Πανεπιστήμιο Πειραιώς – Τμήμα Πληροφορικής

Πρόγραμμα Μεταπτυχιακών Σπουδών

«Προηγμένα Συστήματα Πληροφορικής»

Μεταπτυχιακή Διατριβή

Τίτλος Διατριβής	**Ανάπτυξη Διαδικτυακής Εφαρμογής στη μεριά του εξυπηρετητή, συμβατής με desktop, mobile και tablet συσκευών, για τη διαχείριση αθλητικών δεδομένων αθλητών στίβου, με χρήση της μεθοδολογίας Scrum Agile για την ανάπτυξη λογισμικού.** **Implementation of a cross-device server-side web application for managing and analyzing personal performances for track and field, using Scrum Agile software development methodology.**
Ονοματεπώνυμο Φοιτητή	**ΜΠΑΛΑΣΗΣ ΓΕΩΡΓΙΟΣ**
Πατρώνυμο	**ΑΘΑΝΑΣΙΟΣ**
Αριθμός Μητρώου	**ΜΠΣΠ 12048**
Επιβλέπων	**Ευθύμιος Αλέπης, Επίκουρος Καθηγητής**

Ημερομηνία Παράδοσης **Σεπτέμβριος 2015**

Τριμελής Εξεταστική Επιτροπή

(υπογραφή)	(υπογραφή)	(υπογραφή)
Ευθύμιος Αλέπης Επίκουρος Καθηγητης	Μαρία Βίρβου Καθηγητρια	Κωνσταντίνος Πατσάκης Λέκτορας

Abstract

This thesis is about developing a web application that keeps statistics about track and field athletes. For the development, the nowadays popular software engineering methodology, agile, will be used, and more specifically scrum. Scrum promotes teamwork, so I will develop the application with another programmer. I will write the code on the server side, create and manage the schema of the database and handle the API. The other programmer will work on the front-end of the application. Agile will be used in every part of the development process - coding, decisions, feedback.

The development process will be described in parts, called sprints, according to scrum. Every sprint will be the description of a week's work. The first part of the sprint will be the sprint planning, where the goals of the sprint will be set, and there will be a short description of the desired outcome that we should have by the end of the week. The second part is called stand up, which is a sum of all the daily stand ups. This is where I declare what I did each day, while working on the application. Usually some decisions that I took, along with the other programmer, are mentioned. Stand ups are also used in order to describe the difficulties that are faced. The third part of the sprints is the sprint review, where there is a report of what worked and what didn't during the sprint. The outcome is presented and the plans of the next sprint may be revealed.

By using technologies which are not very familiar to us, developing an application from scratch, and getting feedback from users, we expect agile to be suitable for our case, making communication the main aspect that will help us release a full product.

After finishing the development of the application, it is proven that indeed agile greatly helped in the development process. The application got developed in time, all the developed features are useful for its users, and the quality of the service is very good. The small iterations that had a working application as the outcome, every time, helped us get feedback early on, and improve fast. Overall, the agile methodology proved to be appropriate for our case, and quite possibly, much more suitable than any plan-driven process.

Περίληψη

Αυτή η πτυχιακή αναφέρεται στην ανάπτυξη μίας διαδικτυακής εφαρμογής που κρατάει στατιστικά για αθλητές στίβου. Για την ανάπτυξή της θα χρησιμοποιηθεί η δημοφιλής μεθοδολογία ανάπτυξης λογισμικού agile, και πιο συγκεκριμένα scrum. Το scrum προωθεί την ομαδικότητα, οπότε η ανάπτυξη θα γίνει σε συνεργασία με έναν άλλο προγραμματιστή. Εγώ θα γράψω τον κώδικα για τον server, το σχήμα της βάσης δεδομένων και το API. Ο άλλος προγραμματιστής θα αναπτύξει τον client της εφαρμογής. Η μεθοδολογία agile θα χρησιμοποιηθεί σε κάθε τμήμα της διαδικασίας ανάπτυξης - στον προγραμματιστμό, στην λήψη αποφάσεων και στην ανατροφοδότηση.

Η διαδικασία της ανάπτυξης θα περιγραφεί σε τμήματα, τα οποία αποκαλούνται sprints στο scrum. Κάθε sprint στο κείμενο θα είναι η περιγραφη της δουλειάς μίας εβδομάδας. Το πρώτο μέρος από κάθε sprint θα είναι ο σχεδιασμός του, όπου θα γίνεται ο προσδιορισμός των στόχων και θα υπάρχει μία σύντομη περιγραφή του επιθυμητού αποτελέσματος το οποίο αναμένεται στο τέλος της εβδομάδας. Το δεύτερο μέρος του sprint λέγεται stand up, και στο κείμενο θα είναι μία σύνοψη όλων των ημερήσιων stand up. Σε αυτό μέρος θα περιγράφεται το τι κινήσεις έγιναν για την ανάπτυξη της εφαρμογής κάθε μέρα, τα προβλήματα που αντιμετωπίστηκαν και οι αποφάσεις που λήφθηκαν. Το τρίτο μέρος του sprint είναι η ανασκόπησή του, όπου θα γίνεται μία αναφορά στο τι δούλεψε και τι όχι, κατά τη διάρκεια της εβδομάδας. Επίσης θα παρουσιάζεται το αποτέλεσμα και θα γίνεται μία σύντομη αναφορά στο τι θα ακολουθήσει την επόμενη εβδμάδα.

Χρησιμοποιώντας τεχνολογίες με τις οποίες δεν είμαστε πολύ εξοικειωμένοι, αναπτύσσοντας μία εφαρμογή από την αρχή και παίρνοντας σχόλια από χρήστες, περιμένουμε πως η μεθοδολογία agile θα είναι κατάλληλη για την περίπτωσή μας, κάνοντας την επικοινωνία τον κύριο παράγοντα που θα μας κάνει να παραδώσουμε ένα ολοκληρωμένο προϊόν.

Μετά την ολοκλήρωση της ανάπτυξης της εφαρμογής, είναι φανερό ότι όντως η μεθοδολογία agile βοήθησε στη διαδικασία ανάπτυξης. Η εφαρμογή ολοκληρώθηκε έγκαιρα, όλα τα χαρακτηριστικά της είναι χρήσιμα για τους χρήστες της, και η ποιότητά της είναι πολύ καλή. Οι μικροί κύκλοι εργασίας που είχαν ως αποτέλεσμα ένα δείγμα εφαρμογής που δούλευε κάθε φορά, μας βοήθησε να πάρουμε σχόλια από χρήστες, νωρίς, έτσι ώστε να βελτιώσουμε την εφαρμογή γρήγορα. Γενικά, η μεθοδολογία agile αποδείχτηκε κατάλληλη για την περίπτωσή μας, και πιθανότατα πολύ πιο κατάλληλη από ότι θα ήταν κάποια προσχεδιασμένη διαδικασία.

1. Introduction

1.1. Purpose of the thesis

The purpose of the thesis is to show the agile methodology of software development as it is applied on the development process of the back-end of a web application. It will focus on the frequent iterations between defining the specifications of the application and its development. These iterations will be called “phases” of the application, and according to the agile methodology, these phases will have as a final result a fully functional version of the application.

The flexibility and the efficiency of the agile methodology will be shown in multiple cases by changing the specifications in different stages during the development process and by adding completely new features in phases which were not planned in the previous steps. The selected software engineering method will prove its flexibility as these changes will have little effect on the speed of the development and the quality of the final outcome of the current iteration.

Being a thesis about software engineering, there will be a wide range of diagrams, showing the different states of the application in each phase, how the application was designed before it was developed and the decisions taken before each phase.

Since a big part of the philosophy behind the agile methodology is the efficient cooperation of the team which is developing the application, the front-end development is done by my colleague Theodore Mathioudakis, whose thesis is referenced in this document and who designs the code that will run in the browser, complementing the parts of the back-end that I will design and develop. In each phase, it will be clear how the communication between the browser and the server takes place, and which are the API endpoints that make this communication happen, essentially showing the cooperation of the team in accordance to the agile guidelines.

The parts of the application that will be designed and developed by me will be the code required for the server to run, the logic of the application that exists server-side, the API endpoints and the database. Through the diagrams and the code supplied, it will be clear which features have been implemented on the back-end and which is the schema of the database in each iteration of the methodology.

Finally, after the end of the analysis of the development process of the application, there will be a review of the steps followed and the strong and weak points of the agile methodology will be pointed out, for each of these steps.

1.2. Purpose of the application

The main goal of the application is to keep various data of the performances of track and field athletes, and display them in a way which can help them make useful deductions that will eventually make them improve. It also helps athletes keep an archive of all their performances so that they can track their progress.

Users have to enter raw data in a form, related to the performance. That data is saved in a database and then it can be displayed as distinct activities which can be filtered, or as graphs with various functionalities.

The application also has various settings, regarding the profiles and the accounts. Users can search for other users, using a search input, visit their profiles and view their performances, if their profile permissions allow that. Also, the API of the application allows it to be integrated in other devices like mobile phones or, later, accessories.

1.3. Technologies used and general decisions

The technologies used for the development of the application are all web-based. The server is node.js, a web server running JavaScript on the V8 engine that Google Chrome uses. The MVC pattern is followed so that the models and the interaction of the database is clearly separated from the controllers and the logic of the application. The framework used on node.js is called Express. Express takes care of the routes of the application, the sessions, the controllers and binds all the logic of the back-end system.

For the model part of the MVC, the module name Mongoose is used, which takes care of all the queries and forms the database schemata. The Q module is used for handling the asynchronous database calls. The database used is mongodb which is a NoSQL database that has features that conveniently fit into the application.

The HTML that will be displayed in the browser is generated by the JADE template engine. CSS is generated from the LESS framework. On the front-end, the logic is implemented with the AngularJS JavaScript framework. For the responsive design, the Bootstrap CSS framework is used.

Most of the decisions made, were affected by the nature of the application. The agile methodology was chosen because of its flexibility. The application is developed using the latest web technologies which by the time seemed suitable for the application but we, as developers, were not entirely familiar with them. Also, the functionality and the features of the application is driven by athletes who were asked to test it. Being forgiving to the changes of the specifications, the agile methodology is perfect for this case. Also, since the application is functional between the iterations, athletes can have the demos and test if the features suit their needs.

The application is written in JavaScript, both on the front-end and on the back-end. There aren't many choices for writing the front-end of the application, so JavaScript was an easy choice, but there were many choices for the language of the back-end. Having JavaScript both on the front-end and on the back-end makes the communication of the browser and the server seamless. JSON objects are supported by default on both ends, so this is the format used for the communication between them, instead of XML. Also, when the logic of the front-end and the back-end is similar, code can be reused, which follows the *don't repeat yourself* (DRY) principle of software engineering.

JSON objects are similar to the BSON objects supported by MongoDB, so the next step of communicating without converting any data is using MongoDB. NoSQL databases have a good performance advantage in certain cases. The application doesn't have data that require a relational database, there are no table that need to be joined, so in this case MongoDB is the perfect database and it can scale really well.

There is no native version of the application, just a web version. That's because one of the requirements was that it should be platform free. Not only it should run on desktops running different operating systems, but it should run on tablets and mobile phones as well. The application is about recording the performances of athletes, which do not always have a computer nearby. A web application can run on every platform, as long as there is a browser and the application has a responsive design.

Another decision made is the clear separation between the front-end and the back-end. The application is designed with third-party interfaces and devices in mind. In order to give the opportunity to others to create their own interface and access the API of the application, the back-end should be completely independent from front-end.

2. Agile Software Development Methodology

2.1. Overview

The agile software development is a general method for developing software, which promotes teamwork in small, autonomous teams and makes it easy for them to adapt to the sudden changes of the requirements of the project. Some well-known agile methods are Scrum, Kanban, Lean software development and Extreme Programming. Agile processes of all kinds, share one thing: they embrace change, approaching it as an opportunity for growth, rather than an obstacle [3, 323]. Scrum is the most popular method and it will be the method that will be mainly used during the development process of our application.

No matter which of these methodologies you adopt, you have to do the following: testing in every interval, deliver the product early and often, documentation in every interval, existence of cross-functional teams.

Scrum is a lightweight framework designed to help small, close-knit teams of people develop complex products [1]. It consists of some predefined steps that the team should take every day, in order to organize work, improve communication in the team and achieve its goals. With an agile approach in SCRUM, you begin by creating a product backlog which is a prioritized list of features and other capabilities needed in order to develop a successful product. By following the backlog, you always work on the most important or highest-priority items first. [2, 1078].

A scrum team usually has five to ten people who work in short periods of activity called sprints. The sprints usually range from a week to a calendar month in length. During that period, each independent team does all the work, such as designing, developing and testing, required to produce completed features that can be put into production.

At the end of the sprint, the team reviews the results and gets feedback, which is later used in order to change what must be done in the future and in which way it must be done. An important feature of Scrum is that by the end of the sprint, the team should have a potentially shippable product. Then, the whole process starts all over, and the next iteration is planned.

2.2. Origins

Scrum is not an acronym, it's a term borrowed from rugby, where it refers to a way of restarting a game after an accidental infringement or when the ball has gone out of play [2, 1089]. The first appearance of Scrum was in the article "The New Product Development Game" (Takeuchi and Nonaka 1986) which describes how big companies managed to produce world-class results using a scalable, team-based approach.

In 2001, seventeen software engineers gathered to discuss the future of software development. Part if the discussion was methodologies like scrum, extreme programing and some other development methodologies. They agreed on the name "Agile" and they created the "Agile manifesto" which is a brief set of statements that maps the common philosophical ground of their discussions during that meeting [3, 280].

Highsmith, one of the developers who took part in this meeting, stated "The agile movement is not anti-methodology, in fact, many of us want to restore credibility to the word methodology. We want to restore a balance. We embrace modeling, but not in order to file some diagram in a dusty corporate repository." From that point, agile as a set of methodologies, which can be treated as general guidelines, all aiming towards efficient work and fast reaction to changes, was established.

2.3. Philosophy

Agile has some principles, which is its main philosophy, as stated in the Agile manifesto. [3, 442]

1. Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
2. Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
3. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
4. Business people and developers must work together daily throughout the project
5. Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
6. The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
7. Working software is the primary measure of progress.
8. Agile processes promote sustainable development. The sponsors, developers and users should be able to maintain a constant pace indefinitely.
9. Continuous attention to technical excellence and good design enhances agility.
10. Simplicity - the art of maximizing the amount of work not done - is essential.
11. The best architectures, requirements, and designs emerge from self-organizing teams.
12. At regular intervals, the team reflects on how to become more effective, then tunes and adjust its behavior accordingly.

Taking the bullets one by one, we can see which are the main guidelines of the agile method, how they can help us achieve our goals in the project and make it successful, and how they can be applied to our own specific project.

1. This signifies that the main priority is the customer, and all actions should be taken according to that. So, actions should not be taken because one manager told so, time should not be lost in business meetings that do not related to the product directly, or working on features that may be used in the future, or creating tools that may be needed. Work should always be aimed at delivering the most basic features the customer needs as soon as possible, and all the actions that do not lead to that, should be avoided. In our project, we have a feedback form that will let us hear the opinion of the users as the product grows. Also, we will develop the most basic features first, so that we can make changes to the old features based on users' opinions while we are working on new features.
2. One of the main reasons agile was created, is to make the changes of the requirements during development something which is essential in the life of the product. When PayPal was founded, the aim of the product was to be able to transfer money from one palm-top device to another, and as a secondary feature, it should allow users to send payments by email. That secondary feature became popular, and the company had to adjust and change the whole product to serve this reason. If the team is not so "agile", opportunities may be lost. During the development of our application, we will have one-week long sprints so that if there is an opportunity or a change of plans in one sprint, the context of the work can change in the next sprint.
3. It is essential to deliver working software as often as possible, not only for the customer, but also for the team. The shortest the interval, the shortest the deadlines, and with short deadlines, productivity increases. Automatically all the non-productive procedures like bureaucracy, get removed from the plans. We will have one week intervals for our project, forcing ourselves to work as often as possible, due to the lack of time.
4. When business people communicate with the developers, the have a better idea of how the product is evolving, what problems emerge, which features take more time, so they can adjust parameters like time and money during development. When developers communicate with business people, they can better understand the vision and the philosophy of the product, thus leading to a better result, which will satisfy the former. In the

case of our application, we do not have any business people but communication between the two members of the team will be frequent.

5. Usually developers are already motivated and like what they do. They should just feel that the company will help them even more and make their job even more enjoyable, so that they can be more productive and create high quality software. For this project, we plan to visit and work from different places, to keep things interesting. Also, we will have a "hackathon" atmosphere, working closely and in a good mood to achieve the next goals.
6. Communicating face-to-face is not very common in IT companies. Sometimes employees talk by emails or chat, even though they are located in the same floor or even in the same room. But it has been proven that it's much more possible to misunderstand something in written form. Also, writing to each other is much more slower than speaking. An agile team doesn't have to always communicate face-to-face, but it's always better that week. The team of this project will meet regularly each week in order to achieve the goals faster.
7. Working software is everything. Productivity, funds, good communication, skills etc, mean nothing if by the end of the sprint there is no working software. We will do our best to have working software every week.
8. The development process should have a constant pace. That means that having periods with a lot of work and periods with little work should be avoided. Working long hours has been proven to make the developers more likely to make mistakes, which result in bugs in the software.
9. Taking the time to create code that is more maintainable and better structured, will help the project be more agile than writing code in twice the speed, but with poor structure. Because due to the nature of agile, written code will be revisited quite often, so having written it well the first time, will help the developers work on it faster later. We have separated the tasks of the front-end and the back-end for that reason. There will be no conflicts and we can take the time to think of the whole architecture of the server/client, in order to create clean and maintainable code.
10. Surveys have shown that only 7% of the features of the software projects are always used, 13% of the features are used often and 45% are never used. This is eliminated with agile, by working on the most important features first. In this way, the less important features are naturally removed from the product backlog, and usually they get never implemented. All the unplanned features that may be needed can be added to the project very easily, in the same way.
11. The project should be divided to independent teams that is responsible for what they do as a whole and not as individuals. The result of the team should not be the result of one good member but good work of the whole team equally. In our project, we are only two people so we will divide the work this way, making it easier to find out what works for us and what doesn't.
12. One of the ways for the whole team to improve is to review the work done and fine tune it. With agile, this happens at the retrospective phase, which is one of the most important phases of the agile process. During the development of our application, we will review each other's code and we will discuss the problems we met while coding, in order to learn from each other and improve.

2.4. Why Scrum?

Scrum has the power to transform project management across every industry, every business, and even across life in general. By using Scrum, you'll become more Agile, discovering how to react more quickly and respond more accurately to the inevitable change that comes your way. And by staying focused, collaborating, and communicating, success can be easily achieved.

It is used by major organizations instead of the waterfall method, which is not that efficient in most use cases of the projects. When there is a big up-front design, there is no feedback during

the development process, to make the product better, and this leads to many changes after the release of the product, which means that the product owner will have to pay extra money and the really final version of the software will be delayed.

By prioritizing the work of the project, leads to releasing a fully functional product every one or two weeks, which is a project that has much higher chances to evolve to something better than a project designed up-front. Also, this method speeds up all the processes and makes the team more agile to changes in the environment.

2.5. Scrum roles

2.5.1. Product owner

The product owner is the empowered central point of product leadership. He is the single authority responsible for deciding which features and functionality to build and the order in which to build them. The product owner maintains and communicates to all other participants a clear vision of what the Scrum team is trying to achieve. As such, the product owner is responsible for the overall success of the solution being developed or maintained.

It doesn't matter if the focus is on an external product or an internal application, the product owner still has the obligation to make sure that the most valuable work possible, which can include technically focused work, is always performed. To ensure that the team rapidly builds what the product owner wants, the product owner actively collaborates with the Scrum Master and development team and must be available to answer questions soon after they are posed.

2.5.2. Scrum Master

The Scrum Master helps everyone involved understand and embrace the Scrm values, principles and practices. He acts as a coach, providing process leadership and helping the Scrum team and the rest of the organization develop their own high-performance, organization-specific Scrum approach. At the same time, the Scrum Master helps the organization through the challenging change management process that can occur during a Scrum adoption.

As a facilitator, the Scrum Master helps the team resolve issues and make improvements to its use of Scrum. He is also responsible for protecting the team from outside interference and takes a leadership role in removing obstacles that inhibit team productivity (when the individuals themselves can not reasonable resolve them). The Scrum Master has no authority to exert control over the team, so this role is not the same as the traditional role of project manager or development manager. The Scrum Master functions as a leader, not a manager.

2.5.3. Development team

Traditional software development approaches discuss various job types, such as architect, programmer, tester, database administrator, UI designer, and so on. Scrum defines the role of a development team, which is simply a diverse, cross-functional collection of these types of people who are responsible for designing, building and testing the desired product.

The development team self-organizes to determine the best way to accomplish the goal set out by the product owner. The development team is typically five to nine people in size, its members must collectively have all the skills needed to produce good-quality, working software. Of course, Scrum can be used on development efforts that require much larger teams. However, rather than having one Scrum team with, for example, 35 people, there would more likely be four or more Scrum teams, each with a development team of nine or fewer people.

2.6. Scrum activities

2.6.1. Roles linked to activities

The product owner has a vision of what he wants to create. Because the project can be large, through an activity called grooming it is broken down into a set of features that are collected into a prioritized list called the product backlog.

A sprint starts with sprint planning, encompasses the development work during the sprint (called sprint execution), and ends with the review and retrospective. The number of items in the product backlog is likely to be more than a development team can complete in a short-duration sprint. For that reason, at the beginning of each sprint, the development team must determine a subset of the product backlog items it believes it can complete, an activity called sprint planning.

As a brief aside, in 2011 a change in the “Scrum Guide” generated a debate about whether the appropriate term for describing the result of sprint planning is a forecast or a commitment [2, 1323]. Advocates of the word forecast like it because they feel that although the development team is making the best estimate that it can at the time, the estimate might change as more information becomes known during the course of the sprint. Some also believe that a commitment of the part of the team will cause the team to sacrifice quality to meet the commitment or will cause the team to “under-commit” to guarantee that the commitment is met.

All development teams should generate a forecast (estimate) of what they can deliver each sprint. However, many development teams would benefit from using the forecast to derive a commitment. Commitments support mutual trust between the product owner and the development team as well as within the development team. Also, commitments support reasonable short-term planning and decision making within an organization. And, when performing multi-team product development, commitments support synchronized planning, one team can make decisions based on what another team has committed to do.

To acquire confidence that the development team has made a reasonable commitment, the team members create a second backlog during sprint planning, called the sprint backlog. The sprint backlog describes, through a set of detailed tasks, how the team plans to design, build, integrate and test the selected subset of features from the product backlog during that particular sprint.

Next is print execution, where the development team performs the tasks necessary to realize the selected features. Each day during sprint execution, the team members help manage the flow of work by conducting a synchronization, inspection and adaptive planning activity known as the daily scrum. At the end of sprint execution the team has produced a potentially shippable product increment that represents some, but not all, of the product owner’s vision.

The scrum team completes the sprint by performing two inspect-and-adapt activities. In the first, called the sprint review, the stakeholders and Scrum team inspect the product being built. In the second, called the sprint retrospective, the Scrum team inspects the scrum process being used to create the product. The outcome of these activities might be adaptations that will make their way into the product backlog or be included as part of the team’s development process.

At this point, the scrum sprint repeats, beginning anew with the development team determining the next most important set of product backlog items it can complete. After an appropriate number of sprints have been completed, the product owner’s vision will be realized and the solution can be released.

2.6.2. The product backlog

Using Scrum, we always do the most valuable work first. The product owner with input from the rest of the Scrum team and stakeholders, is ultimately responsible for determining and managing the sequence of this work and communicating it in the form of a prioritized (or ordered) list known as the

product backlog. On new-product development the product backlog items initially are features required to meet the product owner's vision. For ongoing product development, the product backlog might also contain new features, changes to existing features, defects needing repair, technical improvements, and so on.

The product owner collaborates with internal and external stakeholders to gather and define the product backlog items. He then ensures that product backlog items are placed in the correct sequence (using factors such as value, cost, knowledge and risk) so that the high-value items appear at the top of the product backlog and the lower-value items appear toward the bottom. The product backlog is a constantly evolving artifact. Items can be added, deleted, and revised by the product owner as business conditions change, or as the Scrum team's understanding of the product grows (through feedback on the software produced during each sprint).

Overall the activity of creating and refining product backlog items, estimating them and prioritizing them is know as grooming. Before prioritizing, ordering or otherwise arranging the product backlog, the size of each item in the product backlog should be known. Size equates to cost, and product owners need to know an item's cost to properly determine its priority. Scrum does not dictate which, if any, size measure to use with the product backlog items. In practice, many teams use a relative size measure such as story points or ideal days. A relative size measure expresses the overall size of an item in such a way that the absolute value is not considered, but the relative size of an item compared to other items is considered.

2.6.3. Sprints

In Scrum, work is performed in iterations or cycles of up to a calendar month called sprints. The work completed in each sprint should create something of tangible value to the customer or user.

Sprints are timeboxed so they always have a fixed start and end date, and generally they should all be of the same duration. A new sprint immediately follows the completion of the previous sprint. As a rule we do not permit any goal altering changes in scope or personnel during a sprint. However, business needs sometimes make adherence to this rule impossible.

2.6.4. Sprint Planning

A product backlog may represent many weeks or months of work, which is much more than can be completed in a single, short sprint. To determine the most important subset of product backlog items to build in the next sprint, the product owner, development team, and Scrum Master perform sprint planning.

During sprint planning, the product owner and development team agree on a sprint goal that defines what the upcoming sprint is supposed to achieve. Using this goal, the development team reviews the product backlog and determines the high-priority items that the team can realistically accomplish in the upcoming sprint while working at a sustainable pace, a pace at which the development team can comfortably work for an extended period of time.

To acquire confidence in what it can get done, many development teams break down each targeted feature into a set of tasks. The collection of these tasks, along with their associated product backlog items, forms a second backlog called the sprint backlog.

The development team then provides an estimate (typically in hours) of the effort required to complete each task. Breaking product backlog items into tasks is a form of design and just-in-time planning for how to get the features done.

Most Scrum teams performing sprints of two weeks to a month in duration try to complete sprint planning in about four to eight hours. A one-week sprint should take no more than a couple of hours to plan (and probably less). During this time there are several approaches that can be used. One popular approach follows a simple cycle: Select a product backlog item (whenever possible, the next most important items as defined by the product owner), break the item down into tasks,

and determine if the selected item will reasonable fit within the sprint (in combination with other items targeted for the same sprint). If it does fit and there is more capacity to complete work, repeat the cycle until the team is out of capacity to do any more work.

And alternative approach would be for the product owner and team to select all of the target product backlog items at one time. The development team alone does the task breakdowns to confirm that it really can deliver all of the selected product backlog items.

2.6.5. Sprint Execution

Once the Scrum team finishes sprint planning and agrees on the content of the next sprint, the development team, guided by the Scrum Master's coaching, performs all of the task-level work necessary to get the features done, where "done" means there is a high degree of confidence that all of the work necessary for producing good-quality features has been completed.

Exactly what tasks the team performs depends of course on the nature of the work (for example, are we building hardware, or is this marketing work?).

Nobody tells the development team in what order or how to do the task-level work in the sprint backlog. Instead, team members define their own task-level work and then self-organize in any manner they feel is best for achieving the sprint goal.

2.6.6. Daily Scrum

Each day of the sprint, ideally at the same time, the development team members hold a timeboxed (15 minutes or less) daily scrum. This inspect-and adapt activity is sometimes referred to as the daily stand-up because of the common practice of everyone standing up during the meeting to help promote brevity.

A common approach to performing the daily scrum has the Scrum Master facilitating and each team member taking turns answering three questions for the benefit of the other team members:

- What did I accomplish since the last daily scrum?
- What do I plan to work on by the next daily scrum?
- What are the obstacles or impediments that are preventing me from making progress?

By answering these questions, everyone understands the big picture of what is occurring, how they are progressing toward the sprint goal, any modifications they want to make to their plans for the upcoming day's work, and what issues need to be addressed. The daily scrum is essential for helping the development team manage the fast, flexible flow of work within a sprint.

The daily scrum is not a problem-solving activity. Rather, many teams decide to talk about problems after the daily scrum and do so with a small group of interested people. The daily scrum also is not a traditional status meeting, especially the kind historically called by project managers so that they can get an update on the project's status. A daily scrum, however, can be useful to communicate the status of sprint backlog items among the development team members. Mainly, the daily scrum is an inspection, synchronization, and adaptive daily planning activity that helps a self-organizing team do its job better.

Although their use has fallen out of favor, Scrum has used the terms "pigs" and "chickens" to distinguish who should participate during the daily scrum versus who simply observe. The farm animals were borrowed from an old joke (which has several variants): "In a ham-and-eggs breakfast, the chicken is involved, but the pig is committed." Obviously the intent of using these terms in Scrum is to distinguish between those who are involved (the chickens) and those who are committed to meeting the sprint goal (the pigs). At the daily scrum, only the pigs should talk. The chickens, if any, should attend as observers.

It is most useful to consider everyone on the Scrum team a pig and anyone who isn't, a chicken. Not everyone agrees. For example, the product owner is not required to be at the daily scrum, so some consider him to be a chicken (the logic being, how can you be "committed" if you aren't required to attend?). This seems wrong to me, because I can't imagine how the product owner, as a member of the Scrum team, is any less committed to the outcome of a sprint than the development team. The metaphor of pigs and chickens breaks down if you try to apply it within a Scrum team.

2.6.7. Sprint Results

In Scrum, we refer to the sprint results as a potentially shippable product increment, meaning that whatever the Scrum team agreed to do is really done according to its agreed-upon definition of done. This definition specifies the degree of confidence that the work completed is of good quality and is potentially shippable. For example, when developing software, a bare-minimum definition of done should yield a complete slice of product functionality that is designed, built, integrated, tested and documented.

An aggressive definition of done enables the business to decide each sprint if it wants to ship (or deploy or release) what got built to internal or external customers.

To be clear, "potentially shippable" does not mean that what got built must actually be shipped. Shipping is a business decision, which is frequently influenced by things such as "Do we have enough features or enough of a customer workflow to justify a customer deployment?" or "Can our customers absorb another change given that we just gave them a release two weeks ago?"

Potentially shippable is better thought as a state of confidence that what got built in the sprint is actually done, meaning that there isn't materially important undone work (such as important testing or integration and so on) that needs to be completed before we can ship the results from the sprint, if shipping is our business desire.

As a practical matter, over time some teams may vary the definition of done. For example, in the early stages of game development, having features that are potentially shippable might not be economically feasible or desirable (given the exploratory nature of early game development). In these situations, an appropriate definition of done might be a slice of product functionality that is sufficiently functional and usable to generate feedback that enables the team to decide what work should be done next or how to do it.

2.6.8. Sprint Review

At the end of the sprint there are two additional inspect-and adapt activities. One is called the sprint review. The goal of this activity is to inspect and adapt the product that is being built. Critical to this activity is the conversation that takes place among its participants, which include the Scrum team, stakeholders, sponsors, customers, and interested members of other teams. The conversation is focused on reviewing the just-completed features in the context of the overall development effort. Everyone in attendance to help guide the forthcoming development to ensure that the most business-appropriate solution is created.

A successful review results in bidirectional information flow. The people who aren't on the Scrum team get to sync up on the development effort and help guide its direction. At the same time, the Scrum team members gain a deeper appreciation for the business and marketing side of their product by getting frequent feedback on the convergence of the product toward delighted customers or users. The sprint review therefore represents a schedules opportunity to inspect and adapt the product. As a matter of practice, people outside the Scrum team can perform intra-sprint feature reviews and provide feedback to help the Scrum team better achieve its sprint goal.

2.6.9. Sprint Retrospective

The second inspect-and-adapt activity at the end of the sprint is the sprint retrospective. This activity frequently occurs after the sprint review and before the next sprint planning.

Whereas the sprint review is a time to inspect and adapt the product, the sprint retrospective is an opportunity to inspect and adapt the process. During the sprint retrospective the development team, Scrum Master and product owner come together to discuss what is and is not working with Scrum and associated technical practices. The focus is on the continuous process improvement necessary to help a good Scrum team become great. At the end of a sprint retrospective the Scrum team should have identified and committed to a practical number of process improvement actions that will be undertaken by the Scrum team in the next sprint.

After the sprint retrospective is complicated, the whole cycle is repeated again. Starting with the next sprint-planning session, held to determine the current highest-value set of work for the team to focus on.

2.7. Agile compared to plan-driven processes

2.7.1. Plan-driven processes

One pure form of traditional, plan-driven development frequently goes by the term waterfall. However, that is just one example of a broader class of plan-plan driven processes (also known as traditional, sequential, anticipatory, predictive, or prescriptive development processes).

Plan-driven processes are so named because they attempt to plan for and anticipate up front all the features a user might want in the end product, and to determine how best to build those features. The idea here is that the better the planning, the better the understanding, and therefore the better the execution. Plan-driven processes are often called sequential processes because practitioners perform, in sequence, a complete requirements analysis followed by a complete design followed in turn by coding/building and then testing.

Plan-driven development works well if you are applying it to problems that are well defined, predictable and unlikely to undergo any significant change. The problem is that most product development efforts are anything but predictable, especially at the beginning. So, while a plan driven process gives the impression of an orderly, accountable and measurable approach, that impression can lead to a false sense of security. After all, developing a product rarely goes as planned.

For many, a plan-driven, sequential process just makes sense, understand it, design it, code it, test it, and deploy it, all according to a well-defined, prescribed plan. There is a belief that it should work. If applying a plan-driven approach doesn't work, the prevailing attitude is that we must have done something wrong. Even if a plan-driven process repeatedly produces disappointing results, many organizations continue to apply the same approach, sure that if they just do it better, their results will improve. The problem, however is not with the execution. It's that plan driven approaches are based on a set of beliefs that do not match the uncertainty inherent in most product development efforts.

Scrum, on the other hand, is based on a different set of beliefs - ones that do map well to problems with enough uncertainty to make high levels of predictability difficult.

2.7.2. Similarities

Plan-driven and Agile processes share the same goal. What is sometimes lost in Traditional World/Agile World discussions is the fact that both groups have the same goal—to deliver a quality product in a predictable, efficient and responsive manner. Both worlds do the same "types" of things—define, gather, analyze, design, code, test, release, maintain, retire. It's how they do these

things that are different. However, we want to point out that while there are similarities, there are significant differences. The two methods cannot be thought of as the same.

The plan-driven and the agile processes use many of the same principles. The traditional plan-driven process grew out of the perception that the best way to manage the “software crisis” was to:

- plan the work out completely before beginning
- lock down requirements early
- institute multiple reviews
- move forward in a step-by-step, sequential manner
- move forward only when all parts of the previous steps were complete
- capture all details with extensive documentation

Taken individually, it’s difficult to argue with these if they are appropriate (you really can state all your requirements up front) and they are done wisely. For example, gold-plating should be avoided, progress reviews are reasonable management tools, senior leaders do need to be kept informed of progress and issues, designs should be documented to support future work, etc. Because these principles have value, they are used in the Agile World as well.

The plan-driven and the agile processes use the same basic building blocks. They both work with the same basic programmatic building blocks:

- scope
- cost
- schedule
- performance

In its simplest form, the plan-driven process sets the scope up front (through requirements) and then allows cost, schedule, and performance to vary. Again in its simplest terms, the Agile process sets the cost, schedule, and performance up front and then allows the scope to vary.

In addition, both the plan-driven and the agile process use the same technical or development building blocks:

- analyze the requirement
- design a capability to satisfy the requirement
- build the capability
- test the capability to ensure the requirement is met
- deploy the capability

2.7.3. Differences

The most important difference, especially in dynamic environments is that the waterfall method struggles to deliver as it constantly looks back at long-fixed requirements and priorities while the agile method adapts as it delivers by constantly looking forward at evolving requirements and priorities.

The following table points out the most important differences between the traditional development, like the waterfall methodology, and the agile. [8, 8]

	Traditional development	**Agile development**
Fundamental hypothesis	Systems are fully specifiable, predictable and are developed through extended and detailed planning	High quality adaptive software is developed by small teams that use the principle of continuous improvement of design and

		testing based on fast feedback and change
Management style	Command and control	Leadership and collaboration
Knowledge management	Explicit	Tacit
Communication	Formal	Informal
Development model	Life cycle model (waterfall, spiral or modified models)	Evolutionary-delivery model
Organizational structure	Mechanic (bureaucratic, high formalization), targeting large organization	Organic (flexible and participative, encourages social cooperation), targeting small and medium organizations
Quality control	Difficult planning and strict control. Difficult and late testing	Permanent control or requirements, design and solutions. Permanent testing
User requirements	Detailed and defined before coding/implementation	Interactive input
Cost of restart	High	Low
Development direction	Fixed	Easily changeable
Testing	After coding is completed	Every iteration
Client involvement	Low	High
Additional abilities required from developers	Nothing in particular	Interpersonal abilities and basic knowledge of the business
Appropriate scale of the project	Large scale	Low and medium scale
Developers	Oriented on plan, with adequate abilities, access to external knowledge	Agile, with advanced knowledge, co-located and cooperative
Clients	With access to knowledge, cooperative, representative and empowered	Dedicated, knowledgeable, cooperative, representative and empowered
Requirements	Very stable, known in advance	Emergent, with rapid changes

Architecture	Design for current and predictable requirements	Design for current requirements
Remodeling	Expensive	Not expensive

2.7.4. Variability and Uncertainty

Plan-driven processes treat product development like manufacturing. They shun variability and encourage conformance to a defined process. The problem is that product development is not at all like product manufacturing. In manufacturing our goal is to take a fixed set of requirements and follow a sequential set of well-understood steps to manufacture a finished product that is the same every time.

In product development, however, the goal is to create the unique single instance of the product, not to manufacture the product. This single instance is analogous to a unique recipe. We don't want to create the same recipe twice. If we do, we have wasted our money. Instead, we want to create a unique recipe for a new product. Some amount of variability is necessary to produce a different product each time. In fact, every feature we build within a product is different from every other feature within that product, so we need variability even at this level. Only once we have the recipe do we manufacture the product, in the case of software products, as easily as copying bits.

Plan-driven, sequential development assumes that we will get things right up front and that most or all of the product pieces will come together late in the effort.

Scrum, on the other hand, is based on iterative and incremental development. Although these two terms are frequently used as if they were a single concept, iterative development is actually distinct from incremental development.

Iterative development acknowledges that we will probably get things wrong before we get them right and that we will do things poorly before we do them well (Goldberg and Rubin 1995). As such, iterative development is a planned rework strategy. We use multiple passes to improve what we are building so that we can converge on a good solution. For example, we might start by creating a prototype to acquire important knowledge about a poorly known piece of the product. Then we might create a revised version that is somewhat better, which might in turn be followed by a pretty good version that is somewhat better, which might in turn be followed by a pretty good version.

Iterative development is an excellent way to improve the product as it is being developed. The biggest downside to iterative development is that in the presence of uncertainty it can be difficult up front to determine how many improvement passes will be necessary.

Incremental development is based on the age-old principle of "Build some of it before you build all of it" [2]. We avoid having one large, big-bang-style event at the end of development where all the pieces come together and the entire product is delivered. Instead, we break the product into smaller pieces so that we can build some of it, learn how each piece is to survive in the environment in which it must exist, adapt based on what we learn, and then build more of it.

Incremental development gives us the important information that allows us to adapt our development effort and to change how we proceed. The biggest drawback to incremental development is that by building in pieces, we risk missing the big picture (we see the trees but not the forest).

Scrum leverages the benefits of both iterative and incremental development, while negating the disadvantages of using them individually. Scrum does this by using both ideas in an adaptive series of timeboxed iterations called sprints.

During each sprint we perform all of the activities necessary to create a working product increment (some of the product, not all of it). This all-at-once approach has the benefit of quickly validating the assumptions that are made when developing product features. For example, we make some design decisions, create some code based on those decisions, and then test the design and the code, all in one sprint. By doing all of the related work within one sprint, we are able to quickly rework features, thus achieving the benefits of iterative development, without having to specifically plan for additional iterations.

A misuse of the sprint concept is to focus each sprint on just one type of work, for example, sprint 1 (analysis), sprint 2 (design), sprint 3 (coding), sprint 4 (testing). Such an approach attempts to overlay Scrum with a waterfall-style work breakdown structure. This approach is often referred to as scrummerfall.

In Scrum, we don't work on a phase at a time, we work on a feature at a time. So, by the end of a sprint we have created a valuable product increment (some but not all of the product features). That increment includes or is integrated and tested with any previously developed features, otherwise, it is not considered done. At the end of the sprint, we can get feedback on the newly completed features within the context of already completed features. This helps us view the product from a more of a big-picture perspective than we might otherwise have.

We receive feedback on the sprint results, which allows us to adapt. We can choose different features to work on in the next sprint or alter the process we will use to build the next set of features. In some cases, we might learn that the increment, though it technically fits the bill, isn't as good as it could be. When that happens, we can schedule rework for a future sprint as part of our commitment to iterative development and continuous improvement. This helps overcome the issue of not knowing up front exactly how many improvement passes we will need. Scrum does not require that we predetermine a set number of iterations. The continuous stream of feedback will guide us to do the appropriate and economically sensible number of iterations while developing the product incrementally.

Plan-driven processes and Scrum are fundamentally different along several dimensions. A plan-driven, sequential development development process assumes little or no output variability. It follows a well-defined set of steps and uses only small amounts of feedback late in the process. In contrast, Scrum embraces the fact that in product development, some level of variability is required in order to build something new. Scrum also assumes that the process necessary to create the product is complex and therefore would defy a complete up-front definition. Furthermore, it generates early and frequent feedback to ensure that the right product is built and that the product is built right.

At the heart of Scrum are the principles of inspection, adaptation and transparency (referred to collectively by Schwaber and Beedle 2001 as empirical process control). In Scrum, we inspect and adapt not only what we are building but also how we are building it.

To do this well, we rely on transparency: all of the information that is important to producing a product must be available to the people involved in creating the product. Transparency makes inspection possible, which is needed for adaptation. Transparency also allows everyone concerned to observe and understand what is happening. It leads to more communication and it establishes trust (both in the process and among the team members).

Developing new products is a complex endeavor with a high degree of uncertainty. That uncertainty can be divided into two broad categories:

- End uncertainty (what uncertainty): uncertainty surrounding the features of the final product.
- Means uncertainty (how uncertainty): uncertainty surrounding the process and technologies used to develop a product.

In particular environments or with particular products there might also be customer uncertainty (who uncertainty)[2]. For example, start-up organizations (including large organizations that focus on

novel products) may only have assumptions as to who the actual customers of their products will be. This uncertainty must be addressed or they might build brilliant products for the wrong markets.

Traditional, sequential development processes focus first on elimination al end uncertainty by fully defining up front what is to be built, and only the addressing means uncertainty.

This simplistic, linear approach to uncertainty reduction is ill suited to complex domain of product development, where our actions and the environment in which we operate mutually constrain one another. For example:

- We decide to build a feature (our action).
- We then show that feature to a customer, who, once he sees it, changes his mind about what he really wants, or realizes that he did not adequately convey the details of the feature (our action elicits a response from the environment).
- We make design changes based on the feedback (the environment's reaction influences us to take another unforeseen action).

In Scrum, we do not constrain ourselves by fully addressing one type of uncertainty before we address the next type. Instead, we take a more holistic approach and focus on simultaneously reducing all uncertainties (end, means, customer and so on). Of course, at any point in time we might focus more on one type of uncertainty than another. Simultaneously addressing multiple types of uncertainty is facilitated by iterative and incremental development and guided by constant inspection, adaptation, and transparency. Such an approach allows us to opportunistically probe and explore our environment to identify and learn about the unknown unknowns (the things that we don't yet know that we don't know) as they emerge.

2.7.5. Prediction and adaptation

When using Scrum, we are constantly balancing the desire for prediction with the need for adaptation. The five agile principles related to this topic are:

- Keep options open
- Accept that you can't get it right up front
- Favor an adaptive, exploratory approach
- Embrace change in an economically sensible way
- balance predictive up-front work with adaptive just-in-time work

These principles make Agile differ a lot from the traditional plan-driven methodologies. Plan-driven, sequential development requires that important decisions in areas like requirements or design be made, reviewed and approved within their respective phases. Furthermore, these decisions must be made before we can transition to the next phase, even if those decisions are based on limited knowledge.

Scrum contends that we should never make a premature decision just because a generic process would dictate that now is the appointed time to make one. Instead, when using Scrum, we favor a strategy of keeping our options open. Often this principle is referred to as the last responsible moment (LRM) (Poppendieck and Poppendieck 2003), meaning that we delay commitment and do not make important and irreversible decisions until the last responsible moment. And when is that? When the cost of not making a decision becomes greater than the cost of making a decision. At that moment, we make the decision.

To appreciate this principle, consider this. On the first day of a product development effort we have the least information about what we are doing. On each subsequent day of the development effort, we learn a little more. Why, then, would we ever choose to make all of the most critical, and perhaps irreversible decisions on the first day or very early on? Most of us would prefer to wait until we have more information so that we can make a more informed decision. As we acquire a better understanding regarding the decision, the cost of deciding declines (the likelihood of making a bad decision declines because of increasing market or technical certainty). That's why we should wait until we have better information before committing to a decision.

Plan-driven processes not only mandate full requirements and a complete plan, they also assume that we can “get it right” up front. The reality is that it is very unlikely that we can get all of the requirements, or the detailed plans based on those requirements, correct up front. What’s worse is that when the requirements do change, we have to modify the baseline requirements and plans to match the current reality.

In Scrum, we acknowledge that we can’t get all of the requirements or the plans right up front. In fact, we believe that trying to do so could be dangerous because we are likely missing important knowledge, leading to the creation of a large quantity of low-quality requirements.

When using a plan-driven, sequential process, a large number of requirements are produced early on when we have the least cumulative knowledge about the product. This approach is risky, and because there is an illusion that we have eliminated end uncertainty. It is also potentially very wasteful when our understanding improves or things change.

With Scrum, we still produce some requirements and plans up front, but just sufficiently and with the assumption that we will fill in the details of those requirements and plans as we learn more about the product we are building. After all, even if we think we’re 100% certain about what to build and how to organize up front the work to build it, we will learn where we are wrong as soon as we subject our early incremental deliverables to the environment in which they must exist. At that point, all of the inconvenient realities of what is really needed will drive us to make changes.

Plan-driven, sequential processes focus on using what is currently know and predicting what isn’t known. Scrum favors a more adaptive trial-and error approach based on appropriate use of exploration.

Exploration refers to times when we choose to gain knowledge by doing some activity, such as building a prototype, creating a proof of concept, performing a study or conducting an experiment. In other words, when faced with uncertainty, we buy information by exploring.

Our tools and technologies significantly influence the cost of exploration. Historically software product development exploration has been expensive, a fact that favored a more predictive, try-to-get-it-right-up-front approach.

Fortunately, tools and technologies have gotten better nowadays and the cost of exploring has come way down. There is no longer an economic disincentive to explore. In fact, nowadays, it’s often cheaper to adapt to user feedback based on building something fast than it is to invest in trying to get everything right up front. Good thing, too, because the context in which our solutions must exist is getting increasingly more complex.

In Scrum, if we have enough knowledge to make an informed, reasonable step forward with our solution, we advance. However, when faced with uncertainty, rather than trying to predict it away, we use low-cost exploration to buy relevant information that we can then use to make an informed, reasonable step forward with our solution. The feedback from our action will help us determine if and when we need further exploration.

When using sequential development, change, as we have all learned, is substantially more expensive late than it is early on. As an example, a change made during analysis might cost $1. That same change made late during testing might cost $1000. Why is this so? If we make a mistake during analysis and find it during analysis, it is an inexpensive fix. If that same error is not found until design, we have to fix not only the incorrect requirement, but potentially parts of our design based on the wrong requirement. This compounding of the error continues through each subsequent phase, making what might have been a small error to correct during analysis into a much larger error to correct in testing or operations.

To avoid late changes, sequential processes seek to carefully control and minimize any changing requirements or designs by improving the accuracy of the predictions about what the system need to do or how it is supposed to do it.

Unfortunately, being excessively predictive in early-activity phases, often has the opposite effect. It not only fails to eliminate change, it actually contributes to deliveries that are late and over

budget as well. Why this paradoxical truth? First, the desire to eliminate expensive change force us to overinvest in each phase, doing more work than is necessary and practical. Second, we're forced to make decisions based on important assumptions early in the process, before we have validated these assumptions with feedback from our stakeholders based on our working assets. As a result, we produce a large inventory of work products based on these assumptions. Later, this inventory will likely have to be corrected or discarded as we validate (or invalidate) our assumptions, or change happens (for example, requirements emerge or evolve), as it always will. This fits the classic pattern of a self-fulfilling prophecy.

In Scrum, we assume that change is the norm. We believe that we can't predict away the inherent uncertainty that exists during product development by working longer and harder up front. Thus, we must be prepared to embrace change. And when that change occurs, we want the economics to be more appealing than with traditional development, even when the change happens later in the product development effort. Our goal, therefore is to keep the cost-of-change curve flat for as long as possible, making it economically sensible to embrace even late change.

We can achieve that goal by managing the amount of work in process and the flow of that work so that the cost of change when using Scrum is less affected by time than it is with sequential projects.

Regardless of which product development approach we use, we want the following relationship to be true: a small change in requirements should yield a proportionally small change in implementation and therefore in cost (obviously we would expect a larger change to cost more). Another desirable property of this relationship is that we want it to be true regardless of when the change request is made.

With Scrum, we produce many work products (such as detailed requirements, designs and test cases) in a just-in-time fashion, avoiding the creation of potentially unnecessary artifacts. As a result, when a change is made, there are typically far fewer artifacts or constraining decisions based on assumptions that might be discarded or reworked, thus keeping the cost more proportional to the size of the requested change.

Using sequential development, the early creation of artifacts and push from premature decision making ultimately mean that the cost of a change rises rapidly over time as inventory grows. When developing with Scrum, there does come a time when the cost of change will no longer be proportional to the size of the request but this point in time occurs later.

A fundamental belief of plan-driven development is that detailed up-front requirements and planning are critical and should be completed before moving on to later stages. In Scrum, it is believed that up-front work should be helpful without being excessive.

With Scrum, we acknowledge that it is not possible to get requirements and plans precisely right up front. Does that mean we should do no requirements or planning work up front? Of course not. Scrum is about finding balance, between predictive up-front work and adaptive just-in-time work.

When developing a product, the balance point should be set in an economically sensible way to maximize the amount of ongoing adaptation based on fast feedback and minimize the amount of up-front prediction, while still meeting compliance, regulatory and/or corporate objectives.

Exactly how that balance is achieved is driven in part by the type of product being built, the degree of uncertainty that exists in both what we want to build (end uncertainty) and how we want to build it (means uncertainty), and the constraints placed on the development. Being overly predictive would require us to make many assumptions in the presence of great uncertainty. Being overly adaptive could cause us to live in a state of constant change, making our work feel inefficient and chaotic. To rapidly develop innovative products we need to operate in a space where adaptability is counterbalanced by just enough prediction to keep us from sliding into chaos. The Scrum framework operates well at this balance point of order and chaos.

3. Technologies and Architecture

3.1. JavaScript

This project uses JavaScript both on the front-end and on the back end. JavaScript is a scripting programming language that is used mainly on the web, and it is the main programming language ran by the browsers.

JavaScript is a prototype-based, multi-paradigm scripting language that is dynamic, and supports object-oriented, imperative, and functional programming styles. It was created in 1995 in order to add some programming logic in the web pages displayed in Netscape Navigator browser.

After its adoption outside of Netscape, a standard document was written to describe the way the JavaScript language should work to make sure the various pieces of software that claimed to support JavaScript were actually talking about the same language. This is called the ECMAScript standard, after the Ecma International organization that did the standardization. In practice, the terms ECMAScript and JavaScript can be used interchangeably—they are two names for the same language.

There have been several versions of JavaScript. ECMAScript version 3 was the widely supported version in the time of JavaScript's ascent to dominance, roughly between 2000 and 2010. During this time, work was underway on an ambitious version 4, which planned a number of radical improvements and extensions to the language. Changing a living, widely used language in such a radical way turned out to be politically difficult, and work on the version 4 was abandoned in 2008, leading to the much less ambitious version 5 coming out in 2009. We're now at the point where all major browsers support version 5 and version 6 is in the process of being finalized, and some browsers are starting to support new features from this version.

Web browsers are not the only platforms on which JavaScript is used. Some databases, such as MongoDB, which will be used for this project, and CouchDB, use JavaScript as their scripting and query language. Several platforms for desktop and server programming, most notably the Node.js, which again will be used for this project, are providing a powerful environment for programming JavaScript outside of the browser.

Using JavaScript both on the client and on the server has many advantages. First of all, the same libraries can be used on both ends, which makes us, the developers, feel familiar with the code on both sides. Also, code that has the same functionality on the server can be reused in the browser and vice versa. In addition to that, communication between the client and the server is made easy, by transferring JSON objects, which are natively supported by JavaScript.

JSON, the data format that will be used in our application, stands for JavaScript Object Notation. JSON objects is a human readable format of text for data transmission, like XML. They resemble JavaScript objects, but of course do not support methods. It was created for JavaScript but now it is language independent data format.

3.2. Node.js

Node.js is many things, but mostly it's a way of running JavaScript outside the web browser. Node.js allows this popular programming language to be applied in many more contexts, in particular on web servers. There are several notable features about Node.js that make it worthy of interest. Node is a wrapper around the high-performance V8 JavaScript runtime from the Google Chrome browser. Node tunes V8 to work better in contexts other than the browser, mostly by providing additional APIs that are optimized for specific use cases. For example, in a server context, manipulation of binary data is often necessary. This is poorly supported by the JavaScript language and, as a result, V8. Node's Buffer class provides easy manipulation of binary data. Thus, Node

doesn’t just provide direct access to the V8 JavaScript runtime. It also makes JavaScript more useful for the contexts in which people use Node.

V8 itself uses some of the newest techniques in compiler technology. This often allows code written in a high-level language such as JavaScript to perform similarly to code written in a lower-level language, such as C, with a fraction of the development cost. This focus on performance is a key aspect of Node. JavaScript is an event-driven language, and Node uses this to its advantage to produce highly scalable servers. Using an architecture called an event loop, Node makes programming highly scalable servers both easy and safe. There are various strategies that are used to make servers performant. Node has chosen an architecture that performs very well but also reduces the complexity for the application developer. This is an extremely important feature. Programming concurrency is hard and fraught with danger. Node sidesteps this challenge while still offering impressive performance.

To support the event-loop approach, Node supplies a set of “nonblocking” libraries. In essence, these are interfaces to things such as the filesystem or databases, which operate in an event-driven way. When you make a request to the filesystem, rather than requiring Node to wait for the hard drive to spin up and retrieve the file, the nonblocking interface simply notifies Node when it has access, in the same way that web browsers notify your code about an onclick event. This model simplifies access to slow resources in a scalable way that is intuitive to JavaScript programmers and easy to learn for everyone else.

Although not unique to Node, supporting JavaScript on the server is also a powerful feature. Whether we like it or not, the browser environment gives us little choice of programming languages. Certainly, JavaScript is the only choice if we would like our code to work in any reasonable percentage of browsers. To achieve any aspirations of sharing code between the server and the browser, we must use JavaScript. Due to the increasing complexity of client applications that we are building in the browser using JavaScript (such as Gmail), the more code we can share between the browser and the server, the more we can reduce the cost of creating rich web applications. Because we must rely on JavaScript in the browser, having a server-side environment that uses JavaScript opens the door to code sharing in a way that is not possible with other server-side languages, such as PHP, Java, Ruby, or Python. Although there are other platforms that support programming web servers with JavaScript, Node is quickly becoming the dominant platform in the space.

Aside from what you can build with Node, one extremely pleasing aspect is how much you can build for Node. Node is extremely extensible, with a large volume of community modules that have been built in the relatively short time since the project’s release. Many of these are drivers to connect with databases or other software, but many are also useful software applications in their own right.

The last reason to celebrate Node, but certainly not the least important, is its community. The Node project is still very young, and yet rarely have we seen such fervor around a project. Both novices and experts have coalesced around the project to use and contribute to Node, making it both a pleasure to explore and a supportive place to share and get advice. [4, 3]

In our application we chose to use node because there will be a big part of the logic on the front-end, written in JavaScript, so having to write the back-end in JavaScript as well, is much more convenient. Also, for the purpose of the thesis, we wanted to learn a new technology and experiment with it, so node.js was the right choice.

3.3. Express

Express is a node.js framework that includes a small set of common web application features, kept to a minimum in order to maintain the node.js style. It is built on top of the Connect (which is a middleware) and makes use of its middleware architecture. Its features extend Connect to allow a variety of common web applications’ use cases, such as the inclusion of modular HTML template

engines, extending the response object to support various data format outputs, a routing system, and much more. [5, 1129]

When express is used, it helps you have a better project structure, proper configurations and it also helps you break the application logic into different modules. Express organizes the routes of the application as well, using methods for every request type (get, post, put, delete). In the application we will have one file where all the routes will be declared, making it easier to manage. This is where the RESTful API will be formed.

3.4. MongoDB

MongoDB is an open-source document database that provides high performance, high availability, and automatic scaling.

A record in MongoDB is a document, which is a data structure composed of field and value pairs. MongoDB documents are similar to JSON objects. The values of fields may include other documents, arrays, and arrays of documents.

The advantages of using documents are:

- Documents (i.e. objects) correspond to native data types in many programming languages.
- Embedded documents and arrays reduce need for expensive joins.
- Dynamic schema supports fluent polymorphism

MongoDB provides high performance data persistence. In particular,

- Support for embedded data models reduces I/O activity on database system.
- Indexes support faster queries and can include keys from embedded documents and arrays.
- High Availability

MongoDB has the same concept of a database with which you are likely already familiar. Within a MongoDB instance you can have zero or more databases, each acting as high-level containers for everything else.

A database can have zero or more collections. A collection shares enough in common with a traditional table that you can safely think of the two as the same thing. Collections are made up of zero or more documents. Again, a document can safely be thought of as a row. A document is made up of one or more fields, which are a lot like columns.

Indexes in MongoDB function mostly like their RDBMS counterparts. Cursors are different than the other five concepts but they are important enough, and often overlooked, that I think they are worthy of their own discussion. The important thing to understand about cursors is that when you ask MongoDB for data, it returns a pointer to the result set called a cursor, which we can do things to, such as counting or skipping ahead, before actually pulling down data. [6, 4]

One of the main reasons we chose MongoDB is that it uses a data format named BSON (Binary JSON), which resembles json and JavaScript objects. In this way, we avoid converting our data in order to store them in the db, as JavaScript dates, key-value objects, JavaScript arrays, are all accepted without any conversions. Also, MongoDB has very good drivers and support by the Node.js community. We will use one plugin named “mongoose” in order to handle models and queries for MongoDB.

3.5. LESS

Less is a dynamic style sheet language that can be compiled into CSS, or can run on the client-side and server-side. Less is influenced by Sass and has influenced the newer "SCSS" syntax of Sass, which adapted its CSS-like block formatting syntax. Less is open source. Its first version was written

in Ruby, however in the later versions, use of Ruby has been deprecated and replaced by javascript. The indented syntax of Less is a nested metalanguage, as valid CSS is valid Less code with the same semantics. Less provides the following mechanisms: variables, nesting, mixins, operators and functions. The main difference between Less and other CSS precompilers being that Less allows real-time compilation via less.js by the browser.

We will compile less on the browser while the application works in development mode, in order to avoid compiling it often. Later we will use grunt tasks that will compile less to CSS on the server-side, in order to avoid the compilation time in every request.

3.6. JADE

JADE is a template engine used by node and the express framework that we will use. It generates HTML documents dynamically, and it injects variables inside, and also supports conditions, loops and nesting templates into other templates.

We will use JADE for the core of the application, so that only the main page will be written in it, because AngularJS will handle the variables, loops and conditions in the page, since we want live updates in the data. The templates that AngularJS will use will be plain HTML documents, there is no need to compile them from JADE.

3.7. AngularJS

Back in 2009, while building their JSON as platform service, developers Misko Hevery and Adam Abrons noticed that the common JavaScript libraries weren't enough. The nature of their rich web applications raised the need for a more structured framework that would reduce redundant work and keep the project code organized. Abandoning their original idea, they decided to focus on the development of their framework, naming it AngularJS and releasing it under an open source license. The idea was to bridge the gap between JavaScript and HTML and to help popularize single-page application development. [5, 2915]

AngularJS is a front-end JavaScript framework designed to build single-page applications using the MVC architecture. The AngularJS approach is to extend the functionality of HTML using special attributes that bind JavaScript business logic with HTML elements. The AngularJS ability to extend HTML allows cleaner DOM manipulation through client-side templating and two-way data binding that seamlessly synchronizes between models and views. AngularJS also improves the application's code structure and testability using MVC and dependency injection.

The core module of AngularJS is loaded with all the tools needed to create an application. The angular global object contains a set of methods that can be used to create and launch our application. It also wraps a leaner subset of jQuery, called jqLite, which enables Angular to perform basic DOM manipulation. Another key feature of the angular object is its static methods, which can be used in order to create, manipulate and edit the basic entities of the application, including the creation and retrieval of modules.

With AngularJS, everything is encapsulated in modules. Whether you choose to work with a single application module or break your application into various modules, your AngularJS application will rely on at least one module to operate. This core module is usually called application module. The method "module" attached to the angular object, is used in order to retrieve and create modules, and set its dependencies. Our application is relatively small, so we will use just one module, the application module, for the whole application.

The AngularJS team has decided to support the continuous development of the framework by breaking Angular's functionality into external modules. These modules are being developed by the same team that creates the core framework and are being installed separately to provide extra functionality that is not required by the core framework to operate. We will use the "router" external module that we will use in order to create all the routes in the front-end of the application.

In the same way the AngularJS team supports its external modules, it also encourages outside vendors to create third-party modules, which extend the framework functionality and provide developers with an easier starting point. We will use an external module in order to create charts in the “statistics” section of our application.

One of the most popular features of AngularJS is its two-way data binding mechanism. Two-way data binding enables AngularJS applications to always keep the model synchronized with the view and vice versa. This means that what the view renders is always the projection of the model.

Most templating systems bind the model with templates in one direction. This is also the case with the JADE template engine that we will use only for the core of the application, because we need the two-way data binding that AngularJS provides, for the rest of the application. AngularJS uses the browser to compile HTML templates, which contain special directive and binding instructions that produce a live view. Any events that happen in the view automatically update the model, while any changes occurring in the model immediately get propagated to the view. This means the model is always the single source of data for the application state, which substantially improves the development process.

A dependency injection is a software design pattern popularized by a software engineer named Martin Fowler. [5, 2962] The main principle behind dependency injection is the inversion of control in a software development architecture. This principle helps the developers create code and modules that are more testable.

4. Requirement Analysis

4.1. Term Definitions

4.1.1. Activities

I will refer to the performances of the athletes at competitions as activities. They will have several parameters such as the performance, the discipline, the date, the location of the event, the name of the event, the place the athlete got in the event, notes, and whether it is a private activity or not.

4.1.2. Statistics

The diagrams that show statistics of the athletes’ performances through time and the respective section of the application will be referred to as “statistics”.

4.1.3. Discipline

A discipline is the sport that an athlete does. An athlete, as user of the application, has a main discipline, the discipline that they prefer and possibly have most records of.

4.1.5. Filtering

Filtering would be the process of narrowing down the results of the activities or the data that are used in order to create the statistics. Filtering may be applied by discipline or by date.

4.2. Functionality

The final version of the application should be a fully functional tool for the athletes, with all the settings and functionalities that will make the process of tracking their progress easy and satisfying.

Before designing the application, the requirements of the sections and the overall functionality is defined.

4.2.1. Login

As the application will be accessed by individual users, there should be a login section. Since an email address is needed in many cases by the application, the login credentials are an email address and a password, avoiding unnecessary values like usernames. The login section should create a session for the user and keep them logged in, until they decide to logout. It should also be simplistic and look similar to the registration section. There should also be a link to the “Forgot password” section.

4.2.2. Register

There should be a section where the users can create their accounts. The registration section should be simple, look like the login section and after a validated registration, records of the new users should be added to the database. The required fields of the registration section should be five: a first name, a last name, an email, a password and the same password repeated for safety reasons. The first name and the last name are required because they should be displayed in the user’s profile which makes the user easier to find and be recognised by other users. They also make the application feel more personal, which may work as a self inspiration for their progress. The email is used for the registration process and for email notifications. The password is used for the login process and it should repeated in order to avoid typographical errors.

4.2.3. Email validation

The email address that the user submitted during the registration process should be validated. Right after they have registered, an email should be sent to the given address, providing a unique link, created specifically for this user. If they click on this link, the application can assume that it was clicked from the existing email address of the user, that they have access to that address, and therefore it will complete the registration process, allowing the user to use the application. After the validation of the email address, the user should be immediately logged in, since there is no reason to ask for the password when the link is clicked in the user’s own email client.

4.2.4. Automatic detection of user’s preferences

When the user logs in for the first time (after their email address has been validated), some preferences should be preselected. The country of the user should be detected and, using this information, the “country” setting and the “unit system” setting, should be preselected by the name of the country of the user and the unit system that is used in this country respectively, and they should be saved as the default settings of their account.

Also, the profile will be private by default, meaning that other users won’t be able to access it at all. This will be preselected for all users after registration.

When these settings get set automatically, a notification will be showing the values of these settings to the user, stating that if these settings are not correct or if the user wants to change them, they can visit the settings page and do that manually, and a link of that page should be provided in the notification.

4.2.5. Forgot password

Following the “Forgot password” link should lead the users to a page where they can enter their email address, making a request for a new password. After submitting the form, an email should be

sent to the given email address and that email should include a unique link to a page where the user can enter a new password and repeat it once for safety reasons. After submitting this form, the password of the user should have changed to the new one and the user should be logged in without using the login form.

4.2.6. Profile

The profile should be the main section of the application. The registration process and the login process should lead here. In this section, the profile data of the user should be visible and the activities can be created, deleted, edited, and they can be shown and filtered.

The profile section should display the user's profile picture, the first and last name, the discipline and the country. There should also be a list of all the activities previously submitted, which should be editable and deletable, and a form to submit new activities. There should also be a way to filter the activities shown in the list.

4.2.7. Add activity

In the profile page, it should be possible to create new activities. There should be a form where a user can select their discipline, and according to their selection, several more fields should appear. In case the user has chosen a main discipline in the settings, that discipline should be preselected.

There should be one or multiple fields that accept the user's performance. Other fields needed are the date of the activity, a field showing whether the activity is training or not, a field showing the location it took place, a text area for notes, and if the activity is a competition, there should also be fields of the name of the competition and the place that the athlete came.

Since the form is big for the page, and not always needed, the fields should be hidden, when the page is accessed, and the should be shown when the user clicks on the "add activity" option. When the user has submitted the form, the form should close again and the new activity should be added to the list of the activities, in the correct place, according to the date of the activity.

4.2.8. Edit activity

Each activity in the list, in the profile page, should have a way to make the activity editable. The activity should turn into a form, similar to the one that is used to submit new activities, and the data of the activity should be in the appropriate fields. The user should be able to change the values and submit the form. After the form is submitted, the activity should change to its initial state, not editable anymore, with the new data.

4.2.9. Delete activity

In a similar way that an activity can be turned to editable, an activity should be deleted. When the user chooses to delete an activity, a confirmation message should be shown, and if the user confirms the deletion, the activity should be removed from the list. Otherwise, the message should disappear and the list should remain intact.

4.2.10 Activity filtering

It should be possible to filter the activities in the profile. There should be a way to display all the activities or the activities of only one discipline.

4.2.11. User search

When a user is logged in, on every page of the application, there should be a field that can be used in order to search for others who use the application. The search should require a first name or a last name or both, the server should search for the users on every user's key press, and the top five results should be displayed right below the search area. Clicking on one of the results should lead the user to the profile of the selected user.

4.2.12. Visiting profiles

Users should be able to visit the profiles of other users, when they are not private. The profile should have the same interface as when its owner visits it, but there should not be any options to add, edit or delete activities. The whole form used to add activities should be missing, and the options to remove or edit individual activities should not appear on them as well. There should still be a way to filter the activities. Also, in the same way, the statistics of the user should be available for visiting.

4.2.13. Statistics

The data entered in the profile should be displayed as graphs in the "statistics" section. The graphs should have the performance on the y axis and the dates on the x axis. There should be a way to change the discipline shown in the graph and there should be a way to focus on a certain timespan.

4.2.14. Profile settings

There should be a section where the settings and the information that will appear in the profile can be edited. The section of the profile settings, should have as editable fields the user's first name, last name, main discipline, gender, birthday, country, profile picture, privacy and a short text about themselves.

Changing the first name, last name, profile picture, country or discipline, should change the relevant information in the profile page. Also, changing the first and last name should change the way that particular user can be found using the search functionality. The old first and last name shouldn't be used as search parameters anymore.

Changing the privacy of the profile to public should make the profile visible to others, while changing it to private should remove it from the search results and visiting that profile by its url should show a page stating that this profile does not exist.

4.2.15. Account settings

In the same page as the profile settings, there should be a section with editable fields that affect values related to the user's account. These are the password, the language, the date format, the system of measurement and the url of the user's profile.

When the user selects to change the password, three fields should appear, one that will require the old password, and two that will require the new one for safety reasons. Entering the correct old password and the exact same password twice as a new password, should immediately take effect.

Changing the language setting should change the language used for the whole interface of the application. The value preselected when the user first accesses the application, should be the default language of the user's country. The files of the translated values should be separated from the rest of the application, so that changes applied to them will affect all occurrences of the value throughout the application.

The date format input should have two options: day-month-year and month-day-year. Changing this setting should change they way the dates are shown throughout the application, even when visiting profiles of other users, regardless of their settings.

There should be two systems of measurement, the metric and the imperial. A user should have one of them preselected after registration, based on the country they live in. All the units should be displayed in the selected system, even the units shown in other profiles.

The initial url of a user's profile should be the user's id in the database. In the account settings, can be set a new url for the profile, one that a user can choose. Changing the url shouldn't make the profile inaccessible by the user's id.

4.2.16. Privacy

Privacy should be split into two sections, profile privacy and activity privacy. Setting a profile to private, makes the profile visible only to its owner. It should not appear in the search results and when it will be visited by its URL, it should display a message stating that the profile was not found. Accessing an activity of a private profile by using the API should fetch no data.

When a profile is public, activity privacy should apply. Each of the activities should be either public or private. Private activities should be visible only to its owner, and the other users shouldn't be able to see them on the profile page or access them using the API. The public activities should be visible by anyone who can view that user's profile, including the users who are not currently logged in.

4.2.17. API

The API should be RESTful and the application itself should use it. It should mainly interact with the user and activity entities. The GET method should be used for getting data, POST for creating new data, PUT for updating data and DELETE for deleting data.

4.2.18. Responsive design

The application should be displayed correctly on desktop computers and tablets, both in portrait and landscape mode. The controls should be easily accessible for both platforms and the interface should be change in order to adapt to each screen, and possible hide or collapse features in smaller screens.

4.3. Devices

4.3.1. Desktops

The user interface of the application should adapt to the screens of almost all devices. On desktop computers and laptops with medium and large screens, it should reach its maximum size (around 1000 pixels) and not expand more than that. All the elements of the application should occupy 100% of the width of that size, so that there is no empty space.

4.3.2. Tablets

On tablets and laptops with small screens, the width of the application should match the width of the screen. All the options and the features should be available but possible some of them can be grouped into a single option that has to be activated in order to reveal the full set of options, in order to save some space. It should work on both landscape and portrait mode.

4.3.3 Mobile

On mobile phones, the width of the application should match the width of the screen again, and the features should be reduced to the minimum required just for user input. Parts of the application (for example the section of the statistics) should not be available at all. Options should be grouped in order to save space and the application should work in both landscape and portrait mode.

4.3.4. Wearables (Usage of API)

Wearable devices should be able to access the application using the available API. All the features regarding manipulating the data of the activities, should be available using the API, so that users can track their performances using the sensors of their device.

5. Development Process

5.1. Sprint 1

5.1.1. Aim

In this sprint, there will be a requirement analysis, analysis of the technologies that should be used, prototypes will be designed and mockups. No code will be written but everything will be prepared in order to start the next sprint by developing the actual application. This sprint should have a duration of one week.

5.1.2. Requirement Analysis

The application should help track and field athletes, keep track of their performances and visualise their data in a way that they can find out how much they have improved and extract additional data in an easy and fast way. Therefore there are some feature that the application should definitely have and some others that can improve the users' experience and the interaction of each user with the other users. We will use the agile development method, so while the application is being developed, we will be getting feedback from the users and change the requirements accordingly.

The most important feature is the form where users can create, edit and delete activities. The activities are the main data that the application will handle, so this form should be easy to complete so that users can generate data in the application often and without hesitation. This raw data will be the source of all the visualisations and diagrams in the application. So another important feature that should be implemented are the diagrams. The should use the data inserted by the user and display line charts of time spans selected by the user. They should also filter data by discipline. Apart from the diagrams, the profile page should show a list of all the activities that should also be filtered by discipline or year of the activity.

There will be social features so the profile should resemble that of a social network. There should be a profile picture, search input that searches for users, the profile and the individual activities should be set to private, hiding them from the other users and also the diagrams of the users who have their profiles set to public, should be accessible.

An API should be implemented. The client of the application should use it but also it should allow other, future platforms for the client, to access it. All the operations for creation, editing and deletion of activities and user management should happen using the restful API.

The agile development method is flexible towards requirements which are likely to change, and the analysis for the first sprints of the project should be done only for the features that will be implemented in the few next sprints. In this sprint we will discuss the requirements that will likely be

the foundations for the upcoming features, so they will include ideas that may affect the way features are implemented in the future.

The most important feature that should be implemented in the next sprints is the page where the athlete can record and view their activities. This is the core feature of the application. Without activities, there is no content and the application offers nothing to the user. Therefore, the feature that should be implemented in the next sprint, as a prototype, is the way that athletes will record their activities.

Another important feature is the user as an entity in the application. There should be a system for creating users and a system that discerns individual users that use the application, namely, a registration and a login system with user sessions.

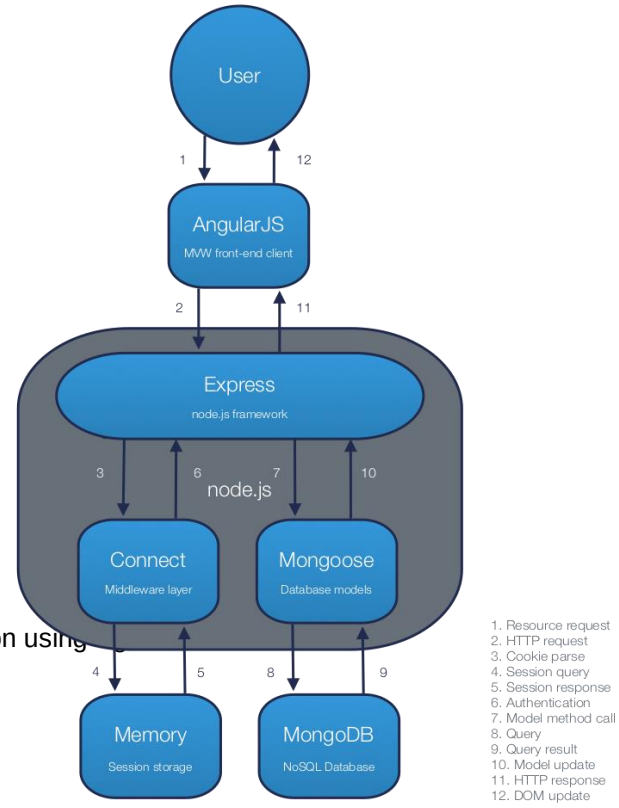
Both features require records in the database, so the design process should include the db schema and its interaction with the application.

When the prototype is ready, the core features should be enhanced and more features should be added in the next sprints.

Apart from the features, the architecture, the technologies and the frameworks that will be used will be decided in this sprint. For the front-end, we chose AngularJS, because it's a framework that represents data in the ui in a very convenient way. It's fast and it updates its values automatically, something that is needed for an application like ours. The models that can be used in AngularJS can match the models that I will use on the server, making it even more appropriate for our use-case.

For the server, we will use node.js, a server that runs JavaScript. The framework we chose to use is called Express. It makes managing the requests, sending responses, managing sessions and combining all the other modules, an easy job. In combination with express, I will use Connect, a middleware that handles user sessions.

For the models and the queries to the database, I will use Mongoose, as it can be used in order to define objects that will keep our database schema consistent, it's queries are similar to the native MongoDB queries and it's extensible. Consequently, we will use MongoDB as our database, which is a NoSQL database that stores BSON objects. Since the whole application will be written in JavaScript, storing our data as BSON objects will make things really easy, as we won't have to



convert types while retrieving or inserting data. We will store the sessions in the memory.

Fig 5.1.1 System Architecture

5.2. Sprint 2

5.2.1. Sprint planning

The features that should be implemented in this sprint are the login page, the registration page, the profile page that allows only creating new activities and simple graphs. I will set the fields for the users and the activities in the database and I will create the server logic for the login and registration functionality, that will also keep the sessions of the users on the server. Then I will create the routes and the functionality for activity creation and retrieval. I will also add a route for the graph, that will just serve the markup.

In order to have the complete functionality, on the front-end, a colleague will implement the functionality and the user interface for all these screens, including the statistics page, that doesn't have any functionality on the server, it only displays some basic graphs, using the data of the activities from the API.

This sprint can be split in 4 stories: the login functionality, the registration functionality, the activities API and sending markup for the 4 pages (login, register, profile, statistics).

The server, should expect from the registration page five parameters: the user's first name, last name, email address, password and the password one more time for verification. The first name and last name values should have only characters and no symbols. The email address should have a valid format and there should be no other such email in the database. The passwords should be at least 6 characters long and they should match.

Nothing more should be implemented regarding the registration process at this time. After registering successfully, the user should be redirected to the profile page. The user's submitted data should be stored in the database as they were given, apart from the passwords which should be encrypted with the sha-1 hash function.

For the login process, the email address and the user's password should be asked. These two pieces of information should match a record in the database in order to let the user login, otherwise, a message should be shown, stating that the credentials were wrong. After submitting the credentials successfully, a session should be created for the users, which should be stored in the memory for the time being, and a cookie should be sent to the browser, allowing the user to browse the private pages without any other authentication.

Sending the markup for the four pages is fairly simple, there will be four routes that will compile jade files and send html to the client. There should be no data from the server on the files, as all the variable will be populated by AngularJS on the front-end.

The API that manages the activities should only support two methods, GET and POST. The resources should be on the route /user/{user_id}/activities. A GET request on this route will fetch all the activities of the user with the given user_id, and if an id is given for an activity, only that activity should be fetched. An example route is GET /user/5/activities/10, which should fetch the activity with id 10 of the user with id 5. When a POST request is made on the resource route, a new activity should be created, according to the posted data. Whenever the API is accessed, the user should be authenticated by their session data. Also, when a user posts an activity, the posted data should be validated before they get stored in the database.

5.2.2. Stand up

Starting from the registration section, the user will have to fill in 5 fields, first name, last name, email address, password and repeat password. Then these fields are submitted, a POST request is

made, on the route /register. The server then checks the validity of the values. If they are not valid, the same markup should be sent back to the browser with messages that will help the user enter the correct values next time. If they are valid, the first_name, last_name, email and the password, hashed by the sha256 algorithm, are stored in the database. There are two tables where the users' data is stored, the "users" table which stores data that is related to the user's account, and the profile table, which stores data related to the users' profile pages. The first name and last name values will be stored to the profile page, while the email address and the password will be stored in the users table. Since there is no email validation, a session is created in the memory of the server, and a cookie with the session id is sent back to the client.

The user should be authorised to access the profile and statistics page, plus the routes that fetch and alter the activities. The sessions are stored in memory, using the middleware named "connect". Whenever a request is made to the server that includes the cookie data, the server will be able to check if the session data in the cookie is the same as the data in the memory and verify the user as logged in. Otherwise, it should redirect them to the login page.

The login page should have two fields, email and password. When the user submits the form, the server should get the posted data, hash the password with the sha1 algorithm and perform a query, searching for a record in the "users" table, that corresponds to this data. If a result is not found, the server should return a message, stating that the credentials entered are wrong. If a match is found, a session should be created, cookie data should be sent to the browser and the user should be redirected to the profile page.

When the user requests the profile page, the data from the table of the database "profile" (first_name, last_name for the time being) should be sent to the browser, personalizing the page. Then, when the front end framework loads, it will request the activities of the user, from the API route /users/{user_id}/activities. The table "activities" should be queried and the activities with the user's id should be fetched and returned to the client as a JSON object.

A big problem that will also be faced is the validation of the activities when they get posted to the server. In the application, activities will be categorised by the performance unit, so there will be three categories. The first category includes the disciplines which are races and therefore they are measured with time. The second category includes the disciplines, the performances of which are measured by distance, like high jump or javelin. The last category includes the disciplines that are measured by points, like the decathlon.

When a user posts an activity, the first step of the validation process would be to check if the posted discipline is a valid discipline. Then, based on the type of the discipline, the performance should be checked for validity. Performance counted in points should be an integer number, distance should be a float number and time should be a string. The last accepted and required value when posting an activity is the date, which should also be validated for its format and it should not be a future date. Then it should be formatted to the format that the database accepts.

After the validations have passed and the activity is considered valid, a new record should be created in the "activities" table, with all the posted data plus the user's id. A JSON object with the formatted activity data should be returned to the clients, in order to update the view with the new activity. In case the data is not well formatted or not valid, a message should be sent to the client stating where the error is located. In this sprint, the activities can not be deleted or edited.

The last feature that will be implemented is the page of the graph. The server side of this page just queries the database for the activities of the user and sends them to the client in order to be rendered as graphs.

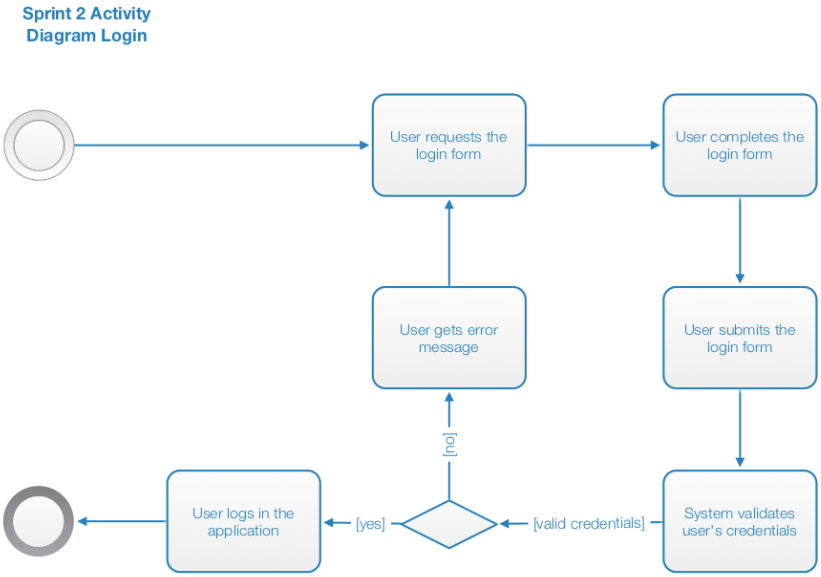


Fig 5.2.1 Login - Activity diagram

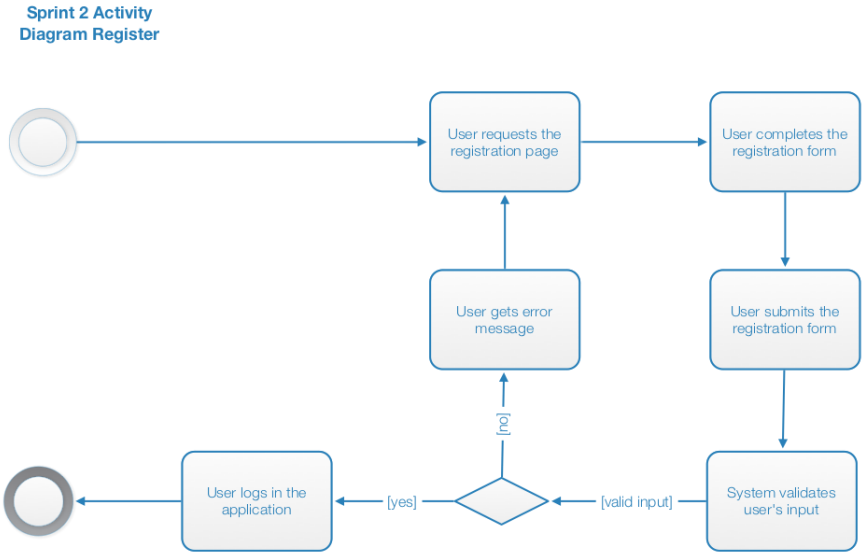


Fig 5.2.2 Register - Activity diagram

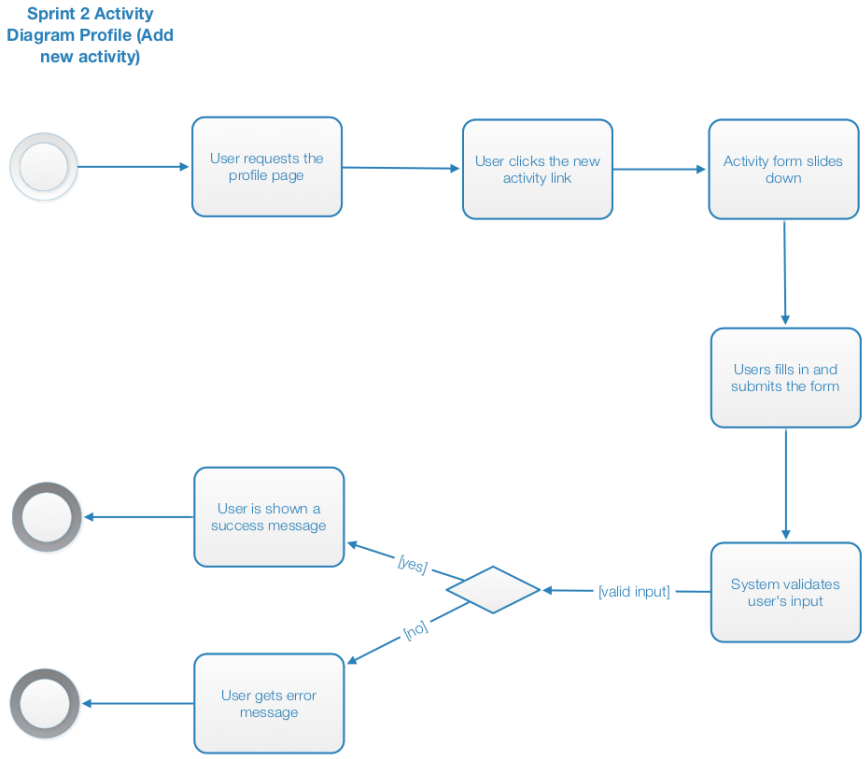


Fig 5.2.3. Profile - Activity diagram

5.2.5. Sprint review

This sprint took a bit more time than expected, as I wasn't very familiar with node, but in the end, all the required features were implemented. As a developer, I became more familiar with MongoDB database and with the node.js server, particularly with the callbacks and the non-blocking style of the server. This is a good starting point for the project, as the database has been set up and the most basic features have already been implemented, so functionality can be added just by working on the existing features.

5.3. Sprint 3

5.3.1. Sprint planning

In this sprint, an essential feature that should be implemented is the deletion and editing of the existing activities. Also, a new section of the application should be created, the settings page, where the user can personalise more his profile page, change his user and contact data and also, in a later sprint, manage the privacy of their profile.

Editing an activity should happen on the activity itself. The element that represents the activity in the client should have an icon, indicating that the activity is editable, which will change the data of the activity to input fields when clicked. When the form is submitted, the changes should be sent to the server and the client should update the activity element with the new data.

There should also be an element that when clicked, will delete the activity, by sending a delete request to the server and then updating the client. Also there should be a pop up, asking the user to verify their action of deletion, protecting the activities from being deleted by mistake.

The settings page should allow the user to change their first name, last name, biography, main discipline, country, profile picture and password. More options will be added in another sprint. Changes made to the first name, last name, main discipline and profile picture, will affect the data visible in the profile immediately. All the data should be edited inline, on the form itself, made editable by clicking an icon, similar to the one used for editing the activities.

5.3.2. Stand up

When the user has submitted the changes to an activity, the data should be parsed and validated by the server, the same way they are validated when a new activity is being created. The request should be a PUT request this time, and not a POST request which is the case in the activity creation. Also, the activity_id value of the existing activity that should be edited should be included in the data sent. After the values have been validated, a query should find the existing record and change all the values according to the new data. If everything goes as expected, a JSON object should be sent back to the client, which will update the view. If something goes wrong, a message should be sent, stating where the error happened, and if the data was not valid, the user should be prompted to correct the data in the form.

Clicking the icon that deletes the activity, and confirming the action in the pop up, should send a DELETE request to the server, along with the id of the activity that should be deleted. The server should check if the activity exists, and if it does, it should delete the activity from the database. If everything goes well, a JSON object should be sent back to the client, so that the view can be updated, removing the element that represented the activity. If there was an error in the process, an error message should be sent.

Both actions of editing and deleting activities, should be validated by the server, regarding user permissions. Users can only edit and delete their own activities, so the id of the user to whom the activity belongs should match the id of the user in the user's session and cookie. If the user tries to edit or delete an activity that doesn't belong to them, the server should not proceed with the requested action and no message should be sent back to the browser, as these actions are not permitted using the interface, so the user has used other means to do achieve it.

The settings page is actually a big form with data that should always be validated by the server. New fields should be added to the database. In the table "profile" the fields "biography", "country", "main_discipline", "profile_pic" should be added, and the model for the profile in mongoose should be changed accordingly. Each field should have its own validation rules. The rules for validating the first name and last name are the same as the rules for their validation in the registration page. The first name and last name should only allow latin characters, apostrophes and spaces. The biography should accept any text but the number of total characters should be limited. The limitation should be stored in a config value in the application so that it can be easily found and changed, depending on the specifications.

The main discipline should have the same accepted values as the disciplines in the profile page. There should be a list of valid countries in the server, that will be used in order to validate the user's country choice. The password should be validated based on the validation rules that were used during the registration process and there should be a second field, ensuring that the user has entered the password that they intended to use.

Uploading a profile picture is a different process than posting the rest of the data in the form. There should be several steps that should be followed during the upload process. First the MIME type and the file extension should be checked, in order to verify the file type. Only jpeg and png files should be accepted. Then the file size should be checked. If the image is too big, it should be rejected. The user is allowed to have only one profile picture, so if there is another picture

posted by them, it should be deleted. Then the new file should be renamed to be the user's id, for fast file retrieval. Then the image should be copied to the file system of the server, the path to the file should be stored in the database, in the profile table and a json object should be sent back to the client, stating the the process was completed successfully. If something went wrong, a detailed error message should be sent to the client. When a new profile picture is uploaded, the image preview and the image in the profile should be refreshed, and the new image should be shown.

There is no need to validate the user in the settings page, as the users can not view the settings page of the others users. The user_id of the values is automatically set to the id of the user in the session. After all the validations, messages should be sent stating if the process was completed successfully or not.

This sprint doesn't have very difficult parts, apart from the image upload which had to be planned carefully. The validations of the fields are not hard to be implemented but they require a lot of time and testing in order to get them right. Also some of the limitations in user input should be put in configuration files in the server, for consistency across the application.

5.3.3. Diagrams

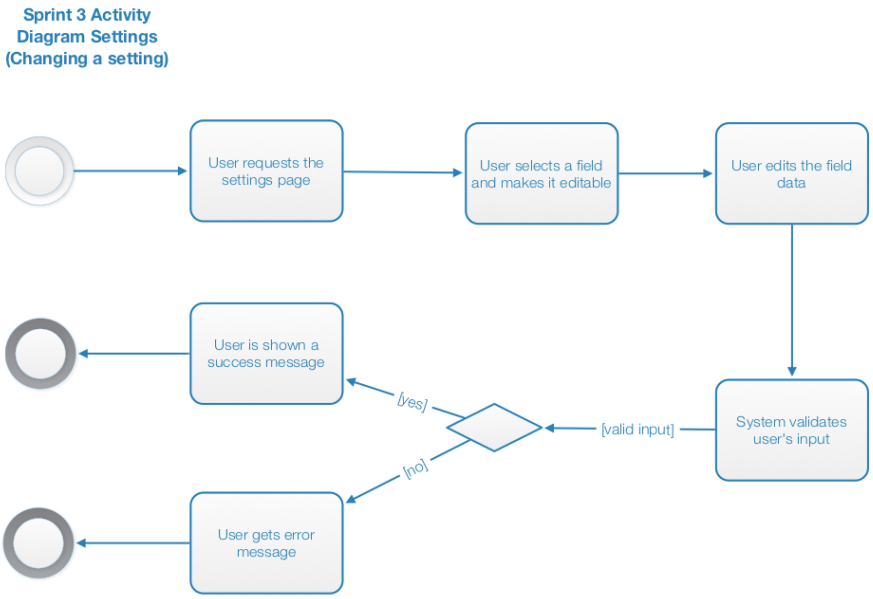


Fig 5.3.1. Settings - Activity diagram

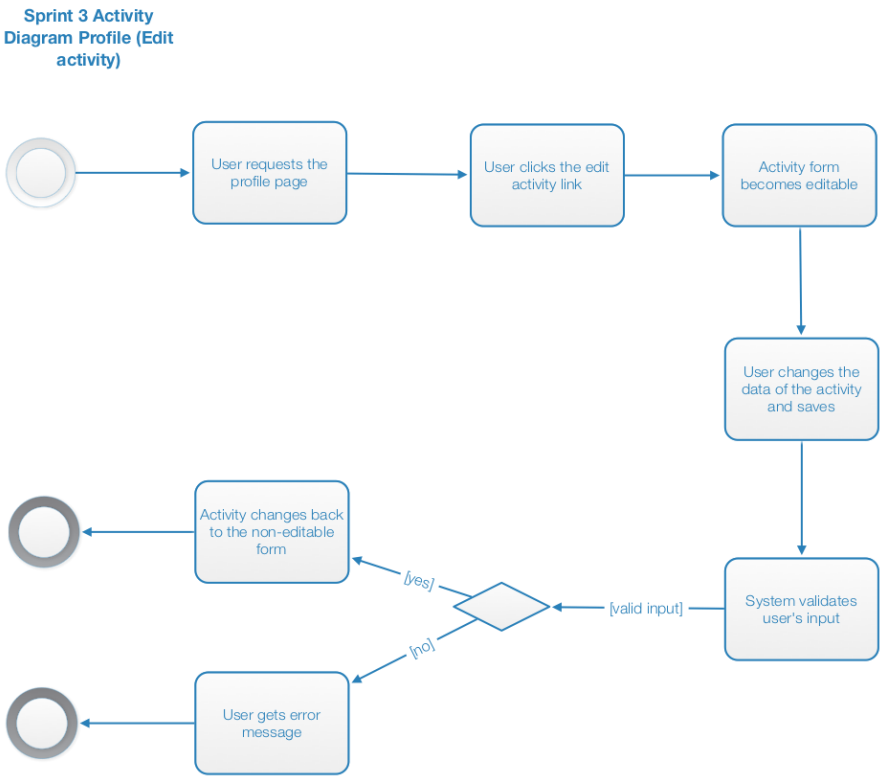


Fig 5.3.2. Edit activity - Activity diagram

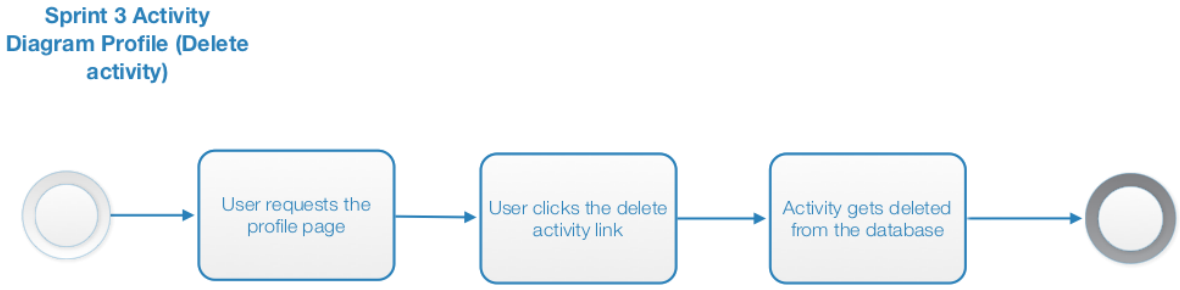


Fig 5.3.3. Delete activity - Activity diagram

5.3.4. Sprint review

This sprint was relatively easy to do, but there was a lot of content that should be written, so it was still time consuming. The only challenge was the file uploading process, for which, the fs library of node was needed. The page of the profile is almost complete. Now users can create and manage activities, so the application is functional and can be released. From this sprint and on, the application can be released in the end of the sprint, as it can help the users in some way, every time with more tools and functionality, which is the idea behind the agile methodology.

5.4. Sprint 4

5.4.1. Sprint planning

In this sprint, more data about the activities will be introduced and there will be functionality for its manipulation and display. Specifically, that data will be, the user's location, the date of the activity, the name of the competition, the place the athlete got in the competition and some notes. These data should be posted when the activity is being created and there should be a way to alter them later. Also, all this information should be available when displaying the list of the user's activities and the date of the activity should be used for sorting and filtering that list.

The main challenge that will be faced in the sprint is the validation of the posted data. Also, the date should be a required field, along with the performance and the discipline.

5.4.2. Stand up

New activity entries should be added in the database with POST requests. The most important piece of information in this sprint, is the date of the activity, and many upcoming features will be based on it. When a date is posted, it should be checked in order to make sure that it's a valid date, and then it should be stored in the database as a date object. When it will be returned to the client, it should be sent as a timestamp as well, since it is UTC time and therefore not related to any timezones.

In order to make sure that the date has a format that is not related to time zones, so that other users can read it correctly in other countries, the date should be converted to a unix timestamp. When the timestamp is sent to the server it should be checked for validity. It should not be a future date and it should be a valid timestamp. Then it should be converted to a javascript date object and it should be stored in the database.

Almost all the other data are just string values that do not need special manipulation. They can also be omitted without causing any problems. The user's location, the name of the competition, and the notes can be stored as they are, by just escaping all the characters that can lead to cross-site scripting attacks. The place that the user got in the competition should be parsed as an integer number, even when values like "1st" or "2nd" are posted. If the values can not be parsed as integers, the process of storing the data should stop and an error message should be sent back to the client indicating that the submitted data was invalid. After creating the new activity, its data, as it's formatted in the server, should be returned to the client which should create a new HTML element in order to display the newly created activity in the markup.

Updating the data should work in a similar way. It should be done with PUT requests and the data that will be sent with it should have the same format as the data in the POST request. The same fields are required, so existing required values can not be deleted, only replaced. After updating the record, the data should be sent back to the client as it got formatted in the server, and the existing HTML element that represents the activity, should be updated with the new values.

Since there is a date in every post now, the data can be displayed sorted in the front-end. There are plans to implement synchronizations of multiple devices later, which means that they will try to access only the data that have been altered lately, so the existing and the newly fetched data should be sorted in the client, for consistency. When the date of an activity has been updated, the server will send the formatted activity as it has been stored in the database back, and the client should move the element that represents the activity in the markup, in the appropriate place in the list of the activities.

5.4.3. Sprint review

This sprint didn't have any challenging tasks, but it was time consuming because of all the data validations and the date parsing. The new data that have been introduced in this sprint, apart from the date, won't play a big role to the next sprints, but the date of the activities is crucial for the functionality and a valuable piece of information for the users.

5.5. Sprint 5

5.5.1. Sprint planning

In this sprint, additional fields will be placed in the settings page, personalizing the application. The new data of the users that will be collected in the settings page are the language in which the user wants to view the application, the username that the user wants to use that will make their profile accessible more easily, the user's birthday, the date format that will be used for the dates displayed throughout the application, and the user's gender.

Changing the language of the application should change all the displayed text in the client - in buttons, labels, inputs, field descriptions, menus and other components of the interface. The application should be in english and there can be a greek version initially, just in order to check if the setting works after the sprint. The username, is not a required field. If it's not set, the default identifier of the user is the id field of the user's record in the database. If a username is set, the user's profile should be accessible by entering the username in the url in the browser, right after the domain name. The user's profile should still be accessible by the user's id of the database record. Changing the date format should change the way the dates of the activities and the dates in the user's profile are displayed. There should be two options, one that shows day/month/year and one that shows month/day/year. Changing this setting, should change the format of all the dates immediately. The birthday and the gender of the user won't play a big role yet, but they will be used later for the general statistics of the athletes that use the application and they can also be used as filters in the search field.

Another task that should be done in this sprint is the verification of users' email during the registration process. When a user submits the data of the registration form, after validating the format of the email address, an email message should be sent to this address, including a link to a unique page that the user can use in order to verify that the account that they created is linked to that email. After validating the account, the user should get automatically logged in. User's that haven't verified their emails, shouldn't be able to login using their credentials.

5.5.2. Stand up

Starting from the settings page, all the types of request in the setting's route should be changed in order to accept / fetch the new values. The client has been designed to let the users change the values of the settings one by one, so there should be cases in the back end, which should match the changed setting and apply that change for only one field in the database.

In order to make the setting of the language of the application functional, translations should be added. The values of the translation and the mapping of the values to the variables should happen using javascript objects in the server. There should be one object for each language, and each object should have the variable names in its keys and the translated values in the object values. And the objects of all languages should be placed in one object that maps the languages to the translations. So, for example, accessing the greek translation of the text "cancel" should be accessible in this way: tr.gr.cancel.

Changing the setting value, should post the two-letter combination of the language to the server. After validating that the combination exists in the object of the translations, the new

language setting should be stored in the database, again, as a two-letter combination. When markup is requested later, the two-letter combination can be used in order to access the correct object of the chosen language and populate the strings in the markup with the values of the requested text. Also, this object of the selected language should be accessible in a route, in order to give access to it to the client.

There were considerations to have the translations in the database instead of an object. If the translations are in the database, they do not occupy space in the memory but they would have to be fetched from the database with each requests, which would affect performance. Also, having the translations hard-coded as objects in the code, makes them easier to manage. New languages can be added more easily and new translations can be entered by just altering the file of the object. The file of the translations should be included in every other file that returns markup. After changing the setting of the language, a success notice should be sent to the client so that it can refresh the browser in order to display the settings page in the new language.

User's profiles are accessed by visiting the domain of the application followed by the id that the record of the user has in the database. By choosing a username in the settings page, the profile of the user can be accessed by entering the username instead of the user's id. So a new route should be created in the server that accepts one parameter after the domain which can be anything, apart from the other routes that the application has.

After a new username has been posted, there should be a check that it is not used by any other users and that there is no route with the same text. Also the usernames should only have latin characters, numbers, dots and underscores. If it's valid, it should be stored in the database. Every time a profile is accessed, there will be an attempt to find the profile by the id. If the id is not found, a query searching for the username should be made. If no profile is found again, a message should be displayed that the profile doesn't exist, otherwise, the profile data should be returned.

When the user posts a new value for the date format, it should be checked if it has one of the two values 'dd-mm-yyyy' or 'mm-dd-yyyy'. If it's valid, it should be stored in the database. Each time markup is requested by the client, all the dates should be formatted and printed in the document according to the selected format. A helper function should accept the date object, parse it, get the day, month and year values and put the in the correct order, as it was chosen by the user, and the return it. Changing the value of the setting should automatically change all the dates in all the pages.

The posted values of the user's birthday will be received by the server as three separate values, day, month, year, and it should be stored this way in the database. All three values should be checked for validity. The user's gender should be true or false, true for male and false for female. If the value of the gender is valid, it should be stored in the db. Both the values of the birthday setting and the gender setting are not used anywhere else in the application, so for the time being the GET request should populate the fields in the settings page.

The email validation, done later in this sprint, requires changes to the already implemented registration process. There should also be a new collection in the database, that will store unique hashes for all the users. These hashes should be generated during user creation, and they should be used in order to generate unique URLs that the users will follow in order to validate their account. When a user submits the registration data, after validating that all the values are correct, the "valid" field of the newly created profile record should be set to "false". A unique has should be created and stored in the database. The hash should be an sha256, and the hashed data should be a combination of the user's id with the current timestamp.

An HTML template will be used for the templates of the emails that will be sent to users. The email should contain the user's first name and last name, to make it appear a legitimate email of our application, and a link to a unique page of the application, that when visited, will set the user's profile record "valid" flag to true. For sending emails, we used the nodemailer library. After sending the email, the user should be notified that an email has been sent to the email address that was used during registration, and there should be a button that re-sends the email, in case the user

didn’t receive it. In order to make this work, the functions of the registration process should be split, and the function that sends the email invitations should be autonomous, in order to work during the registration and as a stand-alone function. That function should check if the hash has been created. It it exists, it should use the existing hash, otherwise, it should generate a new one.

After visiting the link in the email, the user should be taken to a link with no interface. That URL should make the profile valid and immediately login the user and redirect to the profile page. This route should first check if the hash in the URL exists in the database. It should match the hash with the user’s id (which should be another field in the same collection) and then find the user’s profile and update the “valid” field to true. This should enable the user to access the application. The session of the user should be created automatically, following the same process just as if the user has used the login screen to enter the application, and they should be redirected to the profile screen. The record in the database that includes the hash should be deleted, as it will no longer be used.

While the field “active” of the profile record is false and the user tries to visit the profile page, they should be redirected to the login screen with a message that their profile is not valid yet, and a button that re-sends the email with the validation link. There are plans to add another field in the record of the hashes, a timestamp that shows when was the last time the email was sent. A cron job can be ran regularly, checking if there are invitations that were created long ago, and delete their records from the database. A use case for this scenario is when a user tries to create an account with an non-existing email address, so there is no possibility to enable this account. In order to avoid having such permanent and useless records in the database, there should be a regular clean-up of the table.

5.5.3. Diagrams

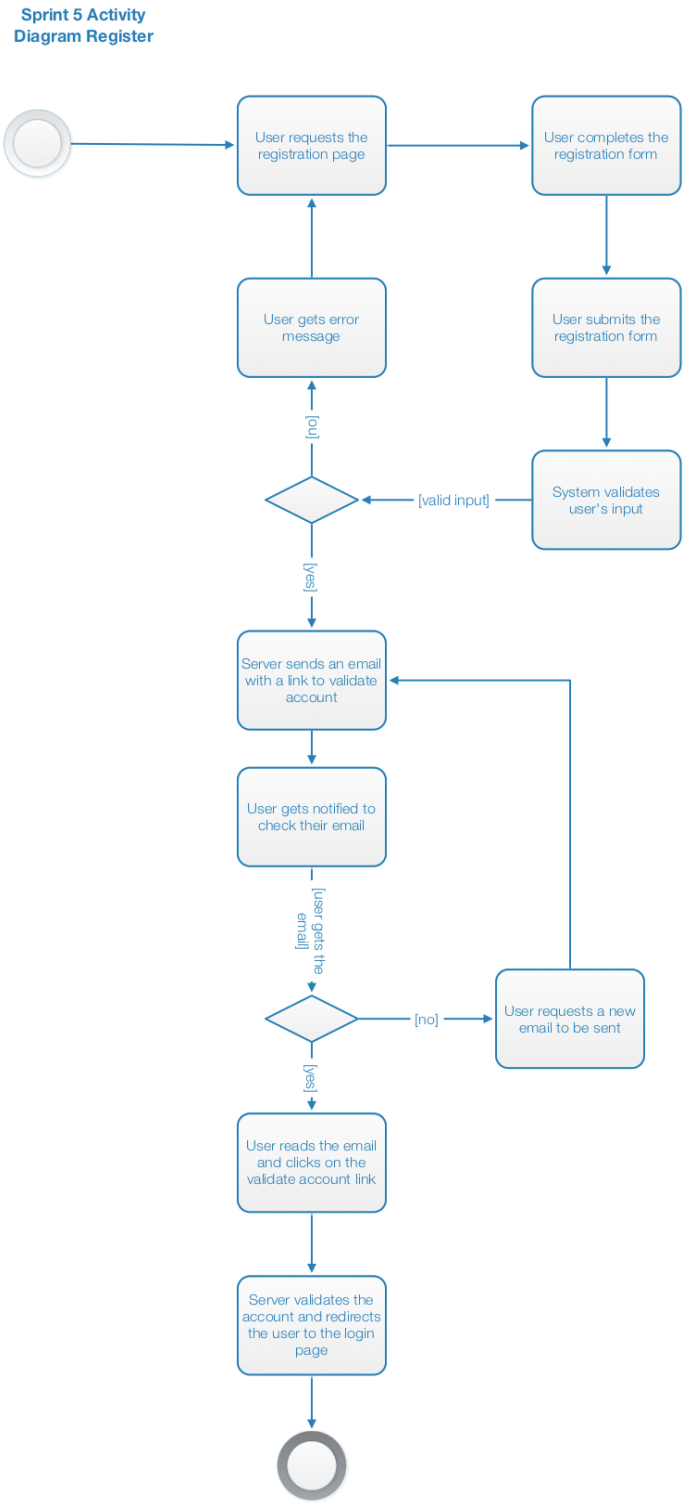


Fig 5.5.1 Register - Activity diagram

5.5.4. Sprint review

Since most of the core features of the applications have been implemented, we could work on features that will improve user experience, without adding anything that changes the workflow too much. Also the validation of the email addresses, prevents users from creating fake accounts.

This sprint was challenging, mainly because it needed to be well planned. Data flow diagrams were mandatory and there were many changes to the class diagrams as well. After all, the email validations took more time than expected, so this sprint took a little longer than planned. Also, managing the routes, because of the new feature of the usernames, added to the complexity of the sprint.

This is why agile suits our needs, this iteration only improved the application, without adding any core functionality. If we had to do this with the waterfall model, it would be a waste of time, as these features should be implemented when the respective sections were developed, but at that there, the application would be functional, and users wouldn't be able to test use it and give us feedback. With these iterations, we can work on new minor features while users use the actual application and provide meaningful feedback for the core features, allowing a better time allocation of work on the bugs, errors and generally for the changes that should be done to the already developed features.

5.4. Sprint 6

5.6.1. Sprint planning

In this sprint, the front-end of the application will be refactored so less decisions should be taken in the back-end, which will have just one change, standardization of the API. Specifically, in this sprint, there will be a shift from vanilla JavaScript to a JavaScript framework named AngularJS. All the functionality of the front-end will be completely detached from the server. Each page will contain no data when it will be fetched. The new API of the server should be able to provide any kind of data and in any format the front-end needs it, free of any markup. So, the browser will ask for HTML files with no data, and data in JSON format and it will populate the data in the markup and make the bindings by itself.

In order to make this work, the API should be designed carefully. Also the translations should be applied to the responses of the requests of the browser, since the markup will be independent now. There should be only one route that returns markup and validate that the user is logged in, the route that is the home of the application. All the other markup should be static files, that are sent to the client without any validations, since they will contain no user data inside. The rest of the routes should be RESTful API routes, with responses to GET, POST, PUT and DELETE requests that manage all the resources of the application.

Since the whole API should be designed, the implementation will be split in several sprints. In this sprint, the design of the API will be done and we will make sure to cover all the cases in order to start developing it and make it secure and consistent. Also the format of the requests and the responses should be finalized.

5.6.2. Stand up

The biggest change in this sprint is that the data should be independent of the markup. All the markup should be plain HTML files with data bindings for AngularJS inside and the data should be sent as JSON objects exclusively. The structure of the data and the types of requests should also not depend on any pages. The API should be autonomous so that it can be used by third parties as well and the format of the URLs should be according to REST. All the requests should send replies

that include meaningful response codes, and when there are errors, the error message should be included in the response JSON object.

For most of the resources, there should be get, post, put and delete requests. The get requests should be used to get all the resources described in the url, a get request with a resource id should return the chosen resource, a post request on a resource should create a new item, a put request with an id should update an item and a delete request with an id should delete the chosen item. Normally, making a delete request without an id should delete all the resources, but since it's dangerous sometimes, and there are not many use cases in our application, it will be avoided.

All the data depend on the users, they are bound to individual persons, so almost all the routes of the API should start with the user. Keeping it simple, the routes should start with /users/:user_id. A get request to the route /users should return a list of all the users, and it will be used with filter parameters for the search functionality of the application. So a get request to /users?first_name=George&last_name=Balasis should search for a user named George Balasis and return all the matching results. A get request with an id should fetch the user that has the given id. For example, a get request to the url /users/54fc5e13946e0a111335070f should return the user that has the given has as the id key in the database.

A post request to the route /users should create a new user, but all the required submitted data should be correct. The submitted data should be similar to the ones sent during registration, and the validations are the same for this route as well. A put request is available for the same route, but it should include the id of the user in the url. The data sent with the put request should have the same format as the data in the post request, and if all the parameters are correct, the record should be updated with the new value. After creating or altering a resource with a post or put request, the record as it was stored in the database should be sent back, confirming that the record was created or updated successfully and the new data can now populate the application with their new formatted values. A delete request to the route /users/:user_id should delete the user that has the given user_id. A delete request to the /users route should not be available, as it is too dangerous and it doesn't provide any useful functionality.

Not all the routes should be available to all the users. Users who are not logged in should be able only to make get requests on the /users route in order to search for users or get the data of one user who has their profile public, and they should be able to make post requests on the same route that will let them create new users. Users who are logged in should be able to use all the routes, but routes that include a user_id should be available for them only if the user_id belongs to them. This way, they won't be able to alter, create or delete data for other users. Also, the get requests to routes of users that have their profile set to private, should return a result similar to the one returned when the resource is not found.

The routes to the activities should follow the same rules, again starting from the users, as there are no activities that do not belong to any users. Generally the routes to the activities should follow the pattern /users/:user_id/activities/:activity_id. A get request to the route /users/:user_id/activities should return all the activities of the user with id :user_id. A get request to the route /user/:user_id/activitıes/:activity_id should return the activity that has this activity_id. A post request to the route /user/:user_id/activities should create a new activity according to the posted data. A put request to the route /user/:user_id/activities/:activity_id should alter the data of the activity with id activity_id, according to the posted data. A delete request to the route /user/:user_id/activities/:activity_id should delete the activity with id activity_id. There should be no batch create, batch update and batch delete of activities. The data accepted and the required data of the post and put requests should be the same.

Only logged in users should have access to these routes for the users that have set their profile to private. Only the owners of the activities should be able to make post, put and delete requests to those routes, otherwise the server should return the error code that corresponds to "forbidden" in http if the user has no access, or the code that states that this record doesn't exist at all, if the profile is private.

One route that will accept only get requests is the /users/:user_id/disciplines. This will be used to return all the disciplines for which the user has records of activities. The returned data of this route should not come straight from the database, but they should be generated by parsing the activities.

For the time being, there are no more resources, so the rest of the routes will not follow the RESTful approach. A get request to the route /settings_data should return the settings that the user who followed the route has stored in the database. So there will be no way and no available route to check the settings of other users. A post request to the same route should alter the settings, but it should accept only one setting at a time.

This separation of the routes that return only data, in json format, from the routes that return markup, will later help us create a friendly API for third-party developers that want to create applications that will use data from our database.

5.6.3. Sprint review

In this sprint, the basics of the API have been designed. In the next sprints, it will be implemented, along with other features. Working on the back-end from now on, would be easier because the data will be clearly separated from the markup, and some of the logic will go to the front-end of the application. This sprint may not seem as hard as the others, since there was no coding involved, but it is very important, as it will be the base for the architecture of the server code in the next sprints.

5.7. Sprint 7

5.7.1. Sprint planning

The new API will be implemented on the profile page first, the most important page that will also access the most API endpoints. The markup should be populated by data fetched from the server, after the page has loaded, and they should not be hardcoded in the page like it was before. This will provide a clear separation of the data in the pages from the page itself, so the data will be manipulated more easily. Also, this means that the same templates can be used for the new activities, the existing activities and the “edit activity” forms, since they will be completely detached from the data they show.

Showing how the data change will be easier as well, since the templates will serve as a visual representation of the array of activities stored in memory now, therefore, when an item of the array is deleted, the view will change automatically, without having code that searches for elements in HTML to do the same work. All these changes will be applied automatically by the newly implemented framework, AngularJS. What has to be done in this sprint is to implement the API in the server and the methods that access it in the client.

5.7.2. Stand up

The markup of the page should not contain any data related to the activities. It should only work as a template that can be populated by the data that will come from the server. So, when the profile page loads, the markup should be shown empty to the user, along with an indication that the application is loading data. The front-end code should make a GET request to the server, asking for all the activities of the user. It should also return a promise that will run again when the results have been returned from the server. The route of the request of the user with id “userid” should look like GET /users/userid/activities. This should return an array filled with JSON objects that represent the activities in the application. Afterwards, this data should populate the template and appear in the

markup as a list of all the activities, and the loader should be removed. An example response with two activities should look like this:


```
[
  {
    "_id": "5468cfa8cf56785d14b6d798",
    "discipline": "discus",
    "performance": "600900",
    "date": "2014-11-16T01:00:00.000Z",
    "place": 1,
    "location": "Saint Petersburg",
    "competition": "Olympic Games",
    "private": false,
    "notes": "I did my best!",
    "formatted_performance": "60.09μ",
    "formatted_discipline": "Δισκοβολία",
    "formatted_date": "16-11-2014"
  },
  {
    "_id": "54564eeb619f4da335000004",
    "discipline": "100m",
    "performance": "00:00:09.25",
    "date": "2014-11-02T18:34:03.436Z",
    "place": 1,
    "location": "Athens",
    "competition": "",
    "private": false,
    "notes": "",
    "formatted_performance": "09.25",
    "formatted_discipline": "100μ",
    "formatted_date": "2-11-2014"
  }
]
```


Every JSON object should be used in the ng-repeat directive of angular in order to populate the profile page, and turn this array of raw data in a nice representation of the user's activities, in a way that our users can easily extract useful information from it. The keys of the json object are:

_id: the id that the activity has in the database

discipline: the discipline of the activity

performance: the performance value as it is stored in the database

date: the date as it is stored in the database. It should be UTC.

place: the place that the user achieved in the competition

location: the location where the competition took place

competition: the name of the competition where the activity happened

private: if the activity is visible to other users. This functionality has not been implemented yet, so the API should always return "false"

notes: additional notes about the activity. It should be plain text, with escaped HTML, CSS, and JavaScript entities, in order to avoid XSS attacks.

formatted_performance: the performance value as it should be displayed in the browser

formatted_discipline: the discipline value, translated into the language the user chose

formatted_date: the date in the format that the user has chosen in the settings page

On the top of the list of the activities, there should be a button that makes a form appear. The form should be used in order to create new activities, and it should make a POST request to the server, with the new activity data. For a user with id userid, the post request should be done to the route POST /users/userid/activities. The data sent with the request should have the format:


```
{
	"selected_discipline":"200m",
	"private":false,
	"performance": {
		"seconds":"19",
		"centiseconds":"70"
	},
	"date":"Thu Apr 02 2015",
	"place":"2",
	"location":"Athens",
	"competition":"Olympic games",
	"discipline":"200m"
}
```


These values are actually the form converted to JSON format. The JSON object will be parsed by the server, it will be validated and the saved in the database. Then the newly saved values will be formatted in the way the server formats the activities before sending them to the client, and then the activity should be passed to the response of the request. A sample response for the previous request should be:


```
{
  "_id": "552145dad19dba060e4203d9",
  "discipline": "200m",
  "performance": "00:00:19.70",
  "date": "2015-04-02T00:00:00.000Z",
  "place": 2,
  "location": "Athens",
  "competition": "Olympic games",
  "notes": "",
  "private": false,
  "formatted_performance": "19.70",
  "formatted_discipline": "200м",
  "formatted_date": "2-4-2015"
}
```


Here the formatted data have been translated to Russian, the language the user chose. This json should generate a new element in HTML, and it should be appear to the appropriate position in the list of the activities, based on the date value.

Each element of the activities, should have an “edit” button or icon. When pressed, the activity should turn into an editable form again, allowing the user to edit the data. When this form is

submitted, a PUT request should be sent to the server. The PUT request for user id userid and the activity with id 5468cfa8cf56785d14b6d798, should be PUT /users/userid/activities/5468cfa8cf56785d14b6d798. The response should be the new data of the activity as they have been saved in the server. The data of the request should have the format:


```
{
        "performance": {
                "distance_1":"60",
                "distance_2":"9"
        },
        "date":"Tue Nov 11 2014",
        "location":"Saint Petersburg",
        "competition":"Olympic Games",
        "notes":"I did my best!",
        "private":false,
        "discipline":"discus"
}
```


A sample response could be:


```
{
  "_id": "5468cfa8cf56785d14b6d798",
  "discipline": "discus",
  "performance": "600900",
  "date": "2014-11-16T01:00:00.000Z",
  "place": 2,
  "location": "Saint Petersburg",
  "competition": "Olympic Games",
  "notes": "I did my best!",
  "private": false,
  "formatted_performance": "60.09м",
  "formatted_discipline": "Метание диска",
  "formatted_date": "16-11-2014"
}
```


After an activity has been updated, the element that represents the activity should be shown, hiding the editable form, and the new data should populate the element. If the “edit” button gets clicked again, the form should contain the new data.

Also, on every activity element, there should be a “delete” icon or button that should be used in order to access the DELETE method of the API. A DELETE request to the server that can be used in order to delete the activity with id 5468cfa8cf56785d14b6d798 for the user with id userid should be done on the url DELETE /users/userid/activities/5468cfa8cf56785d14b6d798. A sample response should look like this:


```
{
        success: true,
```

```
    errors: []
}
```

The success boolean value should show if the deletion was successful or not, and the errors array should contain any string values, giving details about the errors that happened, if any. If the success value is true, the client should remove the element that represents the deleted activity, from the list of the activities. Also a message should be shown, indicating that the activity was successfully deleted.

5.7.3. Sprint review

The implementation of the API didn't require many changes on the back-end, as most of the validation checks existed. Security was the main concern, as all the rules that applied before should be applied to each individual route of the API. Also the JSON objects sent to the client should be consistent. All requests should return the same format of activities. Of course, changes to the routes of the application were made. The most important part of this sprint was the decisions taken, regarding the format of the successful and failed requests, and also the format that the urls of the API have.

5.8. Sprint 8

5.8.1. Sprint planning

In this sprint, the API of the settings will be implemented. The settings, until now, used a POST request that forced a refresh. The POST request was sending the whole form, so the server was parsing everything and it was storing all the values when the whole form was valid. Now, since it will be changed to requests that will not refresh the page, only the changed data should be sent to the server. So, each field should have its own "submit" button that will send the data of the field to the server, and when a response is received, the setting should be updated.

5.8.2. Stand up

The first thing that has to be changed is the form. This is a change on the front-end, but it affects how the back-end will work. Every form row in the settings page should be a different form, sending only its own data. The row should be initially non-editable and there should be an "edit" button or icon. When the button is clicked, an editable field should appear, with the value of the setting entered. Also, the "edit" button should be replaced by "save" and "cancel" buttons.

When the "save" button is clicked, a POST request should be sent to the server. Here, we don't use PUT for update because settings can not be created, they exist since the user got created, so a POST request means "alter data" in this case.

The server should find out which data got changed, check if the value of the field is acceptable and then store it in the database. If the setting is the profile picture, it should check its size, its mime time, and its extension for validity. After validating and updating the value, a response should be sent back to the browser, stating that the settings have changed successfully. All the requests to the api of the settings should be sent to the route /settings. There is no reason to use the id of the user in this request, as there is no way to alter or access the settings of another user. A GET request to the same route should fetch all the settings of the user. Settings can not be accessed by id like the activities, because they do not have their own id, they belong to the record of the user. Also, there is no DELETE request for this route. The POST request should simply look like this:

{"first_name":"George"}

A POST request with that data should change the first name of the user to George. A sample response to such a request should look like this:

{
  "value": "George",
  "translated_value": "",
  "message": "Data was updated successfully"
}

The first parameter, with the key “value” is the new value of the setting. The “translated_value” shows the same value translated into the language that the user has shown. In this case, it’s a name so it won’t be translated. This parameter is meaningful only in the settings “country” or “main discipline” for which the translated values are needed in order to display the result to the user.

Also, a decision has been taken, to separate the settings in “Profile settings” and “Account settings”. The account settings are the user’s password, the language of the application, the date format and the username. The rest of the settings are account settings. This separation exists in the database as well, so the account settings change values that are stored in users table of the database, while the profile settings change the values of the table “profiles”.

5.8.3. Sprint review

This sprint was easy, since the API didn’t change the functionality much, but now the settings page is more easier to maintain, since the fields have been separated and if something goes wrong in the form in one field, that doesn’t affect the other fields of the form. The functions for the validations have been reused so only the code of the controller and route handling has been changed. The decisions for the front-end and the way the settings page would work was the most important part of this sprint, by keeping in mind the effort to make the whole application more maintainable and friendly to 3rd party developers.

5.9. Sprint 9

5.9.1. Sprint planning

In this sprint, the page of the graphs will be implemented in angular, plus, filtering of the activities will be introduced. The graphs use a third-party library, so this won’t change. That library should now get the data from the api though, and also the filters are necessary for the graphs, and it will be much better if they are implemented on the back-end. So changes will be made to the API of the activities, so that it receives parameters that filter the activities. Applying filters should immediately change the graph, and when they are applied to the profile page, they should remove the elements that represent activities that do not match the required filters.

5.9.2. Stand up

Starting from the page of the statistics, the API of the library of the graphs, will remain the same, but it should get the data from the new API of the activities. There is no need for changes to the data that come from the server. Since there is no need to refresh the page in order to get new data now,

the graphs should use a “redraw” method, that will accept the new data and it will draw the graph again.

Previously the server was formatting the data according to what the graphs needed. Now, the data have the same format as the data sent to the profile page, so there should be some formatting/manipulation on the front-end, in order to make the data that come from the API accepted by the library of the graphs.

Both the graphs and the activities in the profile page should be filtered by the server, according to the filters selected by the user on the front-end. For the time being, there should be filters for the start date, end date and the discipline. Choosing a start date, should show all the activities that happened after that date. Choosing an end date, should show all the activities that happened before that date. Choosing a discipline should show only the activities of that discipline.

These parameters should be passed as queries to the GET request. If the user wants to see the activities of the discipline 100 meters sprint, that happened since the beginning of 2015 until the 1st of May of 2015, the request should look like this:

GET /users/54fc5e13946e0a111335070e/activities?discipline=100m&from=2015-01-01&to=2015-05-01

This should return the activities as an array of JSON objects, as usual, but it should filter out the activities that do not match these filters. Any of the parameters can be omitted, leading to different results. Filtering works only for GET requests.

The profile page should have predefined filters on the top, that can be clicked and applied by the users. The main filters should be the years. So, if a user has activities in the year 2014 and 2015, these years should appear as links on the top of the profile page. When, for example, the year 2014 is selected, a request will be sent to the server, with the from parameter set to 2014-01-01 and the to parameter to 2014-12-31, essentially asking from the server the activities of the year 2014.

For the implementation of these filters, the query to the database should be split into several pieces of code. The parameters should be parsed, and based on whether there is value or not, a piece of the query should be added. If the parameter “from” exists, for example, a “where” clause should be added to the query, checking the values “date” to be $gt (greater than) the given date. In the same way, the “to” date and the discipline should be compared.

5.9.3. Diagrams

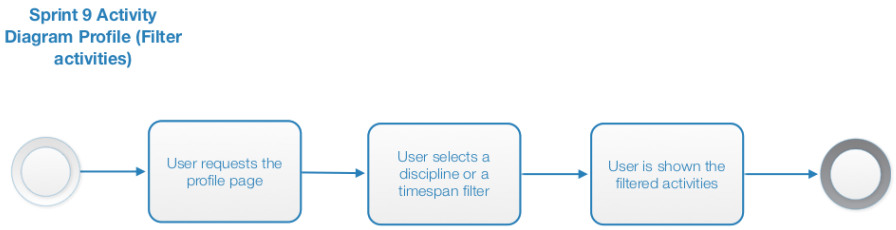


Fig 5.9.1 Activity filtering - Activity diagram

5.9.4. Sprint review

With these changes to the API starts to become more flexible, regarding various use cases in the client-side. Filters like that can be easily used in graphs, search, profiles and also, much later, they will be used in order to load and cache changes in devices that were not yet synchronized. This hasn't been implemented yet, but this feature is planned.

When the user is using the application on different devices, and there are offline data, all the devices should be synchronized. If the user was using the device A for a certain period and then the device B for another period, when he gets back to device A, the client should make a request for the activities, setting a "from" filter set to the last activity it had cached, so that it can synchronize all the changes made from the other device.

Generally, having a powerful API and giving many options to the clients, makes the application easier to develop and extend, and also it gives a great tool to third party developers.

5.10. Sprint 10

5.10.1. Sprint planning

In this sprint, a very important feature will be developed, that will improve the social features of the application: user permissions and visiting other profiles. The permissions will be split to permissions of the profile and permissions of activities. Profiles and activities should be toggled as private or public by the users. Using these features, users should be able to access the profiles of other users who have their profiles set to public.

In order to make these features complete, the search API will be developed, that will be used in order to search for other users, making it easier to find other users and access their data. The search API should also use filters, just like the API of the activities. Of course, by letting users to access data of other users, raises concerns about security and data privacy, so certain actions will be taken in order to prevent users from accessing data that should be private.

Finishing this sprint will make the application fully usable and it will be ready to be released as a fully-featured web application.

5.10.2. Stand up

In order to implement the functionality of the privacy, there should be a setting, so that users can set their privacy settings. The first section that should be protected, is the profile of the users and its setting should be found in the settings page, in the profile tab. Changing the setting, which should be simply a checkbox, should switch the "private" flag in the database, in the "profile" collection.

When a profile is private, it shouldn't be visible at all, to all other users. When a user tries to access a private profile, they should be shown the "Page not found" default page, with a 404 response code. In order to achieve this, whenever a user tries to access a profile, a check should be made. The first condition should check if the profile belongs to the user who tries to access it. If that's true, then the profile should be displayed no matter if it's private or not. If the profile doesn't belong to the user, the "private" flag should be checked. If the profile is private, the "page not found" static page should be explicitly sent to the client and the process should stop there.

The same check should be perform when the user is using the /users route, in order to search the application. The api of this route will be implemented later in this sprint. Also, all of the user's data, the profile page, the activities, should be inaccessible to anyone, by using the same check, before proceeding to any other queries.

Each activity should have it's own "private" flag that can be set to true when the activity is created or edited. When a user tries to get the list of one user's activities, first a check should be

made, to see if the user who makes the request is the user who created the activities. If that's true, then all the activities, private and public, should be returned. If the user is not the owner, only the public activities should be returned. That will be implemented by altering the query, by adding a where clause, restricting the results to not include entries with the flag "private" set to true. The same restrictions to the query should be used when filters are applied. Each element that represents an activity in the markup, should have a switch, that when clicked, sends a put request to the server, and updates the activity, setting the private flag to true or false.

After implementing profile privacy, the functionality of visiting profiles of others can be refactored. The first action that should be performed when a user requests a profile, is to check if the profile belongs to them or to another user. As mentioned before, if the profile belongs to them, it should be sent back to the user regardless of the privacy settings. If the profile belongs to another user, data should be returned only if the profile is public, otherwise, the default "page not found" should be sent back to the user. User profiles can be fetched by id or username, from the url. There is a restriction when creating usernames, that doesn't allow them to be longer than a maximum length, so the id hashes in the database are always longer than usernames. When the server gets a request for a profile, the length of the string should be checked, and it should be determined if it's a username or and id. Then the appropriate field in the database should be queried, searching for the profile. If the profile is found, all the data of the profile should be sent back to the user. If not, the "page not found" static markup should be returned. The returned data should be formatted according to the user's language settings.

The functionality of the search, should work like the rest of the API. It should be a search for profiles/users, and it should work with a GET request. So, the url that should be used will be /users and it should be followed by a query string. The query string, in case we are looking for users named "George" should look like this /users?keywords=George. The search functionality should work with keywords, because people may have more than two names, and later, we will need to have the names available in different language, show a search for George should be a result when searching the same name in Greek. In order to achieve it, there should be a new field in the database, named "keywords" that will be only used for profile searches. This field should be an array of all the names, and all the variants of them, that when matched, will return the profile of the user. A sample response for the above search could be:


```
[
  {
        "_id": "54fc5e13946e0a1113350710",
        "first_name": "George",
        "last_name": "Balasis",
        "discipline": "",
        "country": "",
        "username": "george",
        "formatted_country": "",
        "formatted_discipline": ""
  }
]
```


This is an array with only one result in it, the user George Balasis. This data should be used in order to populate the dropdown with the options in the front-end. When clicked, the _id value should be used in the url, so that the user can be redirected to the profile that they were searching for.

5.10.3. Diagrams

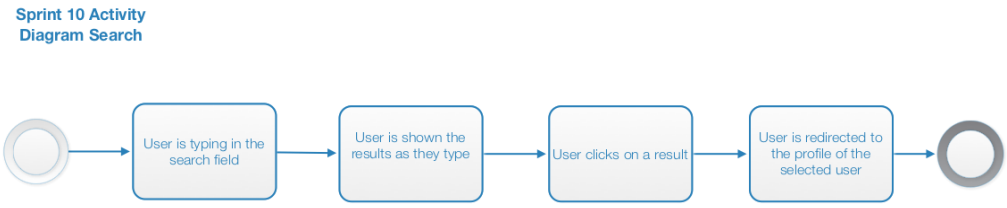


Fig 5.10.1 Search - Activity diagram

5.10.4. Sprint review

This sprint concluded the development of the application. Now the application is ready to be released. It has all the planned features and it's designed in such a way that it will make further expansion and maintenance easier. It also makes it very easy for other users to use the API and access the resources existing in the database.

6. Results

6.1. Agile contribution

The agile methodology, even though it works better in bigger teams, greatly improved our productivity, our motivation and made the project adapt to what the users wanted, really fast, thus resulting in a better product. If we hadn't use agile while developing, quite possibly the application would have been what we thought that the users needed and not what actually the users needed.

First of all, it improved communication. Using stand-ups and reviews made always clear what each one of us was working on, and what is worth working on during each iteration. By the end of each sprint, it was made clear what is necessary to work on during the next one. Communication between us, the developers, was not the only part that agile improved, but it also help us communicate better with people who are relevant to our target group - the athletes. By contacting them personally and by implementing a feedback form, we got all the information we needed to adapt the project to their needs, to the real users.

We almost achieved having a working product by the end of every sprint which helped a lot since the application was partly shaped by our users. They had something to try, since the application was working with every release, and not only they could use it in order to tell their opinion but they could also use it as a tool that helps them track their performances, even before it was ready. It also helped us to find what is working and what is not working from the point of view of user experience and usability.

During the development process, there were some phases that required changes to the specifications, like implementing date formats for different users or having a search input that we didn't include in the initial specifications. Working with the agile methodology made us accept these changes as improvements rather than time consuming tasks that would change the way that we were working, the existing code of the project or throw us off schedule. They were all implemented as if they were part of the initial specifications, and even better.

6.2. The application

The final outcome was close to our expectations. The good communication that was achieved by using the agile methodology, the regular reviews of our work and the short development cycles led to a complete application that is secure, with a client and a server that can be completely detached but still communicate really well, a fast responsive user interface and most importantly, the outcome is a tool that is useful for our users, who contributed while it was being shaped.

These are some screenshots from the latest version of the application.

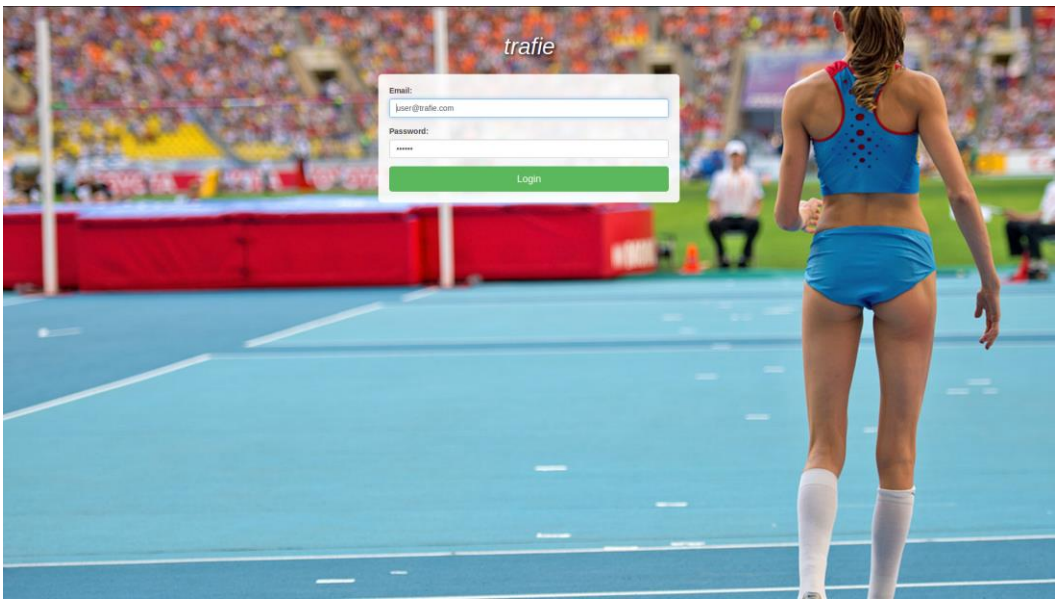


Fig 6.2.1. The login screen

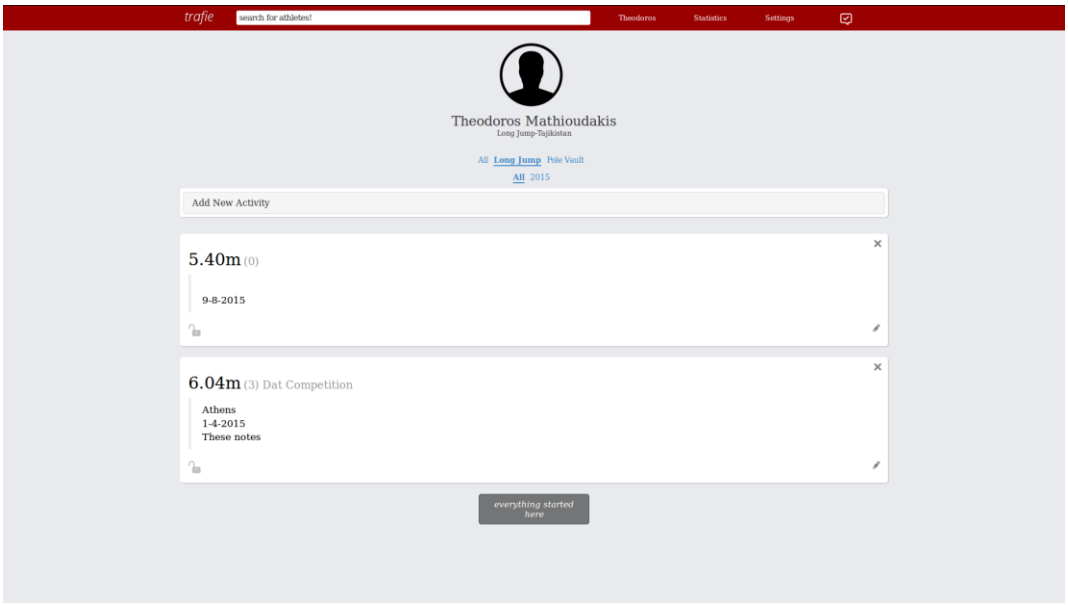


Fig 6.2.2. The profile page

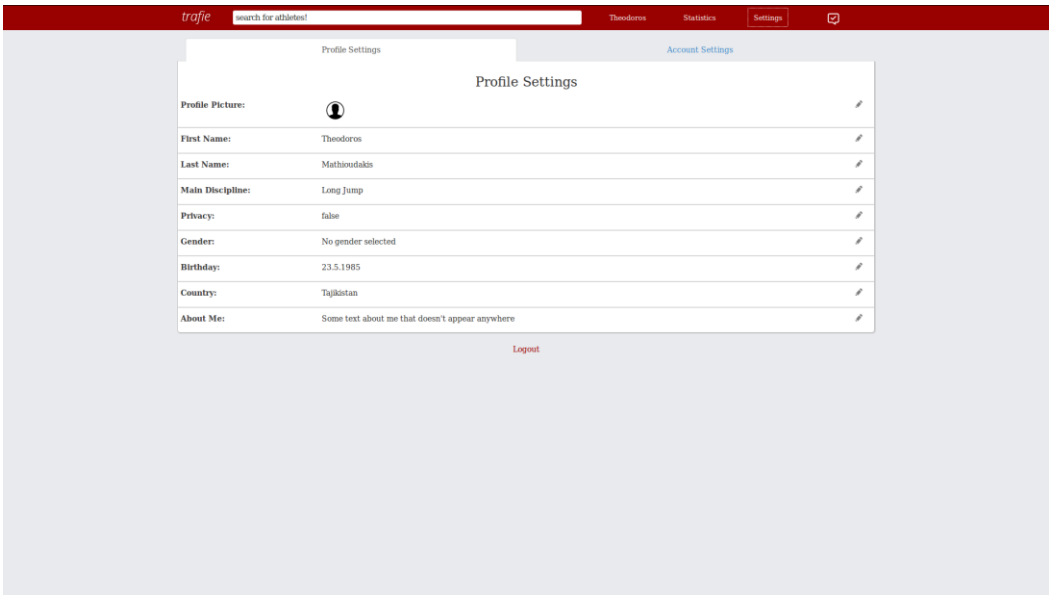


Fig 6.2.3. The settings

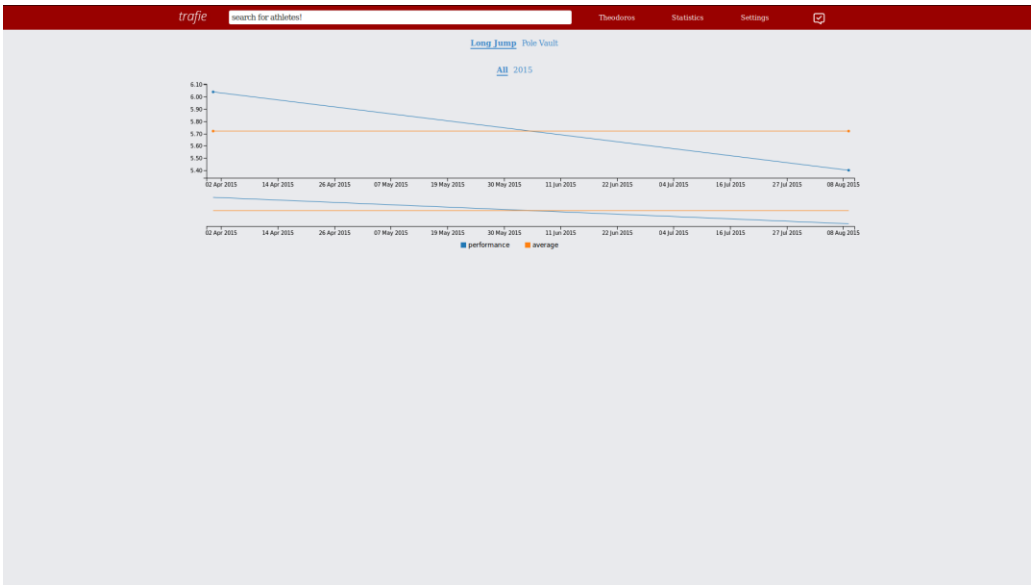


Fig 6.2.4. The statistics

The user interface, which was created by the second member of the team, is fully responsive, showing the same information on tablets and mobile phones. The elements have be placed with the ease of accessibility in mind.

6.3. The API

All the routes with their methods and a short description of what they do, are shown in the following table. The api does not include routes of settings or routes that fetch html documents, as these are not included in the standard format of responses from the server.

API			
Route	**Method**	**Description**	**Parameters**
/users	GET	Returns users, filtered by the parameters. If there is a "keywords" parameters, a search will be performed based on the words separated by spaces. Only public users will be returned.	firstName, last_name, discipline, country, keywords
/users/:userId	GET	Returns the user by id. Only public users will be returned unless a logged in user tries to access themselves.	-
/users/me	GET	The logged in user. If the user is not logged in, nothing is returned.	-
/users/:userId/activities	GET	Returns all the public activities of the user.	from, to, discipline
/users/:userId/activities/:activityId	GET	Returns the activity by id. Only public activities will be returned unless a logged in user tries to access their activity.	-
/users/:userId/activities/:activityId	POST	Creates a new activity.	-
/users/:userId/activities/:activityId	PUT	Edits the data of an existing activity.	-
/users/:userId/activities/:activityId	DELETE	Deletes an activity.	-
/users/:userId/disciplines	GET	Returns all the disciplines that the user has recorded.	-

6.4. The schema of the database

The schema of the database is shown in the following tables. Each table represents a different collection in the mongodb. Each row is the attribute or “column” in the database.

users				
Field	**Type**	**Required**	**Index**	**Description**
_id	id object	true	true	The id of the record, which is automatically set by the database.
email	string	true	true	The email that the account of the user is bound. It must be unique and there should be an index on it, to make searches faster.
password	string	true	false	The password that users use in order to log in. They are required.
valid	boolean	true	false	A flag that shows if the user has validated their account by email.

profiles				
Field	**Type**	**Required**	**Index**	**Description**
_id	id object	true	true	The id of the record, which is automatically set by the database.
first_name	string	true	false	The user’s first name.
last_name	string	true	false	The user’s last name.
username	string	false	false	A username,

				chosen by the user in case they want to have a personalized url.
male	boolean	true	false	The user's sex. It's true if he's male.
birthday	object	false	false	The user's birthday. It's an object with three attributes, day, month, year, all integers.
discipline	string	false	false	The user's main discipline.
about	string	false	false	A short description of the user.
country	string	false	false	The name of the country where the user lives.
picture	string	false	false	The filename of the image that the user has uploaded as their profile picture.
date_format	string	false	false	The date format that the user has chosen. It formats all the dates in the application. The default value is "d-m-y".
language	string	false	false	The language that the user has chosen in the settings. The default value is "en" for english.
private	boolean	false	false	A flag that shows whether the profile is private (hidden by other users) or not. The default value is false.

keywords	array	false	false	An array of keywords for the user, that makes search faster. The keywords are strings.

activities				
Field	**Type**	**Required**	**Index**	**Description**
_id	id object	true	true	The id of the record, which is automatically set by the database.
user_id	string	true	true	This field corresponds to the _id fields of the user table.
discipline	string	true	false	The discipline of the activity performed.
performance	string	true	false	The performance of the activity.
date	date object	true	false	The date that the activity took place. The default value is the current date.
place	string	false	false	The place that the user achieved, if the activity took place in a competition.
location	string	false	false	The location where the activity took place.
competition	string	false	false	The name of the competition that the activity took place.
notes	string	false	false	Notes that the user keeps about

				the activity.
private	boolean	true	false	If the activity is private or not. The default value is false.

user_hashes				
Field	**Type**	**Required**	**Index**	**Description**
_id	id object	true	true	The id of the record, which is automatically set by the database.
user_id	string	true	true	This field corresponds to the _id fields of the user table.
hash	string	true	false	The unique hash stored for the user
type	string	true	false	The type of the hash. There are two types, one for the forgot password functionality and one for the email validation.

7. Conclusion

With the application ready and working, it can be safely said that agile worked and was very suitable for our case. The project was ready on time, the build quality is very high, and all the features that got developed are needed by the users, since there was feedback from early on.

One of the main aspects of agile that helped to successfully finish the application, was the requirement that by the end of every sprint, there should be a product that works and serves at least the basic purpose of the existence of the application. By the end of sprint 2, which was 2 weeks after starting the project, we had a tool that was working and it could already help some track and field athletes in some way. As the time went by, there was a more clear view of the application and its possibilities. Giving these “complete” products to users, helped us get feedback faster and improve the application, without losing much time fixing problems later on.

Another aspect that helped, was the communication part. Before each sprint, there was a discussion that included prioritization of the development time of the features, changes of plans, setting goals, solving problems of the previous iterations and generally planning the current sprint.

This kind of discussions helped us focus on what is important to work on, every time. In every sprint, it was clear what should be done, and which is the most important task that should be finished by the end of the week, so the quality of the service of the application was really good, compared to the time it had been spent on its development, even during development.

Finally, changes had been made during the development process, to the frameworks used, to some of the decisions taken and for some features. Agile helped us adapt to those changes and coordinate our work and implement them smoothly.

All in all, agile proved to be a great choice for this case, making the development a success in every aspect. Changes were smooth, time got spent to features that matter, the project was done one time, the users are satisfied, and the quality of the application is good.

8. Bibliography

[1] Chris Sims & Hillary Louise Johnson. (2012) *Scrum: A breathtakingly brief and agile introduction*. Dymaxion.

[2] Kenneth S. Rubin. (2013) *Essential Scrum: A practical guide to the most popular agile process*

[3] Chris Sims, Hillary Louise Johnson. (2011) *The elements of Scrum*

[4] Tom Hughes-Coucher & Mike Wilston. (2012) *Node: Up and Running*

[5] Amos Q. Haviv. (2014) Mean Web Development

[6] Karl Seguin. *The Little MongoDB Book*

[7] M. Steven Palmquist, Mary Ann Lapham, Suzanne Miller, Timothy Chick, Ipek Ozkaya. *Parallel Worlds: Agile and Waterfall Differences and Similarities*

[8] Marian Stoica, Marinela Mircea, Bogdan Ghilic-Micu. (2013) *Software Development: Agile vs. Traditional*