. In fact each tournament manages a restricted number of players that want to compete against each other until only one player remains. A tournament is organized in rounds, composed by several challenges, according to

the number of participants to the round. A player has to win each challenge to which has been associated during the several tournament rounds, in order to get the win. In conclusion, the tournaments feature is introduced to not only improve the game experience, but to make also the game satisfying for the players that want to gather some friends in order to have fun in a longer session game.

This thesis is structured as follow. This chapter is a little introduction to the motivations that led to Mak07 development, describing, also, the starting point of this final project, motivating the development of tournament feature. The following chapter, instead, has the purpose of describing the main technologies used to develop the back end software that supports the Mak07 tournament feature, analysing their weaknesses and strengths and the reasons due to their usage. Then, the chapter 3, is in charge of defining the requirements for the tournament feature, and describing the difficulties encountered during the design phase, in which a big effort was spent to merge, in the best way, the tournament feature with the existing ones. Finally, the architecture that supports the entire Mak07 system is analysed. In conclusion, the chapter 4 contains a deeper explanation about the algorithm developed for the tournament feature, and the synchronization mechanism needed in a multi-player environment. The last chapter analyses some possible future development.

# Chapter 2

# Used Technologies

This chapter has the purpose of describing the main technologies used to develop this final project. It analyses the Spring framework, pointing the dependency injection and aspect-oriented programming features, and the main components living in the Spring environment. Then, the main relational database limits are analysed in order to explain the reasons why it was decided to use a non-relational database, like MongoDB, to store persistence data. The final section, instead, describes the REST pattern, an which are its benefits.

## 2.1 The Spring framework

Being a Java developer could be hard and frustrating in several situations, but in the last years, a lot of technologies try to make easier the developer's job. Among the different technologies created to support the development in Java environment, Spring has been very approved from the community. Spring is the most popular, and widespread, open source framework for enterprise Java. The first version was written and released by *Rod Johnson* in the October of 2002. This framework was born as alternative to some enterprise Java technologies, in particular the *Enterprise JavaBeans (EJB)*. It has become an innovative framework because it has introduced two primary features: the *dependency injection (DI)* and the *aspect-oriented programming (AOP)*. These features have been very appreciated by the Java community. In fact, they are used also in the development of any Java application. In conclusion, Spring offers a lighter and linear programming model, and it is in continuous evolution, particularly in mobile development, social API, cloud computing and big data areas.

### 2.1.1 Dependency injection

The technology that identifies Spring more than all is the *dependency injection* one. It is a particular case of a more general concept, the *Inversion of Control (IoC)*. These two patterns represent two different things, but strictly connected so much that they are used as synonym.

### Inversion of control

The *Inversion of Control* principle is an architectural pattern, it was born at the end of the 80's, and it is based on the concept of inverting the system *Control Flow* of the traditional programming paradigm. This is a widespread concept in the framework environment.

In order to go deeper inside this analysis, should be clarified what *dependency* means. Whenever a class *A* uses another class or interface *B*, then *A* depends on *B*. *A* cannot carry out its work without *B*, and *A* cannot be reused without also reusing *B*. In such a situation the class A is called the "dependant" and the class or interface B is called the "dependency". Two classes, defined as *A* and *B*, are said *coupled*, if they are dependent on each other. So much higher the dependency level of the code is, so much less the reuse, the readability and the flexibility of the code is. The purpose of the Inversion of Control is to minimize code coupling.

The traditional programming model teaches that the developer, explicitly, defines the control flow. *IoC* inverts the traditional control flow. The developer doesn't have to worry about the dependency management any more, because the framework is in charge of making this, as a consequence of actions.

### How does it work

As described, the responsibility of objects initialization is demanded to an external component, that is in charge of creating the objects and filling their dependencies through the injection mechanism. The traditional control flow is distorted, in fact, with *DI*, the objects dependencies are given at creation time. Dependency Injection in Spring can be done through constructors, setters or fields.

- **Construct-based injection**: the dependencies are injected through the object constructor. The dependencies are represented by the constructor parameters.
- **Setter-based injection**: in this case the dependencies are injected through the setter methods.
- **Field-based injection**: finally, the dependencies are directly injected to the object fields, that are annotated in a proper way.

The last approach might look simpler and cleaner than the previous ones, because it allows to minimize the injection if there is a big number of fields, but it has few drawbacks, such us the use of reflection to inject the dependencies, which has an high cost. The use of a *construct-based* approach makes easy to understand the several fields an object is composed, instead the *setter-based* approach could be so much fragmented if the object has a lot of fields. In Spring, the act of creating associations between objects is commonly referred to as *wiring*. The *application context* loads objects definition and wires them together. There are three main ways of doing wiring:

- exploiting an XML file in which are reported all the object associations needed.
- the same configuration expressed inside an XML file could be defined by code in Java, exploiting suited code annotation provided by Spring.
- finally, a last way lets Spring automatically discover dependencies and create the required relationships.

All these approaches are interchangeable and can live together.

### 2.1.2 Aspect-oriented programming

Until this moment, the software development process was strongly influenced by the *Object-Oriented Programming* (*OOP*) paradigm. The key units of modularity in *OOP* are the classes, and the *DI* helps the developer to write classes decoupled from each other. This model doesn't solve all the problems that a system could rise, so, Spring, in order to improve the code modularity, introduces a new important paradigm, suddenly accepted by the Java community, called *AOP*. The *Aspect-Oriented Programming* paradigm has, as key units of modularity, the *aspects*. In order to better understand this concept, we have to think to a software as composed of several components, and each of them is in charge of managing a specific functionality. Often, these components carry, additional responsibilities beyond their core functionality. An example of this could be the logging component, or the security one, that implement functionalities used a lot in components whose core responsibilities are something else. These kind of system services that span multiple components are called *cross-cutting concerns*. *AOP* allows to decouple code that implements functionality that should be separated, exploiting a new external entity, the *aspect*. An *aspect* is a system component which has a set of API providing cross-cutting functionalities, that are invoked when core components execute their code. This is a powerful concept, because it keeps the cross-cutting concerns separated from the core business logic. Each *aspect* could implement several *advices*. An *advice* is a piece of code that is invoked during the business logic execution. Spring, at configuration time, allows to define when each *advice* should be executed. Inside the configuration file, the time instant at which an *advice* is executed is, also, specified. Spring supports five kinds of time instant:

- **before**: the advice is run before a method execution.
- **after**: the advice is run after a method execution.
- **after-returning**: the advice is run after a method execution, only if the method completes successfully.
- **after-throwing**: the advice is run after a method throws an exception.
- **around**: the advice is run before and after a method execution.

### 2.1.3 Spring IoC Container

The Spring *containers* are a core aspect of the Spring framework. A *container* is in charge of creating objects, wiring them together, configuring them and managing their complete life cycle, from creation till destruction. In Spring the objects are called **beans**, they will be analysed in the detail in the following section. The Spring *containers* exploit *DI* to manage the beans that compose the application. Each *container* needs some meta-data in order to instantiate,

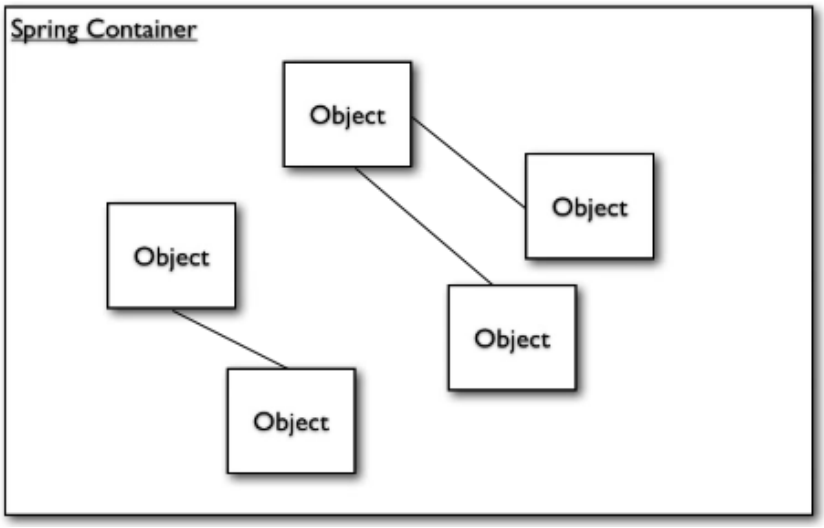


Figure 2.1: Spring container.

configure and associate the objects. These configuration information are provided through XML file, Java annotation or Java code. Starting from classes implementation and meta-data, the Spring IoC container generates a "ready to use" application as final result. Spring supports several container implementations that can be categorized into two distinct types.

- **Spring BeanFactory Container**: this is the simplest one. It provides the basic support functionalities of *DI* and it is defined by the *org.springframework.beans.factory.BeanFactory* interface.
- **Spring ApplicationContext Container**: this one, instead, is an extension of the above factory. It extends the functionalities provided by the *BeanFactory container*, adding more enterprise specific functions. It is defined by the *org.springframework.context.ApplicationContext* interface.

It is possible to use either *BeanFactory container* or *ApplicationContext container* in the same Spring application, but, the first one is, often, too low level for most application. Therefore the *ApplicationContext* is preferred, and it offers different types of application context, among which the most used are: the *ClassPathXmlApplicationContext*, that loads a context definition from one or more XML files located in the classpath, the *FileSystemXmlApplicationContext*, that, instead, stores the XML files in the filesystem, and, finally, the *XmlWebApplicationContext*, that collects the XML files in a web application.

### 2.1.4 Beans life cycle

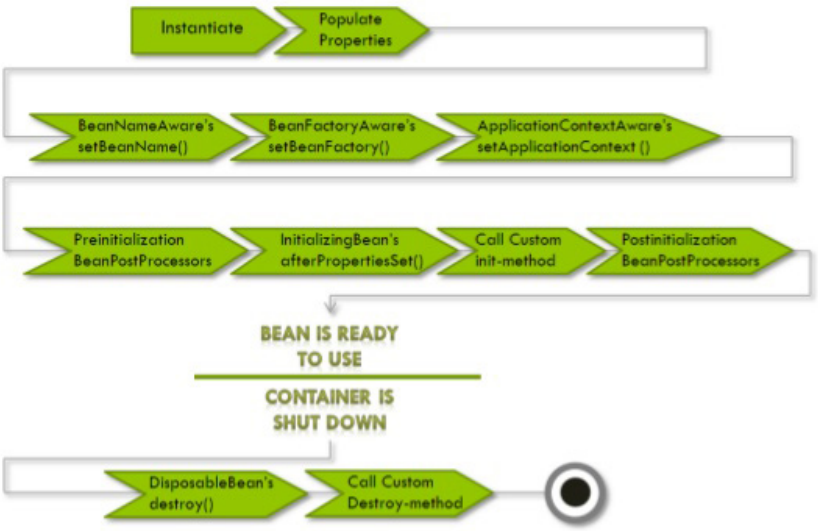


Figure 2.2: Beans life cycle.

In a Spring application, the beans compose the backbone of the entire system. They are managed by the Spring IoC container, that is in charge of creating them, according to the configuration parameter supplied with the container itself. The bean life cycle, in Spring, is more elaborate than the one of a traditional Java application. The Figure 2.2 illustrates the several steps that a bean should follow in its life. A complete knowledge about the bean life cycle, allows the developer to customize some steps in order to improve efficiency inside his system. Each bean crosses the following steps:

1. when a bean is requested, the Spring *IoC* container creates the bean using the class constructor.

2. Now the dependencies are injected using the setter method. Both values and other beans are injected inside the fields.

3. After the dependency injection, if the bean implements the *BeanNameAware* interface, Spring invokes the method *setBeanName()*, passing the bean's ID as parameter. This method sets the name of the bean in the factory that created the bean.

4. Then, if the bean implements the *BeanFactoryAware* interface, Spring invokes the method *setBeanFactory()*, that provides the information about the factory to a bean instance.

5. Following, if the bean implements the *ApplicationContextAware* interface, Spring invokes the method *setApplicationContext()*, in order to inform the bean about the application context.

6. At this point, The *IoC container* invokes the method *ProcessBeforeInitialization()*, if the bean implements the *BeanPostProcessor* interface. Using this method, a wrapper can be applied on the original bean.

7. If the bean implements the *InitializingBean* interface, Spring invokes the method *afterPropertiesSet()*.

8. Finally, the bean instance is ready to be used by the application, and remains in the application context until this is destroyed.

9. In conclusion, if the bean implements the *DisposableBean* interface, Spring invokes the method *destroy()*.

**Beans scope**

Each time a bean is defined, the developer can control not only the various dependencies and configuration values that are injected into an object created from a particular bean, but also can define the *scope* for that bean. This possibility is very powerful, because it makes the code more flexible and prone to several scenarios. The Spring framework supports five possible scopes, three of which are available only in a web-aware application context.

- **singleton**: setting the scope of a bean as *singleton* means that the container, managing that bean, creates only one instance of it, and all the bean requests are going to receive always the same object. Also, any modification performed on the specific bean is reflected in all the references of that bean. This scope is the default one if no other scopes are specified.

- **prototype**: Opposing to the first one, the container returns a different instance of the bean, each time a bean request arrives.

- **request**: a bean, defined with the *request* scope, is strictly related to an HTTP request. In fact, for each HTTP request, the container generates a bean. This scope is available only in a web application.

- **session**: similar to the previous case, the *session* scope ties a bean to an HTTP session. The container creates a new bean each time a new HTTP session is generated. This scope is available only in a web application.

- **global-session**: as for the *session* scope, also the *global-session* scope ties the scope of a bean to a global HTTP session. This scope is available only in a web application.

### 2.1.5 Spring Modules

Spring is a very powerful framework, it can be used for several applications, and to manage different situations. The Spring versatility is due to its modular architecture. Springs is composed of twenty different modules, which can be arranged in six categories of functionality. The figure 2.3 shows the Spring modules organization, and, according to the application you are developing, only some of them can be considered, without worrying about the others. Let's analyse the six categories in which the Spring modules are grouped:

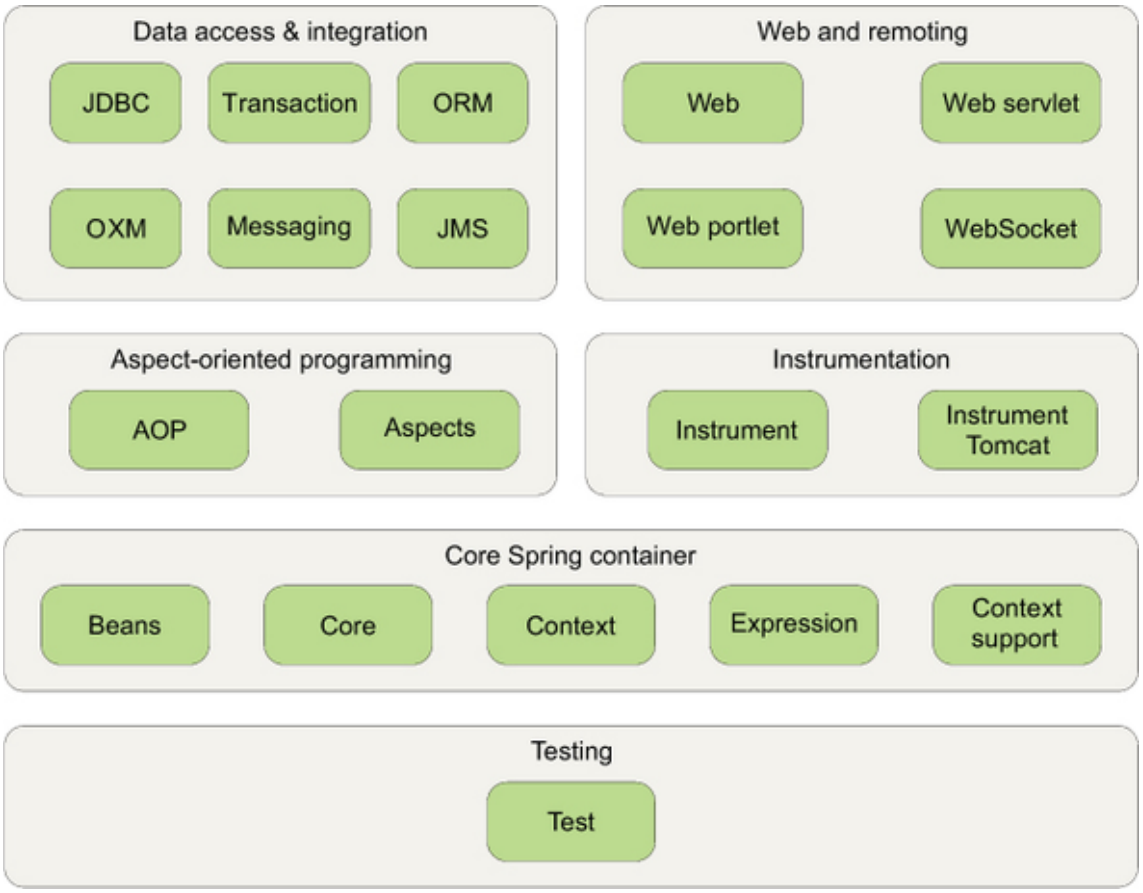


Figure 2.3: The Spring modules.

- **Core Spring container**: It is composed of the *Core*, *Beans*, *Context* and *Expression Language* modules. These modules represent the core of the entire Spring framework. The Core and Beans modules provide the IoC and Dependency Injection features. they are in charge of creating, configuring and managing the beans. The *ApplicationContext* interface is the focal point of the Context module. It provides an access point for any beans created and configured by the Core and Beans modules, it is built on top of them. In conclusion, the Expression module is in charge of providing a powerful expression language for querying and manipulating the beans. It supports the setter and getter methods, and many others, always used to manage the beans.

- **Spring AOP's module**: the aspect-oriented programming is a key functionality of Spring framework, in fact, it is supported by the AOP modules. These modules provides several methods to develop your own aspects for your Spring enabled application.

- **Data access and integration**: this layer is composed by the *JDBC*,*ORM*, *OXM* and *JMS* modules. It provides an easier integration of your application with the most used database. The JDBC module, with the DAO one, simplify the need of getting a connection, creating a statement, processing a result set, and then closing the connection to a database. For those

who prefer using an object-relational mapping, Spring provides, also, the ORM module. Instead the JMS and OXM modules provide, respectively, support for the Java Messaging Service and for the Object/XML mapping implementations. In conclusion, this layer should simplify the integration and access to several common databases.

- **Web and remoting**: like many other frameworks, used to develop web-based applications, also Spring offers a strong support to the *Model-View-Controller* paradigm. This layer collects all the modules that allow the implementation of the MVC, with some additional features, in particular it provides remoting options for building applications that interact with other applications, including the capability of remote method invocation.
- **Instrumentation**: this category collects all the modules that provide support for adding agents to the Java Virtual Machines.
- **Testing**: developing applications means, also, following several good practices, and one of them is to provide some developer-written test used to test the application. Spring supports this good practice. The *Test*: module provides all the feature needed to develop tests for your Spring application.

### 2.1.6 Spring Portfolio

As mentioned many time during this description, Spring is an extremely flexible framework. The functionalities, just described, represent only the core of a greater and wide project. Spring offers a portfolio that allows the Spring programming model to cover several facets of Java development. The whole Spring portfolio includes several frameworks and libraries, built on top of the Spring core, and that live intertwined the ones with the others. In the following, an overview of the main ones is presented:

#### Spring Boot

Spring was born with the purpose of making simpler and more flexible the developing of Java application, minimizing the developer's jobs and eliminating some boilerplate code, needed in some situation. On top of this idea, the *Spring Boot* tool was developed. Its aim is to simplify even more the creation of Spring application, automatizing as much as possible this process. Spring Boot can eliminate most, and in many case all, Spring configuration, providing a ready application, with all dependencies already set.

#### Spring Data

The *Spring Data* framework makes easier the integration of a Spring application with any kind of database. Until a few years ago, the use of relational database was an anchor of any Java application that needs to integrate

data persistence. But, in the last years, it become clear that storing informations as rows and columns is not always the best way, so that the use of non-relational database, that offers a new way of storing data, has become a practice spread in the Java community. Spring offers full support to relational and non-relational database, like MongoDB or Neo4J. Spring Data makes easy, also, the use of data access technologies, of map-reduce framework and of cloud-based data services.

### Spring Social

Social networking is a reality that is born some years ago, and now widespread in the daily life of every person. Nowadays, every available application integrates several social networking sites, such as Facebook, Twitter or Google. So that, Spring releases a new project with the purpose of making easier the social networking integration: *Spring Social*. It helps the developer to make a Java application more "social" and "connected", exploiting REST APIs.

### Spring Web-MVC

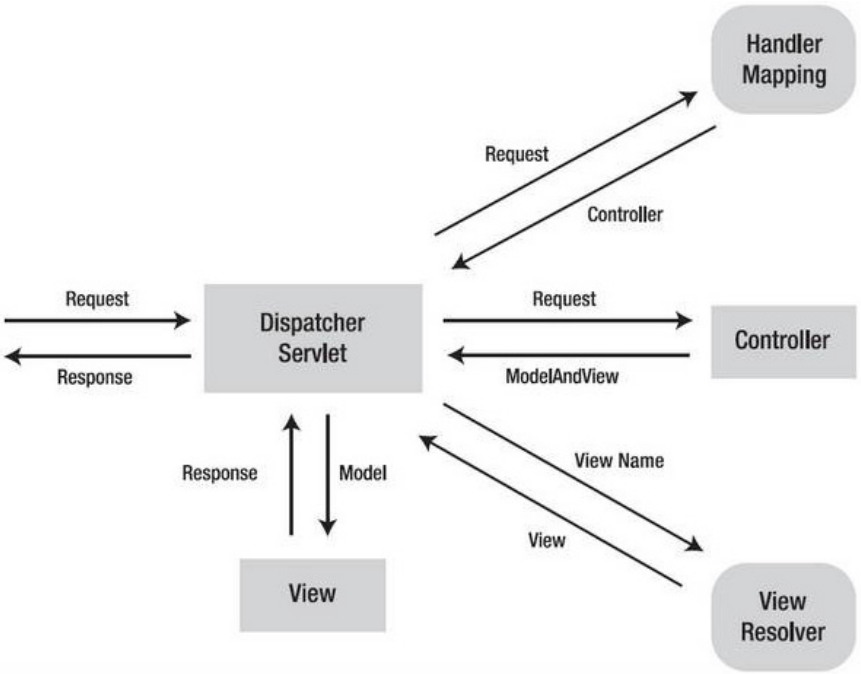


Figure 2.4: The Spring Model-View-Controller.

The Spring Web MVC framework was released to support all web applications that want to exploit the power of Spring and to implement the MVC paradigm. According to the Spring principles, the MVC pattern improves the flexibility and the loosely coupled features, separating the different aspects of an application. In order to better understand how the Spring Web MVC framework works, lets analyse the MVC paradigm. MVC stands for *Model-View-Controller*, these are the three base components of a web application, each of them has specific tasks.

- **Model**: it is in charge of accessing and maintaining the data needed to the business logic.
- **View**: it creates the user interface that shows the data to the final users (i.e HTML pages, mobile interfaces, etc.).
- **Controller**: it implements and executes the application business logic, connecting together the above components. it receives the input from the users, takes the data from the models, executes its logic and, finally, gives the modified data to the View in order to show the result to the user.

Spring Web MVC integrate the approach just described, and the figure 2.4 shows how the framework works. The *DispatcherServlet* is in charge of handling all the incoming HTTP requests and responses, it is the front controller of the whole application. Each time a request arrives, it is forwarded to a specific controller. A web application can implement more than one controller, so that, the *DispatcherServlet* brings into play the *Handler Mapping* in order to dispatch the request to the correct component. The *Controller* processes the request invoking a service method. Each controller disposes of several service methods, any one of which implements a specific aspect of the business logic. The Controller generates a *ModelAndView* object that contains the data to be shown to the user and the name of the view that should be used to show these data. This object is sent to the *DispatcherServlet* that contacts the *View Resolver* in order to receive the view of interest. Finally The *View* component receives the model from the *DispatcherServlet* and merges the data with the view in a proper format. In conclusion the front controller can respond to the request sending back the requested data organized in a proper way.

#### Spring Security

Security is a critical aspect of many applications. *Spring security* is the Spring solution to authentication and access-control issues. It is powerful and highly customizable, in order to meet all the developers customization requirements. *Spring security* is suited for Spring-based application, and it provides authentication and authorization mechanisms at both the web request level and at the method invocation level, exploiting dependency injection (DI) and aspect-oriented techniques. To make secure requests and to restrict access at the URL level, Spring Security uses servlet filters, instead, to secure method invocations, it uses Spring AOP.

## 2.2 MongoDB

Gathering information is one of the most important task in a web based application. Nowadays there are so many types of data and as many possibilities of storing them. Collecting and storing data is not, always, an easy task, and, also, making the correct choice could be cumbersome. Choosing a suited

database, according to the application needs, ensures correct data collection and easy information retrieving. So that, in this section, the relational and non-relational databases model are analysed, describing their main features, and why it is better to use one solution in contrast to another, focusing on pros and cons of both possibilities. In conclusion, MongoDB is analysed, pointing out the main functionalities and explaining why it was chosen as backing storage for Mak07.

### 2.2.1 Relational Databases

In order to better understand the terminology related to the database world, should be clarified that a *database* is a collection of related data, organized in a certain way, instead, a *DBMS*, that means *Database Management System*, is the interface that makes possible the dialogue among database and application. So, a DBMS is a software component, the core of a database infrastructure, that manages three important things: the data, that is a collection of facts and figures that can be processed to produce information; the database engine, that allows data to be accessed, locked and modified; the database schema, that defines how the data are stored. The main purpose of a DBMS is to store data in order to make easier retrieving, manipulating and producing information.

The top most used databases technology is the *Relational* database. It exploits a *RDBMS*, that stands for *Relational Database Management System*. It is a particular type of DBMS, because it uses a *Relational data model* to store and process data. In a relational database, the **table** is the most common and simplest form of data storage. It is organized in **rows** and **columns**. Each row contains the information of an *entity*, that is a real world object that can be easily identified through attributes, which describe the properties of an entity, and that are represented by columns. Furthermore, tables model the important concept of *relationship*. It is a logical connection among entities belonging to different tables, represented by external references inside attributes entities.

A database should be, also, equipped with a query language that can be used to manage the database instances. **SQL**, *Structured Query Language*, is the programming language for relational databases. It is based on the relational algebra and relational calculus principles, and it is composed by two main components: the data definition language and the data manipulation language. The DDL collects the set of functions used to define and to manage database tables and schemas, instead the DML defines all the CRUD commands used to modify the database instances.

In conclusion, the SQL language allows to execute single logical operation on database instances. A group of operations is defined as **transaction**. It is a sequence of tasks representing a unit of work in a DBMS, that must maintain the *ACID* properties.

- **Atomicity**: this ensure that a transaction must be treated as an atomic unit, so, all its operations are executed as they were one.

- **Consistency**: if a database was in a consistent state before the execution of a transaction, must be in a consistent state after the execution of a transaction as well.
- **Durability**: the database must hold all the latest information even if the system crashes. If a transaction updates a chunk of data and finishes with a commit, the updates should not be lost even if they are not persistently stored in the database and the system terminates abnormally. So, the data are going to be updated when the system springs back into action.
- **Isolation**: it ensures that a transaction, that is executed simultaneously and in parallel with other transactions, is performed as if it is the only transaction in the system. So, the execution of a transaction doesn't affect the execution of the others.

These group of properties should be respected by transactions in order to ensure accuracy, completeness and data integrity.

#### Relational databases limits

Technology is, always, in evolution, and new solutions, producing lots of heterogeneous data, are born. The relational databases have been a standard in these last years, but, today, the continuous changes have revealed the limits of this "old" solution.

- Relational databases are not designed to handle changes. They are born in a context in which only little and simple data should be stored persistently, and, even if several improvements are developed, they are not suited, any more, in a context in which changes occur frequently and faster. In a relational database, even a simple change, like adding or replacing a column in a table could be a long and expensive task. This is due to the architecture of the relational databases. Storing a particular data means designing a suited data model, or schema, that involves many resources. It is a long, complex and expensive process, that is not suited in an environment where new heterogeneous data should be stored persistently as soon as possible. An hasty solution could generate waste of resources and querying data could become difficult.
- Relational databases are not designed for heterogeneous data. They require pre-defined schemas before loading data, they are suited to store structured data. But, the growing amount of data variety becomes a problem for relational databases, because any change in a database schema, in order to handle a new data type, could be cumbersome. As described above, it is complex and expensive adding columns, or modifying schemas, in order to store new information, so, this is a limit of relational databases.
- Relational databases are not designed for scale. In the last years, the amount of desktop, tablet and mobile devices is increased, and this has

generated a growth in the amount of heterogeneous data to be stored. Handling this reality, means improving database scalability and elasticity, in order to add the capacity of storing more data. Achieving these two qualities is a huge challenge for a relational database. They are designed to live on a single server in order to maintain the table integrity and avoiding the problem of distributed computing. So that, an *horizontal* scalability can't be applied, but the *vertical* scalability solution should be adopted, buying new powerful and performable hardware into which migrate the system.

- Relational databases are a mismatch for modern App development. Modern applications are built using object-oriented programming languages such as Java, JavaScript, and C#, so that, data structures are treated as objects, containing data and code, and this way of handling data is different from how the RDBMS handles data. The *object-relational mapping*, also known as ORM, that is in charge of creating a mapping between objects and RDBMS data, could be a solution to the problem, but exploiting the ORM means loosing performance and acquiring opportunities for buggy code.

All theses limits describe how the relational databases are not suited any more for the actual environment, so that, new solutions should be considered in order to overcome the difficulties.

### 2.2.2 Non-Relational Databases

> *"This notion of thinking about data in a structured, relational database is dead."*
>
> — Vivek Kundra, Former CIO of the U.S. Federal Government

This statement, according to the description done in the previous section, well describes the current situation of relational databases, and how they are not suited any more to tackle the new challenges due to technology evolution. So that, new possibilities should be considered in order to overcome the relational model limits. In the last decade, the *non-relational* databases represented a solution to data variety and growth, especially due to the rise of Big Data applications. This category of database is, also, known as NoSQL databases, because they are disjointed from SQL language, and they were designed to overcome the relational model limits. In fact a non-relational database is more flexible and scalable than its counterpart. A relational database requires a predefined schema, instead, a NoSQL database offers a more flexible schema, designed in such a way that changes or updates could be handled easily, so it is prone to work in progress changes. But the simplicity of this solution generates the loss of data integrity quality, which is supported by RDBMS, and that, now, must be handled by the application exploiting the non-relational solution. Furthermore, these database are designed to manage unstructured data, i.e. data that are not suited to be stored in rows

and columns. The horizontal scaling, that was not possible with relational databases, is, instead, applicable for the NoSQL one, indeed they take advantages from horizontal scaling. Finally, as mentioned at the beginning, the SQL language can not be used to manage a NoSQL database. It is complex and not suited for a non-relational environment, that, instead, allows the possibility to develop custom APIs to read and write data. In conclusion, NoSQL database represents a important novelty in data storing, widely accepted from the developers community. Today, several options exist, each of them has its main features and weaknesses, that make each different solution suited to satisfy the different developers needs. MongoDB is a non-relational DBMS that gathers most of NoSQL database features, and that has a strong community that, every day, uses and supports it. Since that, MongoDB is the backing store solution adopted to support the Mak07 system, but before deepening its description, an important theorem should be treated.

### CAP theorem

The non-relational databases take advantage from the horizontal scaling, due to the fact that they live in a distributed environment. The CAP theorem, formulated by the professor Brewer from the California University, tries to make aware the developers about the fact that in a network shared data system, there is a fundamental trade-off between *Consistency*, *Availability* and *Partition Tolerance*. The theorem states that it is not possible to provide simultaneously these three qualities.

- **Consistency**: this quality guarantees that every node in a distributed cluster returns the same, most recent, successful write, so, each client has the same view of data.
- **Availability**: every read and write request has to be performed in a reasonable amount of time by each non-failing node.
- **Partition Tolerance**: the system continues to operate despite an arbitrary number of messages being dropped by the network.

At the same time, only two of the above qualities can be ensured in a distributed data system. As a consequence of this statement, the systems can be categorized into three categories: CP, CA and AP, according on two of the three qualities. Furthermore, relational databases are designed on top of a transactional model, ensuring the four principles of the ACID model. These principles are not suited for a non relational environment, so, new principles were formulated as alternative for the NoSQL databases. The *BASE* model, in opposition to the ACID one, sets out three new principles:

- **Basic Availability**: NoSQL databases spread data across many storage systems with an high degree of replication. If a failure destroys the access to a segment of data, the availability is always guaranteed, according to the CAP theorem.

- **Soft State**: The state of the system could change over time, even during times without input. The data consistency is a developer problem, and should not be handled by database.

- **Eventual consistency**: The system will become consistent once it does not receive input any more, and the data modifications are going to be propagated to every node of the distributed data system. The consistency is going to be reached at some point in the future, but no guarantees are made about when this will occur.

### 2.2.3 Types of NoSQL Database

The NoSQL databases provide all the quality required by the modern application: high performance, good horizontal scale and flexibility. Another important things that they have in common is that they do not follow the relational model, in fact non-relational databases are, typically, divided in four categories, according to the way of storing information:

- **Key-Value stores**: this category is the simplest one. Every item in the database is stored as a key-value pair, in which the key is the name, and the value collects the item information.

- **Wide-column stores**: the data are stored together as columns, instead of rows. This storing method is optimized for queries over large datasets.

- **Document database**: it is similar to a key-value database, but the key stores an unique identifier, and the value stores a document. A document is a complex data structure that contains, any different key-value pairs or nested documents. MongoDB is the most popular of these database.

- **Graph database**: these databases are used to store information about networks. The final representation of the data relationships is similar to a graph.

### 2.2.4 MongoDB key features

MongoDB is the most popular of all NoSQL databases, because it preserves the best features of relational databases while incorporating the advantages of NoSQL. MongoDB is a database management system designed to rapidly develop web applications and internet infrastructures. The data model and the persistence strategies are built in order to guarantee high read-write throughput. MongoDB has a schema less storing model, in fact it stores information in documents rather than rows. This is, perhaps, the biggest reason why developers use MongoDB.

A database is defined in large part by its own main features. MongoDB has five main functionalities that make it able to tackle the major today application needs:

- MongoDB provides high performance data persistence. In fact, it supports embedded data models, in order to reduce the Input/Output operations that are less than the relational ones, as well as indexes, in order to speed up queries.

- MongoDB has a rich query language, in fact it supports all the major CRUD operations, as well as the data aggregation, text search and geospatial queries.

- the auto replication feature of MongoDB ensures high availability. In fact, it provides automatic failover mechanism, and data restoring mechanism, due to server fails.

- MongoDB provides, also, automatic scaling functionality. this is ensured by the horizontal scalability quality, that is part of its core functionalities. The technique of automatic *sharding*, instead, distributes data across a cluster of machines, while *replica set* provides eventually-consistency for low latency deployment reads.

- finally, MongoDB supports multiple Storage Engine. These are in charge of storing data in form of document, and how the data are saved on the disks.

### 2.2.5 MongoDB Data Modeling

The data modeling is one important aspect that has to satisfy the needs of the application, the performance of the database engine and the retrieval pattern. Designing a data model means considering the application usage of the data, i.e. how the data are queried, updated, processed. As mentioned above, MongoDB has a document oriented data model. A document is the smaller memorization unit in MongoDB. Documents representing similar data are grouped together in *collections*, that are stored, persistently, in the database instance of MongoDB. A document is a data structure composed of field-value pairs. MongoDB stores document as *BSON*, Binary JSON, objects, which are binary encoded JSON like objects.

Another important quality is how the relationships among documents are represented. MongoDB supports two ways of representing data relationships:

- **Embedded Data** model. It represents the relationships among data by storing data in a single document structure, called embedded document. In MongoDB, It is possible to embed documents inside the fields of another document. This model allow application to retrieve and manipulate data in a single database operation. It is, also, used to represent a one-to-many relationship in which the outer document is the "one", instead the inner documents are the "many"

- **References** model. It stores the relationships among data exploiting links or references from one document to another. Due to the fact that

each document has a unique identifier, this is used to create the connections among documents. This approach is similar to the foreign key of the relational database.

The Embedded data structure of MongoDB documents introduces a *denormalized* data model. Exploiting this model, MongoDB is able to perform write operation in an atomic way, according to a single document. In fact, even if the document is composed of several embedded data, all the information are combined together in order to make atomic the operation performed on the document itself.

### 2.2.6 MongoDB Scaling feature

High performance and good scaling quality are two of the key features offered by MongoDB. These are ensured through the *Replication* and *Sharding* functionalities. These, also, guarantee the availability of data.

#### Replication

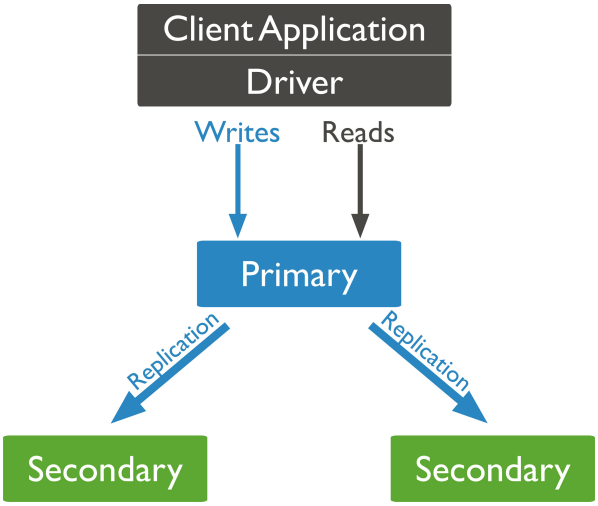


Figure 2.5: MongoDB replication mechanism.

Replication is the practice of keeping identical copies of data on multiple servers in order to maintain the application always on and the data safe. This solution provides redundancy and high availability, ensuring, also, fault tolerance against loss of data. In MongoDB, this is done by the *replica set*. A replica set is a group of processes, named *mongod* (i.e. an instance of MongoDB), in charge of maintaining the same data set. A replica set is composed by data bearing nodes and, optionally, by one arbiter node. About the data bearing nodes, we can distinguish the primary node and the secondary nodes. There is only one primary node, that is in charge of receiving the read and write operations from clients. It applies the received operation on the its set, and then, propagate the changes to the secondary nodes. The secondary nodes are not visible by the clients, that, always, contact the primary node, as the figure 2.5

shows. The primary node is in charge, also, to maintain a Log, named "oplog", in which it saves all the performed operations, that are going to be replicated on the secondary nodes, in order to have all the same updates on all the nodes of the replica set. However, could happen that the primary node is unavailable, so, a secondary node can be elected as new primary node. Optionally, the replica set can contain one or more arbiter node. It is a instance of mongod, as the other nodes, but it doesn't maintain any data. If it exists, it is in charge of managing the election of the new primary node, exploiting the mechanism of *heartbeat*. It is a message exchanged among replica set nodes in order to make aware each node of the existence of the others.

Each secondary node reflects the operation executed by the primary one in as asynchronous way. It applys the updates only after the primary one, however it is possible to force the system to execute suddenly the synchronization, but, obviously, this decreases the performance of the operation. By default, clients read from primary node, but they can specify that want to read from a secondary node. This means that clients can read non up to date data, due to the fact that the updates performed on the primary are reflected asynchronously on the secondary.

### Sharding

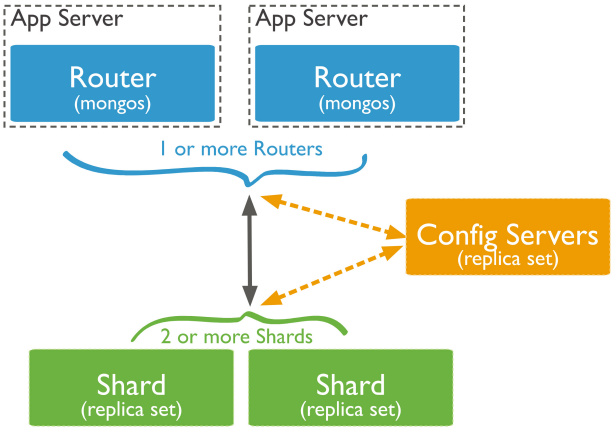


Figure 2.6: MongoDB sharding mechanism.

The Sharding is a mechanism used in a distributed environment to distribute data across multiple machines. This is an example of how MongoDB allows the horizontal scaling. MongoDB uses this mechanism, also, to support the deployments with very large data sets and high throughput operations. In fact, some database systems, with a large data sets, need machines with high CPU capacity in order to ensure powerful data process. The Sharding mechanism is based on the *sharded cluster* architecture, shown in the figure 2.6, that is composed by:

- **shard**: each shard is, essentially, a replica set that holds a portion of data of the entire database. In this sense, a MongoDB collection could divide

in several multiple part, each of them stored on a different shards. To access the full collection must be retrieved all the part from the shards.

- **mongos**: more than one mongos process can exist. Each mongos process is identified as a router, and it provides an interface between client applications and the sharded cluster, in order to make able the clients to query the cluster.
- **config servers**: It stores meta-data and configuration settings for the sharded cluster. It knows the data stored by each shard.

In order to distribute the data of a single collection among the shards, MongoDB uses the *shard keys*. A shard key consists of an immutable field or fields that exist in every document of the collection. Making the correct choice about the shard key is very important because it affects the performance, efficiency and scalability of a sharded cluster.

## 2.3 REST Architecture

The term **REST** was coined by Roy Fielding in his Ph.D. dissertation. It stands for **RE**presentational **S**tate **T**ransfer, and it is an *architectural style* for designing loosely coupled applications over a HTTP connection, that is often used in the development of web services. The REST architecture does not define any rule that should be followed in order to make an application compliant with this style, but it wants only give an high level guidelines on how a well-designed web application should behave. So, the REST style principles have only the purpose of creating a design pattern for web based application. The REST architecture is based on the communication among clients and servers. The clients send requests that are received and processed by servers, that produce responses that are sent back to clients. Requests and responses are built on top of the concept of transfer and represent resources. A resource is an object with a type, a status and associated data, and it has relationships with other resources. Each resource is identified by an URL, and has some methods that can modify its status. Finally, each request points a resource and a specific method of it.

REST defines six architectural constraints which make any web based application really compliant with this pattern.

- **Uniform interface**: This constraint states that should be defined an interface among server and clients in order to decouple the architecture of these two actors. So, clients and server can both evolve separately. The interface is composed by APIs. They are endpoints (URL), that identify the system resources and the operation that can be performed on them.
- **Client-server**: As mentioned, client and server evolve separately. A client should know only the APIs through which can contact the server. The flow information starts from the client that sends a request to the server, that processes it and sends back a response.

- **Stateless**: This is a very important constraint. It means that each client request should carry with it the state information necessary to be processed correctly by the server. In fact, the server does not maintain any state information about the clients that make requests.
- **Cacheable**: Caching brings performance improvement for client side. So REST resources should be declared, when possible, cacheable, in order to make the clients aware on the fact that some resources can be requested only one-time.
- **Layered system**: REST allows you to use layered system architecture. So, at any time, a client can't tell if it is connected directly to the end server or to an intermediate one. This approach optimizes performance of communication and makes load balancing possible.
- **Code on demand**: This means that the server might deliver executable code to the client, such as JavaScript code that is going to be interpreted by the client itself.

## 2.4 Conclusion

Mak07 is a project that is evolved along several implementation phases. It started with, only, a mobile application, and now has a strong back end structure. It will continue to evolve, so, according to this prerequisite, the chosen technologies represent the correct way of supporting it. The framework Spring, that follows the REST pattern, guarantees loosely coupled and easily testable code, Instead, MongoDB increase, not only the scalability quality, but also the possibility to store new unstructured data. All these qualities ensure a strong environment that allow an easy growth for Mak07.

# Chapter 3

# Requirements and Design

The chapter starts with an overview of the game, with a short description of the main feature. Then, the attention is focused on the requirements of the Tournament feature. Therefore the design is analysed, focusing on the server side business logic, that is the objective of this elaborate, looking, always, to the client side, in order to make the communication easy and consistent. Some special cases are also analysed, in order to understand which are the possible real situations the system should deal with in a real environment. Finally, the support architecture is described, with a particular attention to the server structure and used tool.

## 3.1 The Game

The starting point of this master degree project is the game called Mak07, it is a mobile game, and it is already available on Google Play Store and App Store. The game is based on a simple idea, combining together 7 numbers, called "schema" (Figure 3.1a), exploiting the arithmetic operations, sum, subtraction, multiplication and division, in order to obtain a final result equal to zero. Some constraints are inserted in order to do not make harder the management of the game, like the impossibility to exploit negative numbers, or make multiplication by 0. Each time a schema is solved, a new one appears in the screen. The user has two minutes to resolve the highest number of schema in order to increase his partial score. Each schema has several solutions, and according to which operations are used to obtain 0, the partial score could be different. At the end of the two minutes, all the partial scores obtained following the resolution of a schema are summed together, considering also bonus points, to get the game final score. As mentioned, The game is based on a simple idea, but at the same time makes you wonder on which is the best move to do in order to make the highest score, in the shorter possible time.

The game offers a complete multi-player experience, based, essentially, on the challenge modality (Figure 3.1b), in which two users challenge each other to make the highest score. The game requires an initial registration phase, that may take place through Facebook, Google, or the classic username and password, in order to make possible the distinction of the users connected.


(a) Mak07 schema. (b) Challenges page. (c) User profile page.

Figure 3.1: Mak07 game screenshots

Each user has its own profile page (Figure 3.1c), in which are shown some user statistics, like the experience collected and the number of challenges won. Also, to support the multi-player structure, a ranking mechanism is implemented in order to gratify the best players and to encourage to climb the classification.

In conclusion, the game experience tries to involve the player attention and concentration in order to resolve the game schemata as quick as possible, obtaining the requested result, and to encourage the competitiveness among players thanks to the ranking mechanism.

## 3.2 Requirements

As described, the game structure is suited to short, few minutes, but intense interaction, due to the challenge modality (multi-player) and the training modality (single-player). At the same time, the game, thanks to the multi-player challenges and the ranking mechanism, is suited also to longer interaction, in which the user tries to acquire more experiences, improve his skills to climb the classification.

In order to support longer interaction and to attract users to play more the game, it was thought to introduce a new important feature that extends what the game offers, the Tournament feature.

### 3.2.1 Requirements analysis

The tournament feature should support the social and multi-player experience of the game. The main aim of the feature is to gather a group of players that want to challenge each other until one winner remains. A tournament is composed of several phases, the creation phase, the registration phase, and the round phases. For each phase, each user can perform a limited number of

actions, and, in particular, during the round phases, challenges among tournament participants are performed. the last challenge of the last round proclaim the winner.

According to this general description, the tournament feature has to offer some main functionalities in order to allow the possibility to create and play tournaments. Each player must be able to create a new tournament, defining some tournament attributes, like the name or the number of participants. Then, each user should be able to check if there are available tournaments, to sign up to one of them, in order to take part and compete for the victory. According to Mak07 functionalities, that allow each user to have a list of friends that can be challenged or chatted, the possibility of inviting friends to a tournament has been introduced, at creation time and during the registration phase.

The system must also be able to maintain information about tournaments, like played tournaments, participants list, tournament challenges, round winners, in order to have an internal history. The system must guarantee the correctness of the operation performed, and considering the fact that the tournament is a feature of a bigger software, each operation must be consistent not only in the tournament environment, but inside the environment of the entire game. In conclusion, an interesting functionality that improve the user experience is the notification mechanism. The game was thought to live in a mobile environment, and the notification has been introduced to support the challenge feature, so, in order to make the user aware about the starting and end of each tournament phase, notification should be scheduled for the tournament feature too.

### 3.2.2 Use cases

In the following some main use cases, that describe two main functionalities offered by Tournament feature, are analysed.

#### Create a Tournament

- **Description**: The player selects the tournament section of the Mak07 application and creates a new tournament (Figure 3.2);
- **Actors**: Player, Database;
- **Preconditions**: The player must be logged into system;
- **Postconditions**: The player creates a new tournament;
- **Path**:
  1. the player select the tournament section;
  2. the player pushes the button to create a new tournament;
  3. the player sets the tournament name and the maximum number of participants;

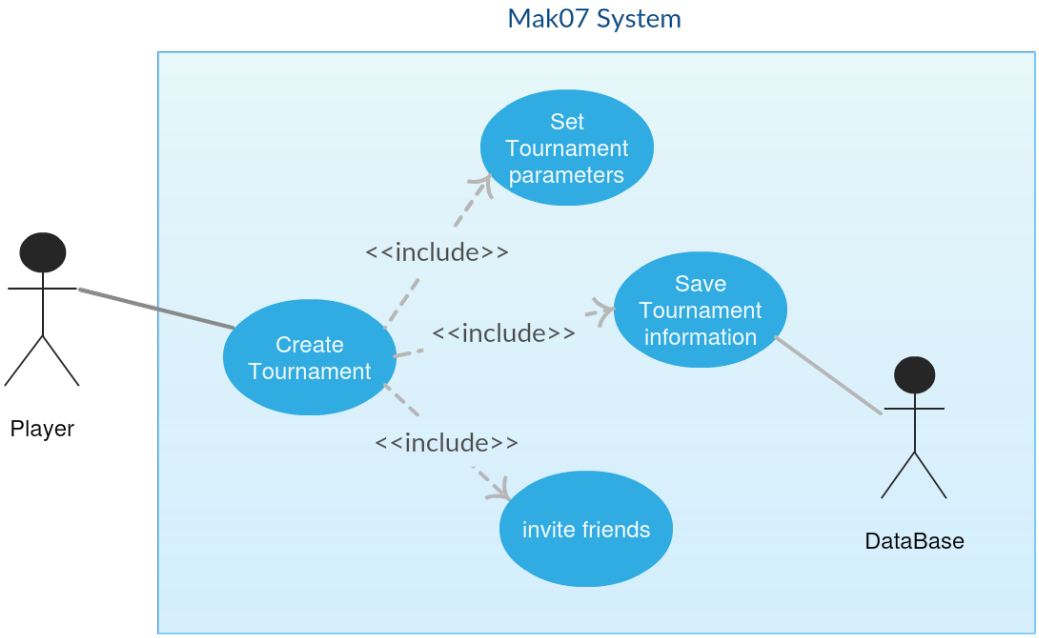


Figure 3.2: Use case - Tournament creation.

4. the player can invite some friends to take part to the tournament;
5. the player submits the request;
6. the system receives the request and save the tournament information into the db;
7. the system notify the invited players;
8. the tournament just created is available for players to sign up.

- **Frequency of use**: around once every 30 minutes.

**Delete a registration to a Tournament**

- **Description**: A player, after the registration to a tournament, can decide to delete his registration, before the tournament starts (Figure 3.3);
- **Actors**: Player, Database;
- **Preconditions**: The player is logged into the system and he is registered to a tournament;
- **Postconditions**: The player is no more registered to the tournament;
- **Path**:

1. the player selects the tournament section;
2. the system shows the information about the tournament to which the player is registered;

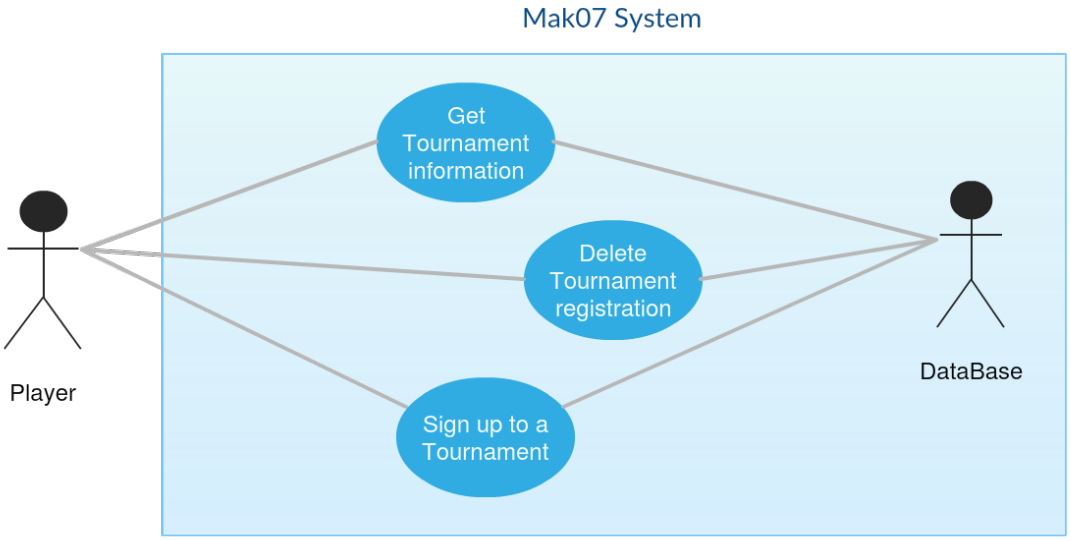


Figure 3.3: Use case - Delete Tournament registration.

3. the player pushes the button to abandon the tournament;
4. the system receives the request and update the tournament information in the Database;
5. the system shows the available tournament to the player;
6. the player can sign uo to a new tournament.

- **Frequency of use**: many times every quarter.

## 3.3 Design

Mak07 is a mobile game designed to entertain and to have fun an heterogeneous community, for this reason a client-server architecture was designed as main support architecture. The tournament feature should be design to leave in this environment. the purpose of this elaborate is analyzing and focusing the attention on the server side software, and due to this reason the following description is focused on the business logic implemented in the back end software to support the tournament functionalities.

During the design phase it was thought, as a correct programming style teach, to do not modify the functionalities already present in the back end software, but to integrate new functionalities that, at most, exploit the existing one.

### 3.3.1 Tournament model

According to the requirements, defined in the previous section, in order to maintain information about tournaments, an object of type tournament has been modelled, that, following, is going to be implemented. The tournament

Listing 3.1: Tournamnet object

```
{
  "id" :String,
  "creator" :String,
  "name" :String,
  "max participants" :Integer,
  "type" :enum,
  "invited friends" :[],
  "participants" :[],
  "participants number" :Integer,
  "challenges" :[],
  "state" :enum
}
```

object (Listing 3.1) has to collect the main information of a single tournament and it is composed by the following fields:

- **id**: Tournament unique id;
- **creator**: the user who creates the tournament;
- **name**: Tournament name, set by user creator;
- **max participants**: maximum number of participants to the tournament, set by the user creator;
- **type**: it defines the tournament type;
- **invited friends**: list of friends invited by tournament participants;
- **participants**: list of actual participants to the tournament;
- **participants number**: number of actual participants to the tournament;
- **challenges**: list of challenges belonging to the different rounds;
- **state**: it describes the actual tournament situation.

The main Tournament information are the *participants list*, that establishes which user takes part to the tournament, the *challenges* list, that collects information about each challenge played during the tournament rounds, and the *state* that represents the actual tournament phase.

### 3.3.2 Functionalities

As mentioned, the Tournament feature should fit the already available features, especially the challenge feature, because each tournament has several related challenges. The challenge phases are managed by the transition from

a state to another. This transition is handled by asynchronous updates performed on the challenge, due to participants actions. In a first time, it was thought to exploit the same mechanism to regulate the tournament phases, but, after, a new solution was found. Considering the challenge as single game, only two users have to be synchronized. After the creation of a challenge, the two participants can play the game when they want. This solution is not suited for a Tournament, where a bigger number of players have to be synchronized, furthermore a challenge belonging to a certain round must be played in a limited amount of time, otherwise, due to the fact that players could play their challenge in different instants of time, the round lifetime could be stretched till the slowest challenge finishes. For these reasons, it was thought to manage the tournament phases through **time intervals**. As well as for the challenges, a *state* field is used, in order to maintain the information about the actual tournament phase.

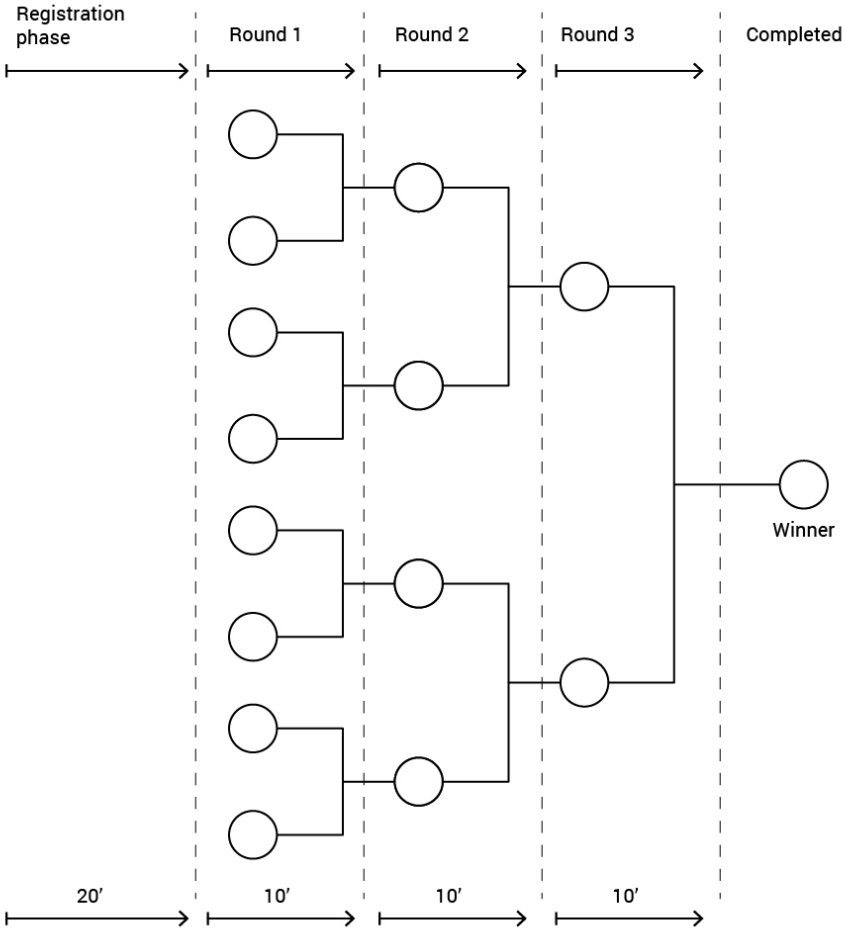


Figure 3.4: Tournament schema.

A tournament is composed by several phases, regulated by time intervals (Figure 3.4). The first phase is the registration one, in which the user can sign up to or invite friends to the tournament. The duration of the registration phase is 20 minutes, that is a reasonable time to collect the maximum

number of participants, considering, also, that a participant could delete his registration and other players can subscribe. After this phase, the tournament round starts. Each round lasts 10 minutes, so, each player, has 10 minutes available for playing his challenge, including also the two minutes needed to play the game, and if he doesn't play the game, the challenge is considered as abandon. Every time a round ends, it is very important that all the round challenges must be in a final state, they must not be upgradable any more, because, the winners of the challenges have to be considered in order to organize the challenges for the following round. The number of rounds depends on the maximum number of participants to the tournament. It was thought to limit the maximum number of participants up to 16 players. as a consequence, the maximum number of round can be 4. At the end of the last phase the tournament is put in a final state in order to be no more upgradable.

To support the tournament feature, several functions are designed:

- **Create a tournament**: each user logged into Mak07 should be able to create a new tournament, settings some tournament parameters, like the tournament name, the number of participants and the tournament type. The tournament creator has, also, the possibility to invite some friends;
- **Sign up to a tournament**: each user should be able to check if there are available tournaments and sign up to the preferred one;
- **Cancel a registration to a tournament**: a registered user to a tournament should be able to cancel his registration, only if the tournament is not already begun;
- **Invite friends to a Tournament**: a registered user should be able to invite his friends to the tournament to which is participating;
- **Tournament history**: each user should be able to analyse his tournament history;

In conclusion, the choice of exploiting time intervals to regulate the tournament state evolution is correct, because a tournament final state must be reached in a limited amount of time in contrast to the final state of a challenge that could be reached after a long time. Since there are these differences, during the development, it was necessary to adapt the challenges mechanism to the tournament one.

### 3.3.3 Special cases

During the design and the development phases, that is going to be analysed in more detail later on, relevant real situations are considered, that could happen in the several tournament phases:

- At the end of the 20 minutes allocated for the registration phase, the number of players could not have reached the maximum number of participants. So a suited number of players must be added to the tourna-

ment in order to allow the beginning, otherwise the tournament can't start;

- An attempt to sign up to a tournament after the end of the registration phase, must be rejected, and, at the same time, an attempt to delete a registration after the end of registration phase must be discarded too;
- An user can not have more than one active tournament at the same time;
- A challenge, at the end of a certain round, can be in a non final state. A challenge could be played by player 1 or player 2 or can not be played by any of them. These situation must be managed, bringing the challenge to a final state;
- As a result of the previous situation, a tournament phase can have a number of winners less than expected. Therefore, a tournament branch could die before the tournament end, and also a tournament can finish before all the expected phases have been played;
- If a player has to compete with the winner of a not played challenge, he directly passes to the next phase;
- A tournament could last a less number of rounds with respect to the normal behaviour, if there is one winner at the end of a certain round;
- If the last round or an intermediate one finishes with no winner, it means that nobody has won the tournament.

As mentioned, all these cases have been taken into account during the development phase, and a suited business logic has been inserted in the code to tackle these special situations.

## 3.4 Architecture

As described in the previous sections, the game Mak07 has been developed to fit a mobile environment, in particular on Android and iOS operating systems. Nowadays, the majority of applications, that are out there, need to connect to the internet in order to operate properly [4], and Mak07 is not an exception. Due to these considerations, it was thought to use a client server architecture (Figure 3.5). The server back end software exploits the *REST architectural style* to organize the functionalities, that are exposed to the outside through suited API, contacted by the clients with HTTP methods. The API invocation is the main method exploited for the client server communications, but, also the Websocket mechanism is used to keep track of the users connected to the system, the players who have the game application in execution on theirs mobile devices, but they are not playing any challenge. In conclusion, to manage the notifications it was thought to exploit the *Google Firebase* notification mechanism.

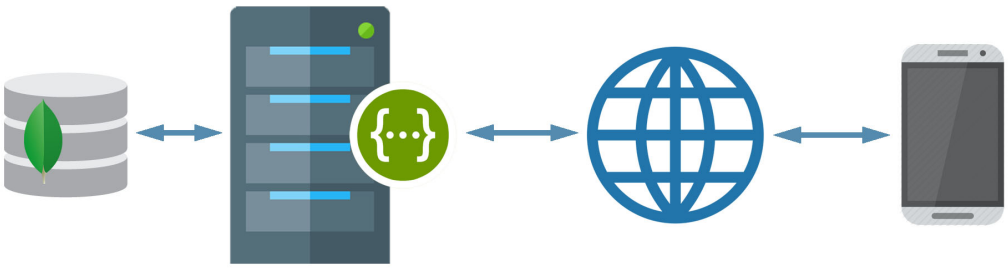

Figure 3.5: General architecture.

### 3.4.1 Server architecture

The server is accessible through the registered domain *game.mak07.com*. The operating system installed on the machine is Ubuntu 14.04, on which several components are virtualized by using a technology, born in 2013 but now widespread, called *container*. The *Docker* tool is exploited to manage and deploy containers. Thanks to containers and Docker, it was possible to build the server architecture needed to support Mak07. It was thought to create four main containers, in order to virtualize the four main components used by the system:

- One container manages the back end software, containing the logic of the entire system. The web logic runs on a Tomcat web container. To make everything lightweight, The Spring Boot application is exploited. Thanks to Spring Boot, we are able to generate a single file *jar* containing all the services of the server logic, that are in charge of handling all Mak07 features, and the Tomcat web server, that makes executable the developed software. This container also exposes to the outside the API and the Websocket endpoint that the clients can contact. Docker is responsible to forward the incoming requests on the real machine to the correct container, which contains the *endpoint* of interest.

- Another container includes the administrator panel web application. As the previous container, also this one exploits the Spring Boot application to have in a single jar the web application and the Tomcat web server. this container hasn't endpoints directly exposed to the outside, but it is accessible from the main container, that includes the game logic. The administration panel is in charge of making statistics about Mak07 players and usage (i.e. number of users, number of challenges, etc.), reading the information from the game databases. This panel is accessible only by the game administrators.

- A further container manages the database instance. It was chosen to exploit a non relational database to support the game features, so, a suited choice is MongoDB. All the game information (i.e. the users information, the challenges information, etc.) are stored persistently in the database. A suited image of MongoDB, that has to be run in containers, is downloaded from Docker Hub, that is the official Docker repository.
- A last container handles the RabbitMQ tool. RabbitMQ is a message broker, in charge of sending the output messages to the correct messages queue, and forwarding the input messages to the correct endpoint. As well as the database image, also the RabbiMQ image is downloaded from Docker Hub.

In conclusion, it has to be specified that all these containers should communicate with each other, and the Docker tool makes it possible. The server logic should communicate with: the database, in order to read and write persistent information, the message broker, in order to manage the messages queues, and also the control panel, in order to forward the administrator requests. Instead, the container deployed for the control panel should be in contact not just with the main container, but also with the database, in order to get information that should be shown to the administrator.

# Chapter 4

# Development of the Work

Inside this chapter, I'm going to analyse, in more detail, the implementation of the back end business logic. As first step, the main models, used inside the tournament feature, are analysed, focusing on the main data collected. Then, the two most important scheduled tasks, implementing part of the tournament logic, are described. Finally, particular attention is given to the analysis of the synchronization mechanisms adopted.

During the development, *good programming practices* have always been kept in mind, in order to write as much as possible the best code.

- To enhance the *readability* quality, it was used code indentation, short but descriptive variable, and function names. It was taken into account to limit the maximum line length and to organize the source code in blocks. Also, a right number of spaces and comments were inserted in order to make the code easy to understand.

- The *maintainability* quality was considered in order to make easy as much as possible future code changes, that could be done due to defect and fault corrections, or efficiency improvements.

- Developing a back end software means giving particular attention on the *reliability* quality. The code execution should never stop in order to offer a continuous service, and should never generate errors or inconsistencies. Even if they are generated, suited actions should be performed to limit system crashes.

- *Efficiency* is another important quality to take care when web applications are under development. The code should be as much as possible responsive to do not waste user time. Efficiency means, also, not wasting system resources, avoiding memory leaks and removing unused data from database.

- The *Security* quality should be, also, considered because the system deals with sensible information, and a lack of security could cause catastrophic consequences on the users.

- Finally, a *flexibility* feature should be ensured, in order to make the system able to grow with feature improvements, adding new functionalities, and modifying the existing ones, to fit the users demands.

## 4.1 Models

This section is in charge of describing the main models related to the tournament feature. Special attention is given to the Tournament model and to the entities strictly connected with it. The information that each entity holds are going to be described. Furthermore, the relationships among the new entities with the existing ones are taken into account. Each model becomes a Json document, that is stored in the system database.

### 4.1.1 Tournament model

This model is the main one of the entire Tournament feature. It collects and preserves all the tournament information, both the participants data and the data related to the tournament rounds. The Listing 4.1 shows a completed Tournament document, stored in MongoDB. Each document has an unique String identifier, assigned by MongoDB, in order to distinguish the different tournaments. The document fields, according to the model defined in the design phase, are:

- **creator**: it stores the unique identifier of the user that is logged into Mak07 system, who creates the tournament.
- **name**: this is a tournament name, given by the creator. it could be not unique in the system, in fact the tournaments are distinguished by the identifier assigned by MongoDB.
- **maxParticipants**: it is an integer number storing the maximum number of participants which can take part to the tournament. It is set by the creator at creation time.
- **type**: it is an enum Java type, and it can have only two values, *PRIVATE* and *PUBLIC*. A tournament is created as *PRIVATE* if the creator wants to play only with its friends, other system players can't see and take part to private tournament. A *PUBLIC* tournament, instead, can be played by all Mak07 players. It was thought to limit, in a first time, the value of *type* field only to *PUBLIC*, and to allow the *PRIVATE* value, in a second time. The *type* field is, also, prone to future development, introducing new possible values.
- **invitedFriends**: this is the list of users identifiers, invited to the tournament. Users can be invited at creation time, by the creator, or during the registration phase, by other participants. An user can refuse an invitation, so his identifier is deleted from the list. The *invitedFriends* list can be empty.

Listing 4.1: Tournamnet document

```
{
  "_id" :ObjectId("5b10f99a36ee652d048a9f60"),
  "_class" :"backend.mak07.model.Tournament",
  "creator" :"5ae3063c36ee651f54741839",
  "name" :"Gold Tournament",
  "maxParticipants" :NumberInt(8),
  "type" :"PUBLIC",
  "invitedFriends" :[
    "5b0bbc7f850940351439df1a",
    ...
  ],
  "participants" :[
    "5ae3063c36ee651f54741839",
    "5b0bbc7f850940351439df1a",
    "5a0eb55e273167a41d1a2bd5",
    "5a0eb568273167a41d1a2bd8",
    ...
  ],
  "participantsNumber" :NumberInt(8),
  "challenges" :[
    {
      "challengeNumber" :NumberInt(1),
      "challengeId" :"5b10f9f436ee652d048a9f65",
      "winner" :"5a0eb55e273167a41d1a2bd5"
    },
    {
      "challengeNumber" :NumberInt(2),
      "challengeId" :"5b10f9f436ee652d048a9f67",
      "winner" :"5ae3063c36ee651f54741839"
    },
    ...
  ],
  "state" :"COMPLETED",
  "created_at" :NumberLong(1527839130329),
  "modified_at" :NumberLong(1527839130329)
}
```

- **participants**: this is another list that collects the participants identifiers. This list will never be empty, because it has, always, a participant, that is the tournament creator. An user can sign up to a tournament or delete his registration, causing the addition or deletion of his identifier from the list. It can be modified only during the registration phase, but when the tournament starts, it is no more modifiable, and it collects persistently the participants information. Its maximum size is established by the *MaxParticipants* field, and it can't grow over this value.

- **participantsNumber**: it is the number of enrolled user to the tournament. Its value changes according to the size of the *participants* list. It was thought to insert this field in order to make the business logic easier, and in particular to make some restrictive queries.

- **challenges**: this is a list of TournamentChallenge objects. This field defines a new document that is injected in the main one, the tournament document. The TournamentChallenge object is going to be described in details in the following paragraph.

- **state**: the *state* field describes the actual situation of the tournament. It is an enum Java type and it can assume the values *CREATED*, *IN PROGRESS 1*, *IN PROGRESS 2*, *IN PROGRESS 3*, *IN PROGRESS 4*, *COMPLETED* and *CANCELED*. At creation time, the *state* field is set to *CREATED*, and it maintains this state until the end of the registration phase. When the tournament starts, it is put in the *IN PROGRESS 1* state, that is associated to the round one. The following rounds have their corresponding *IN PROGRESS X* state. After the end of the last round, the Tournament state is set to *COMPLETED*, in order to bring the tournament in a final state. The *CANCELED* state is created to tackle particular situations that bring the tournament to finish before than expected. For instance, if a tournament, after the registration phase, doesn't reach the minimum number of participants to begin, it is put in the *CANCELED* state. This extra state is a final state. As mentioned during the design section, in the previous chapter, the tournaments are managed by time intervals. A time interval expiration causes an update of the tournament state. This mechanism are going to be deepened during the business logic explanation.

- **created_at**: it represents the timestamp, expressed in milliseconds, of the tournament creation time. This value ss set by MongoDB when the document is inserted in the database.

- **modified_at**: it describes the last modification time of the document. As well as the *created_at* field, it is set by MongoDB and it is expressed in milliseconds.

The list of participants keeps the information about the players. Starting from the user identifier, it is possible to retrieve the tournaments that the user has

taken part or he is participating. In conclusion, it has to be emphasised that inside the tournament model are used *external references*. In particular, the user identifier, that could appear in several tournament fields, is an external identifier of the UserPlayer model, that collects user information, like credential data and player statistics.

#### TournamentChallenge model

As mentioned above, the *challenge* field inside the tournament object is composed by a list of documents. The document injection mechanism is exploited. An object of type TournamentChallenge is directly stored inside the tournament document itself. This kind of object simplifies the business logic mechanism of tournament rounds transition. As we can see inside the *challenge* field of the listing 4.1, the TournamentChallenge document is composed by 3 fields:

- **challengeNumber**: this integer represents the challenge number. Each tournament has numbered challenges. In fact, according to the number of participants, it is possible to know which challenge belongs to which tournament round, i.e. if I want to read the information of the challenges of the second round of a tournament that has 8 participants, I need to read the information of the challenges number 5 and number 6.
- **challengeId**: it is an external reference to the challenge of interest. It is the unique identifier of the challenge document that collects all the relevant information of the challenge itself, like the identifiers of the two players, the schemata solved by player one and player two, the state, etc.
- **winner**: it represents the identifier of the challenge winner, among the two players. The winner identifier is updated by the system when the round, the challenge belongs to, finishes.

### 4.1.2 ActiveTournament model

This entity is designed to ensure the situation in which each player can have only one active tournament at a time. To enforce this, MongoDB functionalities are exploited. As shown by the listing 4.2, the main fields of the Active tournament object are: the **playerId** field that stores the player identifier, and the **tournamentId** field, that, instead, stores the tournament identifier to which the player is registered. Other minor information are stored inside the **created_at** and **modified_at** fields, that collect, respectively, the creation timestamp and the last modification timestamp. An important clarification has to be done about the definition of the *playerId* field. This field is defined as unique, in order to make sure that for each player, only one document can be created. This means that each time a player creates or signs up to a tournament, a document in the ActiveTournament collection is inserted, and an attempt to insert a new document with the same *playerId* generates an exception. This functionality of MongoDB is used to guarantee that each player has

Listing 4.2: ActiveTournamnet document

```
{
  "_id" :ObjectId("5b179c0a36ee6528901c4704"),
  "_class" :"backend.mak07.model.ActiveTournament",
  "playerId" :"5aab9b9736ee65072c327006",
  "tournamentId" :"5b179bf936ee6528901c4702",
  "created_at" :NumberLong(1528273930235),
  "modified_at" :NumberLong(1528273930235)
}
```

only one active tournament at a time. Finally, in order to speed up the search of a document of a specific player, an index is created on the *playerId* field.

### 4.1.3 TournamentPhases model

Listing 4.3: TournamnetPhases document

```
{
  "_id" :ObjectId("5b179c5336ee6528901c4706"),
  "_class" :"backend.mak07.model.TournamentPhases",
  "tournamentId" :"5b179bf936ee6528901c4702",
  "state" :"IN_PROGRESS_1",
  "endedChallenges" :NumberInt(2)
}
```

As the ActiveTournament model, also the TournamentPhases model is created to support the server application algorithm. The listing 4.3 shows a valid document. As it is possible to notice, few information are collected. The **tournamentId** stores the unique identifier of the tournament of interest, furthermore the uniqueness of this field is specified at definition time, to ensure that only one document can be inserted for each tournament. A **state** field collects the tournament state information, and the **endedChallenges** field counts the number of completed tournament challenges. This object is exploited to know at each time the number of completed challenges. The TournamentPhases model was designed to reduce the waiting time between two tournament rounds. As explained during the design section in the previous chapter, each tournament round lasts 10 minutes, but it was thought to reduce this time if all the challenges, belonging to a specific round, ends up before time expiring. How the TournamentPhases document is exploited inside the server business logic is going to be explained later in this elaborate.

Listing 4.4: Challenge document

```
{
  "_id" :ObjectId("5b11097136ee652d048aa0c7"),
  "_class" :"backend.mak07.model.Challenge",
  "player1" :"5a0dd4f427316794f22d52fd",
  "player2" :"5a0dd4f427316794f22d5300",
  "type" :"RANDOM_BOT",
  "seed" :NumberInt(928299316),
  "created_at" :NumberLong(1527843185763),
  "modified_at" :NumberLong(1527843209149),
  "expireAt" :NumberLong(0),
  "state" :NumberInt(4),
  "tournamentId" :"5b11091736ee652d048aa0bc",
  "lastModifiedBy" :"5a0dd4f427316794f22d5300",
  "firebaseIdPlayer1" :"...",
  "firebaseIdPlayer1_timestamp" :NumberLong(1527841567593),
  "schemesP1" :[... ],
  "firebaseIdPlayer2" :"...",
  "firebaseIdPlayer2_timestamp" :NumberLong(1527841567593),
  "schemesP2" :[... ]
}
```

### 4.1.4 Challenge model

In order to better understand the tournament feature, and how its implementation is perfectly merged with the available system, it is necessary to describe and analyse one important model, already existing, but strongly used inside the tournament algorithm, the Challenge model. The listing 4.4 represents a valid and complete challenge model. Each challenge has an unique identifier inside the Mak07 system, stored inside the **_id** field, and the information about the two players of the challenge, registered inside the **player1** and **player2** fields. A challenge is related to two players, but each player can have several challenges. A **type** field, instead, is exploited to describe which kind of challenge i'm dealing with. It can take 3 different values:

- *NORMAL*: the challenge is played by two players in which one invites the other, that has accepted the game.
- *RANDOM*: the challenge is created among two random players connected to the system.
- *RANDOM_BOT*: if a player wants to play a challenge, but no players are connected to the system, a bot player is selected to play the game, so a challenge of this type is created.

According to the tournament feature, all the challenges created for a tournament are of type *RANDOM* and *RANDOM_BOT*, because the players and

the bots, that took part to a tournament, are coupled in a random way in order to generate the games. The *RANDOM* type challenges couple, only, human player, instead *RANDOM_BOT* type challenges can couple human player with bot player, or bot with bot player, if no human players are available. Other relevant information are stored inside the fields **created_at** and **tournamentId**, that collect respectively the creation timestamp and the unique tournament identifier, if and only if the challenge belongs to a tournament. In particular this last field was introduced due to the implementation of the tournament feature, in order to keep track also on the challenge side the tournament information. As in the previous descriptions, the *player1*, *player2* and *tournamentId* contain an external identifier of the documents of interest. Furthermore, the fields *firebaseIdPlayer1*, *firebaseIdPlayer1_timestamp*, *firebaseIdPlayer2* and *firebaseIdPlayer2_timestamp* collect the information about the device on which the players have decided to play the challenge, in order to prevent any attempt to play the same game on two different devices from the same player and generate inconsistency inside the system. Instead, the *schemesP1* and *schemesP2* contain the solved schemata by player 1 and player 2. The schemata are saved inside two lists, as injected documents, more about schemata are going to be described inside the following section. Finally, particular attention should be paid in order to analyse the **state** field, that describes the actual situation of a challenge. It can assume the following values:

- *0 (PENDING)*: player 1 creates the challenge, but player 2 isn't accepted yet.
- *1 (CREATED)*: player 2 accept the challenge, so both players can start playing the game.
- *2 (SCORE_1)*: player 1 has played the challenge, solving some schemata that are stored inside the *schemesP1* list.
- *3 (SCORE_2)*: player 2 has played the challenge, solving some schemata that are stored inside the *schemesP2* list.
- *4 (COMPLETED)*: both player have played the challenge, that is now in a final state and can't be modifiable any more.
- *-1 (ABANDONED_1)*: player 1 leaves the challenge, that isn't modifiable any more.
- *-2 (ABANDONED_2)*: player 2 leaves the challenge, that isn't modifiable any more.
- *-3 (DENIED)*: player 2 denies the invitation to the challenge, that isn't modifiable any more.
- *-4 (CANCELED)*: player 1 has deleted the invitation to the challenge, that isn't modifiable any more.

- *-5 (TIMEOUT)*: if a challenge isn't accepted or refused by no one of the two players, it is put in a *time out* state, in order to be not modifiable any more, and, also, it is going to be deleted from the database.

The transition from one state to another is regulated by a state machine that describes the allowed transitions from a specific state to another. The description of this mechanism is not a purpose of this elaborate, but, however, should be clarified that each transition is a consequence of players actions, as it is self explained by the state description. In order to make atomic each challenge update, the *FindAndModify()* method of MongoDB is exploited, that guarantees atomicity in document access and updates, and ensures consistency and correctness of information. According to the tournament feature, a challenge created in this context is set to the *CREATED* state, because the invitation mechanism is not needed, due to the fact that a challenge created from two tournament participants is accepted by default. A tournament challenge can reach only the following final state: *COMPLETED*, *ABANDONED_1*, *ABANDONED_2* and *TIMEOUT*.

### Schema model

Inside the *schemesP1* and *schemesP2* fields of the challenge document are listed the schemata information solved by player 1 and player 2 during the two minutes available to play a challenge. The listing 4.5 shows the main

Listing 4.5: Schema document

```
{
  "_id" :ObjectId("5b11039a36ee652d048aa06f"),
  "code" :NumberInt(5013),
  "playerId" :"5b0bbc7f850940351439df1a",
  "challengeId" :"5b1102e536ee652d048aa036",
  "maxPoints" :NumberInt(648),
  "score" :NumberInt(45),
  "bonus" :NumberInt(98),
  "xp" :NumberInt(6),
  "time" :NumberInt(22),
  "created_at" :NumberLong(1527841690414)
}
```

information collected inside a schema document. Several external references are saved inside each schema, in particular the player identifier that solved the schema, the challenge identifier to which the schema belongs to, and other information used to evaluate the points assigned to the player for each schema. Particular attention should be paid on the **score**, **bonus** and **xp** fields, that store, respectively, the points obtained from the schema resolution, the bonus points assigned due to the time used, and the experience points earned by the

player. finally, the **code** field, stores the information about the seven number combination composing the schema.

### 4.1.5 Conclusion

As described during this section, the documents are saved in MongoDB to guarantee the persistence of information. Also, in some situations, support documents are created, in order not to, only, save transient data, but also to exploit the MongoDB features in order to guarantee some functionalities that would be difficult to implement using Java. In conclusion MongoDB isn't only a database where to store data, but also an important component with some characteristics useful to enforce the algorithm developed inside the business logic.

## 4.2 Tournament management

This section describes and analyses, in the details, the *tournament APIs*, used by clients to communicate with the server, and the two main tasks that support the tournament business logic. Special attention will be given to the algorithm implemented and to the synchronization mechanism designed to ensure consistency when multiple concurrent requests should be managed by the server.

### 4.2.1 Tournament API

First of all, should be specified that each client, that makes a request to the server, must be authenticated in order to authorize the request. The authorization mechanism exploited by Mak07 system is the OAuth 2.0 protocol. It works by delegating the user authentication to a service that hosts the user account (i.e. Facebook, Google), and authorizing third-party applications (i.e. Mak07) to access the user account. It is, also, fully supported by Spring and in particular by the Spring Security framework, that was used during the implementation of Mak07 authentication. Furthermore, a session mechanism was implemented. It is based on the use of two cookies inside the HTTP header. The *JSESSIONID* cookie is used by the server to identify a session, and the *Remember-Me-Token* is used to make persistent and to update the session if the *JSESSIONID* is expired. These two cookies are used to avoid the user authentication each time he makes a request to the server. The description of the OAuth 2.0 protocol, the session mechanism and how they are implemented in Spring are not a purpose of this elaborate.

In the following, the API endpoints, developed to support the tournament feature, are described and analysed. It should be underlined that all the *RESTful* principles are applied as much as possible during the development, in order to make the API consistent and idempotent. The designed endpoints are:

1. **GET /api/tournaments**: it returns a list of tournaments, that are in the *CREATED* state, so that the user is able to sign up to one of them. An

empty list is returned if there are no available tournaments.

2. **GET /api/me/tournaments**: it returns a list of tournaments, to which an user has participated, that are in the *COMPLETED* state, and the one to which is participating, if any, that is in a non final state. The last seven days are considered, in order to get only the most recent tournaments.

3. **GET /api/me/activeTournament**: it is pointed by a player that wants to know the identifier of the tournament he is participating. If there are no active tournament, the *404* error is returned.

4. **POST /api/tournaments**: this resource is exploited to create a new tournament. The user sends to the server the tournament settings and a list of friends he wants to invite. The tournament just created has only one participant that is the creator itself. Therefore, the invited user are notified by the server and the tournament information are persistently saved in the database. Finally, a task is scheduled after 20 minutes, that is going to manage the tournament beginning.

5. **GET /api/tournaments/{id}**: a tournament object, with the corresponding identifier specified in the path endpoint, is returned to the user that makes the request. Otherwise a *404* error, not found, is generated.

6. **PUT /api/tournaments/{id}/signup**: After checking if there are available tournaments, an user can sign up to one of them contacting this endpoint. It returns the tournament object to which the player signed up, otherwise different errors can be generated. If the registration phase for that specific tournament is expired, a *404* error is returned, instead, if the player has an active tournament yet, or the tournament is full and can't accept new participants, a *405* error is generated.

7. **PUT /api/tournaments/{id}/unsubscribe**: a player can delete his registration to a tournament if and only if the tournament isn't started. Otherwise an error *400* is returned.

8. **GET /api/tournaments/{id}/challenges**: it returns the list of all the challenges information of the tournament identified by the unique id inside the endpoint path. If the tournament challenges are not been created yet, the list is empty, or it could contain a number of challenges not equal to the maximum number expected, if the tournament isn't finished.

9. **POST /api/tournaments/{id}/invite**: it is used if a player wants to invite some friends to a tournament he is participating. The client sends to the server a list of unique identifier of Mak07 users he wants to invite. The server updates the list of invited friends of the tournament, and sends a notification to the interested users. This endpoint works only during the registration phase.

10. **PUT /api/tournaments/{id}/refuse**: it is contacted by a player when it is invited to take part to a tournament, but he doesn't want to participate. If a refuse request is performed, the server updates the list of invited friends, eliminating the user that have made the request. This endpoint works only during the registration phase.

11. **GET /api/me/invitedTournaments**: it returns all the tournaments to which an user is invited, but he is not a participant. This API is designed to allow an user to accept only the invitation of interest.

All the information, exchanged among client and server through these APIs, are compacted in json documents. If a non authenticated user tries to make a request, the server responses with a *403* error, "player not authenticated", refusing the request.

### 4.2.2 Play challenge API

As mentioned in the sections above and during the Design chapter, each tournament has several rounds, up to 4, according to the maximum number of participants, and each round has several challenges, the number of which depends on the the round number. It was thought to exploit as much as possible the code already developed for the challenge feature. In the following, the main endpoints, already available in the server and used by clients to play challenges, are listed:

- **PUT /api/challenges/{id}/play**: in order to play a challenge, each user must inform the server that he is going to start a specific challenge. The server receives the *firebase id* of the device on which the user wants to play the challenge, in order to lock any attempt to play the same challenge from several devices.

- **PUT /api/challenges/{id}**: this endpoint allows each user to update the state of a specific challenge. The challenge identifier is specified inside a path parameter. Also, the client should send to the server the new state to be assigned to the challenge, and the schemata solved during the game, if the user has just played the challenge. The **update** request is always executed after the **play** request. If the update is not allowed, or the user makes a wrong update, a corresponding error message is generated.

During the tournament algorithm implementation, little code modifications are inserted inside the logic of the *play* request, that are going to be explained in the following, instead no modification or insertion are done for the *update* request, that is used as it is.

### 4.2.3 Registration phase and synchronization

After the execution of the request to create a tournament, a document that maintains all the tournament information is inserted inside the database. At

Listing 4.6: Update tournament participants function, due to user registration

```
@Override
public Tournament updateTournamentParticipants(String tournamentId,
    String playerId, int maxParticipantsNumber, boolean signup) {
  Query query = new Query();
  Criteria criteria = new Criteria().andOperator(
    new Criteria().where("_id").is(tournamentId),
    new Criteria().where("participantsNumber")
      .lt(maxParticipantsNumber),
    new Criteria().where("state")
      .is(Tournament.TournamentState.CREATED),
    new Criteria().where("participants").nin(playerId)
  );
  if (signup) {
    criteria.and("created_at").gt(System.currentTimeMillis() -
        TimeUnit.MINUTES.toMillis(20));
  }
  query.addCriteria(criteria);
  Update update = new Update();
  update.inc("participantsNumber", 1);
  update.addToSet("participants", playerId);
  Tournament updatedTournament = template.findAndModify(query,
    update,
    FindAndModifyOptions.options().returnNew(true),
    Tournament.class);
  return updatedTournament;
}
```

this point, the registration phase begins, during which several concurrent requests, that want to modify the tournament document, can be received by the server. The main APIs that modify relevant information are: the request to sign up to a tournament and the request to delete a registration. They both update the list and the number of participants inside the tournament document. In order to prevent inconsistent updates, the *findAndModify()* function is exploited, that guarantees the atomicity of access and update on MongoDB documents. As designed, each user can have only one active tournament at a time. Any attempt of an user, already participating to a tournament, to create a new one or to sign up to an existent one, must be prevented. In order to do this, the collection of *ActiveTournament* document is exploited. When the user performs one of these two requests, the server logic tries to insert a document inside the underlined collection. If a document, related to the user, is already inside the collection, the action is aborted, and an error message is sent to the client. Furthermore, the requests to invite friends and to refuse an invitation modify the invited friends list inside the tournament document, so, the *FindAndModify* function is exploited in these situations too. In conclusion, If one

of the mentioned requests is executed after the registration phase, i.e. when the 20 minutes expires, an error *400 - "time expired"* is sent to the client. This kind of check is made exploiting the MongoDB query parameters. As described inside the section 4.1.1, each tournament document has a *state* and a *created_at* fields. When an update is performed on a tournament document, these fields must be checked, verifying that the *state* field has the value *CREATED*, and the *created_at* field has a value less than 20 minutes (As showed in the listing 4.6). If one of these checks fails, it means that the tournament has exceeded the registration phase any update must be discarded.

### 4.2.4 Start Tournament Scheduled Task

Listing 4.7: StartTournament scheduled task

```
@Override
public void run() {
  Tournament t = tournamentManagement
  .getTournamentRepository().findById(tournamentId);
  try {
    if (t.getParticipantsNumber() >= 2) {
      Collection<String> bots = null;
      List<String> users = t.getParticipants();
      if (t.getParticipantsNumber() != t.getMaxParticipants()) {
        bots = tournamentManagement.getWebSocketGeneralService()
          .getRandomNBot(t.getMaxParticipants() -
             t.getParticipantsNumber());
        for (String bot : bots) {
          tournamentManagement.getTournamentRepository()
            .updateTournamentParticipants(t.getId(), bot,
                t.getMaxParticipants(), false);
        }
      }
      tournamentManagement
        .getTournamentPhasesRepository()
        .save(new TournamentPhases(tournamentId,
           Tournament.TournamentState.IN_PROGRESS_1, 0));
      Collections.shuffle(users);
      tournamentManagement.CreateTournamentChallenges(users, bots,
         t.getId());
      tournamentManagement
        .getTournamentRepository()
        .updateTournamentState(t.getId(),
           Tournament.TournamentState.IN_PROGRESS_1);
      //notify users
      ...
      //schedule new task after 10 minutes and 30 second
      tournamentManagement.getTaskScheduler()
```

```
30          .schedule(new TournamentPhase(tournamentId, 1,
               tournamentManagement), new Date(System.currentTimeMillis()
               + tournamentManagement.getPHASE() + 30000));
31      } else {
32        tournamentManagement
33          .getTournamentRepository()
34          .updateTournamentState(t.getId(),
               Tournament.TournamentState.CANCELED);
35        tournamentManagement
36          .getActiveTournamentRepository()
37          .deleteActiveTournamentByPlayerId(t.getCreator());
38        tournamentManagement.sendFirebaseNotification(t.getName(),
             t.getId(), t.getCreator(), "UPDATE_TOURNAMENT",
             "UPDATE_TOURNAMENT_CANCELED");
39      }
40
41    } catch (UserPlayerNotFoundException e) {
42      ...
43    }
44  }
```

The beginning and the rounds of a tournament are managed by two important tasks. The first of them is the StartTournament task that is in charge of managing the transition from the *CREATED* state to the *IN PROGRESS 1* state for a tournament. This transition happens at the end of the registration phase. In fact, the execution of this task is scheduled after 20 minutes the creation of a tournament. A StartTournament task is performed for each created tournament. In order to develop this task, it was thought to create a class that implements the *Runnable* interface, so, inside the *run* method is contained the task logic.

In the listing 4.7 is listed the task code, that is going to be analysed. After getting the tournament information, the first action to be performed is checking the number of participants. It was thought to limit the number of human participants to at least 2. If the tournament creator is the only participant, the tournament *state* is set to *CANCELED* and the creator is notified that the tournament hasn't reached the minimum number of human players. On the contrary, if there is at least two participants the tournament can start. Once again, the participants number is checked in order to understand if the maximum number of participants is reached. If it is reached, the following step is executed, otherwise a suited number of players must be added in order to obtain this value. To overcome this problem, an already available game feature is exploited, the *Bot player*. Mak07 system makes available a pool of 44 bot players, that can be incremented with a specific server request. A suited number of random bots is selected to be added to the tournament as participants, in order to fill the gaps left by the missing players. Now everything is ready to the next step, creating the challenges for the round one. So, the tournament *state* is set to *IN PROGRESS 1*, the list of users is shuffled in order to create

as much as possible random challenges. The method *CreateTournamentChallenges* receives the list of human and bot participants and the tournament identifier, and it is in charge of creating the challenges among players. It couples a player from the the human list and one from the bots list, up to one of the two lists becomes empty, and the players in the remaining one are coupled together. For each couple, a challenge is generated, and a document of type *TournamentChallenge* is created and inserted inside the list of tournament challenges. The created challenge is in the state *CREATED* and the *RANDOM* type is associated to it, if both players are humans, otherwise the *RANDOM_BOT* type is assigned. Finally, a document inside the support collection *TournamentPhases* is inserted for the actual tournament, storing the tournament id, the state, and the number of ended challenges, that, at creation time, is equal to 0. This support collection is going to be used in order to speed up the end of a round. The last step is to notify the players that the tournament is begun, and that the challenges of the first round are available to be played. Finally, a new task is scheduled that is in charge of terminating a round and starting the following one. In conclusion, as pointed out during the above sections, the tournament document can be updated by the user only during the registration phase, further updates are going to be executed only by the two scheduled tasks.

### 4.2.5 Tournament Phase Scheduled Task

Listing 4.8: Tournament Phase task

```
1  @Override
2  public void run() {
3    try {
4      ...
5      if (!tournamentManagement.getTournamentPhasesRepository()
6           .updateState(tournamentId, oldState, newState)) {
7        return;
8      }
9      ...
10     int numberOfWinners =
           putChallengesInAFinalState(t.getChallenges(), fromIndex);
11     ...
12     if (numberOfWinners > 1) {
13       for (...) {
14         String player1 =
               challengesMap.get(challengeNumber).getWinner();
15         String player2 = challengesMap.get(challengeNumber +
               1).getWinner();
16         if (player1 == null) {
17           if (player2 == null) {
18             tournamentManagement.getTournamentRepository()
```

```
19                .updateTournamentChallenges(tournamentId,
                      newChallengeNumber++, null, null);
20              ...
21            } else {
22              tournamentManagement.getTournamentRepository()
23                .updateTournamentChallenges(tournamentId,
                      newChallengeNumber++, player2, null);
24              ...
25            }
26          } else {
27            if (player2 == null) {
28              tournamentManagement.getTournamentRepository()
29                .updateTournamentChallenges(tournamentId,
                      newChallengeNumber++, player1, null);
30              ...
31            } else {
32              Challenge c = tournamentManagement.getChallengeService()
33                .createTournamentChallenge(player1, player2,
                      tournamentId);
34              tournamentManagement.getTournamentRepository()
35                .updateTournamentChallenges(tournamentId,
                      newChallengeNumber++, c.getId());
36            }
37          }
38        }
39
40        tournamentManagement.getTournamentRepository()
41          .updateTournamentState(t.getId(),
42            Tournament.getTournamentStateFromInt(tournamentPhase+1));
43        //notify users of the new available phase
44        tournamentManagement.getTaskScheduler()
45          .schedule(new TournamentPhase(tournamentId,
              tournamentPhase+1, tournamentManagement),
46            new Date(System.currentTimeMillis() +
                tournamentManagement.getPHASE() + 30000));
47      } else {
48        ...
49        tournamentManagement.getTournamentRepository()
50          .updateTournamentState(t.getId(),
              Tournament.TournamentState.COMPLETED);
51        ...
52        for (String player : t.getParticipants()) {
53          tournamentManagement.sendFirebaseNotification(t.getName(),
              t.getId(),
54            player, "UPDATE_TOURNAMENT", "UPDATE_TOURNAMENT_COMPLETED");
55        }
56      }
57    } catch (UserPlayerNotFoundException e) {
58      ...
```

```
59     }
60 }
```

The other important task, in charge of managing the end of a tournament round and the beginning of the following one, is the *TournamentPhase* task. An abstract is reported in the listing 4.8. It is scheduled for the first time by the *StartTournament* task. Each TournamentPhase task has to receive the information about the tournament it should manage and the round on which the task should focus on.

The task, after retrieving the data related to the tournament from the database, must update the information about the tournament state inside the *TournamentPhases* collection. Since more than one task can be scheduled to manage the same tournament phase, before performing the mentioned update, a query is executed in order to understand if it has already been done by another task. The *FindAndModify()* method of MongoDB is exploited, that, atomically, is able to query and update a document inside a collection. If the query fails, another task has already managed the end of the round of interest, so the current task must die without performing other actions. Otherwise the update can be performed by the current task, and other possible tasks that want to manage the same tournament round are blocked.

At the end of a tournament phase, it is very important that all the challenges, belonging to that round, are in a finale state. If they don't, they must be put in a final condition, in order to not be modifiable any more, and to maintain the system database in a consistent state. The *putChallengeInAFinalState()* method is in charge of doing this, but also performing other actions. It receives, as parameters, the list of tournament challenges, and the starting index from which the challenges must be considered (i.e. let's suppose that the number of tournament participants is 8, and the task should manage the end of the round 2, so it should consider only the challenges that has a challenge number greater or equal to 5). This method scans the input list of challenges, selects the ones of interest, and for each of them it checks the state. According to the challenge states analysed in the 4.1.4 section, some situations should be managed:

- the challenge is in the *CREATED* state. This means that both players have abandoned the game, so there is no winner, and the challenge is put in the final state *TIMEOUT*.
- the challenge is in the *SCORE_1* or *SCORE_2* state. It means that it is played only by one of the two players, that is the winner, because the other has abandoned the game. So, the challenge is put in the final state *ABANDON_1* or *ABANDON_2* according to which player has left the match.
- the last case, instead, describes the situation in which the challenge is already in a final state, represented by the conditions *ABANDON_1*, *ABANDON_2* and *COMPLETED*. The only performed action is declaring the winner.

The method *putChallengeInAFinalState()*, is in charge, also, to update, the information about the winner inside the tournament challenges list, and to return, to the calling function, the number of total winners. Since that a challenge could have no winner, the winner field, inside the tournament challenges list, could contain the value *null*. Finally, each time a player has been defeated in a challenge, he is eliminated from the tournament, so that he can sign up to a new one. This is done, deleting the related document inside the *ActiveTournament* collection.

Knowing the number of winners at the end of each phase, it is possible to understand if a new round should be scheduled or not. If there are one or zero winners, it means that the tournament is finished, so it is put in a final state, and all the participants are notified of the end of the tournament. This situation can happen, not only in a final round, but also during intermediate ones. For this reason a tournament can finish before than expected. Instead, if the number of winners is greater than 1, a new round phase must be scheduled. In order to do this, new challenges must be created. So, the winners of the preceding round, are coupled with respect to the tournament schema 3.4. This step could manage particular situations:

- both the winners of the challenges of the previous round exist, so they are coupled to create a new match for the next round. A new *TournamentChallenge* document (Section 4.1.1) is inserted inside the challenges list of the tournament document, it collects the number and the identifier of the challenge, instead the winner will be registered in a second time.

- if one of the two winners doesn't exist, the new challenge is not created, because it is not needed, due to the fact that the possible winner is already available. This information is stored inside a *TournamentChallenge* document, that is inserted in the tournament challenges list. the document collects, also, the data about the challenge number, instead no information about the challenge identifier are available, for obvious reasons.

- a situation similar to the previous one, could happen if both winners don't exist. In this case, the *TournamentChallenge* document contains only the information about the challenge number, the other fields are left empty.

Finally, the following round can start. The Tournament document is updated to the incoming phase, and all the participants to the new round are notified. As final step, the TournamentPhase task schedules another task of the same type, passing, as parameter, the information about the new incoming phase. In conclusion, a clarification should be done, about the possibility of having more than one task, managing the end of the same round. As explained, a round lasts ten minutes by default, so a TournamentPhase task is scheduled at the end of this time, in order to handle the end of the round itself. But, conceptually, a round can finish before the ten minutes expiration, if and only if all the challenges belonging to that round are in a final state, i.e. all the

players have played the games. In order to guarantee this functionality, a new task of type TournamentPhase is released after the end of the last challenge. Finally, according to the synchronization mechanism explained at the top of this section, only this last released task is going to execute its job, instead the one scheduled at the ten minutes expiration is going to die.

#### Round synchronization

As mentioned several times inside this elaborate, a great effort has been spent in order to integrate the challenge mechanism inside the tournament one. A challenge is composed by two games, one for each player, that can be played in different instants of time. So a challenge can last long time if one of the two player doesn't play his game. On the contrary, each tournament round has a fixed execution time, that must last up to 10 minutes. So the challenges inside each round must finish in a limited amount of time. In order to guarantee this constraint, it was developed a simple but efficient mechanism to limit the updates that an user can make on a challenge outside the 10 minutes available for the round.

The starting point, as explained in the section 4.2.2, is the following: when a player wants to play a challenge, the device, running the mobile application, sends a *play* request to the server, in order to say that the user wants to start playing the game, and an *update* request in order to register the obtained result. In a first time, it was thought to discard an *update* request that arrives in a wrong instant of time, reporting an error to the user. This solution could be not satisfying for the user experience, because, at the end of a challenge, the user expects to see its final result and not an error message. After this approach a new one was developed. Rather than stopping an *update* request, it was decided to block a *play* request, but this solution introduces new problems that should be managed. According to the fact that each challenge lasts two minutes, between the *play* request and the *update* one there is a gap of at least two minutes, because the *play* is executed before playing the game, instead the *update*, later. This means that if the round duration is of 10 minutes, a player must start playing a challenge within eight minutes from the beginning of the round. Any attempt to start a challenge in the last two minutes must be discarded. In this last period, only *update* requests are accepted by the server. Due to this consideration, the *update* request algorithm, developed during past implementations, is not modified, instead, a check is introduced inside the *play* request algorithm, in order to limit the requests, related to tournament challenges, within eight minutes from the beginning of the round. Furthermore, the fact that, the two minutes of time between the *play* and the *update* requests could be enlarged due to network delays, should be considered. To tackle this uncertainty, a suited delay of 30 seconds is introduced in the schedule time of each *TournamentPhase* task. So each task that manage the transition from a phase to another is scheduled, from the previous one, after ten minutes and 30 seconds. *Update* requests, related to *play* requests executed in time, that accumulate a big number of delay are, however, man-

aged by the *TournamentPhase* task. All these expedients guarantee a correct and consistent execution of the code.


Figure 4.1: Architecture used to send push notification from Spring exploiting Firebase.

### 4.2.6 Notification management

Listing 4.9: Firebase notification method

```
public void sendFirebaseNotification (String tournamentName, String
    tournamentId, String userId, String title, String body) {

List<RememberMeToken> result = mongoPersistentTokenRepository
.findTokenByUsername(userId);
List<String> registration_ids = new ArrayList<>();
for (RememberMeToken r : result) {
registration_ids.add(r.getFirebaseId());
}

...

firebaseManagementService
.sendMessageTournament(firebaseMessageTournament);
}
```

As described during the previous chapter, Mak07 is a mobile game, and, in order to improve the user experience, a suited notification mechanism should be implemented. Push notifications are used to enforce all the features of Mak07. Specific notifications were scheduled during the development of the challenge algorithm, not only to inform the final user about specific events, but also to update the user interface if something changes on the server side. Therefore, suited notifications are scheduled in the tournament algorithm too, in order to inform the participants about the tournament main events, like the

beginning of a round and the end of a tournament, or to inform the user that is invited to take part to a specific tournament.

The figure 4.1 shows the architecture exploited to implement the notifications based on the Firebase Cloud Messaging service, provided by Google. The method reported in the listing 4.9 is the one implemented to send notifications due to tournament changes. Each time an information should be notified to the user, this method is invoked. It receives as parameter the *tournament identifier*, the *tournament name*, the *title* and the *body* of the notification, that are going to be shown on the screen of the device. The method, thanks to the *userId* parameter, retrieves, from the database, the *firebase id* of the devices related to the user that should receive the notification. Then, the object, that contains the information to be sent to Firebase and the body of the notification, is built, and passed as parameter to the *sendMessageTournament()* method. This method, belonging to the *FirebaseManagementService* interface, is in charge of sending an HTTP request to the Firebase service endpoint, that is going to forward the message to the user devices.

# Chapter 5

# Conclusion

The starting point of this elaborate is the design and development of a new feature for the Mak07 mobile game: the possibility of having tournament among players. The development of this feature starts from the analysis of the requirements and of the most frequent use cases, focusing, also, on some particular cases that should be tackled during the algorithm definition. After this first study phase, the design phase begins, in which all the data structures needed by the code are designed, and the algorithm that has to manage the tournament feature, along with the synchronization mechanism, is defined. Therefore, during the implementation phase, the defined algorithm is implemented, spending a big effort on making the code as much modular as possible, and merging perfectly the new functionality with the existing ones. In conclusion, a final phase of testing was executed in a developing environment, in order to test the correctness of the application. Now the tournament feature is available as update of Mak07 mobile application.

## 5.1 Feature Development

Mak07 is born with modularity quality and prone to feature development. Therefore, during the development of this final project, possible future developments are always kept in mind. In fact, as explained at design time, each tournament is identified by a type, that could be *PUBLIC* or *PRIVATE*, but only the first of these two possibilities has an implementation inside this final project, instead the other one could be deepened in a future development. Furthermore, another improvement could be the possibility of having achievements inside the game, in order to improve the game experience and to reward the better players. Finally, due to the fact that the game is prone to growth, in the future, could be necessary improving the backbone architecture supporting the Mak07 system. This can easily be done thanks to the used technologies, as MongoDB and Docker, that are suited for horizontal scaling.

# Bibliography


[1] *"REST principles explained"*. 2013. URL: https://www.servage.net/blog/2013/04/08/rest-principles-explained/.

[2] *"WHAT IS MONGODB?"* URL: https://www.3pillarglobal.com/insights/what-is-mongodb.

[3] Matt Allen. *"RDBMS are not designe to handle data"*. URL: https://www.marklogic.com/blog/relational-databases-are-not-designed-for-mixed-workloads/.

[4] Ian Blair. *"How to Create a RESTful API For Your Mobile App"*. 2017. URL: https://buildfire.com/create-restful-api-mobile-app/.

[5] Mike Chapple. *"Abandoning ACID in Favor of BASE in Database Engineering"*. 2017. URL: https://www.lifewire.com/abandoning-acid-in-favor-of-base-1019674.

[6] Loredana Crusoveanu. *"Intro to Inversion of Control and Dependency Injection with Spring"*. 2018. URL: http://www.baeldung.com/inversion-control-and-dependency-injection-in-spring.

[7] Firebase. *"Firebase Cloud Messaging HTTP Protocol"*. 2018. URL: https://firebase.google.com/docs/cloud-messaging/http-server-ref.

[8] Akhil Mehra. *"Understanding the CAP Theorem"*. 2017. URL: https://dzone.com/articles/understanding-the-cap-theorem.

[9] MongoDB. *"Introduction to MongoDB"*. URL: https://docs.mongodb.com/manual/introduction/.

[10] MongoDB. *"Types of NoSQL Databases"*. URL: https://www.mongodb.com/scale/types-of-nosql-databases.

[11] MongoDB. *"What Is A Non Relational Database"*. URL: https://www.mongodb.com/scale/what-is-a-non-relational-database.

[12] Martin O'Shea. *"How does a spring boot work?"* 2017. URL: https://www.quora.com/How-does-a-spring-boot-work.

[13] Pivotal. *"Bean scopes"*. URL: https://docs.spring.io/spring-framework/docs/3.0.0.M3/reference/html/ch04s04.html.

[14] Margaret Rouse. *"RDBMS (relational database management system)"*. URL: https://searchdatamanagement.techtarget.com/definition/RDBMS-relational-database-management-system.

[15] Michael Shrivathsan. *"Use Case Template and an Example"*. 2009. URL: http://pmblog.accompa.com/2009/10/08/use-case-template-example-requirements-management-basics/.

[16] Spring. *"Understanding AMQP, the protocol used by RabbitMQ"*. 2010. URL: https://spring.io/blog/2010/06/14/understanding-amqp-the-protocol-used-by-rabbitmq/.

[17] Tutorialspoint. *"DBMS - Overview"*. URL: https://www.tutorialspoint.com/dbms/dbms_overview.htm.

[18] Tutorialspoint. *"Spring Framework - Overview"*. URL: https://www.tutorialspoint.com/spring/spring_overview.htm.

[19] Tutorialspoint. *"SQL - Overview"*. URL: https://www.tutorialspoint.com/sql/sql-overview.htm.

[20] Craig Walls. *"Spring in action, fourth edition"*. Manning.