

Play with Scala

Functional Programming Course

Martin Zuber, B00066378

Project Supervisor: Luke Raeside

April 24, 2016

# Declaration

I hereby certify that this material, which I now submit for assessment on the programme of study leading to the award of Degree of Honours B.Sc. in Computer Science in the Institute of Technology Blanchardstown, is entirely my own work except where otherwise stated, and has not been submitted for assessment for an academic purpose at this or any other academic institution other than in partial fulfilment of the requirements of that stated above.

# Acknowledgements

In performing my assignment, I had to take the help and guideline of some respected persons, who deserve my greatest gratitude. The completion of this assignment gives me much Pleasure. I would like to show my gratitude to the project supervisor Dr Luke Raeside, Institute of Technology, Blanchardstown for giving me good guidelines for the thesis throughout numerous consultations. I would also like to expand my deepest gratitude to all those who have directly and indirectly guided me in writing this thesis, especially Dr Markus Hofmann.

# Abstract

This paper is dedicated to computer programming with the focus on programming styles found in modern application development. The idea for the project came to life when I, the author and computer science student realized there are programming styles other than imperative object-oriented programming. Namely functional programming paradigm. The lack of a study material dedicated to the functional programming withing the ITB Computer Science course curriculum and my curiosity toward the subject I decided to research and to learn the paradigm.

This paper is discussing functional programming paradigm with Scala programming language and web application development process using Play! Framework. The target audience is any computer science student with intermediate knowledge of a programming language such as Java and with an understanding of some of the object-oriented programming principles such as inheritance and polymorphisms.

Please keep in mind that I had no knowledge of a functional programming, Scala programming language or Play Framework prior to this endeavour. This document is basically a report of a student learning basics of these concepts and tools. I can only hope that I would be able to communicate clearly what I learn and I sincerely hope this document would be useful to someone.

# List of Figures

1.1	<i>Prototyping SDLC diagram</i> . . . . .	9
1.2	<i>Gaant Chart</i> . . . . .	11
2.1	<i>Taxonomy of programming paradigms (Van Roy, page 15)</i> . . . . .	16
3.1	<i>Play Framework Stack</i> . . . . .	30
3.2	<i>Java EE 'lasagna' architecture versus Play architecture</i> . . . . .	31
3.3	<i>Prototype 1.0: Login Use Case</i> . . . . .	36
3.4	<i>Prototype 1.0: Register Sequence Diagram</i> . . . . .	36
3.5	<i>Prototype 1.0: Account Settings Sequence Diagram</i> . . . . .	37
3.6	<i>Prototype 1.0: Main Page Wireframe</i> . . . . .	37
3.7	<i>Prototype 1.0: Dashboard (lectures) Wireframe</i> . . . . .	38
3.8	<i>Prototype 1.0: Account Settings Wireframe</i> . . . . .	39
3.9	<i>Prototype 1.0: Entity Relations Diagram</i> . . . . .	40

3.10	<i>View -&gt; Request -&gt; Controller -&gt; Response -&gt; View Diagram</i>	41
3.11	<i>Play Framework Project Anatomy</i>	42
3.12	<i>GitHub Repository Anatomy</i>	43
4.1	<i>Play WorkFlow Diagram</i>	44
4.2	<i>Lightbend Activator</i>	46
4.3	<i>Main Layout View</i>	48
4.4	<i>Index View</i>	48
4.5	<i>Main Page</i>	49
4.6	<i>Validate New Account E-mail</i>	56
4.7	<i>Validate New Account E-mail</i>	56
4.8	<i>Unsuccessful Log-in Response</i>	57
4.9	<i>Routes</i>	59
4.10	<i>Virtual Machine Endpoints</i>	61
5.1	<i>Activator test run</i>	68
5.2	<i>Selenium IDE Firefox Pluggin</i>	72
5.3	<i>Register Account Twice Actions</i>	73

# Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
1.1	Introduction . . . . .	2
1.1.1	Some of the reasons for Scala success . . . . .	3
1.2	Aims and Objectives . . . . .	4
1.2.1	Proposed functionality of the web application . . . . .	5
1.3	Main Research Questions . . . . .	5
1.4	Justifications / Benefits . . . . .	6
1.5	Feasibility . . . . .	7
1.6	Proposed Methodologies . . . . .	8
1.6.1	Literature Review . . . . .	8
1.6.2	The Web Application Development . . . . .	9
1.7	Project Plan . . . . .	10
1.8	Expected Results . . . . .	12

1.9 Conclusion . . . . .	12
<b>2 Literature Review</b>	<b>14</b>
2.1 Introduction . . . . .	14
2.2 General Concepts . . . . .	15
2.3 Specific Characteristics . . . . .	18
2.3.1 Object Oriented Programming . . . . .	18
2.3.2 Functional Programming . . . . .	21
2.4 Short History . . . . .	23
2.5 Interesting Ideas for Further Study . . . . .	24
2.5.1 A Definitive Programming Language . . . . .	24
2.5.2 Artificial Intelligence . . . . .	25
2.5.3 LambdaFicator . . . . .	25
2.6 Conclusion . . . . .	26
<b>3 Analysis and Design</b>	<b>28</b>
3.1 Introduction . . . . .	28
3.2 Proposed Methodology . . . . .	28
3.2.1 Learning Material . . . . .	28
3.2.2 Web Application . . . . .	29



3.3	Proposed Tools . . . . .	32
3.3.1	Hardware . . . . .	32
3.3.2	Software . . . . .	32
3.3.3	Deployment . . . . .	33
3.4	Web Application Content Design . . . . .	34
3.4.1	The Lecture Structure . . . . .	34
3.4.2	Writing Style Tips . . . . .	35
3.5	Use Cases . . . . .	35
3.5.1	Prototype 1.0 . . . . .	35
3.6	Sequence Diagrams . . . . .	36
3.6.1	Prototype 1.0 . . . . .	36
3.7	User Interface Design . . . . .	37
3.7.1	Prototype 1.0 . . . . .	37
3.8	Database Schema Design . . . . .	40
3.8.1	Prototype 1.0 . . . . .	40
3.9	Back End Design . . . . .	41
3.9.1	Prototype 1.0 . . . . .	41
3.10	Project Repository Design . . . . .	42

3.11 Conclusion . . . . .	43
<b>4 Implementation</b>	<b>44</b>
4.1 Introduction . . . . .	44
4.2 Development Set-Up . . . . .	45
4.3 User Interfaces (Views) . . . . .	47
4.3.1 Prototype 1.0 . . . . .	47
4.4 Data And Helper Classes (Models) . . . . .	50
4.4.1 Prototype 1.0 . . . . .	51
4.5 Business Logic (Controllers) . . . . .	56
4.5.1 Prototype 1.0 . . . . .	58
4.6 Deployment . . . . .	59
4.6.1 Required Steps for Deployment Set-Up . . . . .	59
4.7 Conclusion . . . . .	65
<b>5 Testing and Evaluation</b>	<b>67</b>
5.1 Introduction . . . . .	67
5.2 Unit Tests of Models and Utilities . . . . .	69
5.2.1 Prototype 1.0 . . . . .	69
5.3 Acceptance Tests (Selenium IDE tests) . . . . .	72

5.3.1	Prototype 1.0 . . . . .	73
5.4	Conclusion . . . . .	74
<b>6</b>	<b>Functional Programming with Scala</b>	<b>75</b>
6.1	Introduction . . . . .	75
6.2	Why Would You Read These? . . . . .	76
6.2.1	Is there an alternative? . . . . .	78
6.3	What is Functional Programming? . . . . .	78
6.3.1	Imperative Programming Paradigm . . . . .	79
6.3.2	Functional Programming Paradigm . . . . .	80
6.4	Functions Everywhere . . . . .	82
6.4.1	Pure Function . . . . .	82
6.4.2	General Terms . . . . .	84
6.4.3	Functions as table lookups . . . . .	86
<b>7</b>	<b>Conclusion and Further Work</b>	<b>87</b>
7.1	Introduction . . . . .	87
7.2	Achievements . . . . .	87
7.3	Personal Gain . . . . .	87
7.4	Further Work . . . . .	88

7.5 Conclusion . . . . .	88
Bibliography . . . . .	88

# Chapter 1

## Introduction

### 1.1 Introduction

In the current world of multi-core processor computers, distributed systems and big data, we are witnessing a paradigm shift in the computational model used in application development industry. It is the shift from imperative programming style toward declarative programming and the rise of the functional programming languages[\[1\]](#). JVM ecosystem is no exception.

Even though Java being still most used JVM language, some other languages such as Scala, Groovy and Clojure are getting more popular every day. There are various reasons for the popularity where each language has its own strengths, but what they all have in common is a much richer support for functional programming in comparison to Java. Java itself with recent SE 8 update adapted to this trend and brought limited support for functional programming as well.

Scala programming language [\[2\]](#) is getting a big momentum over the last few years

in the industry[3]. Companies such as Twitter, LinkedIn[4], Netflix, Sony, Foursquare had switched to Scala in their application development. Scala is being a big player in the realms of Reactive programming, Distributed Systems and Big Data[5].

### 1.1.1 Some of the reasons for Scala success

- Scala is an expressive but elegant multi-paradigm programming language with the performance comparable to Java. It is often regarded as *"the most evolved programming language on the planet"*. Scala has better support for object-oriented programming than Java (better object model, higher-kinded parametric polymorphism, multiple inheritance, built-in singleton pattern and more. At the same time, Scala has advanced support for functional programming comparable to ML-like languages, such as Haskell, F# or oCaml. In this context, Scala emphasizes immutability by default, supports first-class and higher-order functions, lambdas, currying and partial function applications, built-in lazy sequences, monads, algebraic data structures, type classes, pattern matching, for comprehension and more.
- Scala compiles to JVM byte code.class files which mean that language is fully compatible with existing Java libraries and frameworks and vice versa. Mixed Java / Scala projects are quite common with Scala compiler support for cyclic compilations. Many projects written in Java were refactored to Scala to utilize the language expressiveness or features which are just missing in Java, such as Scala *Actors* concurrency model.
- Scala is *scalable* by design, with a powerful compiler supporting plug-ins and advanced macros. The compiler is written in Scala itself and both, the language and the compiler are open source projects. Scala supports operator overloading and being a functional programming language, it is a well suited for the

development of interpreters and Domain Specific Languages (DSL).

- The existence of powerful, open source and award-winning frameworks[6], toolkits, and platforms written in Scala, such as Akka, Spark, Play[7] is a powerful drive for language adoption in enterprise development.

## 1.2 Aims and Objectives

The aim of this project is to study Scala programming language, functional programming paradigm and to develop a learning material in the form of lecture prototypes for a fictional "functional programming" college module. The idea came to life in the moment I saw that there is no module dedicated to functional programming in the final year of my studies in ITB. Effectively I realized that functional programming was not mentioned once in any of the modules throughout all four years of Computer Science course.

The secondary aim of the project is to deliver the lectures to potential students who are also interested in functional programming in the form of simple web application. The application will be developed using Play Framework rapid web development platform. In this endeavour, I will concentrate on studying the framework, rather than to dedicate too much valuable time to web application development. I have developed a number of web pages, web APIs and Single Page Applications (SPA) during my studies here in ITB. On the other hand, I did not have a chance to learn any enterprise web development platform other than JavaEE.

### Reasons:

- Strong personal interest in functional programming.

- Lack of modules dedicated to functional programming in ITB curriculum.
- Solid background in JVM platform.
- Desire to pursue a professional career in Java / Scala development.
- Interest in Lightbend Reactive Platform[\[32\]](#) (Play Framework).

### 1.2.1 Proposed functionality of the web application

- A student can register the account.
- The authenticated student will gain access to the functional programming with Scala lectures and exercises.
- The student can adjust the account settings.
- The student will have access to an online Scala interpreter where he could carry out coding exercises.
- The student will be able to communicate with other users through a chat window or other messaging functionality.

## 1.3 Main Research Questions

- What exactly is functional programming and what benefits does it bring to application development?
- What is Scala programming language and how it supports the functional programming?
- How would I explain functional programming to others?
- What exactly is Play Framework and how can I develop a web application with it?
- What are the advantages and disadvantages of the Play Framework in terms of



productivity, performance and scalability and in comparison to other frameworks?

## 1.4 Justifications / Benefits

As mentioned earlier, the functional programming is totally ignored by ITB Computer Science course curriculum. In my opinion, this creates an 'educational gap' in the curriculum. Personally, I feel that the topic belongs in the course and it is the natural progression from imperative object-oriented programming and design patterns which had strong coverage in the third year of the course.

- **Filling the 'educational gap' in the Computer Science course curriculum by creating learning web resource.**

The preliminary research on the current state of the application development industry is suggesting that fluency in a functional programming language is a really valuable skill for any developer to possess. Developers able to code using languages such as Scala, Clojure, F# or Lisp dialects are in high demand. Reasoning about the functional programs requires different 'mind-set' or 'thought process' in comparison with object-oriented programming. Even if one would never use purely functional programming professionally, understanding of the concepts will make one a better programmer overall.

- **Acquiring valuable skills and deeper insides into different programming paradigms and design patterns.**

Also, Scala seems to be very elegant language and takes the good design ideas from many other programming languages. The language was designed with the scalability of the syntax taken in the consideration, which allows the language to 'shape-shift'

towards the needs of its users. In my opinion, the Scala programming language would be the right tool for me to learn functional programming paradigm. In my preliminary research I found number of books dedicated to functional programming with Scala.

- **Learning Scala programming language and Java Virtual Machine (JVM).**

Even though learning functional programming with Scala is the main aim of this project, the additional work will be carried out by exploring Play Framework and developing a web application. The preliminary research suggests the framework is a powerful tool for rapid web development. Play seems to be heavily inspired by Ruby on Rails. It supports reactive web application development and seems to be a cleaner alternative to legacy Java Enterprise stack [8].

- **Building valuable skill set in contemporary web application development.**

## 1.5 Feasibility

### Project requirements

- Access to relevant study material and tutorials dedicated to Scala programming language and Play. There is no material available in ITB library, but there is a number of books published and available over the internet. *Estimated cost is 100 euros.*
- Access to personal computer. I'm the owner of the laptop computer, which should serve as the project workstation without too many constraints. Additional research must be done to decide on the operating system and other tools. Using

free software is most desired. *Estimated cost is 0 euros.*

- Access to the internet, deployment, DNS and versioning control services, printer and thesis binding. *Estimated cost is 300 euro.*
- At least 10 hours per week to carry out the project related work for the whole project duration. This estimation is based on personal experience I gained by working on two academic projects during my studies and on experience working on a research project as part of the summer internship.

In my opinion, all the requirements could be met to assure this project feasibility with the minimal cost associated. With proper planning, use of the personal assets and free software when possible, I should be able to meet the project goals within the given time scale.

## 1.6 Proposed Methodologies

### 1.6.1 Literature Review

The first step is to carry out the research on functional programming [9], Scala programming language [10], MVC design pattern and Play Framework [11]. I'm planning to use dedicated books I acquired recently, plus online tutorials and blogs. I will analyze, summarize, and carry out the coding tutorials. The thorough literature review at the beginning should lay a foundation for following web application development. I will continue to research relevant materials for the whole duration of the project.

## 1.6.2 The Web Application Development

The real challenge lies in the fact that I have a little experience developing web applications using MVC framework and no skills in technologies I decided to use during this project. I have only general ideas how to design such an application. Only layers I can design reasonably well in advance are the database relations and user interface.

Therefore, I must approach this problem using some kind of adaptive methodology and avoid 'a big design in advance' approach. I would argue that Prototyping SDLC is the best methodology to carry out the development. In this model, the developer basically re-analyzes, re-designs, re-implements and re-test application prototype until the product is accepted by the client. Instead of creating 'a big design in advance',

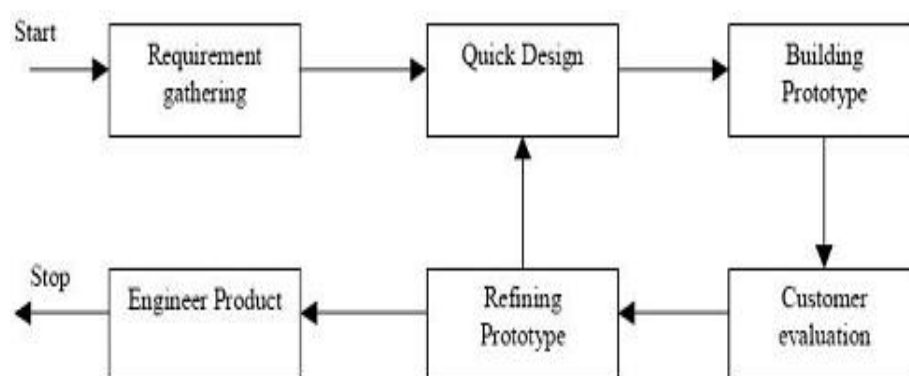


Figure 1.1: *Prototyping SDLC diagram*

prototyping methodology is allowing the developer to change the design with each iteration of the design / development / evaluation / refining circle. I should expect major changes in design based on my lack of experience in web development and technologies I decided to use. The prototyping methodology will allow me to have a simple, but functioning prototype reasonably early. As more insights are acquired from the research, the prototype can be re-designed to implement additional functionality

or re-evaluate the design.

## 1.7 Project Plan

### Work Breakdown Structure (WBS)

#### 1. Research.

- (a) Research on Scala programming language design ideas and syntax, MVC design pattern, Play framework, user interface and database design.
- (b) Literature Review of papers dedicated to the functional programming paradigm and test driven rapid web application development with Play.
- (c) Building development platform.

#### 2. Front-End design.

- (a) Creating wire-frame design for new view.
- (b) Constructing the view.

#### 3. Database Design.

- (a) Adding the database entity for new view.
- (b) Designing the relationships with existing entities.

#### 4. Prototype Development.

- (a) Creating the controller object.
- (b) Creating unit tests.
- (c) Implementing method bodies using test driven development methodology.
- (d) Repeating the steps 1, 2, 3 with additional functionality until the final product is build.

#### 5. Quality Assurance.

- (a) Performing integration tests of the whole MVC pipeline.
- (b) Performing user tests.

## 6. Post Implementation Maintenance.

- (a) Making prototype production ready and deploying the prototype.
- (b) Monitoring deployed application and adding more content.
- (c) Adding content to the application.
- (d) Working on project Report.
- (e) Repeating steps 5.(a), 5.(b), 6.(a), 6.(b), 6.(c), 6.(d)
- (f) Report Binding.

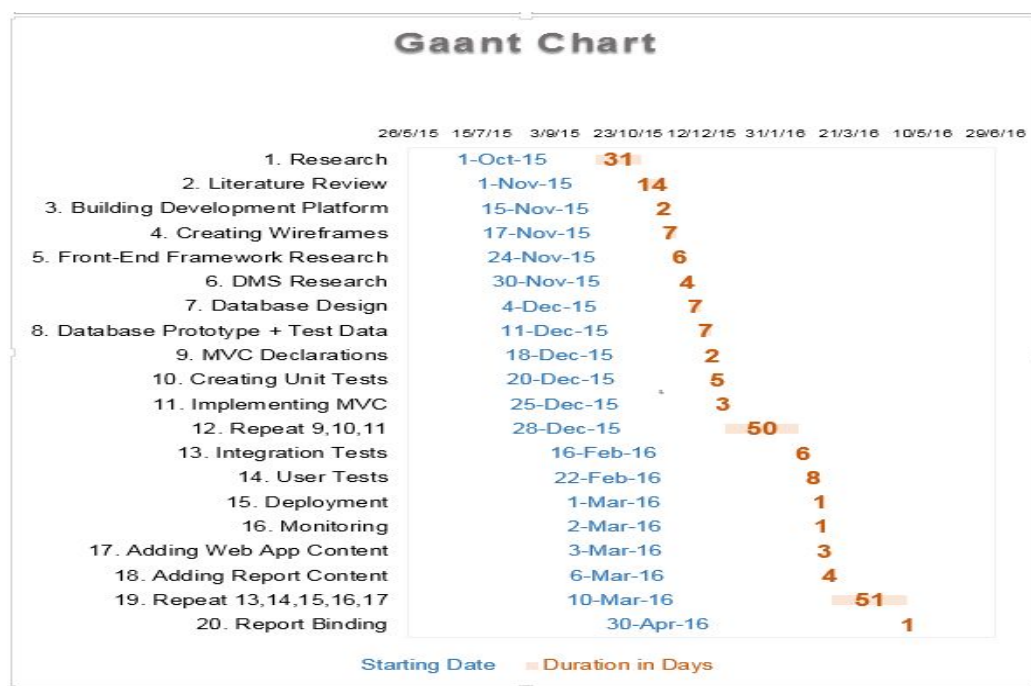


Figure 1.2: *Gaant Chart*

## 1.8 Expected Results

The project would be considered successful if it would meet at least two following criteria and goals. In the case of three or more goals accomplished I would consider project outcome to be very successful.

1. The gain in Scala coding skills and creation of enough learning material content for Scala course web application.
2. The comprehensive research on the functional programming paradigm and design patterns performed and gained the ability to code in the declarative style.
3. The delivery of fully functioning prototype of Scala course learning web application.
4. The acquisition of skill-set in rapid web development using Play framework.

## 1.9 Conclusion

At the end of this project, I will hopefully introduce some of Scala dedicated learning material in the form of an interactive web application. It can be used in a fictional course module for a fictional college. The application main purpose is to promote the interest in Scala programming language and functional programming paradigm. Especially on the personal level. After three years of my studies with ITB, I didn't learn anything about functional programming. I chose this project so I can gain a knowledge and skills which are really missing in my skill-set and which could prove very useful in the future. I hope as a fourth year student I am ready to face this challenge.

This research project will not contribute at all to the discipline area. Maybe

only in a sense that it will hopefully bring one more student with the passion for programming languages to the functional programming paradigm. And maybe if other students or lecturers will see how elegant and declarative functional programming really is, the contribution could be a bit more significant.



# Chapter 2

## Literature Review

### 2.1 Introduction

Computer programming is a fascinating and vast subject to study. Programming languages began as an attempt to translate the human language and the way of human thinking into the language of a computer.

The idea was to make it easier, more efficient for programmers to write the programs. With the evolution of computers, as the problems to solve became more complex, the programs became larger and more sophisticated. The programming languages quickly evolved into the forms commonly referred to as high-level programming languages. In these forms, the programming languages are hiding the internal hardware details and offer a higher level of abstractions allowing programmers to write the programs using familiar terms with an ability to model real life objects.

The purpose of this paper is to review some of the studies dedicated to concepts and design ideas behind the high-level programming languages. The research done in this

field is immense. It is out of the scope of this paper to cover every concept, paradigm, or language and rather study some of the most important concepts in general. Then I will continue to study research papers dedicated to two most popular paradigms - object-oriented and functional programming. I will try to identify and discuss their key concepts, strength, and weaknesses and outline their history in short. I chose reading materials with a secondary intention to lay down a foundation for further studies toward a deeper understanding of the paradigms and transitions between them.

## 2.2 General Concepts

In *The Conception, Evolution, and Application of Functional Programming Languages*[\[12\]](#) author defined a programming paradigm as an approach to a computer programming based on a coherent set of principles or a mathematical theory. The purpose for a paradigm existence is to solve a specific type of a problem. Each paradigm consists of a number of concepts. Any programming language can support one or more paradigms and the language which support more than one paradigm is called the multi-paradigm programming language. Different languages can interpret the concepts of a paradigm differently and often the implementations differ from language to language as well. The author listed around 30 useful programming paradigms implemented by modern programming languages.

Author then identified the two most important properties which differentiate the programming paradigms:

1. *Observable nondeterminism* is when a program is not completely determined by its specification. In other words with each execution the same program can produce different results. Observable nondeterminism is caused by the run-time

scheduler and its usual effect is a race condition, which is often used as the synonym for Observable nondeterminism.

2. *Named state* is an ability of a program to store values in time. This ability is highly influenced by the paradigm it contains it.

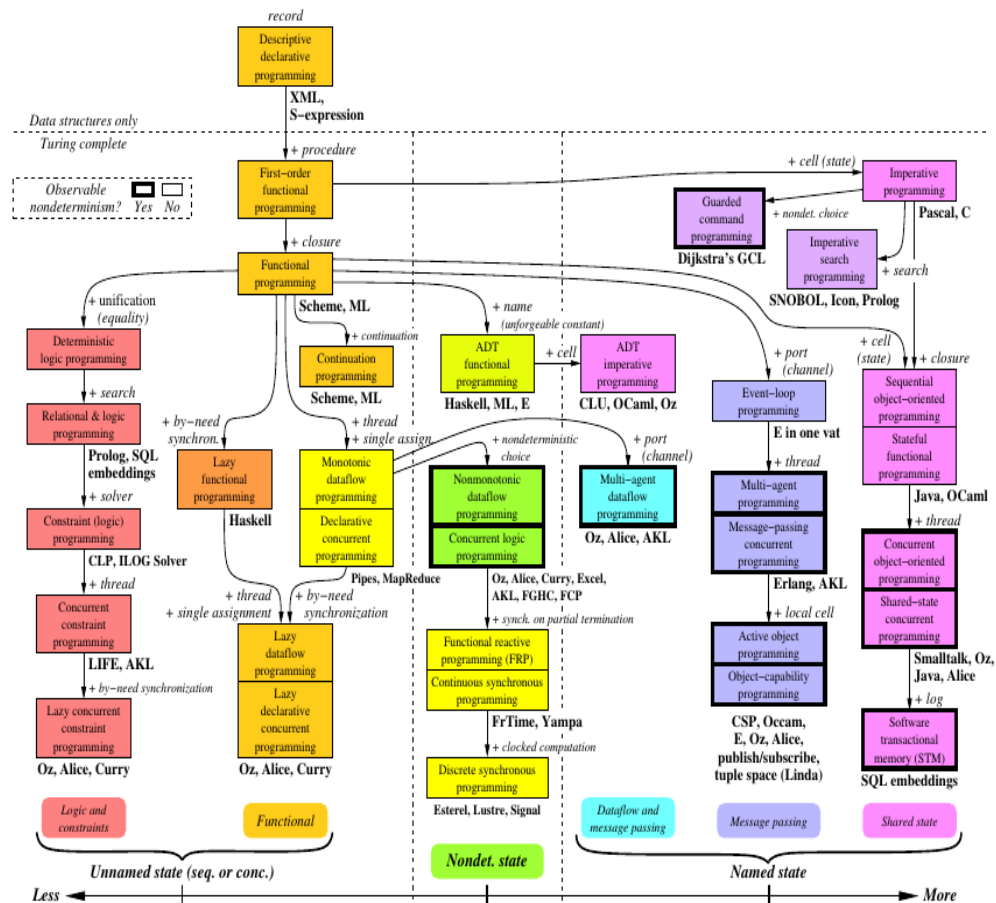


Figure 2.1: *Taxonomy of programming paradigms* (Van Roy, page 15)

As mentioned before, the programming paradigms are based on a number of concepts. Author identified four most important programming concepts:

1. **Record** is a data-structure. Every programming language should be able to work with records. Arrays, lists, strings, trees and hash tables are derived from records.

2. **Lexically scoped closure** is a record storing a function together with an environment referenced at the time of that function definition. Closure is a very powerful concept and constructs such as *objects* or *control structures* have been implemented with closures in many programming languages.
3. **Independence** is when a program is constructed from independent parts. When parts don't interact with each other we call them *concurrent*. When an order of the execution is given, the parts are called sequential. The interaction between the parts is called communication. Author recognizes three levels of concurrency:
  - (a) *Distributed system*, where concurrent activities (parts) are computers.
  - (b) *Operating system*, where concurrent activities are processes. This level of concurrency is often called competitive concurrency because the processes are competing for access to system resources.
  - (c) *Inter Process*, where concurrent activities are threads. This level of concurrency is called *cooperative concurrency* because the threads are cooperating to achieve the result of the process. There are two popular paradigms for inter process concurrency: *Shared state concurrency*, where shared data are accessed by threads using control structures called *monitors*. The second approach is *message-passing concurrency*, where threads are communicating by sending messages to each other.
4. **Named State** Author showed how named state could be implemented with a help of internal memory (variables).

## 2.3 Specific Characteristics

### 2.3.1 Object Oriented Programming

In *A Survey of Object Oriented Programming Languages*[\[13\]](#) authors argued that consensus on what key concepts of Object Oriented Programming still don't exist. They argue that the most fundamental concepts behind the paradigm are as follows:

1. **Class** is a mechanism which group together attributes and methods with common properties. The class describes the run-time behavior of the objects instantiated from it. The well-designed class would have an immutable interface clients can use.
2. **Abstraction** is a simplified view of reality. Presented to clients by class methods and attributes.
3. **Inheritance** is a mechanism to create hierarchical class designs by creating a child class of the original class. A child class inherits the parent class behavior which can be then extended. Multiple inheritance allows for a class to inherit the behavior of multiple parent classes.
4. **Encapsulation** is hiding implementation details within the class. Only the interface is presented to the clients.
5. **Polymorphism** is allowing to similar looking structures to handle a variety of objects.

Authors then discussed inheritance and polymorphism in detail. I will outline the important key points.

## Inheritance

As mentioned earlier the inheritance brings the hierarchical relationship into the class model. Authors pointed out that many languages define the most generic class that is an ancestor for all the classes in the language. This applies that any class can be downcast to the pointer of that ancestor. Authors pointed out that inheritance provides the ability to represent an “*is a*” relationship in software. This relationship can be violated if child class extends the inherited code the way which changes the code semantics. It is the reason for the inheritance to be used with caution.

Inheritance also allows to represent generalization /specialization relationships with method redefinition. This functionality is breaking the encapsulation because the child class has access to parent class hidden methods and attributes. This problem can be addressed by defining a well-defined interface for the descendants. Authors argued that literary every object-oriented language provides the ability for a child class to invoke parent class methods. Even those methods which were redefined in the child class.

Some of the languages support the feature which restrict method redefinition. A method is marked ‘*frozen*’ or ‘*final*’ and no child class can redefine this method. Authors warned that inheritance has also an adverse effect on synchronization requirements of a concurrent object. This problem is usually named ‘*inheritance anomaly*’. The problem arises when a class with concurrent code is derived. The careful redefinition of the inherited methods is necessitated to preserve the synchronization requirements. This necessity denies ‘*reuse*’ benefits of inheritance. The multiple inheritance potential risks were then discussed in details. Authors explained various problems which could arise such as directed acyclic graph in a class hierarchy, method name collisions or repeated inheritance problem.

Authors outlined various solutions to deal with the problems and outlined the alternatives to multiple inheritance used by some of the object-oriented programming languages such as interfaces, mixins or delegation.

## Polymorphism

As mentioned earlier, polymorphism allows programmers to write functions and classes which work uniformly with different types. Authors listed four distinct types of polymorphisms grouped into two categories. *Ad-hoc Polymorphism* and *Universal Polymorphism*. The difference between the categories is based on the fact that ad-hoc polymorphic functions execute code only for a small set of potentially unrelated types while universal polymorphic functions execute the same code for an infinite number of types.

Authors then defined the two types of ad-hoc polymorphism: The first type is *Overloading Polymorphism*, where the polymorphic function has the same name, but a different signature. Within this type authors distinguished between a *method* overloading and *operator* overloading and discussed each in detail.

The second type of ad-hoc polymorphism is *Coercion Polymorphism*. With coercion, the value of an argument can be converted to the value of another argument from the list of a method arguments. Authors pointed out that the difference between those two of ad-hoc polymorphism types is often blurry, especially in untyped and interpreted programming languages. The universal polymorphism is divided to two kinds as well. The first kind is *Parametric Polymorphism* aka generic programming. Generic programming uses type parameters to determine the type of a method argument. Authors pointed out that parametric polymorphism is only relevant for statically typed languages because dynamically typed languages infer types at run-time

and hence have generic programming built into them. The second kind is *Inclusion Polymorphism*, which gives different classes the ability to handle the same functionality.

Inclusion Polymorphism is what we call *Inheritance* in object oriented programming. Inclusion polymorphism is implemented through *dynamic binding* or sometimes called *late binding* because a method is bind to the message at the run-time. It is relevant for situations when a child class has an overridden method of the parent class and it is not obvious which method is being invoked. The search for the right method is then performed by the compiler (interpreter) at run-time. The dynamic binding uses the most specific version of a method.

Authors mentioned that some of the languages implement static binding where methods are bind to messages at compile time and it will always bind to a base class method version.

### 2.3.2 Functional Programming

In *The Conception, Evolution, and Application of Functional Programming Languages*<sup>[12]</sup> the author distinguished four key characteristics of modern functional programming languages.

#### Higher-Order Functions

In functional programming, the functions are treated as '*first class values*', which means that they can be stored in data structures, passed as other function arguments and returned as results. The author pointed out that the function is the primary abstraction mechanism over values. He showed with examples how to compose higher-order functions.



## Lazy Evaluation

Often called Non-Strict Semantics or call-by-need. Its primary feature is that arguments in a function call are evaluated at most once. In some cases, it does no evaluation at all. The author explained with the help of an example how lazy evaluation frees a programmer from concerns about evaluation order and pointed out the ability of the lazy evaluation to compute with infinite data-structures.

## Data Abstraction Mechanisms

The author pointed out that a data abstraction improves modularity, security and clarity of programs. He explained that modularity is improved because one can abstract away from implementation. Data abstractions prohibit interface violations which improves security and that programs are clearer because of the self-documenting quality of the data abstractions. He argues that *strong static* typing eliminates type violations and run-time errors. He continued the discussion by describing *algebraic* (concrete) data types, *type synonyms*, and *abstract data types*.

## Pattern Matching

The lack of side-effects in functional programs allows to apply pattern matching or sometimes called equational reasoning. The author explained the basics behind the feature with the help of code examples and outlined some of pattern matching shortcomings.

In *Why Functional Programming Matters*[\[14\]](#) author summarizes functional programming characteristics and advantages as they are usually used in the literature as follows:

- The functional program consists of functions.
- The functional program uses no mutable variables which, in general guarantee that program contains no side-effects. A program without side-effect is free of a major source of bugs.
- Since a function call produces no side-effects and for given arguments produces the same calculation result independently on when it is evaluated, the order of execution is irrelevant and functional programs are referentially transparent. This freedom makes functional programs easier to reason about.

The author argues that this often used list of strengths of functional programs is describing what functional programming is not (no assignments, no side-effects, no flow of control) and fails to emphasize what functional programming actually is. The list fails to emphasize the modularity as probably the most powerful characteristic and advantage of functional programs. He convincingly argues the ability of higher-order functions and lazy evaluation to increase the modularity by serving as a ‘glue’ for the program fragments. He provides a number of examples with code to support his claims.

## 2.4 Short History

First generally accepted object-oriented programming language is Simula (1967). With the introduction of Smalltalk (1962-1980) the paradigm gained some momentum. Most of the concepts of object-oriented programming were implemented in Smalltalk. In early 1980 the concepts were integrated into C programming language and resulting language was called C++. In the 1990s, the similar language was developed called Java by Sun Microsystems. Java became soon one of the most popular object-oriented

languages. Then in 2000, Microsoft announced .NET platform and C# programming language. C# is in many respects similar to Java. [13]

Functional Programming is heavily influenced by lambda calculus invented by Alonzo Church in 1936. First of the programming languages implementing lambda calculus was Lisp specified in 1958. Next significant language in terms of contributions to functional paradigm was Iswim introduced by Peter Landin in 1966. Probably the first functional language which received wide-spread attention was FP specified in 1978. In mid 70's several research projects related to functional programming emerged in the UK. Specifically the work of Gordon, Milner, and Wadsworth. They developed ML programming language which brought the invention of the type system (Hindley-Milner Type System). In early 80's David Turner at the University of Kent developed three languages which most faithfully characterize "modern school" of functional programming: SASL, KRC and Miranda. [12]

In later 1990s and after 2000 the functional programming has gained a great momentum and penetrated mainstream programming. Haskell, F#, Clojure, Scala are some of the examples of functional programming languages. Many object-oriented languages added functional features and become multi-paradigm languages. Java, Python, C# are examples of such.

## 2.5 Interesting Ideas for Further Study

### 2.5.1 A Definitive Programming Language

In *Programming Paradigms for Dummies: What Every Programmer Should Know*[16] the author is presenting four research projects, each trying to solve a very

different problem, but all four project considered language design as a key factor to achieve success. Turned out that programming languages invented in each project have very similar structure supporting same paradigms: strict functional programming, declarative concurrency, asynchronous message passing and global named state. The invented languages are Erlang, E, Distributed Oz and Didactic Oz. One could infer ideas for a design of ‘perfect language’ and it could be quite interesting to study those four programming languages.

### 2.5.2 Artificial Intelligence

In [14] the author of the paper is using examples of composing programs from lazy evaluated higher-order functions. The final example is the alpha-beta heuristic algorithm. This algorithm is often used in computer games to estimate how good a player position is in the game. The author used Miranda programming language syntax, but it would be very interesting to implement the algorithm in some other functional programming language such as Scala or Haskell.

### 2.5.3 LambdaFicator

Java 8 update brought a few functional programming features in the language. Namely functions as first class values, lambda expressions and closures. Also fluent Stream API which uses monads, lazy evaluation, and higher-order functions. In *Crossing the Gap from Imperative to Functional, Programming through Refactoring*[15] authors presented the analysis, design, implementation and evaluation of LAMBDAFICATOR, the automatic refactoring tool, which converts old-style code prior to Java 8 update into the functional style. Namely tool does two refactorings:

- Anonymous inner classes to lambda expressions
- External iterators to internal iterators (from *for loops* to Stream API *higher order function chains*)

In the paper, authors discussed their motivations, outlined the implementation algorithm in the detail and determined the usefulness with a thorough evaluation. They applied the tool to four open source projects (ANTLRWorks, Apache Ivy, Junit, Hadoop) with very successful results. For example, the first type of refactoring reduced the number of source code lines by 2213 with 100 percent accuracy.

The tool is open source and available for download and would be very interesting to study it deeper and implement it using some other programming language such as Scala or Haskell.

## 2.6 Conclusion

The main purpose of this research was to study two most popular programming paradigms: object-oriented and functional programming. To identify and understand the key concepts and to find the similarities and differences. The first step was to understand general programming concepts such as record, closure, state and concurrency.

I continued to study characteristics of object-oriented programming languages. I learned that main building blocks of object-oriented programs are classes organized into hierarchies based on inheritance. I learned about polymorphism, the powerful feature of programming languages in general.

Also, I studied papers dedicated to functional programming and I found out that

functional programs are built from functions, which are then composed together to modules and programs. I was shown the elegance of lazy evaluation and pattern matching. I learned that functional programs are trying to eliminate state and to stay referentially transparent.

# Chapter 3

## Analysis and Design

### 3.1 Introduction

In this section, I will discuss the approach to the content and web application design. The main aim of this project is to explore functional programming methodologies, develop a learning material content dedicated to the functional programming with Scala programming language. The secondary aim is to develop a simple web application which would deliver the learning material to a potential consumer.

### 3.2 Proposed Methodology

#### 3.2.1 Learning Material

The lectures will have a form of static text with code examples and coding exercises, The references to cited paragraphs and the source study material will be included. There will be a link to online Scala REPL[\[18\]](#) available in each lecture,

offering to a student an environment to test the code examples and carry out the exercises. Lectures are proposed to be short, covering only one concept or a feature at the time. Lectures will be formatted using Markdown. Markdown is a text-to-HTML conversion tool for web writers. Markdown allows you to write using an easy-to-read, easy-to-write plain text format, then convert it to structurally valid XHTML (or HTML)[20].

Scala is an elegant but complex programming language. I realized that to be able to study functional programming with Scala, the student will need to learn the language syntax first. Even though Scala programs resemble Java programs in many ways and they can seamlessly interact with code written in Java[17], it doesn't mean that Scala is just Java with slightly different syntax. In contrary, Scala is purely object-oriented, has a different object model, a richer type system with the type inference, supports multi-inheritance through mixins, pattern matching, implicit references, partial function applications, operator overloading and much more. Basically, Scala is much richer language than Java and considered by many to be the most evolved programming language.

The lectures will cover the functional programming paradigm using Scala programming language in proposed form. These lectures will not cover the Scala syntax, but rather how to design functional programs and how to reason about them.

### 3.2.2 Web Application

Secondary aim of this project is to develop a simple web application which will publish the learning material over the internet to the potential students. Idea is to develop this web application using Scala and functional programming style. This could be quite a challenge, because I must first learn the language and the style, which is



actually the main aim of this project. If I fail to grasp the concepts I will be not able to build the application. I chose Play Framework to carry out the implementation of the application because it supports both Java and Scala programming languages and therefore in the case I will feel not comfortable enough to carry out development in Scala I can work in Java.

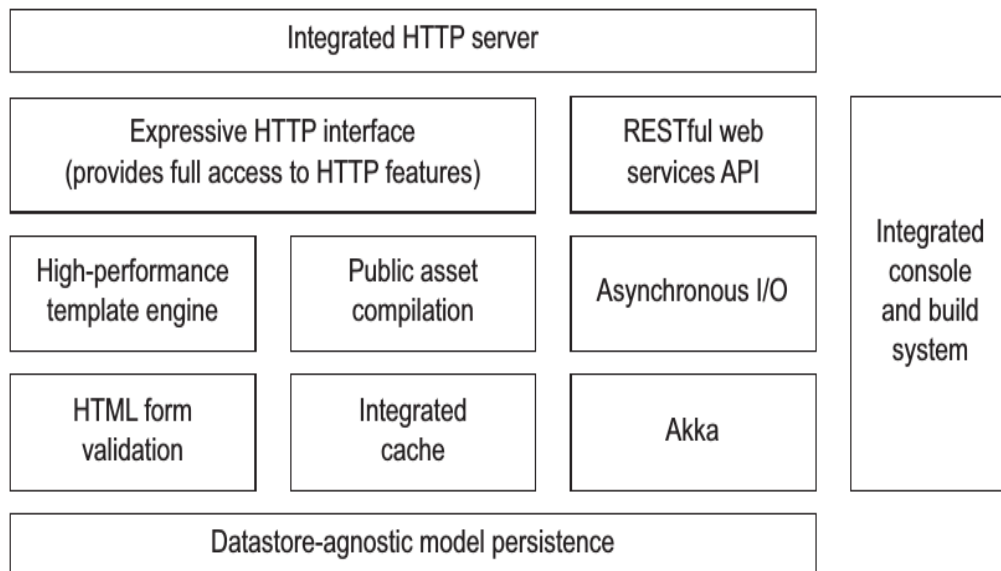


Figure 3.1: *Play Framework Stack*

### Play Framework Key Features Overview[\[11\]](#)

- Declarative application URL scheme configuration
- Type-safe mapping from HTTP to Scala API
- Type-safe templates
- Embraces HTML5
- Live code changes when you reload the page in your web browser
- Full-stack framework (fig. 3.1), including persistence, security and internationalization

### Play versus Java EE

From fig. 3.2 is apparent how the framework embraces the simplicity in comparison with layered Java Enterprise Edition architecture. Whole framework stack is build on top of Netty[19] client-server framework and doesn't require an application server container. Netty assures high performance of the stack and simple deployment of the application. Play has both, Java and Scala APIs (Application-Programming Interface) which will allow me use functional programming with Scala. I will discuss the framework in more depth in Back End Desing section 3.8 and Implementation chapter 4.

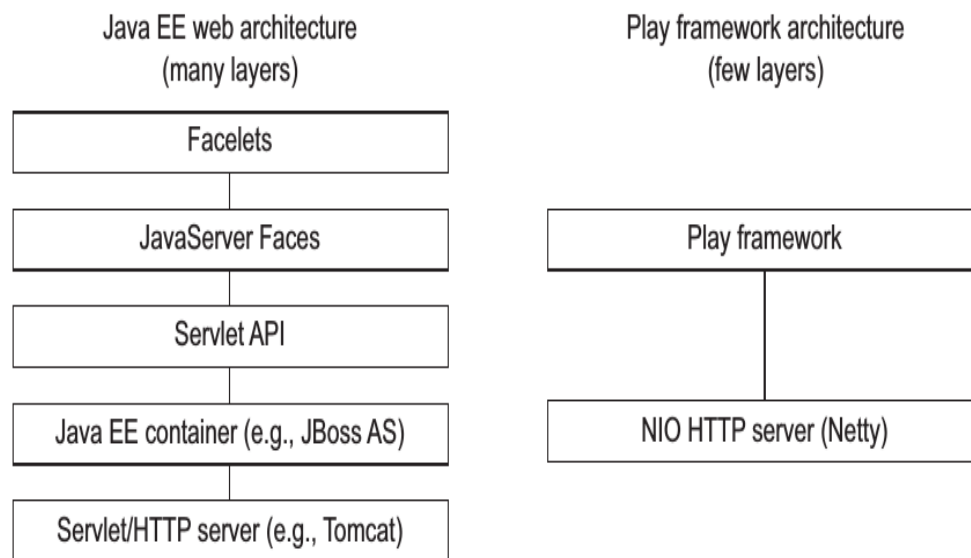


Figure 3.2: *Java EE 'lasagna' architecture versus Play architecture*

## SDLC - Prototyping Model

As discussed in Proposed Methodologies for The Web Application Development section 1.6.2, I will use Prototyping Software Development Life-Cycle Model to develop the web application. Each step in development cycle is already listed and explained in Work Breakdown Structure section 1.7 of Project Plan.

## 3.3 Proposed Tools

### 3.3.1 Hardware

The project will be developed on DELL Latitude E5540 laptop machine lent to me by School of Computer Science, UCD where I work part-time as a research assistant. The laptop has following specification and should be sufficient to support this project development.

- **Procesor:** Intel Core i5-4310 CPU 2.00GHz x 4
- **Memory:** 8GB RAM
- **Graphics:** Intel Haswell Mobile
- **Hard Disk:** 463 GB

### 3.3.2 Software

I decided to use an open source, or at least free software tools. The thesis report will be written in LaTeX instead of an office suite. Learning material will be written using the Markdown format, which require just a simple text editor. For the web application development I will need Java and Scala Development Kits with an IDE Studio. Additional tools will be needed to draw diagrams, take and edit screen shots, CSS, JSON or XML parsers and validation tools. For that I can utilize the operating system tools or online tools.

- **Operating System:** Ubuntu 14.04 LTS 64-bit[\[21\]](#)
- **Word Processing:** LaTeX suite with Gummi simple LaTeX editor[\[22\]](#)
- **Markdown editor:** Remarkable[\[23\]](#)

- **IDE:** IntelliJ IDEA[24]
- **Web Development Framework:** Play Framework 2.4[28]
- **Distributed revision control system:** Git[25]

### 3.3.3 Deployment

Play Framework has very good deployment model. The application doesn't require a Java application server container, which greatly simplifies the deployment process in comparison to Java EE applications. There are various deployment options available, for example the application can be deployed as a standalone package and just copied to the deployment machine.

Some of the cloud providers have built-in support for Play application deployment. Additional dependencies are handled by Simple Build Tool (SBT)[29] which is included with the framework. For example, during the development, the framework is utilizing memory H2 database, but for the deployment the database management system can be replaced with MySQL for example. All need to be done is to edit the web application configuration file and SBT will handle the rest automatically.

- **Heroku:** The Heroku Cedar stack natively supports Play framework applications. A Postgres database is automatically provisioned for Play framework applications.[26]
- **Azure:** Microsoft Azure Cloud supports Play applications deployment on a web server with *Azure Toolkit for Eclipse*. Another option is to dedicate a Virtual Machine (VM) to the application. Azure has numerous Linux images available, Ubuntu Server 14.04 included. The deployment should be then a simple question of moving the packaged application to the VM after all the required technologies

are installed on the VM, such as Java and Scala runtime and database server.[\[27\]](#)

## 3.4 Web Application Content Design

In preparation for learning material writing, I collected some of the guidelines and recommendations of ways to structure the learning content with some tips on the writing style. Following material is only a short extract from the research conducted on numerous online resources dedicated to e-learning.

### 3.4.1 The Lecture Structure

- **Introduction**

- Title
- Lecture content list
- Requirements (previous work or expected level of knowledge)
- Learning outcome
- Expected time requirements

- **Ending**

- Summary of main points
- Optional extra work

- **Signposts**

- Route the lecture by headings, which are related to objectives and emphasize the the structure of the lecture
- Indicate beginning and ending
- Emphasize important points

- Show relationships within the subject matter

### **3.4.2 Writing Style Tips**

- Keep it simple
- Keep it short
- The 20 words per sentence is the maximum
- 5 - 7 lines per paragraph
- Maintain the variety in the writing style
- Use positive rather than negative expressions
- Use visuals and examples often

## **3.5 Use Cases**

### **3.5.1 Prototype 1.0**

#### **Prototype 1.0: Login Use Case**

1. User logs in.
  - (a) User reads the lectures.
  - (b) User adjusts the account settings.
    - i. User resets password.
      - A. User confirms password reset in email.
    - ii. User changes the address.
2. User logs out.

### 3. User registers the account

(a) User validates the account in email.

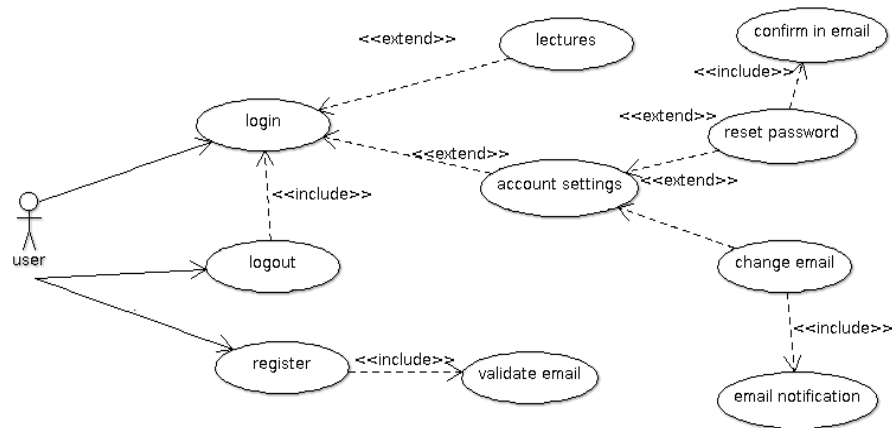


Figure 3.3: *Prototype 1.0: Login Use Case*

## 3.6 Sequence Diagrams

### 3.6.1 Prototype 1.0

#### Register Sequence Diagram

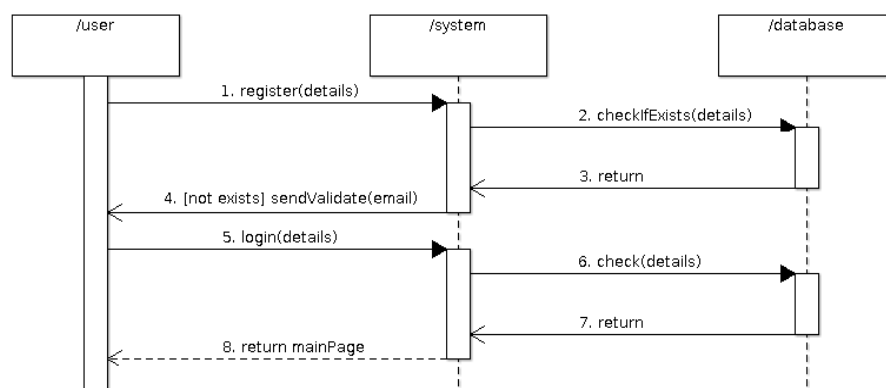


Figure 3.4: *Prototype 1.0: Register Sequence Diagram*

## Account Settings Sequence Diagram

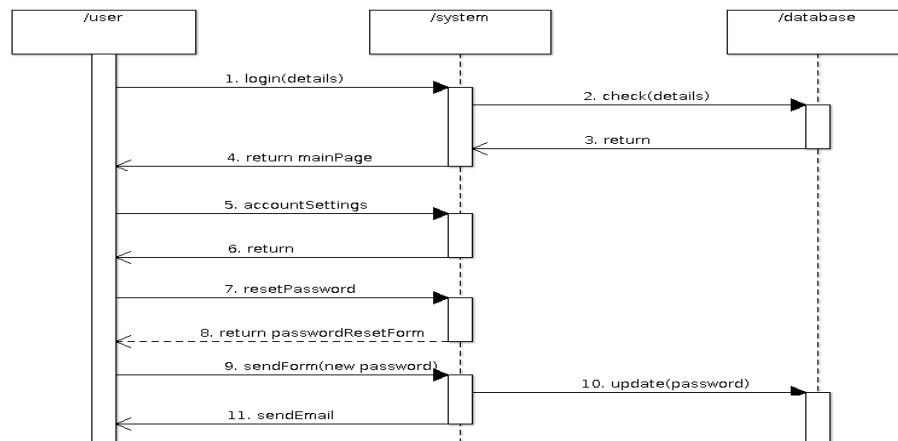


Figure 3.5: *Prototype 1.0: Account Settings Sequence Diagram*

## 3.7 User Interface Design

### 3.7.1 Prototype 1.0

#### Main Page Wireframe

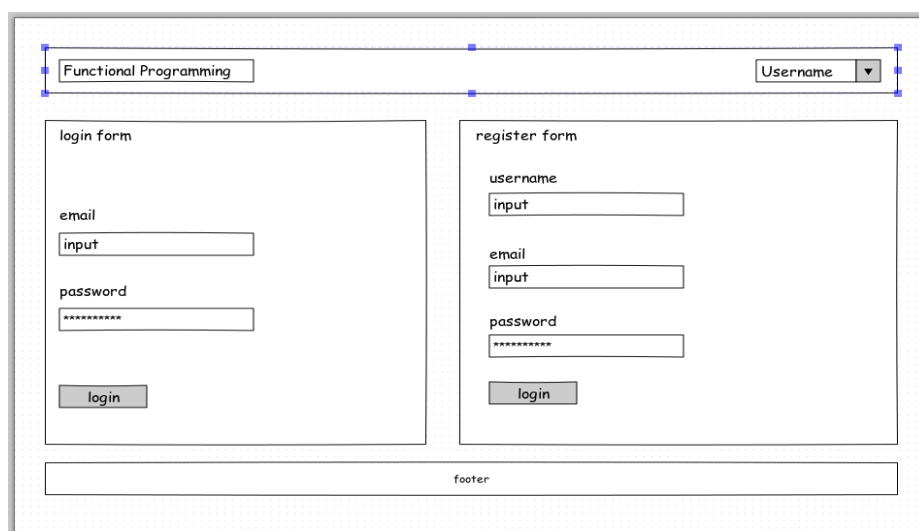


Figure 3.6: *Prototype 1.0: Main Page Wireframe*



On the main page, the user can log-in or register a new account. If the user attempts to log-in without the account, the attempt is rejected. The successful log-in will result in redirection to the dashboard page. Upon successful account registration, the user will receive the email with a confirmation link which he must click to validate the registration.

### Required Elements

- header (label, hidden drop-down menu)
- login form (labels, text input, password input, button)
- register form (4 labels, text inputs, password input, button)
- footer

### Dashboard Wireframe

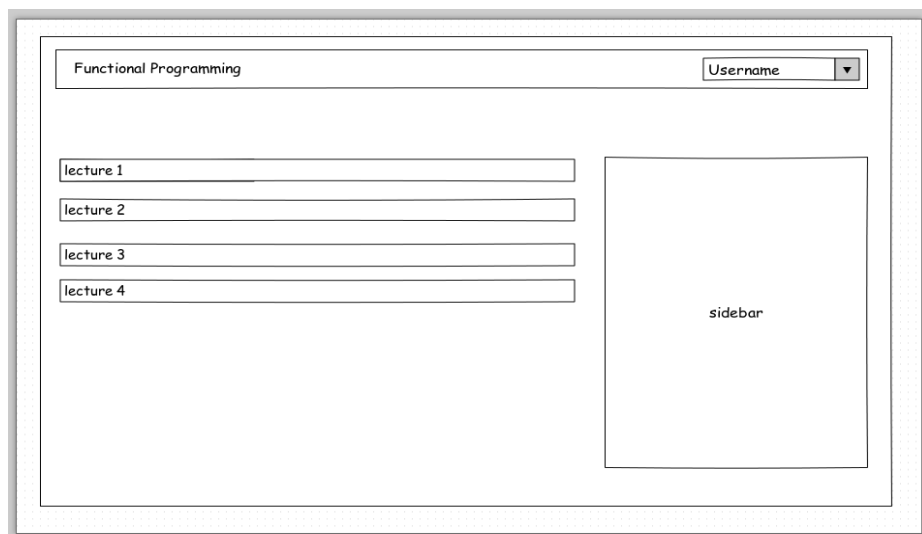


Figure 3.7: *Prototype 1.0: Dashboard (lectures) Wireframe*

On the dashboard page user will have access to the lectures. Also, the drop-down menu in the header will be visible and will allow the user to request the account settings page or log-out. The account setting request will result in the account settings

page redirection and the log-out request will result in the redirection to the main page.

### Required Elements

- header (label, visible drop-down menu)
- main area (links to the lectures)
- side area (for future functionality)
- footer

### Account Settings Wireframe

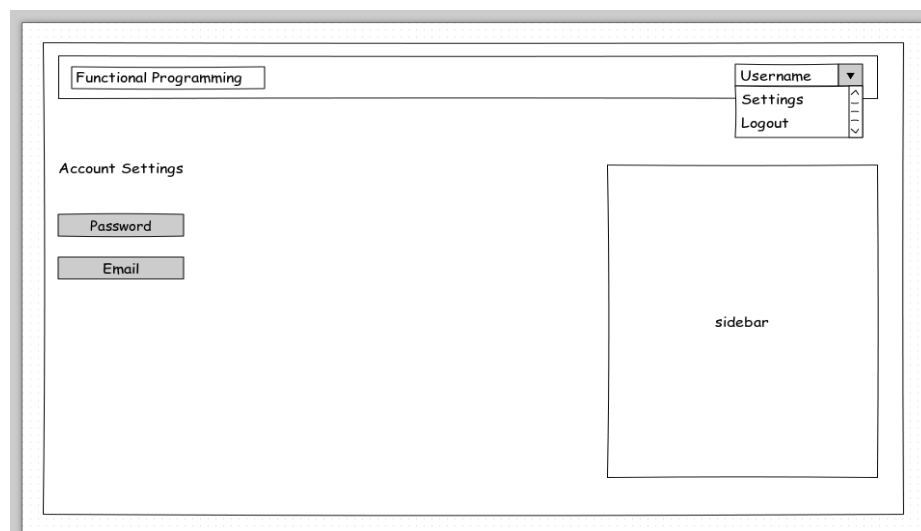


Figure 3.8: *Prototype 1.0: Account Settings Wireframe*

On the account setting page, the user can change the password or the email address. Upon the password change request, the email will be sent to the user with the confirmation link. Upon the successful email change, the user will receive the notification email.

### Required Elements

- header (label, visible drop-down menu)

- main area (2 buttons)
- side area (for future functionality)
- footer

## 3.8 Database Schema Design

### 3.8.1 Prototype 1.0

For the prototype 1.0 functionality, I will need the entity which will represent the user and the entity which will represent a random token. I will generate a token and associate it with the user every time a user requests registration, password change or email change. These tokens will then be embedded into an URL which will be sent to user's email. It's basically a way of identifying the user actions. The user then must click the URL to validate (confirm) the action. There should be a general expiration period associated with each token. Let's say one day. If the user doesn't confirm the requested action the action will be cancelled. For this functionality, I will need to save the token creation time-stamp as well.

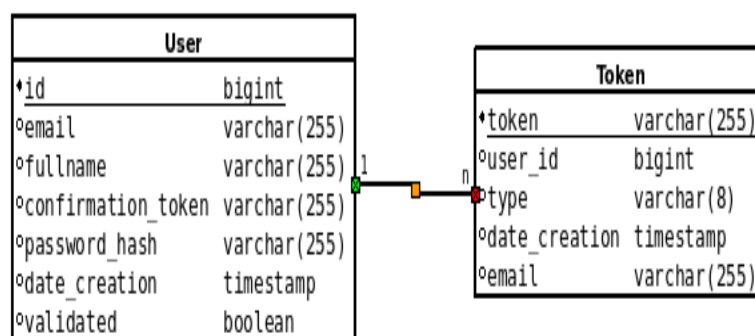


Figure 3.9: *Prototype 1.0: Entity Relations Diagram*

## 3.9 Back End Design

### 3.9.1 Prototype 1.0

In Play! Framework the user interfaces are represented by views. The view consists of HTML, Javascript and embedded Scala[30] code. The views are compiled to Scala classes and stored in /views folder. The views are rendered on demand by the server.

The user interacting with the views creates the requests to the resources. The incoming requests are handled by controllers. Each request is routed to the one controller method. Controller methods are processing the request and returning responses which in turn render the views. Controllers are stored in /controllers folder.

The requests are routed to the controllers by rules stored in /conf/routes file. In this file, the mappings from URIs to controller methods are defined. Play Framework uses REST like approach to routing. The /conf directory serves as a storage for all configuration files.

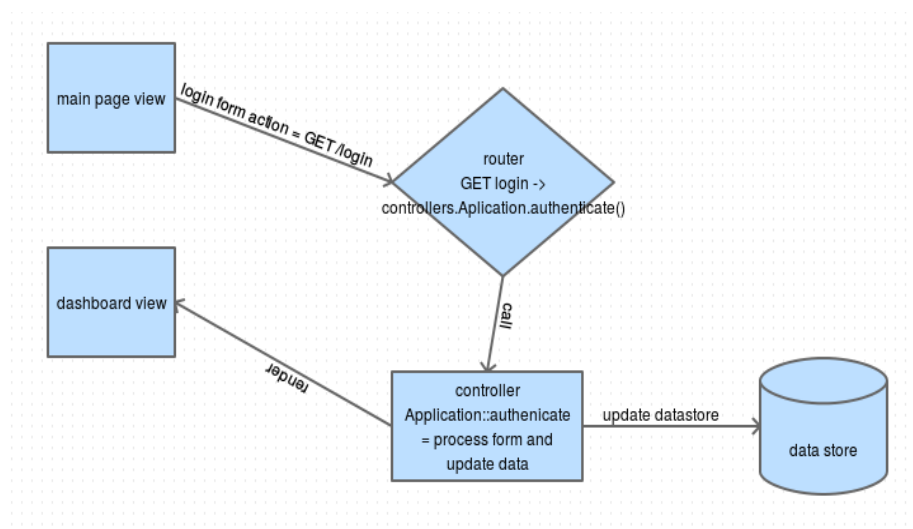


Figure 3.10: *View -> Request -> Controller -> Response -> View Diagram*

Play Framework is persistence agnostic, but it provides Java Ebeans ER mapper and H2 in memory database by default. H2 can be changed for other database implementation for the production mode and any other ER mapping framework can be used instead of Ebeans. In Ebeans, each entity is represented by a POJO annotated class called model. Models are stored in `/models` directory.

The layout of a Play application is standardized to keep things as simple as possible. After a first successful compile, a Play application looks like this:

1	app	→ Application sources
2	└ assets	→ Compiled asset sources
3	└ stylesheets	→ Typically LESS CSS sources
4	└ javascripts	→ Typically CoffeeScript sources
5	└ controllers	→ Application controllers
6	└ models	→ Application business layer
7	└ views	→ Templates
8	build.sbt	→ Application build script
9	conf	→ Configurations files and other non-compiled resources (on classpath)
10	└ application.conf	→ Main configuration file
11	└ routes	→ Routes definition
12	dist	→ Arbitrary files to be included in your projects distribution
13	public	→ Public assets
14	└ stylesheets	→ CSS files
15	└ javascripts	→ Javascript files
16	└ images	→ Image files
17	project	→ sbt configuration files
18	└ build.properties	→ Marker for sbt project
19	└ plugins.sbt	→ sbt plugins including the declaration for Play itself
20	lib	→ Unmanaged libraries dependencies
21	logs	→ Logs folder
22	└ application.log	→ Default log file
23	target	→ Generated stuff
24	└ resolution-cache	→ Info about dependencies
25	└ scala-2.11	
26	└ api	→ Generated API docs
27	└ classes	→ Compiled class files
28	└ routes	→ Sources generated from routes
29	└ twirl	→ Sources generated from templates
30	└ universal	→ Application packaging
31	└ web	→ Compiled web assets
32	test	→ source folder for unit or functional tests

Figure 3.11: *Play Framework Project Anatomy*

## 3.10 Project Repository Design

Project repository will be hosted on GitHub page: [https://github.com/zubidlo/itb\\_honours\\_project](https://github.com/zubidlo/itb_honours_project). I propose following directory structure:

```
1 /lectures markdown
2   - 1. lecture
3   - 2. lecture
4   ...
5 /poster
6 /thesis
7   /diagrams
8   /figures
9   /tables
10  - thesis.latex
11 /web_app (Play Framework Project)
```

Figure 3.12: *GitHub Repository Anatomy*

## 3.11 Conclusion

In this section I discussed the preliminary design of the project. I argued that the prototyping software development methodology would be the appropriate approach in the development of this project. I outlined the form the learning material should take and I decided to use Scala programming language to communicate functional programming ideas in code.

The web application publishing the learning material for students will be developed using Play! Framework. The application design will follow the standardized anatomy of a Play application and I will use Git as versioning system and GitHub repository to store the project. Play supports both Java and Scala programming languages and I have not yet decided what programming language I will use to develop the application. Also, I listed all the tools I will use in the project implementation.

With the help of UML, I outlined proposed design for web application prototype version 1.0. In this prototype iteration, the application will support user log-in and registration, account settings and log-out. The application will require email validation with expiration date and password should be stored as encrypted hashes. The application will also list all the learning material and reserve a space for an additional functionality.

# Chapter 4

## Implementation

### 4.1 Introduction

In this section, I will discuss some of the web application implementation details. As outlined in chapter 3.9.1 the application will be rendered on the server side and will consist of views, routes, controllers, and models. The views are basically HTML pages returned by the server on user requests. They are all the user is seeing and interacting with.

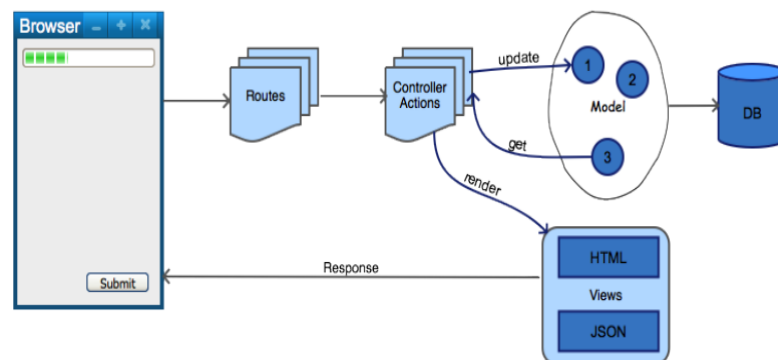


Figure 4.1: *Play Workflow Diagram*

The rest of the Play infrastructure is working in "the background" to make sure that views are returned to user filled with appropriate data. The router routes the requests such as an HTML form POST action to a controller method which will process the form. For example, log-in form POST request will be routed to the appropriate controller method, which will first validate the form. Next, the user data will be fetched from the model and compare to the data in the form. If there is a match, the controller method will render the dashboard view and send it back to the user. Also, the server will maintain the session during the user interaction with the views as long as the user is logged-in.

The application will be developed locally on a laptop and deployed on a live internet server. I chose Microsoft Azure Cloud for the deployment. The reason is that I already have the access to the Azure and I'm familiar with the service management console.

## 4.2 Development Set-Up

The Play Framework is the part of Lightbend Reactive Platform[32]. The platform uses Lightbend Activator User Interface for the development. The user installs the activator locally and starts the service. The user interface is spawned in the browser (figure 4.2) in which user can create new web application or develop, build, run and test existing web application. I used activator and eclipse for the application development.

The new application development doesn't start from scratch, but user rather creates a new application by choosing one of 400 templates. These templates are scaffolds consisting of standardized Play project directory structure and already imported libraries required for chosen type of the web application. Also, the templates



serve the role of tutorials for the beginners. I created the new Play Java application from a template tutorial showcasing the user log-in and I imported additional Javascript libraries for user interface enhancement such as Scala code highlighting.

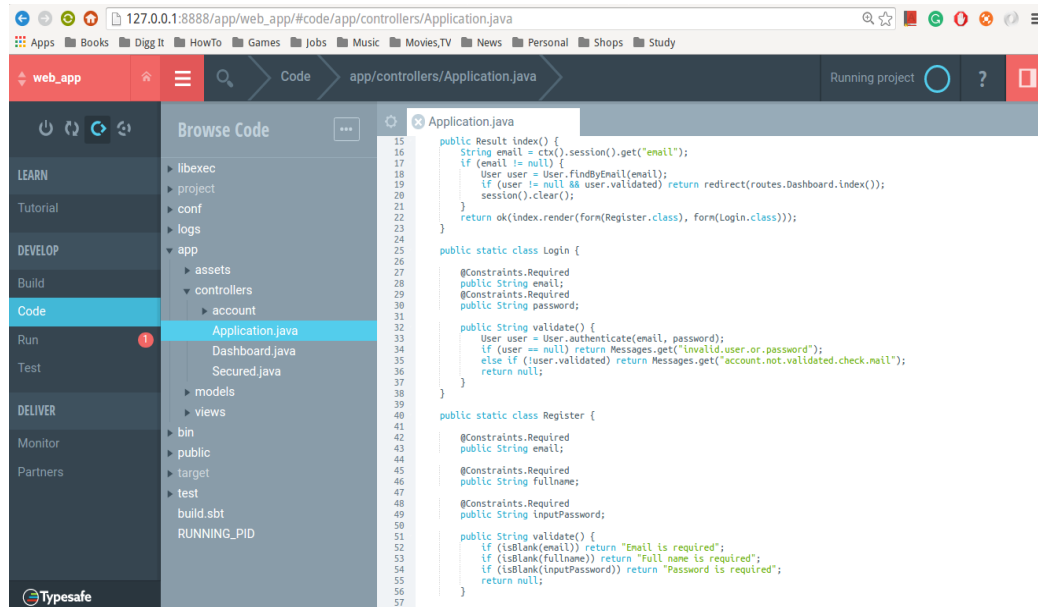


Figure 4.2: *Lightbend Activator*

To start an application development from a template is not specific just to Play Framework, but it is a common practice in most of the web application development platforms. Java EE and .NET offer the same functionality. Because I'm an absolute beginner in Play web application development I found this approach very useful. I choose the template which is in turn extension of <https://www.playframework.com/documentation/2.2.x/JavaGuide1> tutorial.

There is a slight issue with Play approach to backward compatibility. The current Play version I installed is 2.5 and the template and tutorial I followed are working with the version 2.2. Play Framework evolved quite a bit between the versions and there are migration steps need to take place to ensure the application is working. Play documentation includes migration guides, but it is mandatory to follow migration guide from version 2.2 to version 2.3, then from 2.3 to 2.4 and finally from 2.4 to 2.5,

which effectively requires a study of Play Framework evolution.

## 4.3 User Interfaces (Views)

The user interface is contained in package "views". The views are developed with usage of Scala templates. Each view is compiled to the Scala object and the HTML can be then rendered by calling the `render()` method on the given view object. This action is usually done in a controller method as the last step before the rendered HTML is inserted in the response. The view can be arbitrary nested, allowing for creating common layouts (header, footer) just once. The views can carry state-full information as well, such as references to some useful values existing outside the views. This can be used to maintain the session state for example. Every view must have a reference to a user object, which gets created when a user successfully logs in. In the case the user object is not available, a view will redirect to the index page. This way only the logged-in user would get an access to certain views.

### 4.3.1 Prototype 1.0

#### Main Layout View

The main layout view consists of a header (navigation bar) and footer. References to the User and additional HTML `@content` are imported at the top of the view. Every additional view is then inserted on `@content` placeholder. Also, scripts and CSS are imported in this view.

```

layout.scala.html x
1 @ (user: User = null) (content: Html)
2
3 <!DOCTYPE html>
4
5 <html lang="en">
6   <head>
7     <meta charset="utf-8">
8     <meta http-equiv="X-UA-Compatible" content="IE=edge">
9     <meta name="viewport" content="width=device-width, initial-scale=1">
10    <title>@Messages("title")</title>
11    <link rel="shortcut icon" type="image/png" href="@routes.Assets.at("images/favicon.png")">
12    <link rel="stylesheet" href="@routes.Assets.at("stylesheets/bootstrap.min.css")">
13    <link rel="stylesheet" href="@routes.Assets.at("stylesheets/main.css")">
14    <link rel="stylesheet" href="@routes.Assets.at("stylesheets/ie10-viewport-bug-workaround.css")">
15    <link rel="stylesheet" href="@routes.Assets.at("stylesheets/agate.css")">
16    <script src="@routes.Assets.at("javascripts/jquery-2.2.1.min.js")"></script>
17    <script src="@routes.Assets.at("javascripts/bootstrap.min.js")"></script>
18    <script src="@routes.Assets.at("javascripts/highlight.pack.js")"></script>
19    <script src="@routes.Assets.at("javascripts/password.js")"></script>
20    <script>hljs.initHighlightingOnLoad();</script>
21  </head>
22  <body>
23    <header>
24      <nav class="nav navbar-default navbar-fixed-top">
25        <div class="container">
26          <div class="navbar-header">
27            <ul class="nav navbar-nav navbar-right">
28              <li><a href="@routes.Application.Index()">Functional Programming in Scala</a></li>
29            </ul>
30          </div>
31          <div>
32            @logged(user)
33          </div>
34        </div>
35      </nav>
36    </header>
37    <section class="container">
38      <div class="row">
39        @content
40      </div>
41    </section>
42    <footer class="container">
43      <div class="nav navbar-default navbar-fixed-bottom">
44        <div>
45          <!-- <p class="text-center">Created by <strong>Martin Zuber</strong>, student number: <strong>B00066378</strong><p -->
46        </div>
47      </div>
48    </footer>
49  </body>

```

Figure 4.3: *Main Layout View*

## Index View

The index view consists of 2 forms, the login and register form. The view is inserted into main layout view and rendered by the server as the main page of the web application.

```

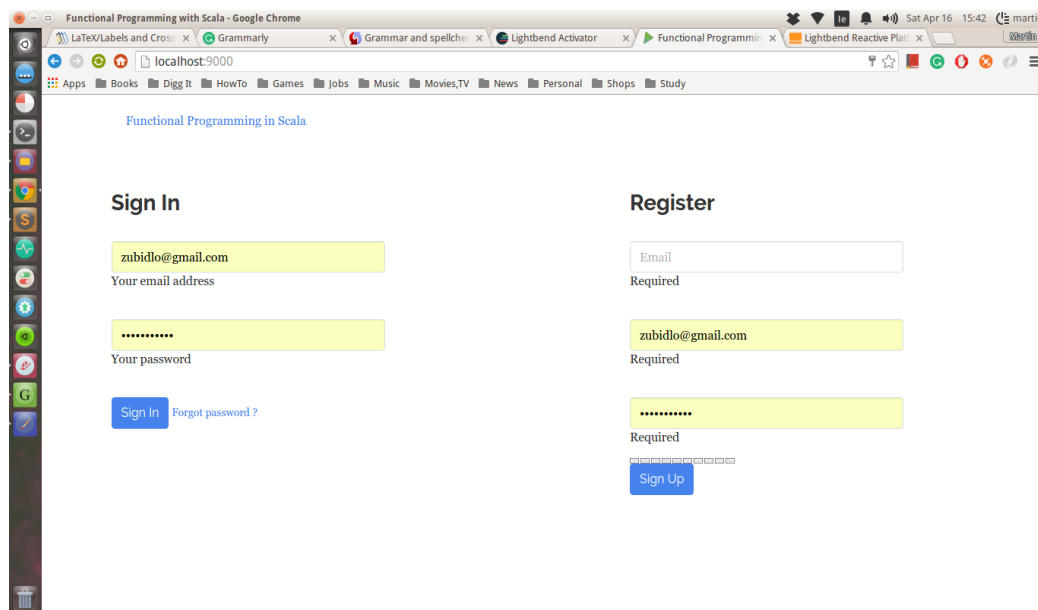
index.scala.html x
1 @ (signupForm: Form[Application.Register], loginForm: Form[Application.Login])
2
3 @layout(null) {
4
5   @wflash()
6
7   <div class="col-lg-4 col-md-4 col-sm-4 col-xs-6">
8     @login(loginForm)
9   </div>
10
11   <div class="col-lg-4 col-md-4 col-sm-4 col-xs-6 col-lg-offset-3 col-md-offset-2 col-sm-offset-1 col-xs-offset-0">
12     @views.html.account.signup.createFormOnly(signupForm)
13   </div>
14 }

```

Figure 4.4: *Index View*

The figure 4.5 is a snapshot of the rendered main page.

All the user interface implementations in Prototype 1.0 are following this pattern. The layout view is a container in which appropriate views are inserted in. This nested view is then rendered into HTML.

Figure 4.5: *Main Page*

## Password Strength

Each form which contains the password input field is enhanced with animated success bar like control under the field. The bar is mimicking the password strength as the user writing the password in the field. This functionality was taken from <http://benjaminsterling.com/wp-content/files/passwordstrength.htm> and consists of `assets/javascript/password.coffee` and `assets/styles/main.less` files. The important fact is that Play Framework compiles *coffee scripts* into *javascript* and *less* into *css* sheets during the compilation.

## Internationalization

Play follows the common I18N scheme to address the internationalization. The developer specifies the languages using language tags, such as "en" for English. These are defined in `conf/application.conf` file by:

```
1 play.i18n.langs = [ "en", "en-US", "fr" ]
```

The text is externalized to text files *conf/messages.en*, *conf/messages.fr* and so on. The message text file consists of simple mappings:

```
1 title=Functional Programming with Scala
2 files.summary=The disk {1} contains {0} file(s).
```

which are referenced from Java by:

```
1 Messages.get("title")
2 Messages.get("files.summary", d.files.length, d.name)
```

The current language is found by looking at the *lang* field in the current Context and can be changed at any time. The Context's *lang* value can be determined by other means:

1. Context's lang field explicitly
2. *PLAY\_LANG* cookie in the request header
3. the *Accept-Language* headers of the request
4. the application's default language

## 4.4 Data And Helper Classes (Models)

The model package consist of Ebean entities and some useful utility helper classes. Ebean framework supports common JPA annotations for entities and simple fluent API for handling database updates.

```
1 FIND.where()
2   .eq("email", email)
3   .findUnique();
```

In this example the chained call will return the user with given address or null. FIND is a reference to model.Finder class which handles User entity.

#### 4.4.1 Prototype 1.0

The model package in this prototype iteration consist of following classes:

- User
- Token
- package utils
  1. AppException
  2. Hash
  3. Mail

##### **User**

The User class is annotated JPA Entity which directly maps to User database table. The class has a set of static methods which allow to find a user by email or by the confirmation token, authenticate the user, change user password and confirm the user.

##### **Token**

The Token is annotated JPA Entity which directly maps to the Token database table. Token functionality allows to create a new token and to find already existing token. The token purpose is to enable email confirmation functionality for registering the new account, resetting the password or changing email of an existing account.

**Token usage in password reset workflow:**

1. User clicks on "Reset" button beside "Password" label in the account settings view. This button is "submitting" empty form with POST action which matches *runPassword()* of *Password* controller method.

```
1  POST    /settings/password    controllers.account.settings.  
    Password.runPassword()
```

2. Method *runPassword()* will check if there is *email* stored in the current session and if so the user with this email address will be fetched from the database, the new genuine token will be created and associated with the user and email with the confirmation link will be sent to the email address.

```
1  @Security.Authenticated(Secured.class)  
2  public Result runPassword() {  
3      User user = User.findByEmail(request().username());  
4      try {  
5          Token t = new Token();  
6          t.sendMailResetPassword(user, mailerClient);  
7          flash("success", Messages.get("resetpassword.mailsent"))  
8      };  
9      return ok(password.render(user));  
10     } catch (MalformedURLException e) {  
11         flash("error", Messages.get("error.technical"));  
12     }  
13     return badRequest(password.render(user));  
14 }
```

**Notice:** Play Framework is binding every request to the Request object. The request object can be retrieved by calling `Controller:: request()` method. The Request object represents an HTTP request with header and body, but it

has an additional *username* field, which can be set at any time. This field serves as an authentication mechanism. In this application design the *@Security.Authenticator* controller injects the *email* address from the current session into the `request.username` field assuming that during an user log-in his email address was stored in the session. This way Play allows me to identify each request with a logged-in user.

3. The user receives the email asking him to click on <http://localhost:9000/reset/61ec0e88-9db6-4103-bf9d-d299ba03f9f8> to reset the password. This link includes the immutable universally unique identifier (UUID) of the created token and matches following route:

```
1 GET    /reset/:token    controllers.account.Reset.reset(token:
    String)
```

When user clicks on the link the `Reset::reset(String token)` controller method will be invoked.

4. The `reset()` method will check if given token exists, if it is of token type used for resetting passwords and if the token is not expired. The tokens expire in one day. If everything is right, the password reset view is rendered and reset form and the token is passed to it.
5. In the reset view the user input his new password in the form and on the form submission the form post matches:

```
1 POST   /reset/:token    controllers.account.Reset.runReset(token:
    String)
```

and `Reset::runReset(String token)` controller method is invoked with token as the argument.

6. The `runReset()` method will again validate the token as before in `reset()` method and if that is successful the user will be fetched from the database based on the token `userId` field and his password will be updated. The confirmation email



will be then sent to user email account and the control will be redirected to dashboard view.

## AppException

The AppException is just a simple exception class, which allows me to create custom exceptions.

## Hash

The Hash is a collection of two static methods which uses `org.mindrot.jbcrypt`[\[35\]](#) library to encrypt and decrypt the passwords.

```
1 package models.utils;
2
3 import org.mindrot.jbcrypt.BCrypt;
4
5 public final class Hash {
6
7     public static String createPassword(String clearString) throws
AppException {
8         if (clearString == null) throw new AppException("No password
defined!");
9         return BCrypt.hashpw(clearString, BCrypt.gensalt());
10    }
11
12    public static boolean checkPassword(String candidate, String
encryptedPassword) {
13        if (candidate == null) return false;
14        if (encryptedPassword == null) return false;
15        return BCrypt.checkpw(candidate, encryptedPassword);
```

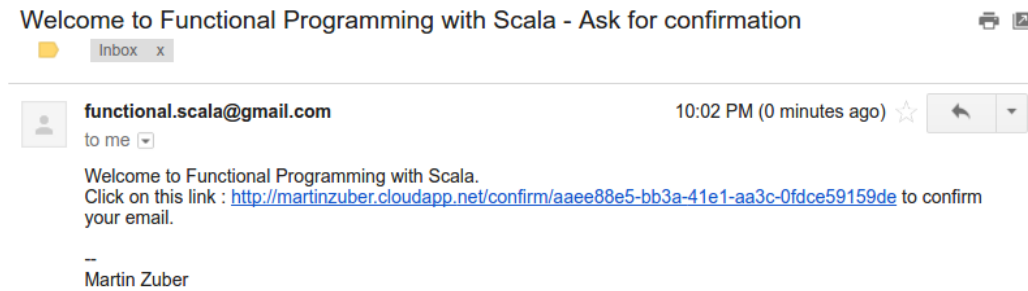
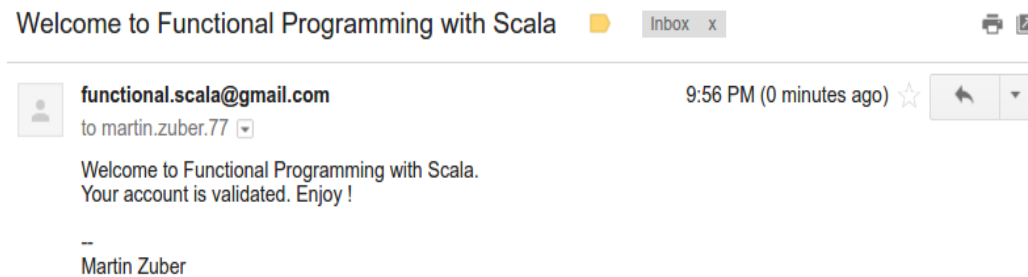
```
16     }  
17 }
```

## Mail

The Mail class is the implementation of email client using Play built-in email support library. The emails are build and send from a separate thread one second after the thread is started. Play includes very powerful support for concurrency in the form of Akka toolkit.[33] Akka is an award winning open source project for building highly concurrent, distributed, and resilient message-driven applications on the JVM based on Scala Actors threading model. Akka is capable of handling 50 million msg/sec on a single machine with a small memory footprint with 2.5 million actors per GB of the heap.

The Mail client uses dedicated email account at [functional.scala@gmail.com](mailto:functional.scala@gmail.com) to send emails to users. The connection is done through SMTP protocol over SSL with and port number 456.

In the case of a user registration processing the Signup:: sendMailAskForConfirmation controller method creates a new random token and assigns it to the new user. The email message is then constructed which includes a link such as [HTTP://localhost:9000/confirm/5ef75942-86ba-4841-a413-70126f9d8e65](http://localhost:9000/confirm/5ef75942-86ba-4841-a413-70126f9d8e65). Clicking on this link will resolve in route to the *controllers.account.Signup.confirm(confirmToken: String)* controller method which will validate the new user account and send the confirmation email.

Figure 4.6: *Validate New Account E-mail*Figure 4.7: *Validate New Account E-mail*

## 4.5 Business Logic (Controllers)

As outlined in section 4.1 the user requests are routed to the controller methods. These methods consist all the business logic of the application. Each method is processing the request and creating the response which usually contains a new HTML page rendered from a view. The mappings from HTTP requests to the controller methods are stored in routes file.

```
1 POST    /login          controllers.Application.authenticate()
```

In the example above is the route from log-in form action to the corresponding controller method. When user triggers the log-in form action (POST) by clicking on Login button, the Application::authenticate method is invoked.

```
1 public Result authenticate() {
2     Form<Login> loginForm = form(Login.class).bindFromRequest();
3     Form<Register> registerForm = form(Register.class);
```

```
4      if (loginForm.hasErrors()) return badRequest(index.render(  
      registerForm, loginForm));  
5  
      else {  
6          session("email", loginForm.get().email);  
7          return redirect(routes.Dashboard.index());  
8      }  
9  }
```

The authenticate method fetches data from the POST request body and bind them to the POJO class representing the form data (email, password). Because the index page has both forms, also an empty registration form is bound to it's POJO representation. The log-in form is then validated against the User model and if it contains errors the index page is rendered and send back to browser inside the `badRequest` response (code 400). From the user point of view, the index page is reloaded with error message included.

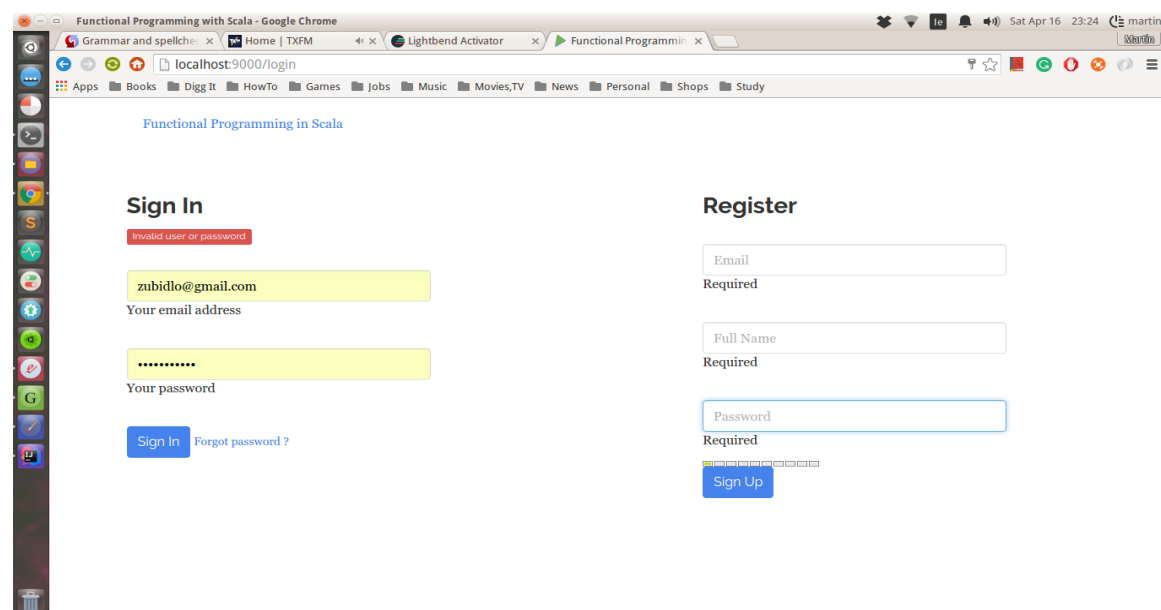


Figure 4.8: *Unsuccessful Log-in Response*

In the case, the user log-in form contained valid email and password, the user's email is saved in the session storage and the user interface is redirected to dashboard

view. The dashboard view can be only accessed as long as the session storage contains an email of a registered user or the session is not expired. Actually, every view which requires logged user is checking the session storage for valid email matching the existing user row in the database. In the case the session is empty the views redirect back to index page.

### 4.5.1 Prototype 1.0

Figure 4.9 is a screenshot of all the routes in prototype 1.0 implementation. Special case is the last line:

```
1 GET /assets/*file controllers.Assets.at (path="/public", file)
```

which handles location of static assets such as CSS styles and script files stored in /public directory of the applications. These files must be handled as Java resources because in a production environment the application will be deployed as **.jar** archive.

Controller classes are implemented in the Java programming language in this prototype iteration. The reason for this decision was purely pragmatic. Being Scala and Play beginner, I planned to learn the Play Framework in the first prototype implementation. In later iterations, I have planned to refactor all the Java code to Scala as soon as I would become comfortable enough with the new language. Scala is often referred to as the most evolved programming language on the planet and I soon found out that Play uses advanced Scala features such as *implicit*.

There has been enough complexity involved just with a study of Play Framework alone. After all, the main point of prototyping approach to a software development is to be able to have a working prototype as soon as possible.

```

# Routes
# This file defines all application routes (Higher priority routes first)

GET    /                controllers.Application.index()
GET    /dashboard      controllers.Dashboard.index()
GET    /lecture/:file   controllers.Dashboard.lecture(file: String)

POST   /login           controllers.Application.authenticate()
GET    /logout         controllers.Application.logout()

GET    /settings       controllers.account.settings.Index.index()
GET    /settings/password controllers.account.settings.Password.index()
POST   /settings/password controllers.account.settings.Password.runPassword()
GET    /settings/email  controllers.account.settings.Email.index()
POST   /settings/email  controllers.account.settings.Email.runEmail()

GET    /signup        controllers.account.Signup.create()
POST   /signup        controllers.account.Signup.save()

GET    /confirm/:confirmToken controllers.account.Signup.confirm(confirmToken: String)

GET    /reset/ask      controllers.account.Reset.ask()
POST   /reset/ask      controllers.account.Reset.runAsk()

GET    /reset/:token   controllers.account.Reset.reset(token: String)
POST   /reset/:token   controllers.account.Reset.runReset(token: String)

GET    /email/:token   controllers.account.settings.Email.validateEmail(token: String)

GET    /assets/*file    controllers.Assets.at(path="/public", file)

```

Figure 4.9: *Routes*

## 4.6 Deployment

The application is deployed on Azure Cloud Virtual Machine <http://martinzuber.cloudapp.net/>. Play Framework doesn't require an application server for deployment. Play includes its own HTTP server and is deployed as the standalone application. Activator creates distribution version.zip archive which contains the startup script. This archive can be then copied to live server, unpacked and run.

### 4.6.1 Required Steps for Deployment Set-Up

#### 1. Create a virtual machine.

Azure cloud offers classic virtual machines. The user chooses data centre location, a number of processor cores, memory size, disk size, and bandwidth. The selection

will dictate the price per month. Virtual machine specifications can be adjusted dynamically later if the requirements change. For example, the machine can be migrated to a different geographical location, cores, memory can be added and so on. Microsoft offers a wide range of pre-configured operating system images to be installed automatically during the machine creation. The user can choose from quite a rich selection of Linux distribution images and specialized appliances are available as well. All images come with configured SSH server.

#### Deployment Virtual Machine Specifications:

- Type: Basic A1 (1 Core, 1.75 GB memory)
- OS: Ubuntu Server 14.04 LTS
- Location: Europe North (Dublin West Data Centre)
- DNS name: martinzuber.cloudapp.net
- Public Virtual IP: 40.85.129.158

## 2. Install and configure all required software

To be able to deploy Play web application I need to install following software:

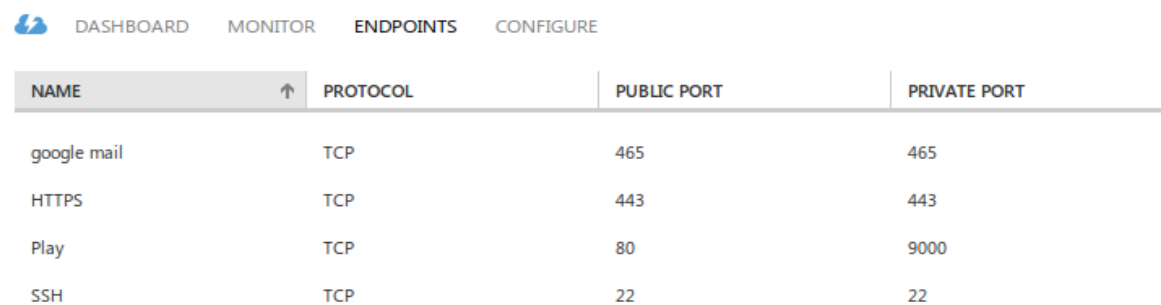
- Java SE Platform
- MySQL database All the Play distribution build needs is installed Java SE Platform. Netty server and all the dependencies are included in the build. The H2 in-memory database will be replaced by MySQL in production when the application is fully developed.

### 3. Configure Endpoints.

Endpoints are mappings from Azure internet facing ports to internal ports on the virtual machine. For example, to be able to connect to the machine with SSH protocol, the following endpoint must be created:

```
1 name: SSH, protocol: TCP, public port: 22, private port: 22
```

This endpoint would make virtual machine port 22 available from the outside the cloud. I configured the following endpoints:



The screenshot shows the Azure portal interface with the 'ENDPOINTS' tab selected. Below the navigation bar, there is a table listing configured endpoints. The table has four columns: NAME, PROTOCOL, PUBLIC PORT, and PRIVATE PORT. The rows represent different services: google mail, HTTPS, Play, and SSH.

NAME	PROTOCOL	PUBLIC PORT	PRIVATE PORT
google mail	TCP	465	465
HTTPS	TCP	443	443
Play	TCP	80	9000
SSH	TCP	22	22

Figure 4.10: *Virtual Machine Endpoints*

### 4. Create and configure web app distribution build.

Even though to create the production build in Play is a question of one activator command, the necessary changes have to be made in the production build configuration:

1. The web application is sending emails consisting of a URL to validate user actions such as account registration as I discussed in the section 4.4.1. When the application is deployed its domain name changes and so must the URL in those emails. The domain name is therefore not hard-coded, but rather stored in the `conf/application.conf` file and referenced from the code by name.

```
1 server.hostname="martinzuber.cloudapp.net"
```



2. Java Ebean ORM supports the database evolutions. This functionality allows having different database schema for each prototype or build. The schemas are stored in *conf/evolutions/* directory and with each restart of the application, the developer is asked to choose an appropriate schema. This functionality is not allowed in the production mode and the database must be configured with the default schema at application start-up:

```
1 /path/to/web_app/app -Dplay.evolutions.db.default.autoApply=true
```

or turned off in *conf/application.conf* file:

```
1 evolutions=disabled
```

3. Application in production mode requires a private secret key. The key is used for signing session cookies, CSRF tokens and in built-in encryption utilities. Anyone with the access to the secret key can generate any session they please, effectively allowing them to log into the app. Therefore is highly recommended to store the key only on the production server and not to check the secret to source control. Also, if there are more instances of the application deployed the secret key must be the same for each instance. The secret key can be created in activator console with:

```
1 playGenerateSecret
```

and stored in *conf/application.conf* by:

```
1 play.crypto.secret="</N_v'W4xv8GY^c7e=UCDYF]vXSguUvABIHoi[
  JNm1j@lf2C6S9TL[?FNM?ImDd; "
```

4. In the development mode, the Play application uses H2 in-memory database, which only store data for the application runtime and schemes are deleted with each restart. The H2 driver configured in *conf/application.conf* by:

```
1 db.default.driver=org.h2.Driver
2 db.default.url="jdbc:h2:mem:play "
```

In the production mode, MySQL server could be used instead so the data are truly persistent.

```
1 db.default.driver=com.mysql.jdbc.Driver
2 db.default.url="jdbc:mysql://localhost/playdb"
3 db.default.username=playdbuser
4 db.default.password="a strong password"
```

5. As seen in steps 1-4, the production build differs from development in configuration of multiple items. For these cases Play provides an option to specify different configuration file for the production environment in the form of a command line argument:

```
1 /path/to/web_app/app -Dconfig.file=/full/path/to/conf/application
   -prod.conf
```

## 5. Create Upstart service.

In Ubuntu 14.04, Upstart is a replacement for traditional *init daemon* and it is responsible for bringing the computer into a normal running state after the boot-up. Upstart handles the starting, stopping and monitoring processes while the operating system is running. For each process to be managed by the Upstart daemon the `.conf` script must be created and stored in `/etc/init/` directory. Processes defined by these scripts are called services.

I wrote following Upstart script `/etc/init/play.conf` which turned the web app into a service daemon. The service is started automatically at server startup as soon as the file system and network interface are alive. Also, running Play application stores its process id (PID) in a file which has to be deleted before the application can be restarted.

```
1 description "My Play Application"
```

```
2
3  env USER=martin
4  env GROUP=martin
5  env APP_HOME=/home/martin/web_app-1.0-SNAPSHOT
6  env APP_NAME=web_app
7
8  env EXTRA=" "
9
10 start on (filesystem and net-device-up IFACE=eth0)
11 stop on runlevel [!2345]
12
13 respawn
14 respawn limit 30 10
15 umask 022
16 expect daemon
17
18 pre-start script
19     #If improper shutdown and the PID file is left on disk delete it
20     #so we can start again
21     if [ -f $APP_HOME/RUNNING_PID ] && ! ps -p $(cat $APP_HOME/
22     RUNNING_PID) > /dev/null ; then
23         rm $APP_HOME/RUNNING_PID ;
24     fi
25 end script
26
27 exec start-stop-daemon --pidfile ${APP_HOME}/RUNNING_PID --chdir ${
28 APP_HOME} --chuid $USER:$GROUP --exec ${APP_HOME}/bin/${APP_NAME} --
29 background --start -- -Dplay.evolutions.db.default.autoApply=true
30 $EXTRA
```

The Upstart script syntax validity can be checked with:

```
1 $ init-checkconf /etc/init/play.conf
```

The Play web application can be managed from terminal with commands:

```
1 $ service start | stop | restart | status
```

## 4.7 Conclusion

In implementation chapter, I started with the description of a general structure of every Play application and how the pipeline from user interface toward the persistence storage is designed in the framework. In the comparison with multi-layered Java EE platform, the Lightbend Reactive Platform feels lightweight but more powerful.

Scala Twirl templates are closer to .NET Razor templates than to Java Server Faces. The templates are compiled to Scala objects and therefore are type-safe with the support of all the language features. On the other hand, they should handle only simple tasks and all the business logic is placed in the controller actions.

The requests coming from views are routed to controller methods which return responses usually in a form of another view. Play follows Model-View-Controller architectural pattern, but with REST-like resource oriented design. This approach feels very natural and logical. The framework is persistence agnostic and comes with default EBeans ORM mapper and various database drivers.

With each change in the code, the web application is recompiled and if compilation or runtime errors are printed in the browser window in development settings. Everything in Play application is therefore statically typed code. Play supports coffee script and less compilation as well.

The last section of the implementation chapter was dedicated to the application deployment in a lot of detail. The reason for such a detailed description was to

document the process for a future reference. Overall, I had a quite positive experience using Azure cloud services.

# Chapter 5

## Testing and Evaluation

### 5.1 Introduction

Testing is an imperative part of any software development project. Eradicating errors and bugs is vital to the user experience. The successful test on one which identifies the bug in the application. There are various testing strategies to be performed:

- Unit testing – testing of a single function, procedure. The test subject is treated as a "black box" meaning that we test the subject functionality not the underlying implementation. Integration testing – checks that units tested in isolation work properly when put together.
- System testing – the emphasis is to ensure that the whole system can cope with real data, monitor system performance, test the system's error handling and recovery.
- Regression Testing – checks that the system preserves its functionality after maintenance and/or evolution tasks.

- Acceptance Testing – checks if the overall system is functioning as required. It is usually performed by the end user on working application prototype.

The Play Framework comes with automated testing capabilities powered by the sbt build tool. Test classes are to be located in `/test` directory and every method annotated with `@Test` will be included in the test run. The test can be run with "`$ activator test`" command or performed in the activator user interface.

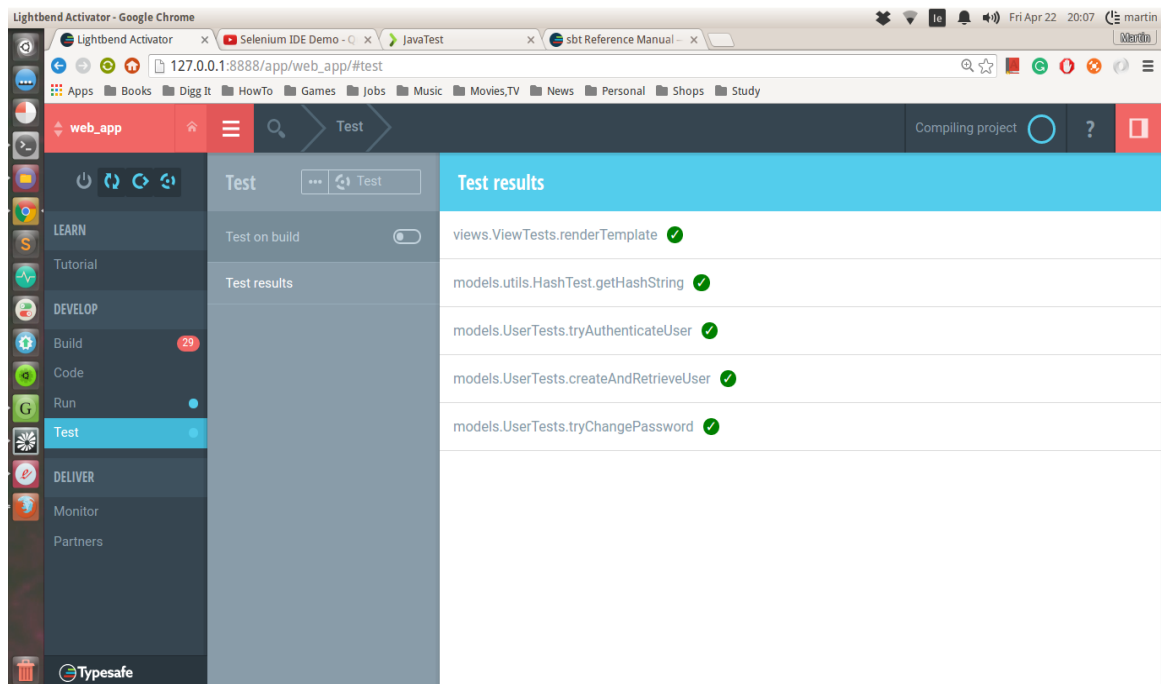


Figure 5.1: *Activator test run*

The Play Framework supports various test frameworks out of the box with the additional functionality of so called test helpers. Test helpers help to create mock instances of various Play components such as application, router or controller to make possible to test a component in isolation. Play comes with JUnit, WebDriver, ScalaTest support, but any testing framework can be used if needed.

In my test suite, I will concentrate to unit test the application models and utilities with JUnit test tool and test the user interface with Selenium IDE. In the following

sections I will outline my approach to testing and discuss every found bug.

## 5.2 Unit Tests of Models and Utilities

Test are stored in */test/models* directory of the web application project. All test for an unit will be contained in one class.

### 5.2.1 Prototype 1.0

#### Hash

```
1 @Test
2 public void getHashString() throws AppException {
3
4     String password = Hash.createPassword("hello world");
5     Assert.assertNotNull(password);
6
7     boolean matches = Hash.checkPassword("hello world", password);
8     Assert.assertTrue(matches);
9
10    boolean badPassword = Hash.checkPassword("some other password",
11        password);
12    Assert.assertFalse(badPassword);
13 }
```

#### User

```
1 public class UserTests extends WithApplication {
2     private String token;
3     private User user;
```



```
4      @Before
5      public void setUp() throws AppException {
6          start(fakeApplication(inMemoryDatabase()));
7          String hash = Hash.createPassword("password1");
8          token = UUID.randomUUID().toString();
9          user = new User("martin@gmail.com", "Martin Zuber", hash, token);
10         user.save();
11     }
12
13     @Test
14     public void createAndRetrieveUser() {
15         User martin = User.findByEmail("martin@gmail.com");
16         assertNotNull(martin);
17         assertEquals("Martin Zuber", martin.fullname);
18
19         martin = User.findByConfirmationToken(token);
20         assertNotNull(martin);
21         assertEquals("Martin Zuber", martin.fullname);
22     }
23
24     @Test
25     public void tryAuthenticateUser() {
26         assertNotNull(User.authenticate("martin@gmail.com", "password1"));
27
28         assertNull(User.authenticate("bob@gmail.com", "badpassword"));
29     }
30
31     @Test
32     public void tryChangePassword() throws AppException {
33         user.changePassword("password2");
34         User martin = User.findByEmail("martin@gmail.com");
35         assertNotNull(martin);
```

```
35     assertNotNull(User.authenticate("martin@gmail.com", "password2"));
36 }
37 }
```

In User tests I tested all the User model methods with the mock application and a database with one user created in `UserTests::setUp()` method annotated with *@Before*. All the tests passed.

## Token

```
1 private String token, token1;
2 private User user;
3 private String email;
4
5 @Before public void setUp() throws AppException {
6     start(fakeApplication(inMemoryDatabase()));
7     String hash = Hash.createPassword("password1");
8     token = UUID.randomUUID().toString();
9     token1 = UUID.randomUUID().toString();
10    email = "martin@gmail.com";
11    user = new User(email, "Martin Zuber", hash, token);
12    user.save();
13    new Token(token, user.id, Token.TypeToken.password, email).save();
14    new Token(token1, user.id, Token.TypeToken.email, email).save();
15 }
```

In listing above the `Token::setUp()` method creates the user, password reset token and email change token.

```
1 @Test public void tryIsExpired() {
2     Token t1 = Token.findByTokenAndType(token, Token.TypeToken.password);
3     assertNotNull(t1);
4     assertNotNull(t1.dateCreation);
}
```

5 }

The test above was successful and revealed the bug: **dateCreation is never assigned a value**. The User model test was then adjusted to test if *dateCreated* is not null on a user fetched from database and the same bug was found. The rest of the tests passed.

### 5.3 Acceptance Tests (Selenium IDE tests)

The Selenium allows controlling a web browser actions (simulated user input) in a various way. The developer can use WebDriver API to control the browser from code, or to use Selenium IDE Firefox plugin. Selenium IDE is an automated web testing tool that allows the user to navigate and interact with a website and have Selenium track those navigations and changes. You can then choose to let Selenium re-run these while looking for any inconsistencies or errors. The tests are stored in HTML files where action steps are stored in a table.

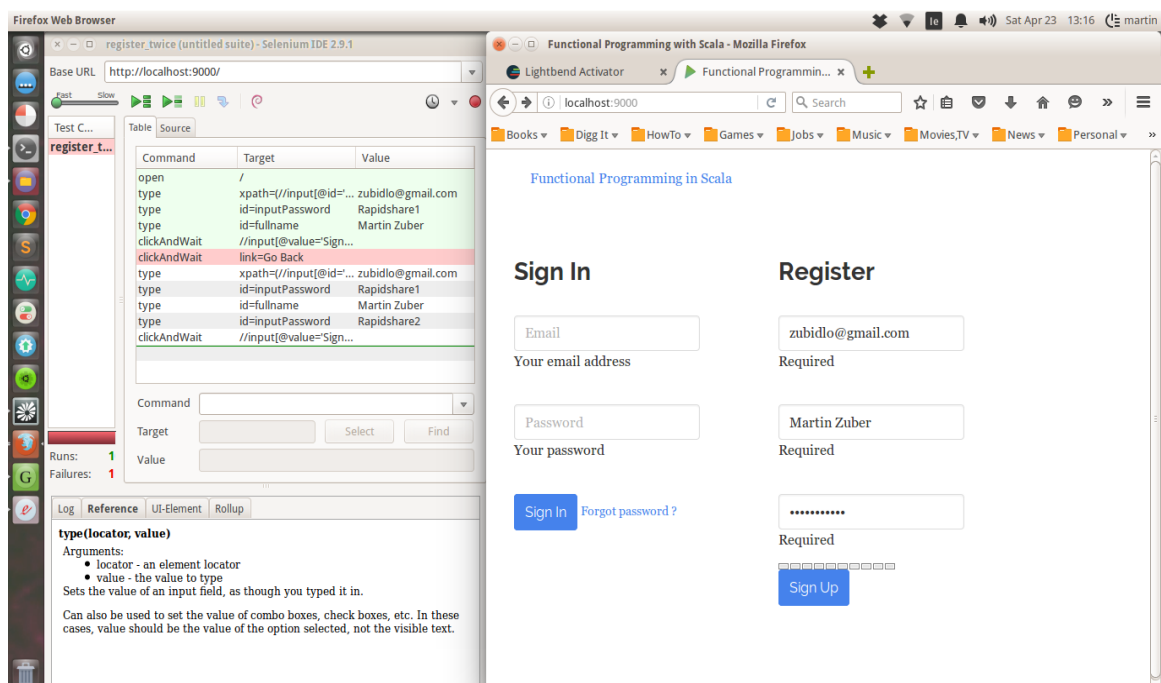


Figure 5.2: *Selenium IDE Firefox Plugin*

All the Selenium tests are stored in the `/web_app/test/selenium_firefox/` directory.

### 5.3.1 Prototype 1.0

#### Register Account Twice Test

User clicks register. The page will redirect to the account registered page which will inform the user that he will receive an email to confirm the registration. User clicks on *"GO BACK"* button. The user is then redirected back to index page. The user then repeats the registration process with the same email supplied and clicks on the registration button.

register_twice		
open	/	
type	xpath=("//input[@id='email'])[2]	zubidlo@gmail.com
type	id=inputPassword	Rapidshare1
type	id=fullname	Martin Zuber
clickAndWait	//input[@value='Sign Up']	
clickAndWait	link=Go Back	
type	xpath=("//input[@id='email'])[2]	zubidlo@gmail.com
type	id=inputPassword	Rapidshare1
type	id=fullname	Martin Zuber
type	id=inputPassword	Rapidshare2
clickAndWait	//input[@value='Sign Up']	

Figure 5.3: *Register Account Twice Actions*

**Bug:** The page will redirect without any explanation of just a registration form again and will not register any additional account.

**Suggested Fix:** The application should print a warning message that the account with this email address already exists and redirect to the index page.

### Dashboard or any of the lecture request test

**Bug:** If the dashboard page or any of the lecture pages are requested without a user login the pages are delivered which is not desired behaviour.

**Suggested Fix:** Check if dashboard checks if user email is in the session.

## 5.4 Conclusion

In the test phase I applied following test strategies:

1. Unit tests of the models.
2. User Interface acceptance tests.

Even though Play application supports the test of the whole application stack, I left out the testing of view rendering and controllers. The reason was that I could not make supplied test helpers work. Tutorials referring to the Play 2.2 version does not reflect the 2.5 version of the testing framework.

I found various bugs in the application, especially in the user interface which have inconsistent behaviour, not in compliance with the application specifications. After each bug fix, the regression tests were performed not the re-factored code.

In conclusion, the Play Framework has rich support for testing the web applications with some migration issues.

# Chapter 6

## Functional Programming with Scala

### 6.1 Introduction

As a Computer Science student at ITB, I was introduced to computer programming through Java programming language. First, we learned the syntax of the language. Next, we have been introduced to the object-oriented approach to designing software and how to work with libraries such as Swing. Later in the third year, we learned the basics of concurrency and some design patterns for solving reoccurring problems.

Around this time I crossed the initial steep part of the programming learning curve and I was comfortable enough with Java to learn some additional language features and Java Standard Edition platform libraries. Programming with Java became my hobby and I found a time to read additional material such as *Thinking in Java* from Bruce Eckel or *Effective Java* from Joshua Bloch.

At the same time, Oracle introduced important update 8 to Java Standard Edition platform which, among other things introduced the syntax for anonymous functions (lambdas) and a number of libraries dedicated to functional programming style. I was interested in this new functionality and started to learn to implement it into my code. By playing with *lambdas*, *streams*, and *optionals* I got really curious about the functional programming and wanted to learn more.

Any student who tries to learn functional programming within JVM ecosystem will very soon become aware of *Scala* programming language. The reason is that Scala has more richer support for functional programming than Java and hence is better suited as a vehicle for the study.

The following material is an overview of what I learned about functional programming in the period of 2 semesters. I uses Scala syntax to communicate ideas with code.

## 6.2 Why Would You Read These?

```
1 In the last section , I tried to articulate my interest in functional
   programming and my reasons for working on this project .
2
3 In this section , I will try to give you a reader some reasons to
   continue to read these pages .
```

In my experience, writing software is fun. For a while at least.

Before I write a software, I think about the design. I would think how to represent data and how to manipulate them. Then I start to write the code and I'm excited. It's fun. I'm transforming my ideas into a working program!

Very soon, though, I start to notice that not everything works well. I must handle exceptions, null checks and to deal with another problems not related to my design ideas. I find myself solving problems related to underlying computer and programming language.

As the software grows it starts to resemble a monster. My beautiful design is scattered around null checks, buried inside nested try-catch blocks. Every time I add something new, the program breaks in five places. Everything seems to be connected, dependent. Complexity grows exponentially and my capability to reason about the program degrades. It's just so many things to keep in mind. It is not fun any more! It's rather hard and mundane.

At that point, I'm ready to give up.

Well, in imperative programming languages, the methods are too powerful with no constraints. They can do whatever they want. They can change objects in place, write to a file, put something on a screen. They are liars too. For example, consider some 'write(file)' method. She says she will write to a file. But instead she could throw 4 different exceptions. There are methods which say they do something and return a result. But they can do something else in the process and hide that fact from the programmer.

*Metaphorically it is similar to a situation when one gets a powerful car which allows him to drive anywhere. The car can drive through fields for example. One could try to reach a destination by driving straight line path. It looks like a great idea in the*



*beginning until one ends up in a ditch with no way out.*

### 6.2.1 Is there an alternative?

Functional Programming of course :-) !

Functional programming is the programming with functions, not methods. Functions are much less powerful. They follow constraints. In exchange though, they have various very useful properties which make reasoning about them straight forward. Functions do what they say they do and never lie. They don't throw exceptions or return nulls.

Functions are associative, pluggable and composable, the same way the 'Lego' blocks are. One can build a software by composing simple functions the same way as one would build a house from Lego blocks. The software complexity doesn't grow exponentially neither. Instead, a program is just a function composed of another functions. The capability to reason about the software stays constant.

Also, the solution space is smaller. One just need to learn a few ideas and how to compose them together. There are design patterns in functional programming of course, but there is not 40 of them.

*It is like to have a car capable of driving on the roads only. One has much less power at the beginning, but one will safely reach the destination following the roads.*

These methaphores were quoted from *Functional Programming is Terrible* talk presented by Runar Bjarnason.

## 6.3 What is Functional Programming?

```
1 In the last section , I made a suggestion that there is a way to have fun
   while creating a software. I claimed that functional programming
   gives us a better way to design a software and reason about it .
2
3 In the following sections , I will discuss the functional programming
   definition in comparison to imperative programming paradigm .
```

### 6.3.1 Imperative Programming Paradigm

Imperative programming uses statements that change the program's state. We construct programs with programming language commands and which describe how the program operates. Our programs basically tell the compiler (interpreter) how to accomplish desired results step by step.

With the addition to statements, we use subroutines call procedures or functions to modularize imperative programs. Imperative programming with procedures is called *procedural programming*. Procedures have no properties or restrictions defined, for example, they may or may not take arguments and may or may not return a result value. They may modify the state of the program or crash with an error.

What imperative programs do:

1. They modify mutable variables with assignments.
2. They use control structures such as loop, break, continue, if-else statements.
3. They use procedures with no restrictions on their properties.

#### Example:

```
1 var numbers = List(1,2,3,4,5)
2 var sum = 0
3 do {
```

```
4     sum += numbers.head
5     numbers = numbers.tail
6 } while (numbers != Nil)
7
8 //Result:
9 numbers: List[Int] = List()
10 sum: Int = 15
```

In the example above the program calculates the sum of numbers from 1 to 5. Variables *numbers* and *sum* represent the mutable state which is modified in the *do while loop* control structure. After the program executes the list *numbers* is empty and *sum* is 15.

### 6.3.2 Functional Programming Paradigm

Functional programming is a programming style where the fundamental application is the application of pure functions to arguments. Functional programs are composed of functions.

What functional programs do:

1. They transform immutable values to other values.
2. They use functions and function compositions as control structures.
3. They use pure referentially transparent functions.

#### Example:

```
1 var numbers = List(1,2,3,4,5)
2
3 def sumOfNumbers(numbers: List[Int]): Int = numbers match {
```

```
4  case Nil => 0
5  case head :: tail => head + sumOfNumbers(tail)
6  }
7
8  val sum = sumOfNumbers(numbers)
9
10 //Result:
11 numbers: List[Int] = List(1, 2, 3, 4, 5)
12 sumOfNumbers: (numbers: List[Int]) Int
13 sum: Int = 15
```

In the example above the program calculates the sum of numbers from 1 to 5 with the usage of pure function *sumOfNumbers*. This is a function which takes a *List[Int]* and returns *Int*. The *sumOfNumbers* is defined in terms of its argument *List[Int]* as follows:

- If the function receives empty list ‘Nil’ it returns 0.
- If the function receives a list of one or more elements it adds the first *head* to the recursive call to itself with the new list *tail*. The *tail* just original list without *head*. Note that *head :: Nil* would match list with one.

The function application *sumOfNumbers(numbers)* will in effect build following call stack:

```
1 (1 + sumOfNumbers(List(2,3,4,5)))
2 (1 + (2 + sumOfNumbers(List(3,4,5))))
3 (1 + (2 + (3 + sumOfNumbers(List(4,5)))))
4 (1 + (2 + (3 + sumOfNumbers(List(4,5)))))
5 (1 + (2 + (3 + (4 + sumOfNumbers(List(5)))))
6 (1 + (2 + (3 + (4 + (5 + sumOfNumbers(Nil)))))
7 (1 + (2 + (3 + (4 + (5 + 0)))))
```

And when it folds it returns 15. Notice that we did not define any mutable variables anywhere in the program. Everything is immutable value *val* and cannot be changed. All we do is to transform a list of numbers to its sum without changing anything.

## 6.4 Functions Everywhere

```
1 In the last section , I compared the idioms of imperative and functional
  programming with simple examples. I stated that the building blocks
  of functional programs are pure functions.
2
3 In the following section , I will discuss functions and their
  characteristics in more detail.
```

### 6.4.1 Pure Function

- The function **always** evaluates to the same result given the same argument. The word **always** is emphasized here. The pure function can return the same result for different arguments, but it can never return different results for the same argument.
- Evaluation of the result does not cause any semantically observable **\*\*side effect\*\***. Well, what are these side effects then? In Computer Science, a function or expression is said to have a side effect if it modifies some state or has an interaction with calling functions or the outside world. Basically, a function has a side effect if it does something else than returning a result. Some examples of side effects:
  1. Modifying a variable or a data structure in place
  2. Throwing an exception or halting with an error

3. Writing to or reading from the console or a file
4. Drawing on the screen, updating a database, printing on a printer

Every function with the characteristics of a pure function is called **referentially transparent** and allow us to replace the expression in code which applies one of these functions with its result value without affecting the behaviour of the program in any way.

#### Example: Pure vs Impure

```
1 sin(x)
2 length(list)
3 random(seed)
4 printf(string)
```

Functions *sin* and *length* are pure functions because they always return the same result for the same argument and do nothing else. Function *sin(0)* returns always 0 and *length(List(1,2,3,4,5))* of the same list is always 5. In other words function application *sin(0)* can be replaced with value 0 and function application *length(List(1,2,3,4,5))* with value 5.

Functions *random(seed)* and *printf(string)* are impure functions because *random(1)* will not always return the same result. It will return a random number from 0 to 1. Function *printf("hello world")* will print the string literal to the console and return no result. Even that is not guaranteed. Sometimes it can throw an exception if the console is not available for example. Impure functions are called **procedures** in computer science.

#### Example: Function definition in Scala

```
1 def add(a: Int, b: Int): Int = a + b
```

The function *add* is defined as a function which takes two integers and returns one integer. The *add* function signature is  $add = (Int, Int) \Rightarrow Int$ . We can think of functions as mappings from input to output. In this case *add* is a mapping from  $a, b$  to  $a + b$ . We can express this fact with the following statement:

```
1 val add: (Int, Int) => Int = (a, b) => a + b
```

Because the Scala compiler supports type inference we can express the function with shorter statement:

```
1 val add = (a: Int, b: Int) => a + b
```

The compiler will *infer* that *add* can only return *Int* value. In a specific cases we will be able to omit the arguments types as well and just write:  $add = (a, b) \Rightarrow a + b$ . This form of function description is called a *lambda expression*. It is basically an anonymous function, a function without name. We can read  $val\ add = (a: Int, b: Int) \Rightarrow a + b$  as: *Immutable value add is anonymous function which takes a,b and returns a + b.*

### Notice:

```
1 In Scala and functional programming languages in general, functions are
  values. The same way the objects are values. We can assign functions
  to variables, pass them as arguments to other functions, create
  them inside the functions and return them as results of function
  computations.
```

## 6.4.2 General Terms

Consider a function:

```
1 val intToString: Int => String = i => i.toString
```

```
2 val s = intToString(123)
3
4 //Result:
5 intToString: Int => String = <function1>
6 s: String = 123
```

The immutable value *intToString* is a function which takes an *Int* and returns *String* such as it applies *toString* function to the given *Int*. The immutable *String* value *s* is the result of *intToString* function application to arguments *123*.

1. The *Int* and *String* are **types** and represent sets of values.
2. The type *Int* is a **domain** of *intToString* function.
3. The type *String* is a **codomain** of *intToString* function.
4. The *intToString* is **total function** because it is defined for every value from its domain. In other words *intToString* will return a value for any possible value of type *Int*

Consider:

```
1 val stringToInt: String => Int = s => s.toInt
2 val i = stringToInt("123")
3
4 //Result:
5 stringToInt: String => Int = <function1>
6 i: Int = 123
```

In the contrast the *stringToInt* is a **partial function** because is not defined for every value from its domain, for example application *stringToInt("abc")* would crush with *java.lang.NumberFormatException* exception.



### 6.4.3 Functions as table lookups

As we discuss above pure functions are mappings from domain to codomain. We can always express functions in a form of a table look up. For example, we can define *stringToInt* as follows:

```
1 def stringToInt(s: String): Int = s match {  
2   case "1" => 1  
3   case "2" => 2  
4   case "3" => 3  
5   case s => ???  
6 }  
7  
8 val i = stringToInt("1")  
9  
10 //Result:  
11 stringToInt: (s: String)Int  
12 i: Int = 1
```

The *stringToInt* is only defined for domain: "1", "2", "3" and for any other value it would throw *scala.NotImplementedError: an implementation is missing* error.

#### Notice:

```
1 Any function can be defined as a table lookup. We can think of common  
  data structures as partial functions as well. \emph{Map ("1" => 1,  
  "2" => 2, "3" => 3)} is the same partial function as 'stringToInt '  
  and in Scala every Map inherits the 'PartialFunction' trait. Every  
  Scala Map is a function from its keys to its values. If fact, Many  
  Scala data structures are partial functions.
```

# Chapter 7

## Conclusion and Further Work

### 7.1 Introduction

//TODO

### 7.2 Achievements

//TODO

### 7.3 Personal Gain

//TODO

## 7.4 Further Work

//TODO

## 7.5 Conclusion

//TODO

# Bibliography

- [1] Neil Ford, *Functional thinking: Why functional programming is on the rise*, <http://www.ibm.com/developerworks/library/j-ft20/>, (2013-01-25)
- [2] Typesafe, *Scala, Object-Oriented Meets Functional*, <http://www.scala-lang.org/>, (2002-2015)
- [3] Typesafe, *Scala in the Enterprise*, <http://www.scala-lang.org/old/node/1658>, (2012-01-19)
- [4] Brikman, Yevgeniy, *The Play Framework at LinkedIn: Productivity and Performance at Scale*, [https://www.youtube.com/watch?v=8z3h4Uv9YbE&ab\\_channel=NewCircleTraining](https://www.youtube.com/watch?v=8z3h4Uv9YbE&ab_channel=NewCircleTraining), (2013-06-26)
- [5] Wampler, Dean, *We Won! How Scala Conquered the Big Data World.*, [https://www.youtube.com/watch?v=AHB6aJyhDSQ&ab\\_channel=NewCircleTraining](https://www.youtube.com/watch?v=AHB6aJyhDSQ&ab_channel=NewCircleTraining), (2015-03-01)
- [6] theotown, *Akka wins 2015 JAX Award for Most Innovative Open Source Technology*, <https://www.typesafe.com/blog/akka-wins-2015-jax-award-for-most-innovative-open-technology>, (2015-03-23)
- [7] Wikipedia, *Free software programmed in Scala*, [https://en.wikipedia.org/wiki/Category:Free\\_software\\_programmed\\_in\\_Scala](https://en.wikipedia.org/wiki/Category:Free_software_programmed_in_Scala), (2015-01-05)
- [8] Wikipedia, *Play Framework*, [https://en.wikipedia.org/wiki/Play\\_framework](https://en.wikipedia.org/wiki/Play_framework), (2015-10-04)
- [9] Paul Chiusano, Runar Bjarnason, *Functional Programming in Scala.*, Shelter Island, NY 11964, Manning Publications Co., 2015. ISBN 9781617290657.
- [10] Dean Wampler, Alex Payne, *Programming Scala, Second Edition.*, 1005 Gravenstein Highway North, Sebastopol, CA 95472, O'Reilly Media, Inc., 2015. ISBN: 978-1-491-94985-6.
- [11] Peter Hilton, Erik Bakker, Francisco Canedo, *Play For Scala.*, Shelter Island, NY 11964, Manning Publications Co., 2014. ISBN 9781617290794.

- [12] Hudak, P., *The Conception, Evolution, and Application of Functional Programming Languages.*, s.l.:Yale University Department of Computer Science, 1989
- [13] Hristakeva, M., Vuppala, R., *A Survey of Object Oriented Programming Languages.*, Santa Cruz: Univ. of California. 2009
- [14] Hughes, J., *Why Functional Programming Matters.*, Goteborg Institutionen for Datavetenskap, Chalmers Tekniska Hogskola, 1984
- [15] Goyri, A., Franklin, L., Dig, D., Lahoda, J., *Crossing the Gap from Imperative to Functional, Programming through Refactoring.*, Saint Petersburg: s.n., 2013
- [16] Roy, P. V., *Programming Paradigms for Dummies: What Every Programmer Should Know.*, s.l.:IRCAM/Delatour, 2009
- [17] Odersky M., Altherr P., Cremet V., Dragos I., Dubochet G., *An Overview of the Scala Programming Language*, s.l.:IRCAM/Delatour, Second Edition, 2006
- [18] Wikipedia, *Read-Eval-Print Loop*, [https://en.wikipedia.org/wiki/Read-eval-print\\_loop](https://en.wikipedia.org/wiki/Read-eval-print_loop), (2015-11-21)
- [19] Netty is an asynchronous event-driven network application framework ..., *Netty Project*, <http://netty.io/>, 2016
- [20] Markdown, *Daring Fireball*, <https://daringfireball.net/projects/markdown/>, 2002-2016
- [21] Ubuntu 14.04.3 LTS, *Canonical Ltd.*, <http://www.ubuntu.com/download/desktop> 2016
- [22] Gummi, The Simple LaTeX editor, *alexandervdm*, <https://github.com/alexandervdm/gummi>, 2016
- [23] Remarkable, *Jamie McGowan*, <http://remarkableapp.github.io/>, 2015
- [24] IntelliJ IDEA, *JetBrains s.r.o.*, <https://www.jetbrains.com/idea/>, 2000-2016
- [25] Git, *Linus Torvalds, Software Freedom Conservancy*, <https://git-scm.com/>, 2005-2016
- [26] Heroku Play Framework Support, *Heroku.com*, <https://devcenter.heroku.com/articles/play-support>, 2016-01-22
- [27] Deploying Play Framework Apps with the Azure Toolkit for Eclipse, *Kirk Evans[MSFT]*, <https://blogs.msdn.microsoft.com/kaevans/2015/05/12/deploying-play-framework-apps-with-the-azure-toolkit-for-eclipse/>, 2015-05-12
- [28] The High Velocity Web Framework For Java and Scala, *Typesafe*, <https://www.playframework.com>, 2016

- [29] The Interactive Build Tool, *Typesafe*, <http://www.scala-sbt.org/>, 2015
- [30] Play Framework, *Scala Twirl Engine*, <https://github.com/playframework/twirl>, 2016
- [31] Avaje, *Ebean ORM for Java/Kotlin*, <http://ebean-orm.github.io/>, 2016
- [32] LightBend Inc, *LightBend Reactive Platform*, <https://www.lightbend.com/products/lightbend-reactive-platform>, 2016
- [33] Lightbend Inc, *Akka*, <http://akka.io>, 2016
- [34] W. Richard Stevens, Stephen A. Rago *Advanced Programming in the UNIX Environment, Third Edition*, Addison-Wesley 2013 May, ISBN-13: 978-0-321-63773-4
- [35] mindrot.org, *jBCrypt*, <http://www.mindrot.org/projects/jBCrypt/>, 2016-01-30