Czech University of Life Sciences in Prague

Faculty of Economics and Management

Department of Information Technologies

Diploma Thesis

Web Applications Development in Ruby on Rails

Author: Jiří Procházka

Supervisor: Ing. Pavel Šimek, PhD.

CZECH UNIVERSITY OF LIFE SCIENCES PRAGUE

Department of Information Technologies

Faculty of Economics and Management

DIPLOMA THESIS ASSIGNMENT

Bc. Jiří Procházka

Informatics

Thesis title

Web applications development in Ruby on Rails

Objectives of thesis

This thesis will deal with a development of web applications with databases using Ruby programing language with a framework Ruby on Rails. The main purpose is to analyze problems connected with the process of the development and introduce few approaches how to implement a web application. Partial goals of this thesis will include comparison of relational and object oriented databases and description of various tools and utilities commonly used for the development. Practical part of this thesis will cover real development, implementation and deployment of a web application using the theory from the theoretical part.

Methodology

Methodology of the thesis is based on study and analysis of information resources. The practical part is focused on developing a web application with relational and object oriented database, and deploying it to the Internet. On the basis of theoretical knowledge and author s own work and experiences, the conclusion of the thesis will be formulated.

The proposed extent of the thesis

70-80 pages.

Keywords

Ruby, Ruby on Rails, MongoDB, Mongoid, Twitter Bootstrap, PostgreSQL, OpenShift, Web application, SQL, NoSQL, Git

---

Recommended information sources

Bootstrap. [on-line]. Available at WWW: <http://getbootstrap.com>
FLANAGAN, David a Yukihiro MATSUMOTO. The Ruby programming language. First edition. Sebastopol, CA: O'Reilly, 2008, 429 p. ISBN 978-0-596-51617-8.
CHODOROW, Kristina. MongoDB: The definitive guide. Second edition. Sebastopol: O'Reilly Media, 2013, 432 p. ISBN 978-144-9344-689.
LOELIGER, Jon and Matthew MCCULLOUGH. Version control with Git. Second edition. Sebastopol, CA: O'Reilly Media, 2012, 434 p. ISBN 14-493-1638-7.
MATTHEW, Neil and Richard STONES. Beginning databases with PostgreSQL: from novice to professional. Second edition. Berkeley, CA: Apress, 2005, 637 p. ISBN 15-905-9478-9.
OBE, Regina a Leo HSU. PostgreSQL: up and running. First edition. Sebastopol, CA: O'Reilly, 2012, 147 p. ISBN 978-1-449-32633-3.
Ruby on Rails Guides. [on-line]. Available at WWW: <http://guides.rubyonrails.org>
THOMAS, Dave, Sam RUBY and David HANSSON. Agile web development with Rails 4. Dallas, Texas: Pragmatic Bookshelf, 2013, 480 p. ISBN 978-193-7785-567.

Expected date of thesis defence

2015/06 (June)

The Diploma Thesis Supervisor

Ing. Pavel Šimek, Ph.D.

Electronic approval: 31. 10. 2014	Electronic approval: 11. 11. 2014
Ing. Jiří Vaněk, Ph.D.	**Ing. Martin Pelikán, Ph.D.**
Head of department	Dean

Prague on 15. 03. 2015

Declaration

I declare that I have worked on my diploma thesis titled “Web Applications development in Ruby on Rails” by myself and I have used only the sources mentioned at the end of the thesis.

In Prague on

Jiří Procházka

Acknowledgement

I would like to thank to Ing. Pavel Šimek, Ph.D. for his valuable advices and supervision of this diploma thesis.

Vývoj webových aplikací v Ruby on Rails

Web Applications Development in Ruby on Rails

Souhrn

Diplomová práce se zabývá vývojem webových aplikací s užitím relačních a dokumentových databází v programovacím jazyce Ruby a frameworku Ruby on Rails. Teoretická část shrnuje informace o programovacím jazyce Ruby a Ruby on Rails. Dále práce představuje relační (PostgreSQL) a dokumentovou (MongoDB) databázi s jejich základní architekturou a propojením s Ruby on Rails. Poslední teoretická část popisuje několik nástrojů a utilit (např. Boostrap či Git), které se obvykle používají při vývoji webových aplikací.

Praktická část se zaměřuje na vývoj webové aplikace s PostgreSQL a MongoDB databázemi. Dále práce popisuje nastavení vývojářského prostředí jako přípravu pro vývoj. Samotný vývoj se skládá z implementace datového modelu v PostgreSQL za pomocí generátorů Ruby on Rails. Následně jsou implementovány controllery a views, které vytvářejí aplikační logiku. Design je vytvořen za pomocí frameworku Bootstrap. Další část popisuje proces refactoringu, kde byla PostgreSQL databáze nahrazena MongoDB databází. Aplikace je v obou případech pokryta testy pro ověření všech funkcionalit. Nakonec se praktická část zabývá nasazením aplikace na bezplatné cloudové platformy (Heroku a OpenShift) i s popisem celého procesu.

Klíčová slova: Ruby, Ruby on Rails, MongoDB, Mongoid, Bootstrap, PostgreSQL, OpenShift, Heroku, Webová aplikace, NoSQL, Git, SQL, RVM

Summary

The thesis deals with development of web applications with relational and document-oriented databases using Ruby programing language and Ruby on Rails framework. The theoretical part summarises information about Ruby programming language and Ruby on Rails. The next part introduces relational (PostgreSQL) and document-oriented (MongoDB) databases with their basic architecture and connection with Ruby on Rails. The last section from the literature review describes a few tools and utilities typically used for web applications development such are Bootstrap or Git.

The practical part is focused on developing a web application with PostgreSQL and MongoDB databases. It describes setup of the developer's machine environment as a preparation for the development. The development itself consists data model implementation with Ruby on Rails' generators and PostgreSQL. Afterwards, the relevant controllers and views are implemented in order to create the application logic. The design is created with Bootstrap framework. The following section describes the refactoring process, where PostgreSQL database was changed to MongoDB database. The application is covered by tests for both databases to test all of its functionalities. Lastly, the practical part deals with deployment to free cloud platforms (Heroku and OpenShift) and a description of the whole process.

Keywords: Ruby, Ruby on Rails, MongoDB, Mongoid, Bootstrap, PostgreSQL, OpenShift, Heroku, Web application, NoSQL, Git, SQL, RVM

Table of Contents

1 Introduction

The present time is tightly connected with information technologies and life without them is almost unmanageable for many people. With the expanding variety of Internet-enabled devices it is vital for any application to adapt and work on any of these environments. A convenient solution to this problem is the usage of web applications.

Web applications are becoming more and more popular nowadays. Majority of all typical tasks performed by users are done via some web application - it can be a simple using of an email account, checking the weather forecast, transferring money via Internet banking, online shopping or just browsing social networks and chatting with friends. The Internet, with its increasing availability, is becoming accessible to almost everyone and creates a suitable environment for web applications.

One of the greatest advantages of web applications is the platform independence - it can work on almost any device, which has an Internet browser. Upgrade or security patches can be applied at once on the server without any necessary cooperation of users. Distribution is simple - application is available on particular URL without the need of downloading or installing anything. A proper web application incorporates responsive design to deliver the content in a form that is adjusted for any device. These properties make web applications popular from the point of users, developers and companies.

Web applications can be developed in many programming languages and its frameworks. Ruby on Rails is one of the most popular frameworks, which focuses on very fast development process, with minimal configuration and maximum security. This framework is cross-platform (for the development) and open-source with wide community support and continuous development. Since Ruby on Rails is for free and has a very readable syntax, the costs of development are minimal. The framework uses Model-View-Controller pattern, which enables developers to change individual layers with minimal affects for others. Therefore, the underlying database can be switched for another one if necessary. All of these features make Ruby on Rails optimal choice for any web application development.

2 Thesis objectives and methodology

The thesis will deal with a development of web applications with databases using Ruby programing language and the framework Ruby on Rails. The main purpose is to analyse problems connected with the process of the development and introduce few approaches to implement web applications. Partial goals of this thesis will include comparison of the relational and document oriented database and description of various tools and utilities commonly used for the development. The practical part of the thesis will cover real development, implementation and deployment of a web application using the theory from the literature review.

The methodology of the thesis is based on study and analysis of information resources. At first, the theoretical part summarised information about Ruby programming language and its features, followed by possibilities of extending Ruby with additional libraries. The next part dealt with the introduction of Ruby on Rails framework for web application development. Consequently, relational (PostgreSQL) and document-oriented (MongoDB) databases were introduced with basic architecture and connection with Ruby on Rails. The last part from the literature review described a few tools and utilities typically used for web applications development such are Bootstrap or Git.

The practical part is focused on developing a web application with database. At first, an assignment of the web application was formulated and formalised by diagrams and description. Afterwards, as a preparation for the development, it was necessary to set the developer's machine environment. The development itself began with data model implementation with Ruby on Rails' generators and PostgreSQL. Once the models were created the relevant controllers and views were implemented in order to match the application logic. The design was created with the aid of Bootstrap framework and author's experiences and skills. The last part of the development was the creation of tests covering the application. The following section of the solution was to describe the refactoring process, where PostgreSQL was changed to MongoDB with corresponding modifications to models. The application was again covered by tests with the new database. Lastly, the practical part dealt with deployment to free cloud platforms and description of the whole process. The application in version with PostgreSQL was deployed to Heroku, and with MongoDB to OpenShift. The conclusion of the thesis will be formulated on the basis of theoretical knowledge and author's own work and experiences.

3 Literature review

3.1 Ruby programming language

Ruby is genuine object-oriented programming language from scripting family. By the object-oriented properties indication, Ruby supports encapsulation, inheritance through classes, and polymorphism. It is an interpreted language, which uses the interpreter instead of the compiler. Interpreter is directly translating statements into the machine code one after another and not the whole program at once like in Java or C. Ruby is also a dynamically typed language, which means that types are bound at the execution time rather than compile time. This particular property might be a bit controversial in the terms of flexibility vs. execution safety. (TATE, 2010)

3.1.1 History

Yukihiro Matsumoto created Ruby in February of 1993 as a result of searching for pure object-oriented, easy-to-use scripting language. It combines features of Perl, Smalltalk and Scheme (LISP) in a simple syntax and is purely object-oriented. The first public release of version 0.95 happened in December of 1995 for Japanese domestic newsgroup. (MAEDA, 2002)

The development continued with more people involved, which led to release of Ruby 1.0 in December 25, 1996. In 1997 the first article about Ruby appeared on the Internet and Matsumoto announced that Netlab hired him as a full-time Ruby developer. The following year, he founded Ruby Application Archive, which was an online repository of applications for public language development, and also Ruby home page for the first time in English – that helped to spread Ruby outside of Japan. (SIEGER, 2006)

Another assistance for revealing this language to the world came in the form of a printed book in English – “Programming Ruby”. It was published in year 2000, covering version 1.6, and distributed freely online. The biggest expansion and spread came in 2006 with publish of Ruby on Rails framework. (THOMAS, 2001) (TATE, 2010)

3.1.2 Numbers

Five built-in classes and 3 additional classes from the standard library define all numeric data types in Ruby. The hierarchy and relations between these classes are captured on Figure 1 below. All numeric data type classes are subclasses of Numeric; therefore any number object is an instance of Numeric class.

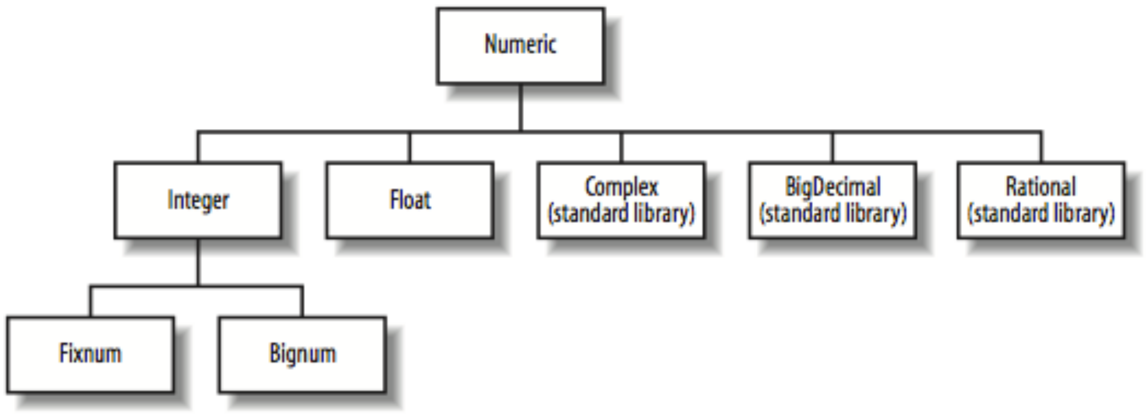

Numeric
Integer
Float
Complex (standard library)
BigDecimal (standard library)
Rational (standard library)
Fixnum
Bignum

Figure 1 - Numeric data type classes

Source: (GAMBLE, 2013)

Integer objects are representing whole numbers and can be created by integer literal as sequence of optional sign followed by numbers with no prefix as decimal representation (0, 12, 3456, etc.), prefixed by 0 as octal representation (0377, 0777, etc.), prefixed by 0x as hexadecimal representation (0x3A, 0xFF, etc.) or prefixed by 0b as binary representation (0b11001010, etc.). The Integer class has two subclasses – Fixnum and Bignum. Division of all integers into these classes is dependent on the size of the number. If the number fits within 31 bits of its value then it's a Fixnum instance, otherwise it's a Bignum instance. Conversion between these classes (to preserve memory) happens automatically, however Fixnum is more commonly used.

The next native built-in class for numbers is Float. Instances of this class are objects of real numbers (or rather approximation of them) in native floating-point representation of the platform. Decimal numbers can be created simply by floating-point literal which consists of optional sign, one or more decimal digits, decimal point (the "." character), one or more additional digits, and an optional exponent part beginning by "e" or "E" character and followed by integer literals (e.g. 0.0, -3.14, 6.23e-20, etc.).

3.1.3 Arithmetic operations

Ruby can perform all basic arithmetic operations as other languages. This includes addition, subtraction, multiplication, and division for all numeric types described earlier, entered as infix input. The resulting data type of arithmetic operation depends on the data types of operands, where bigger range dominates; for example division of Float and Fixnum results in Float. However when dividing two instances of Fixnum, the result is Fixnum again, which can lead to rounding error because the resulting Fixnum will strip all of the decimal places with no rounding (e.g. 5/2 => 2). Integer division by zero throws an error, as defined in mathematics. For Float type this behaviour is different. When dividing Float type by zero, it returns `Infinity` as result. Apart from the basic four operations, Ruby also supports modulo operation (% operator) which returns remainder after integer division (e.g. 10 % 3 => 1) and exponentiation operations (** operator) which powers the input number (e.g. 3**2 => 9).

Arithmetic operations in Ruby have useful properties to avoid overflowing. When for example multiplying or powering large numbers the result won't throw an error, it just simply returns Infinity after achieving a maximum value or zero after achieving a minimum value of Float. (FLANAGAN, 2008)

3.1.4 Strings

Strings are sequences of characters in Ruby. In versions before Ruby 1.9 strings were holding sequences of 8-bit bytes. All strings are objects of String class and they can hold printable characters as well as binary data. Creating string can be done via constructor of String class, but an easier way is to create it by string literals (similar as in Numeric class).

String literals or strings in general are defined within two delimiters – typically single quote ('), or double quote ("). As Ruby is very flexible, the delimiters can be changed to almost any desirable character by preceding it with % character thus avoid escaping the delimiter characters in the string thanks to this change (e.g. `my_string = %& This is "the" string&`). The difference between double- and single-quoted strings is the degree of escaping characters. In single-quoted strings only a very few characters can by escaped by preceding backslash ("\"). Double-quoted strings, on the other hand, supports many escape sequences preceded by backslash (e.g. `\n` for a new line, `\t` for Tab, `\uxxx` for any Unicode character, etc.). Double-quoted strings can also execute any Ruby code inside itself, bounded within curly brackets with hash tag at the beginning - `#{code}`.

There are three ways of constructing strings - by the single-quoted, double-quoted (also substitutable by `%q` and `%Q` retrospectively) literals described earlier and "here documents" or "here docs". This last way is created by encapsulation text starting with two "<" characters and arbitrary keyword and ending with the same keyword (e.g. `my_string = <<DOC text DOC`). "Here documents" construction keeps all characters within the keywords as they are, including non-printable characters, encoding or whitespaces. It's suitable for long strings or documents, where for example indentation might be important to keep.

Almost everything in Ruby is an object and strings are not an exception. Therefore, strings respond to method calls defined by String class and they have particular properties. All strings have encoding defined. This property can be changed, but by default it is set as US-ASCII in Ruby 1.9 and UTF-8 in Ruby 2.0 and above. (THOMAS, 2013)

3.1.5 Ranges

Range objects are modelling real world cases of ranges. It can be range of numbers, letters or other objects where comparison between them is possible. Ruby is using ranges in three main areas – sequences, conditions, and intervals. Syntax for creating a range is - the first element, two or three dot symbols, and the last element (e.g. 1...9, 'a'..'z', etc.), where ranges with two dots are including the ending value (inclusive range), and ranges with three dots are excluding the ending value (exclusive range).

Ranges as sequences are used for defining finite number of discrete objects between two boundaries. The range produces successive values in sequence from the low defining value to the high value (depending on number of dost between). Sequence ranges can be converted to an array, where all elements from start value to end value are listed. It is also possible to iterate through the sequential range via methods each, while, for, etc. Range is sequential (can be enumerated, is discrete) only if each of its objects have deterministic successor (responds to the `object.succ` method), and as in all ranges have to be comparable by "<=>" operator. Thanks to this, any custom objects implementing these methods can be represented in sequential range.

Ranges as conditions can be used like conditional expression – returning Boolean values. It is composed as range of conditions where the first value of the range returns true until it reaches the last value of the range where it starts to return false.

Ranges as intervals can be simply used for tests whether some value belongs to the specific interval (range). Result of this test is true or false value. The operator for comparison between range and tested value is three consecutive equal signs ("==="). Ranges as intervals can work with Float numbers as well, therefore the tested value can be number with many decimal places and it can fall to interval range defined by two native numbers (e.g. `(1..10) === 3.14159 # => true`). (THOMAS, 2013)

3.1.6 Symbols

Comparison of strings is usually very expensive operation, yet sometimes it's necessary for clean code to denote some constants by characters rather than numbers. In many programing languages, programmers have to create global variable with desired name, and then assigning it a simple integer number to solve this issue. With this approach only simple inexpensive comparison of integer is happening instead of string comparison. In Ruby, this problem is solved by symbols. A symbol is basically a string prefixed by colon (e.g. `:north, :wooden`). Two strings objects can hold the same content and still can be completely different objects. This doesn't apply to symbols. Symbols with same content are always the same objects, because they have assigned values on background that makes them simple to compare. Therefore symbols should be used when assigning some category or other unique identifiers in general. Symbols are widely used in various internal comparisons, methods and classes names, etc. (FLANAGAN, 2008)

3.1.7 Arrays and Hashes

Arrays and hashes are collections of objects, which are indexed and accessible through a unique key. The difference between these two types of collections is a type of the key. Arrays use an integer value as a key, starting from zero to n-1, where the n is number of elements in the array. Hashes, on the other hand, can use any object as a key, but with costs of efficiency compared to arrays. Size of arrays or hashes can be dynamically changed and it can grow as needed, while inserting new elements. Both collections support storing of objects of different types at the same collection – for example it can store an integer, a float, a string, another hash or array at the same time dynamically without defining it at the creation.

Arrays can be created by the Array class constructor or by using an array literal. Creating array by constructor can be done via calling `Array.new` for a new empty array, optionally with one argument for defining initial size of array, or with two arguments for size and for default object. Creating array by literal can be done by simply putting objects into square braces or writing empty square braces for empty array. As mentioned before a unique integer key (or index) can be used to access an array element. Accessing an object from array is possible by calling the array's name following square braces with the key number inside (e.g. `arr[3] # => “value”`). Retrieving more objects from array can be done the same way with two numbers in the braces (first is defining a start element and second number of elements) or a range of indices. There are a few approaches how to add a new element to already initialized array. First is to assign object or value to particular index (e.g. `arr[3] = “new value”`). Similarly, in the same way as retrieving elements by two numbers (start and length) or by range, it can be used to add new elements into an array. These methods can rewrite any element on the position specified by the index (indices, range), however this overriding might be potentially dangerous, it is sometimes desirable. Another approach of adding new element is to call “`.push`” method, which simply adds object in arguments to the end of array. Removing elements from an array can be done by calling “`.pop`” method (to remove last element) or to assign “`nil`” object to undesirable element. (THOMAS, 2013)

As mentioned before, hashes can use almost any object as a key. Every key has to be unique and is associated with particular value. Hashes can be created by the constructor as a “`Hash.new`”, which creates a new empty hash, or by hash literal. Hash literal is an expression closed within curly braces containing zero or more key-value pairs (zero for empty hash). Syntax of a pair is the key, which could be string, number, regex, symbol, etc., two-character arrow “=>”, and the value, which could be any arbitrary object (e.g. `{:one => 1, “key” => [1,2,3]}`). Most common type of keys is symbol, because of the best efficiency in hashes. Since Ruby 1.9 hash literals with symbols as keys have alternative simplified syntax, similar to JavaScript notation. The simplification lies in dropping the arrow and moving the colon prefix to postfix of the symbol – e.g. `{one: 1,` `key: “value”}`. Addition of new elements can be done by assigning a new key with value to already existing hash, like in arrays (e.g. `my_hash[:two] = 2`), or calling “`.store`” method with key and value as two arguments (e.g. `my_hash.store(:three, 3)`). (FLANAGAN, 2008)

3.1.8 Regular expressions

Regular expression is a special expression, which specifies a particular characters pattern that can be matched to a certain string. Class for regular expressions in Ruby is called Regexp. Object of this class can be created by a constructor of the class or by literals as expression within “/” characters or within “`%r{`“ and “`}`” characters. The expression can be extended by modifiers, which can modify the pattern with special options. Regular expressions in Ruby can be used to return a specified pattern from a string or just to test if the pattern is present and its first occurrence. There are two main methods for matching the Regexp’s pattern – “=~” operator and “`match()`” method. The “=~” operator returns index with position of the first occurrence of the pattern, which equals to true when running Boolean test. Otherwise it returns nil, which equals to false. The “`match()`” method is called on the Regexp object with examined string in arguments. It returns MatchData containing the matching substring, when there is a match (this equals to true), or it returns nil when there is no match (this equals to false). Both methods set a special variables “$~”, which stores the MatchData. The syntax of Regular expression is universal for many programing languages and Ruby is not an exception. Basic syntax is captured in Table 1 below. (GOYVAERTS, 2013)

Pattern	**Explanation**
[abc]	A single character of: a, b, or c
[^abc]	Any single character except: a, b, or c
[a-z]	Any single character in the range a-z
[a-zA-Z]	Any single character in the range a-z or A-Z
^	Start of line
$	End of line
\A	Start of string
\z	End of string
.	Any single character
\s	Any whitespace character
\S	Any non-whitespace character
\d	Any digit
\D	Any non-digit
\w	Any word character (letter, number, underscore)
\W	Any non-word character
\b	Any word boundary
(...)	Capture everything enclosed
(a|b)	a or b
a?	Zero or one of a
a*	Zero or more of a
a+	One or more of a
a{3}	Exactly 3 of a
a{3,}	3 or more of a
a{3,6}	Between 3 and 6 of a

Table 1 - Regular Expressions

Source: http://rubular.com

3.1.9 Naming conventions

Names of objects, classes, modules, methods, etc. are very important for identification and it's good to keep some rules of naming them for better readability. Ruby is very benevolent and free with characters in names, yet there are some arbitrary conventions for a couple groups of entities. Names of the first group, which includes local variables, method parameters, and method names, should begin with a lowercase letter or an underscore (e.g. `subject`, `insurance_company`, `_type200`, etc.). The underscore is used as a word delimiter in more complex names. These rules are usually also applied on symbols as well. The second group, containing classes, modules and constants, has to start with an uppercase letter. This group is then using capital letters to distinguish words in multiword names (e.g. `InsuranceCompany`, `Object`, etc.). (RUBY, 2013)

3.1.10 Variables

Variables in Ruby are simply names for values or references to objects. They can be created and assigned by an assignment expression (`variable_name = “text”`), while it’s on the left-hand side. Other occurrence of name of the variable than left-hand is a reference to the variable and it returns its value. There are four kinds of variables in Ruby – global, instance, class, and local. Global variables are prefixed with “$” character and are visible everywhere in the whole program. The next kind are instance variables, which begin with “@” character. Scope of these variables is within an instance of a class and it is accessible only within instance methods or using accessors. Class variable is similar to instance variable but it has a scope through the whole class and is prefixed by “@@” characters. Its reference is the same across all instances of the class and it is accessible within the definition of the class, class methods and instance methods as well. Class variables are not visible out of the class – they are encapsulated like instance objects. The last kind of variable is the local one. It has no special prefixes and usually begins with lowercase letter or an underscore. Its scope is only within a method or a block of code; therefore it’s not visible outside of it. (FLANAGAN, 2008)

3.1.11 Classes

As already mentioned, Ruby is purely object-oriented language, where every value is an object and every object is an instance of a class. Every class is like a template for an object – it defines a set of relevant methods on which object responds. Classes can subclass or extend other classes, consecutively inherit or override methods. All objects in Ruby have property of strict encapsulation of its state. The object’s state can be accessed only through instance methods defined by appropriate class. Instance variables can be also accessed only via instance methods, particularly by setters and getters. These two methods are called accessor methods or attributes. Ruby has two main methods for easily defining instance variables – “`attr_reader`” and “`attr_accessor`”. Both of these methods take name of the variables as arguments. The “`attr_reader`” method defines only getters for the passed variables, and the “`attr_accessor`” extends it with setter methods as well. Classes are opened for adding new methods to already existing classes programmatically on the fly by other Ruby programs. Furthermore new methods can be added to single instantiated object – “singleton methods”.

Definition of a new class is denoted by keyword “`class`” followed by a name of the class with first capital letter and is closed by keyword “`end`”. Between the name and the “`end`” keyword is the class content, where properties, attributes and methods can be defined. Class methods can be called almost anytime from everywhere and their names are prefixed with the class name or “`self`” keyword where applicable. In contrast, instance methods can be called only from class instance perspective as a message to object.

Calling “`new`” method on the class object can create a new instance of the class. This method call will return a new object of that particular class. Each class can have a special method called “`initialize`”, which is invoked on creation of new instance. Procedures needed during the creation of a new object or input arguments should be located in this method. Arguments passed to the “`new`” method are automatically passed to the “`initialize`” method. This can serve for initialization of instance variables or other properties of the new object. (FLANAGAN, 2008)

3.1.12 Methods

Ruby methods are similar to functions of any other programming language. Method is a named block of code suitable for repetitive calls without the need of rewriting the same code again. Methods are associated with one or more objects – it is always invoked on some object because in Ruby everything is an object. They can also have optional zero or more arguments. Syntax for creating a method begins with keyword “`def`” followed by a space and a name of the method and closed with a keyword “`end`”. The body of the method is then between the method name and the keyword “`end`”. The method name can be prefixed by a receiver object, if not it is automatically defined for “`self`” of the scope, which can be a global method when the method is defined outside of a class or an instance method when it’s defined within a class body.

There are three main types of methods – class methods, instance methods and singleton methods. Singleton methods are defined only for one particular object. The method name is prefixed with the object and a dot. Instance methods are defined only for instance objects of a class where it’s defined. They have no prefix before the name and they are defined within the body of the class. Instance methods can be invoked only on an existing instance of the class. If a method is defined out of a class body it is still within an Object class and it acts as an instance method for Object class. The class methods are prefixed with a name of a class to which it belongs. If it is defined within the class body, the class name in prefix can be substituted with "`self`" keyword. Class methods can be invoked directly on the class with no need for any instance of that class.

Ruby has a system of access control to all methods in three levels of visibility – public, protected, and private. All methods are by default public, which means that everyone can invoke them from everywhere. Only exceptions are the initialize method and methods defined within the Object class, which are always private. Protected methods can be invoked only on explicit instances of defining class or its subclass or within the class or subclass implementation. Private methods are for internal usage and can be called only from other instance method within the defining class or subclass. These methods can't have explicit receiver – it's always "`self`". Private methods serves only for internal class purposes and it is recommended from security reasons to make private all methods, which could possibly invalidate the object. (FLANAGAN, 2008)

3.1.13 Modules

Module is similar to class – it contains methods, class variables and constants. However, module can't be instantiated or subclassed. It stands alone without hierarchy of inheritance. Syntax of modules is almost the same as classes', only the starting keyword is module instead of class. Name should also begin with capital letter and keeps the same rules as classes' names. Modules have two main applications – namespaces and mixins.

Module as namespace serves for grouping methods for better readability and usability. It should be used when there is no need of creating any instances, hence using class for this would be unnecessary. Methods included in the module for namespace purposes should be strictly in form of class methods. All of the defined methods, constants or class variables are accessible through a prefix of the module name, a dot, and the method (with optional arguments) or variable name.

As mentioned, modules cannot be instantiated, yet it is possible to define instance methods within its body. Module with such methods is called mixin module. These methods are accessible via any class, which includes the module. When the module is included, all of the instance methods of the module are available for the class instances as well – the methods are mixed together. For including a model into a class it is necessary to add a keyword "`include`" followed by a space and a module name within the class body. Usage of modules as mixins can (to a certain extend) substitute subclassing and inheritance of classes. It's one of Ruby's very powerful features. (FLANAGAN, 2008)

3.2 RubyGems

RubyGems is packing and installation framework for Ruby libraries and applications, which can extend Ruby programs. RubyGems simplifies the procedures of this extension by managing the location, installing and upgrading these libraries, called Gems. Gem is a single file containing a bundle of a libraries or an application, stored in some repository on the Internet. RubyGems also take care of dependencies between the installed gems. It provides a simple command-line tool called gem for managing gems (e.g. "`gem install rails`", "`gem list`", etc.). In version before Ruby 1.9, RubyGems has to be installed manually as a separate tool; with Ruby 1.9 and following, this tool is already integrated. Gems can be included into a Ruby program by simply calling a require command with a name of desired gem at the beginning of the file. (THOMAS, 2013)

3.3 RVM

Ruby Version Manager (RVM) is a command line utility for installation, management and work with several Ruby environments on one system. Having multiple Ruby environments can be helpful in case of testing new Ruby versions, upgrading, or managing more Ruby applications each with different version. RVM also provides functionality called gemsets, which manages RubyGems for different Ruby versions. Each Ruby can have multiple gemsets to meet the developer needs. It can be manually created for each project, user, folder, etc. Gemsets keep the Gems separate, preferably for each project, and install only Gems that are required. Gems' installations are managed centrally and they are linked (not copied) to every gemset, thus saving space and complexity. RVM can run multiple Ruby environments on one system at the same time in different folders or even in different shell window.

Rails developers can use RVM's capabilities especially when maintaining more projects where different Gems are used. Also the process of upgrading Rails versions (or any other Gem) is easier with RVM – it allows developers to create a new Gemset with upgraded gems and debug it while keeping the original working version separate and intact. In combination with different branch (for example of Git) it can be switched in between almost instantly. (SEGUIN, 2014)

3.4 Web Applications and Frameworks

Web applications are becoming more and more popular nowadays. Almost all typical users' activities are done via some web application. Whether it's about checking email, transferring money, online shopping or chatting with friends, all these things are online and accessible through a web application. In general, web-based software has a lot of advantages compared to traditional software. One of the biggest is the distribution way – traditional software has to be copied on some media and then installed on a computer, or placed on the Internet for downloads. Web applications are already running on servers and users can simply use them via web browser without any installations. This is also helpful with bug fixing and software updates. Web application is updated once on the server and then the users have the new version directly after a single refresh of a browser. Traditional software has to be updated via downloading more OS specific files and executing them in order to have a new version.

Another advantage is the platform independence. The only requirement is to have a network connection and web browser on a device. Web application then doesn't care about the underlying operating system; it uses the features of web browsers. That means, it can run on smart phone device, on Linux, Windows, Mac OS or almost any operating system available. Without the need for installation, web-based software can be run from anywhere and any device. Traditional software is developed only for specific operating systems, usually separately for each.

Even though there are many advantages of web-based software, there are some constrains as well. Various types of browsers from different producers may cause a slightly modified behaviour than expected. The problem is particularly displaying the content via HTML and CSS. Although, there have been some attempts to unify standards and improve cross-browser compatibility, there is still need for browser specific coding. Aside of client-side development it is also quite complicated to develop the server-side. Server-side can be quite complex with all protocols, ports, database servers, network structure and configurations working together. Solution for this complexity came in a form called Framework.

Framework is a collection of libraries and various tools made for simplifying web applications development. It should serve for automation and shortcutting of repeated tasks and pre-coding default behaviour as a start-up. Framework is basically a portion of already written code prepared for further development. It typically includes fixes to various vulnerabilities managed by a community, which gives a great way of support. A good framework should have several properties. It should be a full stack – it means that the framework should be a bundle for building a complete applications included with everything needed for the development. Framework should also be an open source – thanks to this all developers can participate on new versions and fixing its bugs. Another important property is cross-platform compatibility – a framework should run on any platform and stay as neutral as possible. A good framework also provides specific structure of its projects, where everything has a proper place.

A database abstraction layer is another important aspect – developers shouldn't be dealing with low-level database access, furthermore it shouldn't be constrained to a particular database engine. Lastly, a good framework should guide developer by conventions used thus liberating him and letting him focus on important aspects of development. (GAMBLE, 2013)

3.5 Ruby on Rails

Ruby on Rails (or simply Rails) is a web application framework for Ruby programming language. Rails has the properties of a good framework, which were mentioned earlier – it is complete, open source, cross-platform, with powerful database abstraction layer, and multilayer system for organizing program files and concerns. Rails framework is focused on productivity and code readability.

David Heinemeier Hansson originally created the Ruby on Rails framework from a wiki application called Instiki. The first public release of Rails was in July 2004. It was created from an already functional web application Basecamp created by developers' group 37signals – they took out the application specific parts and crafted a simple generic framework for web applications in Ruby. Because of this extraction, Rails includes only features that are really helpful and commonly used. The framework solves around 80%

of all problems connected with the development such as naming conventions, application construction, database abstraction, etc. The other 20% of problems are considered unique and should be solved within the particular application.

Ruby on Rails framework is an open source and distributed under the MIT license, which is one of the most free software licenses. Thanks to that anyone can download, modify and use the source code as wanted. The open license also allowed to create a whole community of developers contributing with updates and fixes to the framework.

Most developers using Rails are running UNIX based operating systems (typically Mac OS or various Linux distributions) but it also supports Windows environment. The framework has included stand-alone webserver WEBrick, which can instantly run the currently developed web application. There is no need for any Integrated Development Environment, because Rails can be coded in any simple text editor. (GAMBLE, 2013)

As the development of Rails framework had continued, functions, features and popularity of it were growing. Thanks to the license, the community contributes to the development process and the framework has been improving by every release. A brief overview of the most significant versions with some of the important changes is shown in Table 2 below.

Version	**Release**	**Notes**
Rails 1.0	Dec 2005	Polishing and closing pending issues from the first release; inclusion of Prototype and Scriptaculous
Rails 1.2	Jan 2007	REST and generation HTTP appreciation
Rails 2,0	Dec 2007	Better routing resources, multiview, HTTP basic authentication, cookie store sessions
Rails 2.3	Mar 2008	i18n, thread safe, connection pool, Ruby 1.9, JRuby
Rails 3.0	Aug 2010	New query engine, new router for controller, mailer controller, CRSF protection
Rails 3.1	Aug 2011	jQuery, SASS, CofeeScript, Sprockets with Assets Pipeline
Rails 3.2	Jan 2012	Journey routing engine, faster development mode, automatic query explains, tagged logging for multi-user application
Rails 4.0	June 2013	Russian Doll Caching, Turbolinks, Live Streaming, Declarative etags, extracting some components into separate gems as optional
Rails 4.1	April 2014	Spring, Variants, Enums, Mailer previews and secret.yml

Table 2 - Rails version history

Source: (HANSSON, 2014), (HANSSON, 2013), (BASU, 2013)

3.5.1 Agile Software Development

Web applications development is often considered as difficult to work withand complicated to maintain. Therefore Rails tried to advocate as many agile methods as possible to simplify the development process. Agile development is based on iterative and incremental methodologies, and close cooperation with a customer. There are four main principles that Rails uses. The first prefers individuals and interactions to processes and tools. The second principle promotes working software over comprehensive documentation. The third one concerns about the collaboration with the customer rather than cancelling a contract. Every issue can be solved somehow, but it's essential to communicate and manage all problems. This is directly connected to the last principle of responding to a change over following a plan. Things can change and the development should adapt to these changes in order to deliver desired results for all parties. These principles were defined in Agile Manifesto by a discussion among 17 figures in the field of software development. (GAMBLE, 2013)

3.5.2 Convention over Configuration, DRY

Ruby on Rails tends to a philosophy of using less software. This basically means to follow few principles to make things simple. Convention over configuration is one of the really helpful during the development. It means that if a developer sticks with conventions (of naming, files locations, definitions, etc.) it is then already configured with defaults and ready to use. Many frameworks are difficult to configure, but Rails framework has a very little configuration needed. It guides the programmers through a certain path of conventions and rules for a very fast development without any low-level decisions. It is of course possible to change almost all of the framework's configuration but with the cost of loosing efficiency.

The DRY acronym stands for "Don't Repeat Yourself". This term is fairly self-explanatory. The principle is about not duplicating the code; it should be written only once and stored on one place only. It prevents difficulties when updating and fixing bugs. In Rails it applies on values and credentials as well – for example database credentials can be stored in one file and then accessed from any file of the Rails' project. (GAMBLE, 2013)

3.5.3 The MVC Pattern

The creators of Rails framework decided to implement the MVC pattern, which is well known and widely used architectural concept. It is dividing the application logic into three categories – the model, the view, and the controller (MVC). The model represents the data, the view represents the user interface, and the controller handles all the interaction between the model and the view. The power of the separation to these three layers lies within its isolation and independency. Any of the layers can be modified without any affects on any other layer. This independence simplifies maintenance and helps the reusability of components. In Rails, the layers work together without any extra coding as long as the conventions are kept. There are more software design patterns, but the MVC performed as fast and very efficient for web applications development.

The typical control flow of the MVC cycle is captured on Figure 2. The cycle can start with user's interaction with the interface, triggering some event that is handled by the controller, which can access the model for data retrieval or update. The next step again handles the controller with updated information from the model, renders a new view, which is again ready for user interaction.

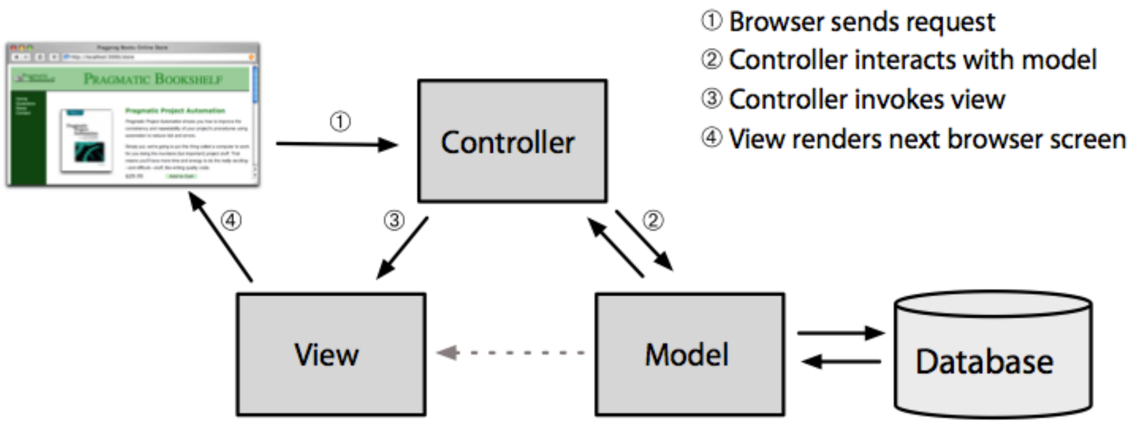

① Browser sends request
② Controller interacts with model
③ Controller invokes view
④ View renders next browser screen
Controller
View
Model
Database
①
②
③
④

Figure 2 - The MVC cycle

Source: (RUBY, 2013)

As already said, the model layer represents the data, which in web applications is a database. Rails application typically embeds several models where each is mapped to a particular database table or collection in case of NoSQL databases. If the developer is following the conventions, then for example a class (model) called “Lecture” inheriting from ActiveRecord class is by default mapped to SQL table “lectures” and all its fields. This mapping is called Object-Relational Mapping (ORM). The ORM maps the SQL tables from relational database to classes, records (rows) to objects and individual fields to attributes. Rails have a built-in ORM library within a class ActiveRecord, which is also a database abstraction layer. It defines many methods for working with database records such as creating, reading, updating, deleting, etc. In general, models should also contain all operations manipulating with the data – this includes validations, associations, calculations, which should be called before or after save, update or destroy (callbacks). (RUBY, 2013) (GAMBLE, 2013)

The next important part of this concept is a controller. Controllers are very important hence they link everything together and manage operations of the web application. The most significant is the connection between views and models. Controllers are accepting requests from the users (or other interacting entities), processing the input and then passing the control to the view layer, which displays the result. The model layer is not necessary included in all of the controllers’ actions – such actions are then processing inputs without the database access. However, more common are actions with the database aid. Typical controller that interacts with a model usually performs four basic operations with data – creating, reading, updating and deleting. These operations are often abbreviated as the CRUD. For a simplification, controller’s work can be described as performing the CRUD operation(s), setting variables for the view and rendering that particular view(s).

Controllers define instance methods, which are called actions and are usually linked with a same named view. Controllers are designed to manage single area of an application. For example a data model with some lectures will be linked with a Lectures controller, which will include all the appropriate actions. The decision, which action should be executed is made by parameter in the requesting URL – the parameter is mapped to a specific action of a controller. A simple default controller for lectures with basic actions is captured in Appendix 10.1.

The last part of the MVC pattern is the view layer. Views are templates, which contain HTML to be rendered for the end user and are injected with Ruby code. Views should contain as little of the application logic as possible without any interaction with the model (it should be done via relevant controller). The default templates are called "erb" – Embedded Ruby and for HTML documents have the "*.html.erb" extension. They are composed as a plain HTML with the addition of Ruby code encapsulated within "`<% %>`" or "`<%= %>`" characters, depending if the Ruby output should be rendered in the template or not. Rails also provide special set of helpers, which are helping to the connection between the view and the controller (e.g. prepopulating forms, formatting currencies, etc.). The combination of plain HTML, simple injections and logic out of the view helps to divide the development between designers and programmers. An example of a simple view presenting a list of lectures from index action of relevant controller is defined in Appendix 10.2.

The Rails framework is composed of many different libraries. There are three of them, which are directly mapped to the MVC pattern described above. The model is mapped to the ActiveRecord library (handles database abstraction and interaction), the view is mapped to the ActionView library (templating system that generates the HTML documents with the Ruby code), and the controller is mapped to the ActionController library (for manipulating application flow and the data on its way from database to user's display). (GAMBLE, 2013)

3.5.4 Rails Console

Ruby provides an interactive interpreter as tool to help the development. When Ruby is installed on a system, the Interactive Ruby is accessible via command "`irb`", where developer can execute Ruby commands and read the outputs. Rails ships with its own interactive console, which is based on the irb. It has all of its function and additionally it is working within the Rails project environment. This means that developer can access the application's variables, settings, and even access the models and database. Therefore it can be used for testing the application behaviour and modifying data from command line environment. The console can be run by command "`rails console`" or "`rails c`" from Rails application project folder. (GAMBLE, 2013)

3.5.5 Application Initialization

With Ruby, RubyGems and Rails installed on a system, it is very easy to create a new empty application. Rails have many tools and generators to make developer's life easier. Creating a new application can be done by execution of "`rails new <application name>`" in a command line (or shell, bash, etc. - depends on the operating system). This command will generate a folder containing default structure and configuration. Developers should follow this directory structure, because everything has its optimal location for better uniformity. The root directory structure of Rails application is captured in Table 3.

Folder/File	Description
app	All the components of the application
bin	Executables to support Rails
config	Configuration files for all of the components of the application
config.ru	A file used by rack servers to start the application
db	Folder containing database connected files with subfolder for migrations
Gemfile	File with list of gems used in the application, used by bundler gem
Gemfile.lock	Canonical resource of what gems should be installed
lib	A folder for libraries, that could be used in the application
log	A folder with all of the log files needed
public	A folder with static assets served by the application
Rakefile	Lists available for tasks used by Rake
README.rdoc	Human readable file for the application description
test	Directory containing various tests for testing the application
tmp	Folder for temporary files
vendor	Folder with external libraries, gems, plugins

Table 3 - Rails directory structure

Source: (GAMBLE, 2013)

All Rails applications have three basic environments by default – development, test and production. This separation allows developers using different configurations, database connections, etc. in different environments. The development is for development machines of programmers; the test environment is for running unit, functional, integration and other tests. Lastly, the production environment is meant for a server where the applications will eventually run.

After running the command for creating a new application (mentioned earlier) without any extra arguments, the application will be configured with SQLite database, with different name for each environment. By passing an extra argument "`-d`" and relational database engine name after "`rails new`", the command will configure the application with the particular database (options are: mysql, oracle, postgresql, sqlite2, sqlite3, frontbase, or ibm_db). There is also an option of no database configuration in case of manual setup or NoSQL database. When the configuration of the database is complete, it is necessary to create it. Rails provide another tool for database manipulation without any complicated DB engine specific procedures. Running command "`rake db:create`" will ensure that the application database is created with the previously setup configuration. Once all these operations are done, Rails can provide direct access to the database query engine by calling "`rails dbconsole`". If this command works, the application is successfully connected with the database.

When this basic configuration is done, a local web server can be started up to test the application. As already mentioned, Rails ships with built-in web server WEBrick, which can be launched by executing "`rails server`" from the application directory (see Appendix 10.3). If the web server starts up successfully, a welcome page should be accessible by web browser at http://localhost:3000 (see Appendix 10.4). (GAMBLE, 2013)

3.5.6 Generators

Rails framework provides special tools called generators, which simplifies some frequent tasks. Each generator is ought to be run within the project folder starting with "`rails generate`" command. In general, generators create new files; modify other files or configurations, in order to speed up development. It is also possible to create custom generators to automate application specific tasks or setting up custom engines, gems or other plugins. There are four main built-in generators – model, controller, scaffold, and migration.

Model generator is executed by "`rails generate model`" with additional arguments. First argument is a name of the model, which is mandatory. After running this command a few files are created – model, migration for the model, unit test file for the model and fixtures. The test file and fixtures will be described in detail later in chapter 3.5.7. The model has a name that was passed to the generator, which should be camel-cased and singular, and it is mapped to a database table with the passed name in plural and lowercased. The model file is located in “app/models” folder. The migration file contains procedures for creating the new table and lies within “db/migrations” folder. The model generator can be extended with more arguments describing attributes (or fields) of the model in form “`[field[:type][:index] …]`” (e.g. “`rails generate model Lecture name:string`”). For creating the new table in database it is necessary to run command “`rake db:migrate`” to execute all pending migrations.

The controller generator has a similar syntax to the model generator – “`rails generate controller`” followed by controller name in plural form. Additionally, it can be extended by controller’s actions separated by space (e.g. “`rails generate controller Lectures index show`”). The generator creates a controller file named as the passed name with “_controller.rb” at the end in “app/controllers” folder (e.g. “app/controllers/lectures_controller.rb”). Furthermore, this generator creates appropriate templates for all of the controller’s actions in new folder named as the passed name within the “app/views” directory, same-named JavaScript and CSS files in “app/assets” folder, a test file for functional tests of the controller, and a helper file for the views.

The functions of migration generators are used within the model generator, as already mentioned. Migration generator creates a migration file, which is used for changing the database tables. The file contains commands for adding, removing, and renaming fields, or even whole tables. However, majority migration files are typically used for modifying existing tables. It can be created by calling the command “`rails generate migration`” with additional argument of name of the file. Optionally, the arguments can be extended with field name and its data type for adding a new column of the table.

The migration file can contain a change method or up and down methods. Database modifications are within these methods. The up method is for running the migrations by command “`rake db:migrate`”, the down method is for rolling back the previous state before the modifications by command “`rake db:rollback`” (see Appendix 10.5). The change method is substitution for both methods, which is written in the “up” form and the “down” form is generated automatically (see Appendix 10.6).

The last of the four main generators is the scaffold. It is one of the most powerful generators from the whole framework suitable for fast prototyping. It basically combines features of model and controller generators and links everything together. Running command "`rails generate scaffold`" with arguments same as the model generator had, will cause creation of controller, model with migrations and basic views for actions. The basic views include listing all objects of the model, form for creating a new one, view for showing details of one and form for updating. These views are bound to actions in controller respectively – index, new, show, edit – with few more for actions without need of view – create, update, destroy. All these actions, views, model, and controller creates together a simple application only by running the generator. As was with the model generator, it is necessary to run the pending migration after generating the scaffold. (GAMBLE, 2013)

3.5.7 Tests

Testing application is essential for developing and producing bug-free functional applications. Every user tests the application visually by using it and observing the outputs. This is sometimes called manual testing. Since this approach might be sufficient for small applications, it is definitely not the correct way. A proper way of testing is to create and use automated tests. An automated test is a code designed to exercise the software and confirm assumptions or expectations of the outputs. Then it can be run repetitively to check if the application works, as it should. Tests are very useful when programmers are refactoring code, adding new functionalities or creating new parts of any software – to see if any functionality is not working or acting differently than normal. The best practise is to write test right away with the code – e.g. when writing methods for creating user, a test to check all object's attributes and validations is in order. As already mentioned, testing is very important and useful tool for developing process particularly for large applications or financial systems where every mistake can be fatal for the business.

Rails framework and its community encourage writing tests. The framework gives a structure for testing by providing test directories for controllers, models, mailers, helpers, and integration tests, fixtures for working with database data and a whole environment created for testing purposes.

Fixtures are representing database data in textual form formatted in YAML (data-serialization format). Before execution of the tests all fixtures are loaded into the database to test against. The fixtures folder contains files for each database table with matching names. Rails generators are creating these files when generating a model, but it can be added manually as well. YAML files can be manually filled with desired data to test with. The test database records can be created or modified during the test execution from individual tests; fixtures are prepopulating the test database before the execution. After the tests are finished, the database is dropped.

Rails test files typically contain a class inheriting from one of following classes: "ActiveSupport::TestCase" (for unit tests), "ActionController::TestCase" (for functional tests), "ActionMailer::TestCase" (for mailer tests) or "ActionDispatch::IntegrationTest" (for integration tests); and a requirement statement for test_helper. These entities provide several methods for testing purposes. Actual tests are implemented as blocks using "`test`" keyword at the beginning of every block followed by test description within quotes and closed by the "`end`" keyword. Testing is done via using assertions methods. These methods are verifying some condition or event and returning true or false depending on the result. Successful assertion is denoted by dot symbol, failed assertion by "F" and if the test produces an error it is marked with "E", in the summary at the end of test run. Tests can be run individually, by groups ("`rake test:models`", "`rake test:functional`", etc.) or all together ("`rake test`").

One of the most significant areas of testing is the unit testing. In Rails, unit tests are for testing models and its methods. It is recommended to create test cases for the basic CRUD operations for starter and then adding more specific tests validating behaviour of all methods within the model. Another important area of testing is the functional testing. This basically covers tests of controllers' methods in the same way as unit tests with addition of a web context. Functional test cases have the ability to test the full request/response cycle. Thanks to that it can test the application in how it runs on a web server (e.g. testing URL with correct parameters etc.). Mailer tests are considered as a part of functional testing and it works the same way as unit tests with the context of mail server configured. The last group of tests are the Integration tests. This group of tests is very similar to functional tests, however it can span over more than one controller and it supports complete session. Integration tests are the closest simulation of a running web server application. It is possible to follow the flow of redirects until reaching some point to test within one test case. These tests can be created as possible scenarios of user's work and the test the output correctness. (GAMBLE, 2013)

3.6 PostgreSQL

PostgreSQL is a Database Management System (DBMS), which includes relational model database and SQL standard for queries. DBMS is typically a set of several libraries, applications and utilities that helps developers to efficiently create working databases without the need of low level programming such as storing and managing the data. There are several alternatives on the market (commercial or free) to PostgreSQL – for example MySQL, Oracle, IBM DB2, Microsoft SQL Server, etc. A proper DBMS should be able to create database, provide query and update facilities, multitasking (simultaneous database access), maintain an audit trail (logs about changes for certain time period), manage the security of the database (authentication and authorization of users), and maintain referential integrity (data correctness).

Databases can be divided into four main categories according to following a particular database model. First model is the Hierarchical Database Model. It is composed of collections of records, which are then hierarchically broken down until the final "leaves" of the connected records. The next model is called the Network Database Model. This model is based on wide usage of pointers. Each record has a pointer within itself pointing to succeeding connected record. This fact allows very fast data retrieval, however it is very slow in selecting some particular data from inside of the "chains". Probably the most recent database model is the Object-Oriented or Document-oriented Database Model, which will be discussed later in chapter 3.7.

As already mentioned, PostgreSQL incorporates the Relational Database Model. This concept had lied down some new approaches that became very popular. The model introduced the idea of relations therefore it was able to closely relate to real-life objects and describe their properties (data). E.F. Codd published this concept in 1970 and it is still extensively used nowadays. Relational Database Model emphasizes the importance of data integrity and referential integrity. Stored data has to be consistent and comprehensible at any time. (MATTHEW, 2005)

3.6.1 History

The PostgreSQL started to shape in 1977 at the University of California at Berkley by relational database called Ingres. Ingres was being developed until 1985 and was widely used among universities and various research facilities. The whole project was then commercialised thus becoming one of the first commercially available Relation Database Management System (RDBMS).

The original development of database server at Berkley continued and changed the name to Postgres. In 1994 project Postgres was equipped with SQL features and changed the name again to Postgres95. Around 1996 the database was very popular and widely used; therefore the developers decided to open the development to the community for contributions. The last change of name to the final PostgreSQL happened shortly after these events.

Since then PostgreSQL is open-source DBMS available for everyone, supported by the community of developers with various suggestions, fixes and new features. It also has commercial support provided by several companies. Since the first release, PostgreSQL is getting more stable, reliable, with more functions by every newer version. Every upcoming release is extensively tested over sufficient period of time to ensure that no major bugs or disadvantages are included. The current stable version is now 9.3.5, and beta release 9.4 – almost ready for deployment. All current and past releases are captured in a timeline on Figure 3. (MATTHEW, 2005)

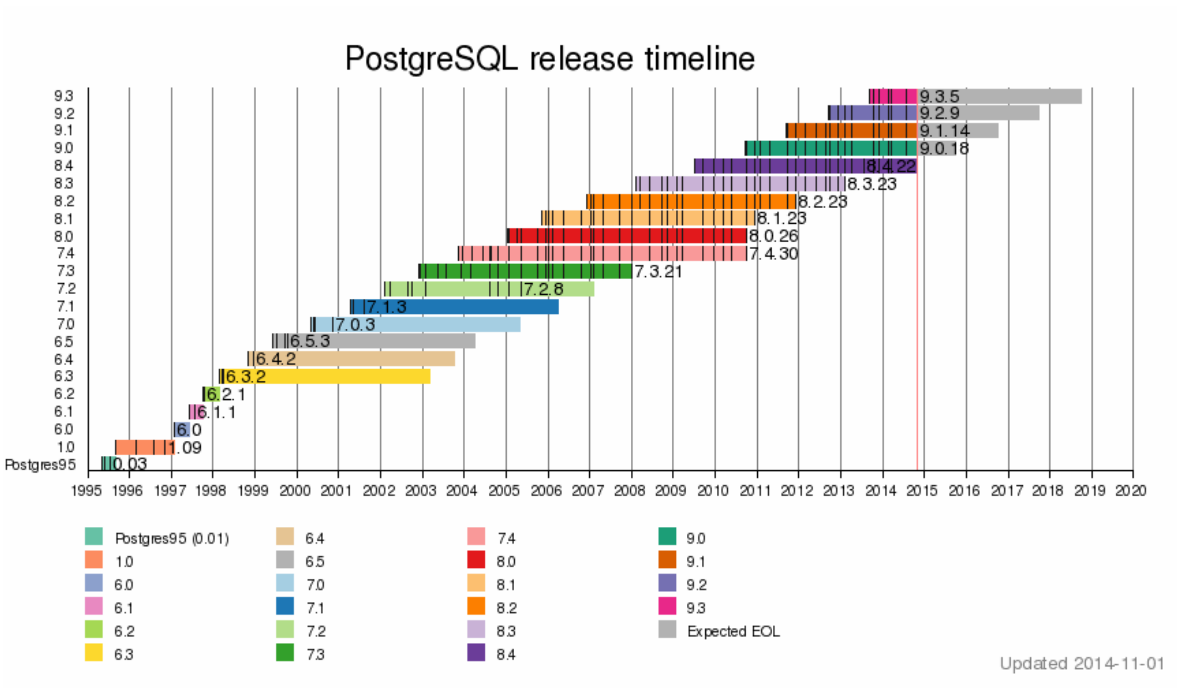


Figure 3 - PostgreSQL release timeline

Source:
http://upload.wikimedia.org/wikipedia/en/timeline/4ecb8fb0a423cb0f51bcc5d8d108c2c1.png

3.6.2 Architecture

PostgreSQL is DBMS that is very reliable, capable, with good performance characteristics. It can run on any UNIX-like platform, including FreeBSD, Linux, UNIX, Mac OS X, and also on some Windows systems such as Microsoft Windows Server or Windows XP/Vista/7/8 for development.

PostgreSQL can be used in a client-server environment. The core is the database process, which runs on a server. The only way how to access the stored data is through this process via some client running on the same or different environment. This separation of server and client allows applications to be distributed, therefore the database process providing the data can run on some UNIX machine and a client accessing these data can run on a different machine with Windows on network for example. Condition for the connection is that the network has to be a TCP/IP network (e.g. LAN, the Internet). The database server can serve multiple clients at the same time. All clients connect to the main server process called Postmaster, which then creates a new process specifically for the particular client that handles his requests. The client-server communication is done via PostgreSQL message protocol. It is also possible to use special interfaces, which allows applications without the ability of using the native protocol, to communicate with the server (e.g. Open Database Connectivity – ODBC, Java Database Connectivity – JDBC). The basic architecture of PostgreSQL is shown on Figure 4.

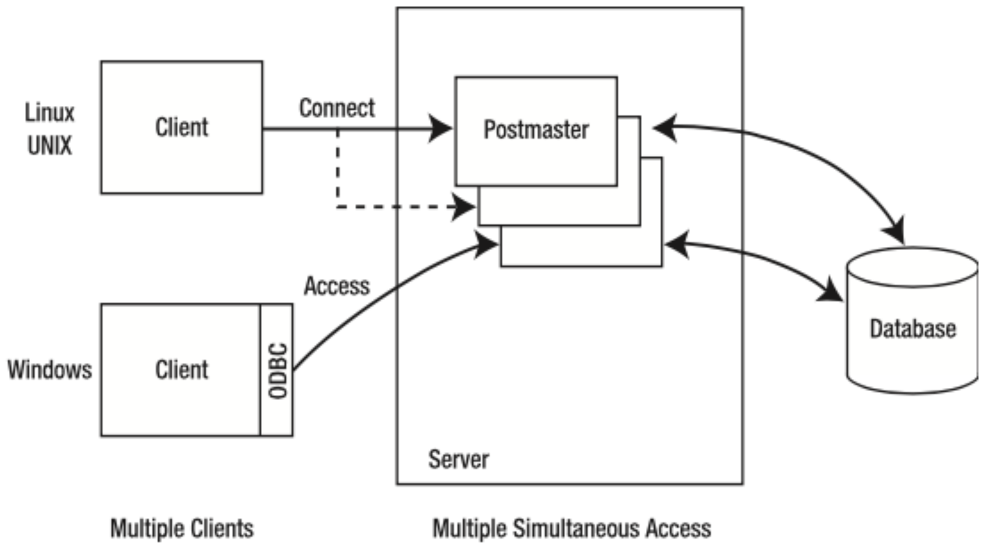


Figure 4 - PostgreSQL Architecture

Source: (MATTHEW, 2005)

This architecture allows to easily and efficiently maintaining the data's integrity. The main process can be run only once thus makes it impossible to scale the load on more physical machines. Nevertheless, there are some solutions (usually commercial) how to effectively scale the load, with a certain amount of compromises between speed, data consistency, or implementation difficulty. (MATTHEW, 2005)

PostgreSQL has many features including transactions, subselects, views, foreign key referential integrity, sophisticated locking, user-defined types, inheritance, rules, multiple-version concurrency control, and since version 8 the list expanded by native Microsoft Windows version, table spaces, ability to alter column types, point-in-time recovery and much more. It also supports most of the SQL:2008 data types such as INTEGER, NUMERIC, BOOLEAN, CHAR, VARCHAR, DATE, INTERVAL, and TIMESTAMP. It is even possible to store Arrays, Hashes, or large binary data within a field, which might be a complete image, sound or video file. PostgreSQL has many programming interfaces for all major programming languages (e.g. C/C++, Java, .Net, Perl, Python, Ruby, Tcl, etc.). However its features, it also has some limitations for data storing, summarized in Table 4. (MATTHEW, 2005) (PostgreSQL: About, 2014)

Limit	**Value**
Maximum Database Size	Unlimited
Maximum Table Size	32 TB
Maximum Row Size	1.6 TB
Maximum Field Size	1 GB
Maximum Rows per Table	Unlimited
Maximum Columns per Table	250 - 1600 depending on column types
Maximum Indexes per Table	Unlimited

Table 4 - PostgreSQL Limits

Source: (PostgreSQL: About, 2014)

3.6.3 Backups and Replication

Data backup is very important feature for every database system. Even with everything seamlessly working a failure can occur anytime. There are several potential threads for data loss. Typical cause of data loss or corruption is hardware malfunction. This is often minimized by removal of single points of failures (e.g. using multiple hard drives with RAID) and using various analytical tools to prevent hardware failures (e.g. S.M.A.R.T.). Another cause of data loss might be done by poorly written application code, which transform data to unusable form.

Avoid to these threads is essential, but having backups is also important for emergency situations when everything else failed. PostgreSQL offers full database backups and in later versions incremental backups in addition. The full backup needs the database service to be down for consistent data copy. Similarly it is possible to dump and store only some parts (tables) of a database. The incremental backup is also called Point-in-Time Recovery (PITR). It is recording all transaction that modifies the data to a special log file – write-ahead log (WAL). Starting point for these records is initial full backup to which the changes are recorded. With this log file it is then possible to recover any database state from the past. During the recording it is no need for any down time, because it is recording in parallel without interfering the main service. PITR also save disk space in opposite to everyday full backups. (YONG, 2012)

Within certain conditions, PostgreSQL can be replicated on more physical servers. This division can work in two modes – it can serve same data from more servers (load balancing) or backup all data and take over if the primary server fails (high availability). As mentioned earlier, it is not possible to seamlessly scale database service load. There are some solutions, which typically runs another database service and pulling the data from primary server. There are some issues with speed and/or data consistency when the database is not read-only.

In the terms of high availability usage, one server is the primary or master, which accepts all requests and other servers are secondary or slaves, which accept some requests. All servers have to have the same data. Master can modify them – it accepts read/write requests. Slave server can be a "warm standby" or "hot standby". Warm standby server doesn't accept any queries until it is marked as primary. Then it substitutes the failure, takes over and became a new primary. Hot standby server works similarly as warm with addition that it also accepts read requests even before it is marked as primary. (High Availability, Load Balancing, and Replication, 2014)

3.6.4 Rails with PostgreSQL

Majority of Rails applications include databases in its design for data storage. As mentioned earlier, Rails can work with many database systems (through the ActiveRecord class) including PostgreSQL. Definition for a particular database lies within database.yml file, which is located inside of the "config" folder of Rails project. This configuration file contains adapter name that is used to communicate with the chosen DBMS, database name, credentials, and other database specific configuration for each of Rails' environment. This file can be created manually or during the Rails project creation.

As mentioned earlier, the most important thing when comes to communication with the database is the database abstraction layer ActiveRecord. ActiveRecord is basically translating pure Ruby code into the database language and vice versa without the need of low-level database specific commands. The translation lies in transforming Ruby classes into tables, objects into records, and attributes into table columns of relational database. This process is called Object-Relational Mapping (ORM). ActiveRecord implements many methods for data manipulation without the need of raw SQL, yet it is possible to execute direct SQL queries.

Rails' migrations are an important part of the application-database cooperation. A migration is a file containing procedures, which are altering the database schema. The structure was already described earlier in this thesis. Manipulation with objects/records and perform of the basic CRUD operations can be easily done via ActiveRecord from a running application. Creation, drop or any modification of table properties has to be done via migration files, which are executed separately. For example, when creating a new model with attributes, it is necessary to create relevant migration to create table in the underlying database and execute it via rake task. Migrations allow changing the database schema incrementally during the development and subsequent maintenance, without the need of rebuilding the whole database or loosing data. (GAMBLE, 2013)

Not all of the PostgreSQL's features are natively supported by Rails framework, but there are many gems that make these available for the application. During the development of Rails framework itself, these gems are sometimes integrated within the framework. This happened with features of "postgres_ext" gem, which was adding ability of rails to save Array as a database field. In Rails 4, this ability was included without the need of external gem. PostgreSQL has the ability of saving Arrays and Rails made it very simple to implement. Within a migration file where the Array field is created, developer needs to specify an extra attribute "array: true". After the migration is run, the attribute can be filled with simple Ruby Array and then saved to database. It is also possible to query the content of Array in two ways – select records, which contain all specified values in the array or any of the specified values. (SANDERSON, 2013)

Another very useful ability of PostgreSQL is to store Hashes in field. The data type is called "Hstore" and it is available as an extension, which has to be included in the database. By executing SQL command "`CREATE EXTENSION hstore`" by a Rails migration, the database is all set. Within a creation of a particular field, the data type has to be set to "hstore". Afterwards, Rails can store and query hashes from the attribute. This extension is giving PostgreSQL a property of NoSQL databases, where attributes do not have to be specified before their usage. (BADAWY, 2014)

3.7 MongoDB

MongoDB is a powerful, flexible, and scalable general-purpose NoSQL database written in C++ programming language. From NoSQL family of databases it is defined as a document-oriented database. The concept of a document replaced row from relational databases. A document can contain various data types, arrays, embedded documents and even complicated hierarchical structures within a single record. Representing all relations of a document within a single record allowed MongoDB to be very easily scalable. This concept doesn't define any fixed schemas; therefore for example attributes (keys) can be added dynamically without any special modification of database. Data types and size of keys and values are also not fixed, which allows faster development of data model or whole applications.

Scaling can be distinguished between two types: scaling up and scaling out. Scaling up is improving performance by increasing the power of a single machine (server) running a single database instance. It is typical for relational DBMS. Scaling out is, on the other hand, improving the performance by adding more machines and distributing the load between them (see Figure 5). This is an advantage of MongoDB (and other NoSQL DBs) – it can be easily scaled out. (CHODOROW, 2013)

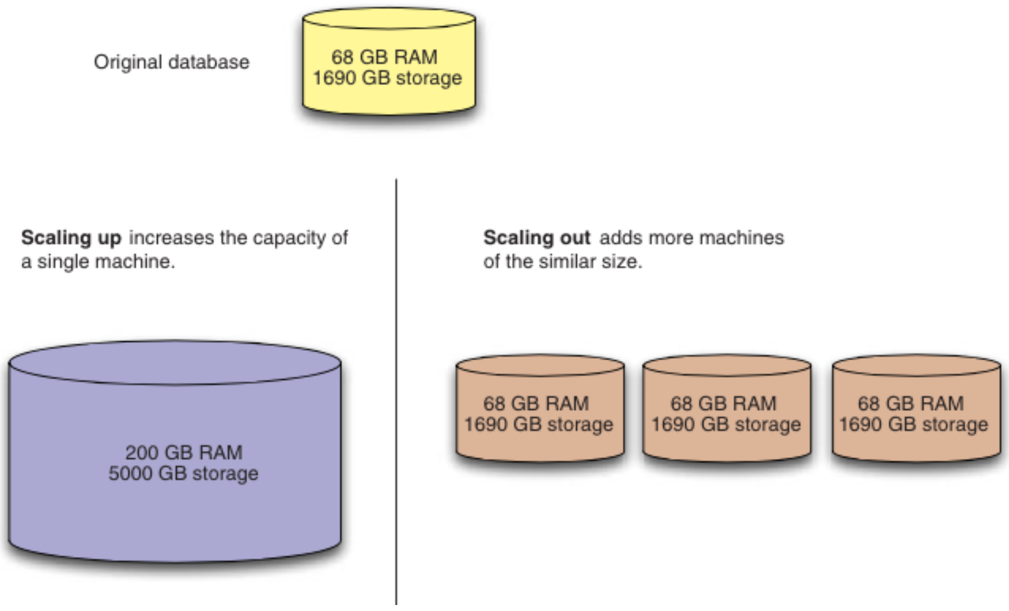


Figure 5 - Scaling types

Source: (BANKER, 2012)

3.7.1 History

The history of MongoDB is quite short since it was created quite recently. It began with foundation of company called 10gen in 2007. The company was founded by leading people from DoubleClick (the Internet ad serving services), namely by Dwight Merriman, Eliot Horowitz and Kevin Ryan. The company had several start-ups where they encounter the same problem with database scaling. Therefore they started to develop app engine, which should solve the problem. The app engine was called Babble with underlying database MongoDB. The project as whole was not that popular and had very few users. In 2009, the creators decided to tear off MongoDB out of the engine and made it open-source. This decision turned out to be appropriate and within a short period of time MongoDB had a solid user base and several contributing developers. In 2013, 10gen changed their name to MongoDB, Inc. to associate more with its flagship and main product.

MongoDB is nowadays an open-source licensed under the GNU-AGPL. This license defines that the source code is available for free and is open for contributions, however all changes made have to be openly published for the benefit of the community. MongoDB, Inc. provides also other licences for commercial usage. (CHODOROW, 2010)

3.7.2 Architecture

MongoDB is written in C++ language and it compiles on all major operating systems – Mac OS X, Microsoft Windows and most of Linux distributions. The development was influenced by relational databases – features, which were considered as well working and useful, were kept in MongoDB, others were removed or redesigned.

The main database service is provided by process “mongod”. This process receives commands over network socket using custom binary protocol. It can be run in several modes, depending on servers’ structure. Mongod process is easy to configure in the contrast with other relational databases. The memory management is handled by the operating system, where it should be done in the best possible way. Data files are simply mapped to virtual memory by calling system functions of the OS kernel (e.g. `mmap()`).

Speed and durability are important to every database. Write speed can be defined as a volume of inserts, updates and deletes that the database is capable of processing within a certain time frame. Durability refers to level of assurance that the write operations have been successfully done and stored. MongoDB has two possible settings regarding speed and durability, where each aims to prefer one or the other. First mode is called "fire-and-forget" and it considerably prefers speed to durability. In this mode, all write queries are sent to the database service without acquiring any response. It is faster, because application doesn't have to wait for any acknowledgement. However, with this technique there is a potential thread of not saving the data. When it is essential to ensure that the data was successfully and permanently stored, MongoDB offers second mode called "safe mode". With this mode, all write operations have to be confirmed as successful or cause an error. This mode prefers durability and can be configured to block the process until the modification is saved on all servers (when using replica sets).

MongoDB supports ad-hoc queries, which means that queries accepted by the system do not have to be defined in advance. This featured is well known from relational databases, where typically all well-formed SQL queries are accepted and evaluated by the database service. Main property of ad-hoc queries is the possibility of querying over arbitrary combinations of attributes. Ad-hoc queries are very powerful feature, however for better querability it is essential to use indexes as well. Once the database grows to a certain level it would be very slow for queries without index.

Indexes help to speed up the query process by minimizing the set of records with indexing query field(s). MongoDB implement indexes as B-trees, similarly to most of the relational databases. Indexes are crucial mechanism for optimising the system performance. There are many types of indexes supported by MongoDB – Unique, compound, array, TTL (time to live), geospatial, sparse, and text search indexes – each suitable for different optimisation of queries. (BANKER, 2012)

3.7.3 Replication

MongoDB provides similar data replication as PostgreSQL mentioned earlier. Topology of this replication is called "replica set". The set is composed of exactly one primary (master) node and one or more secondary (slave) nodes. Replica set distributes data across connected machines for backing up all data. It is also possible to scale reads from secondary nodes and in a case of failover of primary node the cluster automatically sets a new primary node from any of the secondary nodes, which becomes the new primary node. Thanks to this process the database service can continue serving all requests without any downtime. A simple scheme of automatic failover in replica set consist of three nodes is showed on Figure 6.

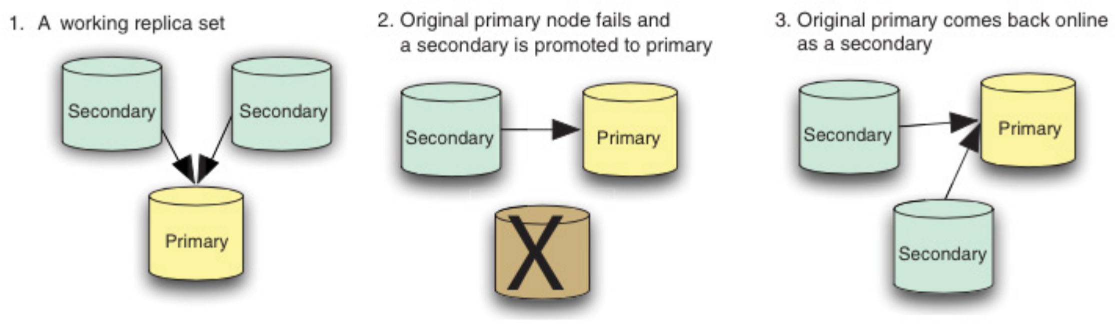

1. A working replica set
Secondary
Secondary
Primary
2. Original primary node fails and a secondary is promoted to primary
Secondary
Primary
X
3. Original primary comes back online as a secondary
Secondary
Primary
Secondary

Figure 6 - Replica set failover

Source: (BANKER, 2012)

3.7.4 Scaling

Scaling up is often easy solution to improve performance. However, upgrading a single physical machine has some limitations. At certain point it might get cost ineffective and later technologically impossible to upgrade more. Because of these reasons, scaling out is (in most cases) the better solution. MongoDB was designed to support horizontal scaling (scaling out) from the beginning. The scaling is called sharding in MongoDB. Every node that helps to scale the cluster is called a shard. Sharding enables evenly distribute the data across all connected nodes and add extra nodes when necessary to extend capacity. It is possible to scale out even relation databases, but the sharding logic have to be inside the application logic, which is not easy to configure. MongoDB's sharding is working seamlessly where the user (from the point of application logic) doesn't need to know if there are multiple or just single shard used.

A sharded cluster is composed of three main components – shards, mongos routers, and config servers. Each shard stores certain portion of the cluster total data. If the replication is enabled, replica set is considered as one shard. Every shard of any cluster can be accessed directly, yet data stored in it would be just a part of the total cluster content. To access the data as whole consistent database, it is necessary to use mongos routers. Mongos router process typically runs on the same server as the application and it handles the database requests. The application queries the mongos process, which takes care of retrieving the data from the corresponding shard. Mongos process is non-persistent; therefore it needs to store the cluster state to some durable place, where it can be retrieved anytime. This is what config servers are made for. Config server stores configuration of the cluster, locations of each database, collection, the ranges of data and other important metadata. Every time when the mongos process starts it fetches all metadata from the config server. Access all data stored in the cluster without these metadata would be impossible, hence there should be at least three config servers running on different machines for redundancy. Modifications of config servers' data is done via two-phase commits, which shows how important it is to keep it available. All mentioned components of a shard cluster are shown on Figure 7 as a working schema. (BANKER, 2012)

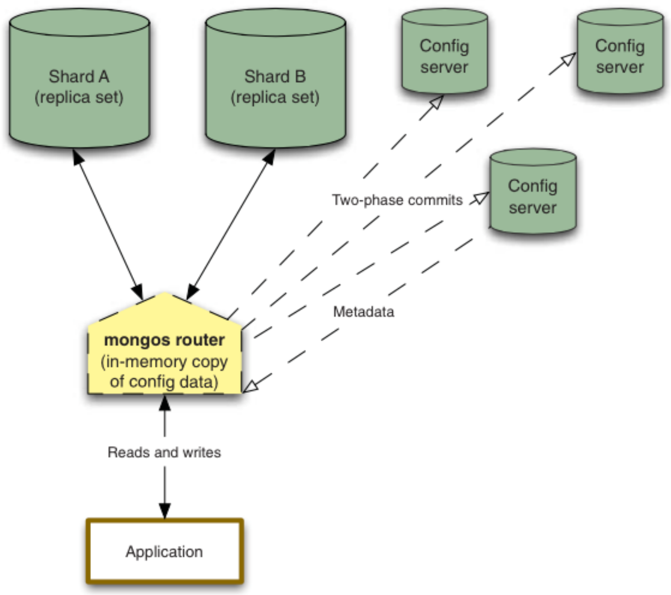


Figure 7 - Shard cluster

Source: (BANKER, 2012)

3.7.5 MongoDB Shell

MongoDB provides simple command-line tool (shell) for administrating the database and manipulating data. It can be run by executable “`mongo`”, which connects to specific mongod process that is running on the system. The mongo shell is JavaScript-based; where instead of SQL it uses JavaScript functions, expressions and JSON-like syntax. It is possible to execute all basic CRUD operations within this environment, creating, deleting and modifying databases and collections, analyse queries, read logs and change various configuration and administrative settings. Because of its nature, the shell is capable of executing native JavaScript functions such as loops, conditions etc.

Apart from mongo shell, MongoDB is bundled with few more command-line tools. On of the most important is mongodump and mongorestore, which are standard utilities for backing up and restoring data. Another similar tools provided are mongoexport and mongoimport that can export and import data in various exchange formats such is JSON, CSV, or TSV. There are more command-line tools shipped with MongoDB, such as `mongosniff` (translating operations sent do DB to a readable form), `mongostat` (providing various system statistics), etc. (BANKER, 2012)

3.7.6 CRUD Operations

MongoDB is capable of performing all basic CRUD operations via its powerful shell interface. Documents are stored in form of key-value pairs. This form is JSON-like and JavaScript shell is representing it this way. However, documents are stored as BSON format, which is a binary representation of JSON with additional type information. Documents are stored in collections (analogy with tables) from which they can be queried, modified, deleted, etc.

Each query has to specify over which collection should be performed. It can be also extended with various conditions for selecting a specific group of documents. Resulting selection can be then projected (show only defined fields) or modified to impose limits, skips or sort orders. MongoDB shell defines a several functions for these operations. A simple example of a query is on Figure 8, which selects all users whose age is greater than 18 and sorts the result in ascending order.

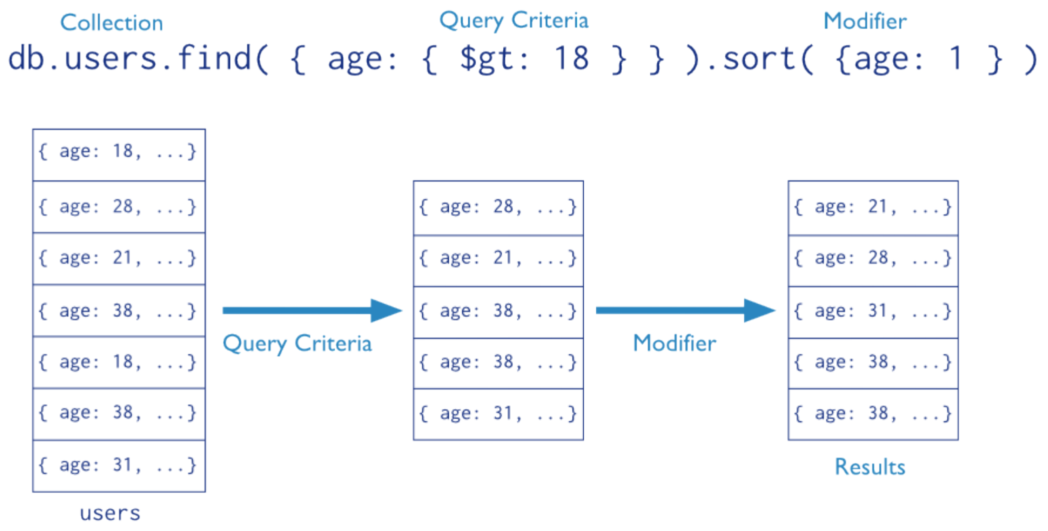
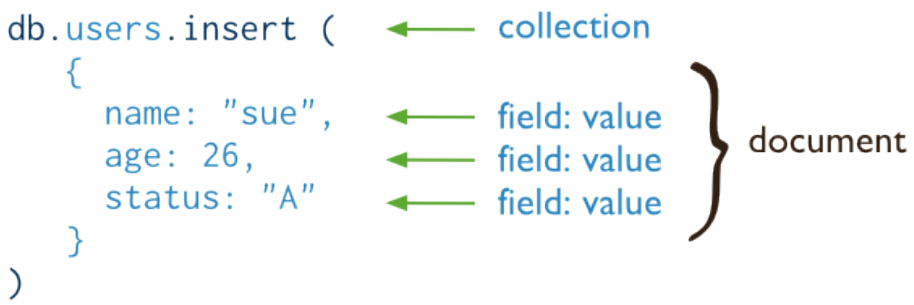


Figure 8 - MongoDB shell query

Source: (MONGODB, 2014)

Similarly to read operations above, MongoDB shell provides various methods to create, update and delete documents from collections. The syntax is analogous to read queries. Remove function accepts query criteria and deletes all matched documents. Update is alike remove with additional attributes, defining changes to matched documents. Inserting of a new document is shown on Figure 9, where a new user with name, age and status is created in collection named users. (MONGODB, 2014)



```
db.users.insert (
   {
      name: "sue",
      age: 26,
      status: "A"
   }
)
```

Figure 9 - MongoDB shell insert

Source: http://docs.mongodb.org/manual/_images/crud-annotated-mongodb-insert.png

3.7.7 Rails with MongoDB

Most applications have a database connection for data storage. Ruby (and Rails) has typically two main components for smooth access. First, which is compulsory, is to provide database driver for Ruby language. This driver then provides an interface for communication with database. It is typically a gem that developer needs to install and include into the application. This is the same as the "pg" gem that developer needs to include before communicating with PostgreSQL database.

Second component is optional, but for bigger applications almost necessary. It is usage of a DataMapper. In general, DataMapper is a process, framework, or library that maps two different sources of data. Obviously in this case, it is necessary to map MongoDB documents to Ruby objects. Rails ships with already mentioned DataMapper ActiveRecord, which is ORM. By its and name and definition it suits only for relational databases. Therefore, an external DataMapper needs to be installed and included for Rails application that communicates with MongoDB. Mongo DataMappers usually includes or depends on a MongoDB driver gem.

Most popular MongoDB driver is mongo-ruby-driver and is packed within gem simply called "mongo". It is licensed under Apache license, distributed for free and maintained by the community. It gives a basic ability to communicate with MongoDB database from Ruby or Rails applications. This means establishing connection, specifying collection, then executing raw queries and obtaining results in hash form. Mongo gem has some dependencies on "bson" and "bson_ext" gems, which are essential for working with MongoDB's BSON objects. This communication might be enough for simple applications, yet it is probably better to turn more abstract and include DataMapper as well, which will keep the possibility of executing raw queries as well.

There are two widely used MongoDB DataMappers for Ruby/Rails – MongoMapper and Mongoid. Both mappers provide more or less the same methods and functionalities with small differences. Basic methods for working with objects are close to ActiveRecord for easier implementation. After installation DataMappers generate configuration files and set up defaults for connection. Rails model class with Mongoid or MongoMapper simply includes module of the particular mapper (in opposite to inheritance when using ActiveRecord), which then dynamically creates a collection with name of the class. Attributes are defined by special keyword and data type definition. Example of a class definition is shown in Table 5 for both DataMappers, where the similar syntax is quite noticeable. (REGE, 2012)

MongoMapper	Mongoid
`class Person`	`class Person`
`  include MongoMapper::Document`	`  include Mongoid::Document`
`  key :name, String`	`  field :name, type: String`
`  key :age, Integer`	`  field :age, type: Integer`
`  key :height, Float`	`  field :height, type: Float`
`  key :born_on, Date`	`  field :born_on, type: Date`
`  key :interests, Array`	`  field :interests, type: Array`
`  key :is_alive, Boolean`	`  field :is_alive, type: Boolean`
`end`	`end`

Table 5 - MongoMapper and Mongoid model definition

Source: Own

3.8 Git

3.8.1 Version control

Git is a distributed version control system, which helps its users to track changes in documents, source code, or files in general. Version control system (VCS) records changes of these files in a time and makes possible to move back in history to its older versions. This tracking is applicable to all types of files and in addition it is possible to log users who made the changes. Versioning of files helps to prevent data loss by accident or other failures.

VCSs were evolving over a time, which led to several forms of nowadays. First type is a Local version control system. This system basically records all versions of files in history and stores it on a local machine. However, there is a potential thread of data loss, since the local machine is the single point of failure. Also it is not very well adapted for team collaboration. Centralized Version Control System is a next type, which moved the version storage from local machine to central server, where more users can work on the same project and track the changes. Nevertheless its teamwork improvements, the single point of failure remained – just moved to server. Users within a team can keep just single snapshots of their work. The ultimate solution supposed to be the Distributed Version Control System. This type combines all advantages of the previous two – it uses a central storage for better team work, and also mirrors the whole repository to clients where it remains as a backup and users can work even without connection to the central server. A simple scheme of distributed version control is shown on Figure 10. (CHACON, 2014)

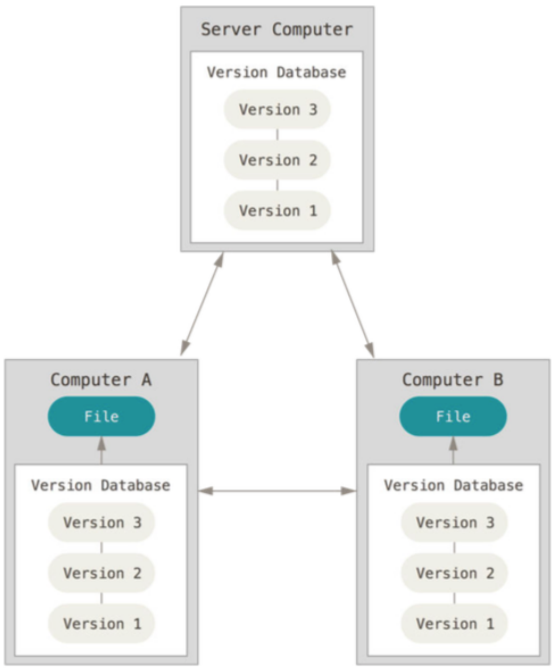


Figure 10 - Distributed version control

Source: (CHACON, 2014)

3.8.2 History

The history of Git is associated with Linux kernel development. Since 1991 until 2002 changes of the kernel were done via patch files or archived files passed around. In 2002 the Linux kernel project started to use distributed version control system called BitKeeper. This software solution was more than sufficient and it improved the development process of such a large community. Unfortunately, the company of BitKeeper decided to charge for using their software in 2005. The Linux kernel community and particularly Linus Torvalds (the creator of Linux) started to create their own DCVS based on features of BitKeeper called Git. Git was a success and perfectly substituted old VCS. It included several key features, which remained till nowadays, such as speed, simple design, strong support for non-linear development, fully distributed, ability to handle large projects (speed and data size), etc. Git is still being developed even though it was almost flawless at the beginning. (CHACON, 2014)

3.8.3 Basics

Git is fast and simple command-line tool. It has many third-party GUIs, but these typically implement only a subset of all Git commands, therefore it is recommended to use the CLI. Git can be installed on all major operating systems, or build from source.

In comparison to other VCSs (Subversion, CVS, Bazaar, etc.), Git tracks changes in a different way. It is not working with differences from origin file, but with snapshots of whole file. Basically, it creates a snapshot of all files within directory with each version, while maintaining data efficiency (unmodified files are just linked, etc.). From its distributed nature, users can work with Git locally without network connection and send their changes later when online. Almost all data are downloaded on a local machine, where user can browse the history of different versions and locally commit new modifications. Once connected, these changes are synchronized with central data store and it updates local files with changes of other members. Git provides advanced data integrity by creating check-sum hashes (SHA-1) of all files and refers to changes by these hashes. With this feature, it is impossible to make changes without knowing it or corrupt some files.

After installation of Git, user should set some basic credentials in order to label author of commits. To track changes within a specific project folder, a new Git repository needs to be initialised by command "`git init`". After executing this command a new folder ".git" is created where Git stores all metadata and object database for the project. It also stores current stage or index of the project.

There are two main states of files within Git project directory – tracked and untracked. Untracked file is not included in git repository and changes are not recorded. Tracked file is watched by Git and is included in last snapshot. It can be modified, unmodified, or staged. Unmodified files are those, which haven't been modified since the last snapshot in contrast to modified, which were somehow changed. User can mark modified files as staged (added to index) to be candidates for commit. It simply means that when user makes a commit, changes of these files will be added to the next snapshot. Lifecycle of files with different statuses is shown on Figure 11.

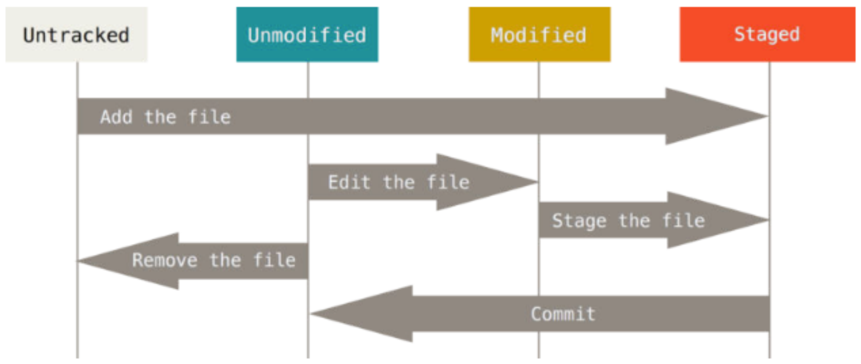


Figure 11 - The lifecycle of file statuses in Git

Source: (CHACON, 2014)

Once the changes are final and staged, user can commit these changes. Commit creates a new snapshot, which is saved as a new version in tracked history. Commit can be pushed to central database where it remains stored and available for other project members to download.

As many VCSs, Git provides branching. Branching means that users can diverge from the main line of development (master branch) and can work on the project without affecting the main line. Practically, it allows developing many features in different branches at the same time, while keeping the possibility of modifying the main branch without modifications from other branches. Branching is a non-linear programming technology. (CHACON, 2014)

3.8.4 GitHub

Git can perfectly work only on a single local machine, but most of its features lie within its distributed nature and network cooperation with other users. With network connection it is possible to download or clone remote repositories to a local machine. Git also support shared repositories, where users can collaborate on same project over the network. To enable this functionality on local machine, Git needs to be provided with "remote" which is basically a reference to location of a repository. Users with permissions, who added the particular remote to Git, can work together within the same repository and synchronize their work over the network. It is possible to setup own Git instance providing shared repository, however there are many servers providing storage for remote repositories already available. The biggest and most popular is called GitHub.

GitHub is a web application with ability of sharing Git repositories. User needs to create an account and then it is possible to create unlimited number of public repositories of open-source code. User can then set who can also write (therefore collaborate) to any of his repositories. These services are free and users have to pay only for private repositories, which are hidden and visible only to users selected by the owner. GitHub has several options of paid accounts for individuals or organizations.

GitHub was originally made for cloning public repositories from their users, but with growing popularity and user base; it added loads of new features. It offers an interface to repositories over three protocols (https://, git://, and git+ssh://). These functionalities are offered by other companies as well – Google Code, SourceForge, etc., but GitHub has a status of a social coding site. It combines social networking and programming in a form of watching other developers' activities, forking other repositories and modifying them, pull requests or offering some improvement to source code of other users, etc. People can also comment code, create tickets with encountered issues or create common wiki pages. Thanks to these, GitHub is very unique and well-know site for programmers around the world. (LOELIGER, 2012)

3.9 Bootstrap

Bootstrap is open source front-end framework originally created by Twitter employees – Mark Otto and Jacob Thornton. It was developed in 2010 as a Twitter Blueprint for internal purposes in order to unite various tools, plugins, libraries and styles used by Twitter's developers and to provide basic guidelines. After a year of internal usage, Bootstrap was released publicly as an open source framework (August 19, 2011). Its first version was basically CSS-driven, but it evolved and included various JavaScript plugins, images and fonts. Bootstrap has optional responsive functionality since version 2 for easier development on multiple screen sizes. The current version (Bootstrap 3) has the responsive functionality enabled by default with mobile first design approach. (SPURLOCK, 2013) (About Bootstrap, 2014)

3.9.1 Architecture

Bootstrap is distributed in two main forms – precompiled or source code. Precompiled Bootstrap contains compiled and minified versions of both CSS and JavaScript files logically divided in corresponding folders. The whole package is also equipped with fonts folder containing basic fonts and icons. The folder structure of precompiled version of Bootstrap 3.3.1 is shown in Table 6. The second form of Bootstrap's distribution is more comprehensive. It includes all precompiled CSS, JavaScripts and fonts with extra source of Less, JavaScript, documentation and examples. This package folders structure is shown in Table 6.

Precompiled Bootstrap	Source code
`bootstrap/`	`bootstrap/`
`├── css/`	`├── less/`
`│ ├── bootstrap.css`	`├── js/`
`│ ├── bootstrap.css.map`	`├── fonts/`
`│ ├── bootstrap.min.css`	`├── dist/`
`│ ├── bootstrap-theme.css`	`│ ├── css/`
`│ ├── bootstrap-theme.css.map`	`│ ├── js/`
`│ └── bootstrap-theme.min.css`	`│ └── fonts/`
`├── js/`	`└── docs/`
`│ ├── bootstrap.js`	`  └── examples/`
`│ └── bootstrap.min.js`	
`└── fonts/`	
`├── glyphicons-halflings-regular.eot`	
`├── glyphicons-halflings-regular.svg`	
`├── glyphicons-halflings-regular.ttf`	
`├── glyphicons-halflings-regular.woff`	
`└── glyphicons-halflings-regular.woff2`	

Table 6 - Bootstrap packages

Source: Own

As briefly mentioned, Bootstrap can be downloaded as a Less source code if developer wants to use pre-processor. There is also Sass version available, which was ported from Less for users' convenience. Bootstrap website offers online customisation of the current version, where user can change various settings, mainly values of Less variables. Resulting package is then compiled and downloaded. (Getting started, 2014)

Main advantages of Bootstrap are predefined CSS styles, ready for use by assigning classes to particular elements and predefined graphical elements such are buttons, form groups, lists, navbars, etc. Bootstrap also includes JavaScript plugins with wide functionality available by simple calls (modals, scroll spies, dropdowns, tabs, popovers, alerts, etc.). All of these components are made to adjust their properties to fit the used device and its screen size.

One of the most significant features of Bootstrap is the Grid system. It is used for creating pages with help of series of rows and columns, which hold the content. Horizontal division of the content is done via rows and vertical division is done using columns within these rows. By default, each row has up to 12 columns, depending on their width. Since Bootstrap 3 incorporates responsive design, it is possible to define different grid for various screen sizes; using multiple classes for columns (see Table 7).

	Extra small devices (<768px)	**Small devices (≥768px)**	**Medium devices (≥992px)**	**Large devices (≥1200px)**
Grid behaviour	Horizontal at all times	Collapsed to start, horizontal above breakpoints		
Container width	None (auto)	750px	970px	1170px
Class prefix	.col-xs-	.col-sm-	.col-md-	.col-lg-
# of columns	12			
Column width	Auto	~62px	~81px	~97px
Gutter width	30px (15px on each side of a column)			

Table 7 - Bootstrap screen sizes

Source: (CSS, 2014)

There are 4 main classes for columns; each defines its width according to screen size. Every column class for particular screen size is appended with number (by default from 1 to 12), which sets how much spaces will this cell occupy within a parent row – e.g. to divide the row within 4 equal parts, each column class has to have size of 3 – 4x3=12. Example usage of Bootstrap's grid system is shown on Figure 12. Columns can be also moved to right by defining offset class in the same scale and manor as columns – e.g. column moved to right by size of one column is supplied with class "`.col-md-offset-1`". (CSS, 2014)

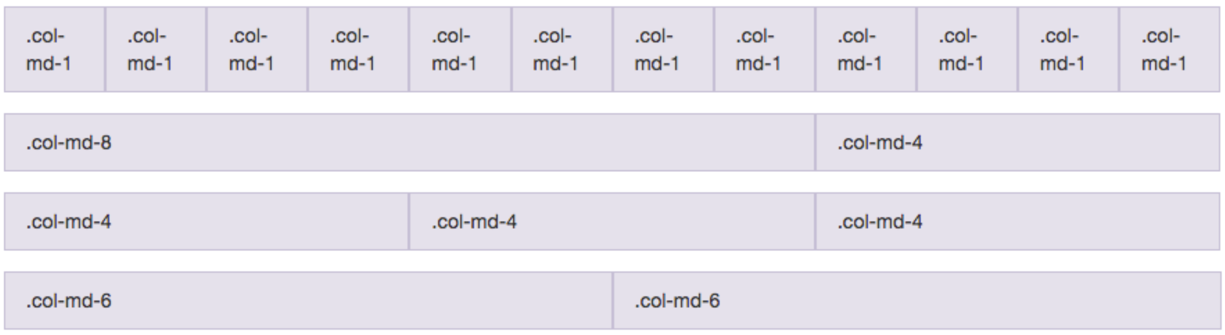


Figure 12 - Bootstrap Grid System

Source: (CSS, 2014)

4 Practical part

The aim of the practical part is to create web application for a company from building industry. The company offers various kinds of reconstructions, building smaller objects or revitalisation of gardens. The desired web application should create a simple way how to communicate with customer and handle majority of paperwork connected with any project. Customers should be able to list all of their projects and their details such as progress, budget, future plans, etc. They will also be able to confirm any changes of budget or plan, or add some requirements for the project.

4.1 Assignment

The management of the company wanted a custom tailored application to suit all their needs, requirements and future development as the demands might change. The initial application solution should be very simple and iteratively improved as necessary. Other software on the market was either too complicated or not fitting the desired needs. It was also estimated, that creating and improving web application from scratch will be cost effective. The company will own the application source code and can sell it further to increase its profit.

The company and the developer agreed to conduct a meeting in order to specify the basic structure and future usage of the application. This first meeting resulted in summary of requirements and resulting assignment. During the session some basic diagrams were created as a draft for the application. It was decided to use Bootstrap for application design and layout.

The first part of the assignment was summarized by simple use case diagram. It was stated that the application would be used by three participants' roles - customer, manager and administrator. Administrator and manager have the same use cases since they both operate under the company side towards to the customer. However, the administrator has few more use cases, and is privileged to perform tasks that manager is not. This relation was formalised as inheritance, where administrator is inheriting manager's use cases and adds few of his own.

The customer uses the system for submitting new concepts, which are the origin of any project. Once a project based on the concept is created, customer can approve individual parts of the project plan (jobs), insert additional requirements for the project, or just to check progress of the project. All of these actions can be done after the customer creates his own account and logs in.

The manager also has the option to create account with credentials, however it has to be assigned with manager role by the administrator. Manager can also submit a concept, but more importantly he can create projects and all of its parts. Subsequently, manager can record the progress of individual jobs, and update all of its attributes.

Lastly, the administrator (among all of the manager's use cases) can change users' roles and perform all basic CRUD operations on most of the records in the application. Listing of all of these use cases was summarised and is shown on Figure 13.

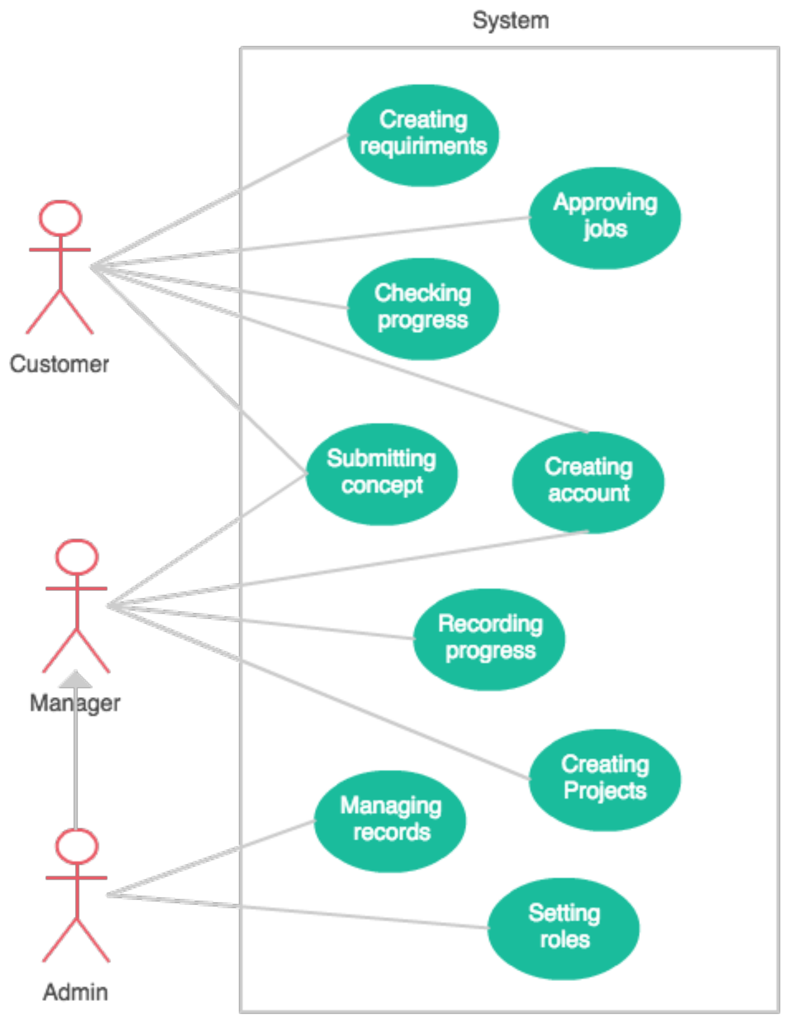


Figure 13 - The application use case diagram

Source: Own

The use case diagram is very useful when specifying requirements of a client. Even people without any IT knowledge can understand this diagram and it helps to define what should the application be capable of.

The next part of the assignment is formalising the data model. This task requires experiences with software development. In case of this thesis, the developer created the data model using class diagram and then explained and finalised the diagram with the client from the company. The resulting diagram is represented on Figure 14.

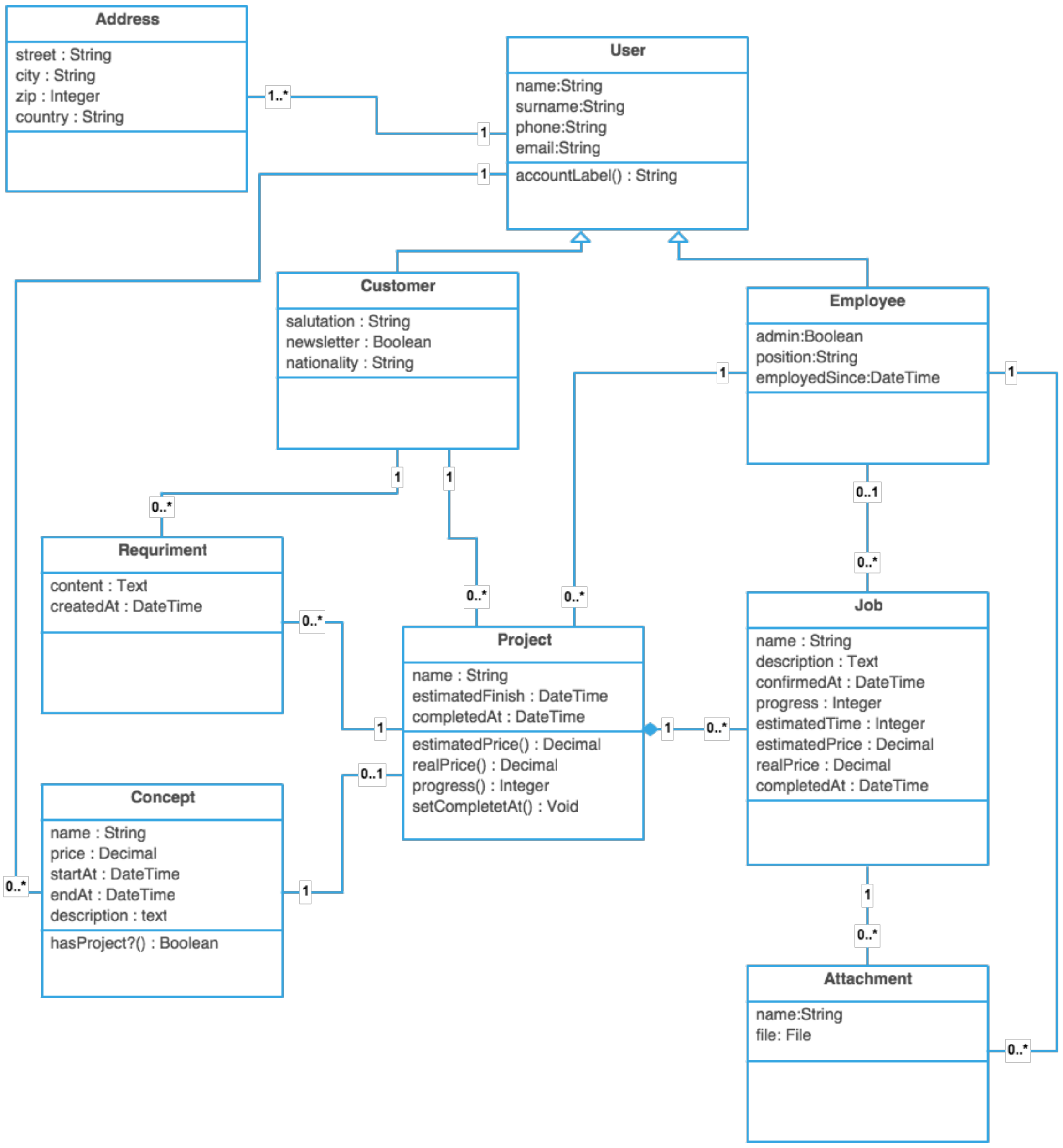


Figure 14 - Application class diagram

Source: Own work

Additionally, all objects contains timestamps attributes `created_at` and `update_at` by default. These attributes were omitted in the class diagram for better readability.

4.2 Prerequisites

As already mentioned, the practical part of this thesis will cover the development process of the web application. This task requires some prerequisites. Technologies (software and hardware), which were used for the development, are as follows:

- Apple computer with Mac OS X Yosemite operating system
- Ruby 2.2.0
- Ruby on Rails 4.2.0
- RubyGems 2.4.5
- Ruby Version Manager (RVM) version 1.26.9
- Sublime Text 2
- Creately.com (online diagramming tool)
- PostgreSQL 9.4
- MongoDB 2.6.7
- Git 1.9.3

4.3 Development

4.3.1 Environment setup

In order to start the development process, a system environment has to be properly setup. For the beginning, it was necessary to install Ruby interpreter, RubyGems, PostgreSQL and some editor for code writing. Ruby and RubyGems were installed via RVM with Rails gem included. The installation can be simply done by running a command from rvm.io ("`\curl - SL https://get.rvm.io | bash -s stable –rails`") in shell (bash, command line). With addition of any text editor (in this case Sublime Text 2), the Rails development can be started with SQLite database or without any database. The developed web application needs a better database; therefore next step is to install PostgreSQL. In case of Mac OS X operating system package manager Homebrew was used for installation of PostgreSQL, where is minimal need for configuration. Once these components are installed the actual development in Rails can begin.

The whole process begins with creation of a new Rails application. It was done via command "`rails new buildit -d postgresql`" in shell. This command created a new web application with name "buildit" (commercial name Build IT), setup its database to PostgreSQL and run "`bundle install`" to install all basic gems. It is possible to run the web server of this project after these steps; however opening it in browser window at this state causes an error because there is no database created for this project. Developer has to run "`rake db:create`" from the application folder to avoid this error. If there is no output to shell after this command, everything is correctly configured and databases for each environment were created (buildit_development, buildit_test and buildit_production). At this state it is possible to flawlessly run the web server ("`rails s`") from the application folder, open a browser on address "`localhost:3000`" and see Rails' welcome screen with basic environment details.

4.3.2 Git

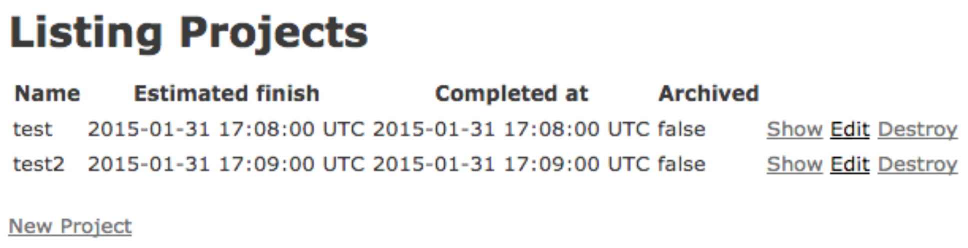
The next part of development, after the application had been initialized, was to create models with attributes and associations between them according to project specifications and the class diagram (Figure 14). Before the following programing, it is the best practice to initialize a new Git repository to track changes within the application folder, thus preventing any potential data loss. At first, "`git init`" was called to create an empty Git repository. However, all files from the application folder were labeled as untracked and not included in the new repository (checked by "`git status`"). To track all changes in history, it was necessary to add all files to Git's index by executing "`git add --all`". All files were added to the index and labeled as new, ready for commit. After these actions the repository was committed for the first time by "`git commit -am "init"` ". To use all advantages of Git and its features, it was important to connect this local repository to a Git server – in this case GitHub.

GitHub offers several plans for personal account, where user can choose how many private repositories are needed. It was decided to use a free plan, where all repositories have to be publicly visible. This may be changed later, but at the current state of development it is the optimal solution. The next action after the registration process was to add the repository to GitHub via its web interface. The online repository needs to be synchronized with the local repository. Adding credentials to the local Git installation –

commands `git config --global user.name “NAME”` and `git config global user.email “EMAIL”` prepared the local environment. Authentication of user was done by uploading public key of the local machine on GitHub server. Consequently, a remote origin was added to local repository by executing `git remote add origin git@github.com:<USER_NAME>/buildit.git`. Last step of this synchronization was to push the content from local machine to the server (`git push -u origin master`). All changes within the application folder have been tracked and could be easily pushed to remote server, where they can be displayed in GitHub web interface.

4.3.3 Data model

The development will focus on data model implementation in this section. According to the assignment there were a few classes defined. The Project model was the first one to implement, since many other models are dependent on it. Scaffold generator was used to create relevant controller and views for the model at the same time. Attributes of the model with their respective data types were passed to the scaffold generator. The command was as follows: `rails g scaffold Project name:string estimated_finish:datetime completed_at:datetime archived:boolean`. This action created a working MVC fragment for the projects, with basic views and controller with actions. The index view (with two added records as an example) located on running server at `localhost:3000/projects` is shown on Figure 15. A necessary condition to make this example working was to run pending migrations (`rake db:migrate`), which were created by the generator. The migration file contained instructions for creating a database table “projects” with passed attributes.



Listing Projects

Name	Estimated finish	Completed at	Archived			
test	2015-01-31 17:08:00 UTC	2015-01-31 17:08:00 UTC	false	Show	Edit	Destroy
test2	2015-01-31 17:09:00 UTC	2015-01-31 17:09:00 UTC	false	Show	Edit	Destroy

New Project

Figure 15 - Default index page of projects

Source: Own

According to the class diagram from the assignment, the Job model was the next to create. The procedure was basically the same as for the Project, only with different name and attributes. However, in case of Job, there is a relationship to Project class, which means it has to contain a reference to it. This was achieved by supplying extra attribute to the scaffold generator, which defines the association - `project:belongs_to`. This extra attribute added column `project_id` with index on it and also foreign key constrain to ensure referential integrity on the database level. The foreign key constrain validates if a referenced record from the other table actually exists. If not, it throws an exception, which originates at DBMS. The generator added the reference to Job class definition as `belongs_to :projects`. To use this association bidirectional, it was necessary to manually add `has_many :jobs` to Project class definition.

The next model to implement was Concept. Concept has an association 1:1 with Project. The assignment proposed that there should be no Project without a Concept. This led to implementing the reference column within the Project - Project belongs to Concept and Concept has one Project. After the scaffold generator created MVC for the Concept, its migration had to be manually added with two extra lines defining this association - `add_reference :projects, :concept, index: true` and `add_foreign_key :projects, :concepts`. Once the migrations were executed it was again necessary to add the relationship definition to respective class definitions.

Furthermore, another association with Requirement model was added to the Project model. Realization of this was similar as the Job model. The scaffold generator was provided with association `project:belongs_to` as well as with other attributes. Attachment model was analogously generated. It has an association to Job model, which was denoted by `job:belongs_to` within the generator command. Both models had to be delegated within the class definition of the other side of the association, similarly as it was done previously.

At this point the application data model is complete, yet without any users involved. As the initial diagram shows, there should be a superclass User with two subclasses inheriting from it - Employee and Customer. The authentication of any user working in the application had to be done and it was decided to use external gem called Devise. Repository commits were done during the development process to leave the option of reversing changes and track history. Before installation or generation of new files the project was committed to local VCS and pushed to GitHub as well.

4.3.4 Authentication with Devise

Devise is a gem that was used for authentication of users (registration, login). This powerful gem solves all basic problems connected with authenticating of users. It has wide variety of functions, but for purpose of this web application only few of its modules were used – Database Authenticable (credential storage in DB and password encryption), Registerable (register process of users), Timeoutable (session expiration after inactivity of user) and Lockable (account lock after several failed sign-in).

Once the gem is included in Gemfile and command "`bundle install`" executed, it is included in the application. Consequently, it was necessary to initialize Devise with database table of users, where credentials and other attributes can be stored. Devise gem provides generators to do so. A command "`rails generate devise:install`" was run for basic initialization of the gem followed by "`rails generate devise User`" for creating and linking to model User and its relevant database table. The generator created a new model with list of modules used, which was modified according to requirements mentioned earlier, and a migration containing instructions for creating a new relational database table of users. This migration file was also altered to match desired attributes. The last configuration of Devise gem was to extract views from the gem in order to edit their appearance. This was done by calling another generator "`rails generate devise:views`".

4.3.5 Users

Before creating any associations with other models it was vital to solve the User superclass-subclasses relations. There were two subclasses defined (Employee and Customer) inheriting from User class. This relation was settled due to majority of similar attributes, yet different in few. Ruby on Rails offers inheritance called Single-table inheritance (STI), which maps all subclasses to the same database table and distinguishes them by a particular string attribute with model name. The table contains union of all subclasses' attributes, where different subclass instances use chosen attributes. Rails code then implements separate classes with inheritance as described in theory earlier. Parent class User was already defined therefore it was necessary to extend it with string attribute named "`type`", where subclass indication can be saved. Since all records are saved to one table (STI), it needed to be extended by union of all attributes. Addition of these columns can be done in a separate file but it was decided to include it into User migration. The desired data model links User or its subclasses to all other models. There were 6 associations, which had to be created. All of them are 1:N relations where the reference was stored on the other side of the association than the User was. This procedure included adding extra column to other tables with constrain of foreign key to users table. The actual implementation of these links was done within the migration file for creating User model (table users) as follows:

```
#add user to concept
add_reference :concepts, :user, index: true
add_foreign_key :concepts, :users

#add customer to project
add_reference :projects, :customer, index: true
add_foreign_key :projects, :users, column: :customer_id
#add customer to requirements
add_reference :requirements, :customer, index: true
add_foreign_key :requirements, :users, column: :customer_id

#add employee to project
add_reference :projects, :employee, index: true
add_foreign_key :projects, :users, column: :employee_id
#add employee to job
add_reference :jobs, :employee, index: true
add_foreign_key :jobs, :users, column: :employee_id
#add employee to attachment
add_reference :attachments, :employee, index: true
add_foreign_key :attachments, :users, column: :employee_id
```

The code does almost the same for each table, where `add_reference` method added a column named as the second argument with extension of “`_id`” at the end to a table named as the first argument. At last, the method also created an index on the column. The second method added a foreign key constrain to ensure referential integrity. Models Employee and Customer do not have database tables because of the Single-Table Inheritance; therefore foreign key is referenced from users table in both cases.

After all pending migration were run, the application is ready for adding Employee and Customer models. The scaffold generator was used again with related attributes (already defined in migration for User model) and definition of the parental class - `rails g scaffold <NAME> <ATTRIBUTES> --parent User`. The generator run for Employee and Customer, and created the basic MVC structure with User as a superclass.

The last model to implement was the Address. It can contain addresses of Customers and Employees as well. Since it has a relationship to both, it was connected with their parent class User. The creation was again done via scaffold generator specifying the relationship by `user:belongs_to` passed to the generator command. After running the migrations, the data model was completed according to the initial design. It was again important to commit all changes to Git repository to track the history of the whole project.

4.3.6 Bootstrap

The next major part of the application development is the user interface and desired behaviour. This suggests that it is focused on implementation of views, controllers and their communication with models. The user interface was developed using Bootstrap framework, which handles various requirements by default - such is responsiveness, basic graphical design, gird layout, etc. There are gems for Rails that includes Bootstrap, wraps it, and offers various methods and generators with its functionality. However, from request of the assigning company and possible future separate UI development, it was decided to use stand-alone static files of Bootstrap. The main advantage is that the future design changes can be done without any server-side development but only within static CSS files.

Bootstrap files were obtained at the official website of this framework, particularly at Customize page (http://getbootstrap.com/customize/). This page enables developers to make changes to various components of Bootstrap. In order to create unique, distinguished and custom design, some of the visual components were modified. After these modifications a compiled static version was downloaded and ready for the application inclusion. Downloaded zip file contains three folders to incorporate to the application. The first is the CSS file, where its minified version was put to `app/assests/stylesheets/` folder of the project. Analogically, minified Bootstrap JavaScript was copied to `app/assets/javascripts/` folder and finally the whole folder called fonts from the downloaded file was put to `app/assets/images/` folder. Because the application is set to include all assets by default, there was no need for any modification and the Bootstrap was loaded into the application.

4.3.7 Design

There are two parts of the application that are always displayed on every page - header section and footer section. The header section contains a navigation bar (navbar) with main links throughout the application. According to the assignment, the navbar should always stay on top, which is fixed-top class of navbar in Bootstrap terminology. The second part (footer) was designed to contain only single line of text with copyright, year, developer and company name. Both of these parts were implemented within `app/views/layouts/application.html.erb` file, which contains a layout rendered with every page. Figure 16 shows the navbar and the footer without the content.

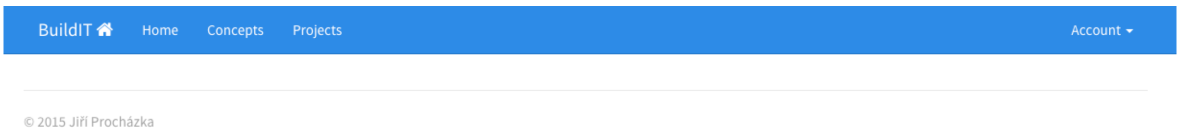


Figure 16 - Navbar and footer

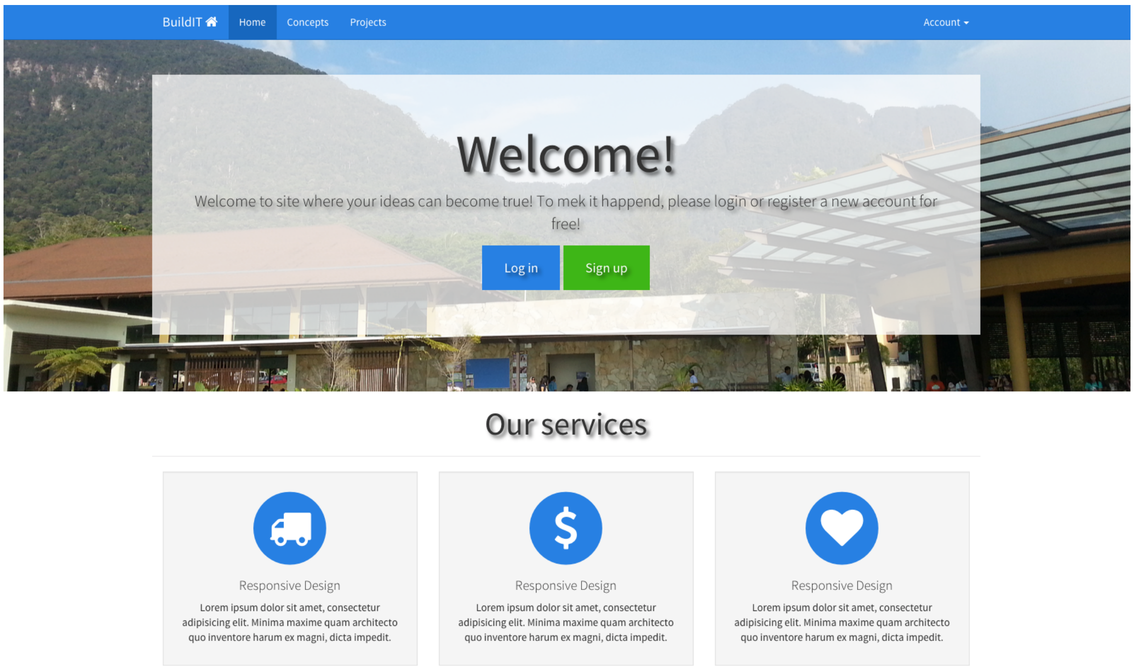
Source: Own

The design development (and its implementation) was separated into sections according to individual pages of the application. Every web application has typically some home page or welcome page. This was also included in this web application. Since there were only views generated by scaffold generators, it was necessary to generate the home page. By initial requirements, it was defined that home page should not interact with database. Therefore, a controller generator was used to create home controller with one action and corresponding view - `rails g controller Home index`. Because this page was going to be the main page (or landing page), application routes were edited to match the root path with this controller.

Composition of this page was predefined by the company's requirements. It basically has to contain a big announcement text box with quick links for signing up and logging in with three columns of provided services or other marketing material. The announcement was implemented with Bootstrap's jumbotron class extended with semi-transparent background. The whole upper section under the announcement was equipped with a static background image. The columns below were put within well classes. The home page was made responsive for smaller screen resolutions, where the content simply shrinks, text wraps itself and cells from the columns are merged into one column with full width of the screen.

The background image underneath the jumbotron box had to use full width of window. Because of this requirement, the whole section had to be put into a partial and rendered directly from application.html.erb layout instead of home page views. The layout is rendered on every page, thus the partial had to be rendered only if home page is active. This was done by a simple condition checking the active controller.

The application was enriched by set of icons called Font Awesome. It was included with CSS file with classes defining individual icons, their various sizes and styles. This feature was used for header part of columns on the home page for better user experience and graphical design. The created home page is shown on Figure 17.



Welcome to site where your ideas can become true! To mek it happend, please login or register a new account for free!

Lorem ipsum dolor sit amet, consectetur adipisicing elit. Minima maxime quam architecto quo inventore harum ex magni, dicta impedit.

Lorem ipsum dolor sit amet, consectetur adipisicing elit. Minima maxime quam architecto quo inventore harum ex magni, dicta impedit.

Lorem ipsum dolor sit amet, consectetur adipisicing elit. Minima maxime quam architecto quo inventore harum ex magni, dicta impedit.

Figure 17 - Home page

Source: Own

Since the home page contains login and sign up buttons, it was appropriate to continue the design development by modifying login and sign in forms. These forms were created by Devise gem and extrapolated by additional command to views folder. The login form needed only few modifications of the design - usage of the correct classes and proper elements’ nesting with usage of Bootstrap. The whole form was put within a panel and centred. Fields from the original form were kept - email and password. The form also contains a submit button to send the credentials. Additionally, a sign up button was added to this form to give an easy way to create a new account. The login form is displayed on Figure 18.

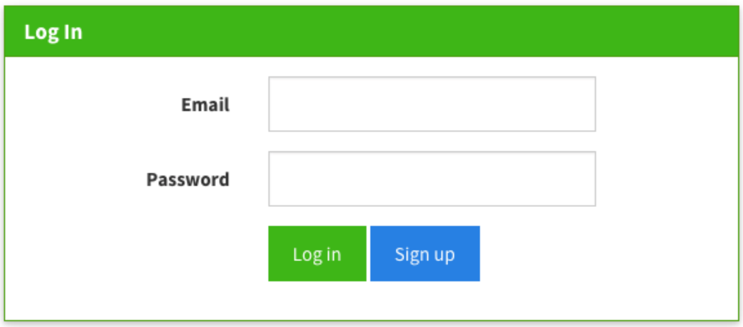


Figure 18 - Login form

Source: Own

The sign in form was also modified in a very similar way of design. However, this form had to contain more attributes, according to the User model. Attributes name, surname and phone were added to the form, which kept the same design as login. Because of the new attributes, the application had to be adjusted in a controller. Every controller contains a method that permits a set of attributes, which can be passed to the model (from security reasons). In this case it was necessary to include these new attributes for all controllers from which users can sign up. The method was put in the ApplicationController file. Additionally, attributes name and surname were made mandatory in the User model definition.

Every new user created with sign up form had been a User class instance. This behaviour had to be changed to create Customer class instances by default. A new method setting the `type` attribute value to "Customer" and using it as callback `before_create` implemented the desired behaviour.

At this point, the creation of new user (customer) works for the mentioned attributes only. Furthermore, it was also desired to include address to each user within the sign up process. Address model had been already generated and it is ready for use. It was necessary to add address fields to sign up form and since it is a different model the fields have to be within a Rails block `f.fields_for :addreses, Address.new`, where the f is the superior block variable. User model also has to contain additional line `accepts_nested_attributes_for :addresses`. The last condition for this to create both objects (Customer and Address) via sign up form is to permit these nested attributes in the ApplicationController as was done with the User. The final appearance of the sign up form is captured on Figure 19.

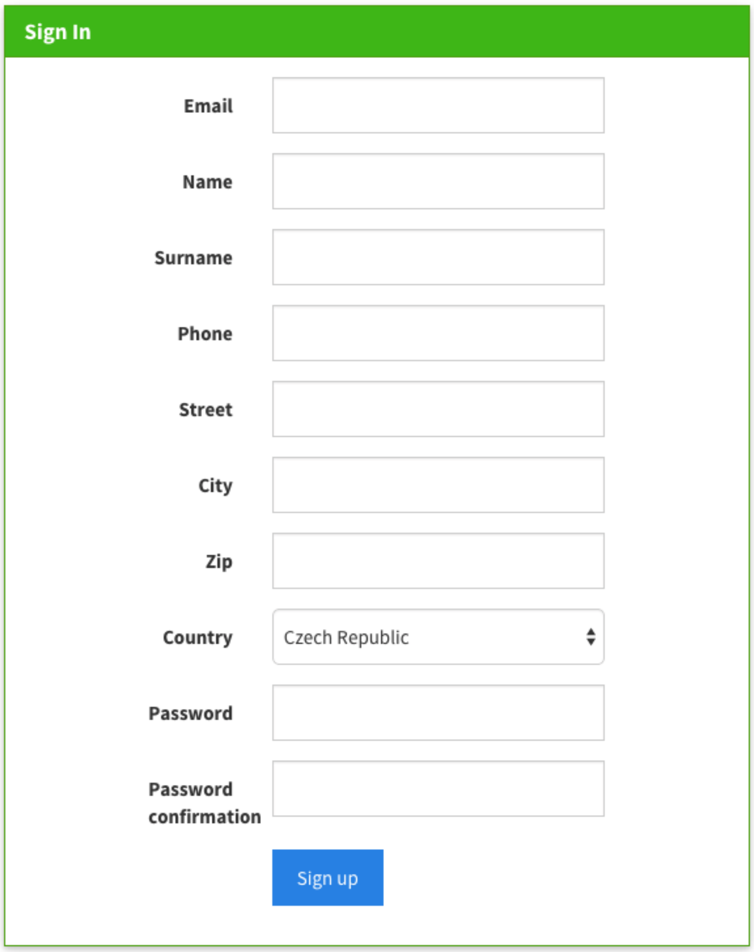


Figure 19 - Sign up form

Source: Own

After implementing the login and sign up logic and the design, the navbar was enhanced with user's dropdown menu. It is a simple dropdown list with links for user's account related matters. Before the user is logged in it shows list of two actions - login and sign up with "Account" as a label button. After login the user can see his name and surname as a label with dropdown menu containing link for editing account and logout link. The name and surname label was provided by an instance method created in User model - it returns a string composition of both attributes. The dropdown menu was placed on the right side of the navbar and both of its states are shown on Figure 20.

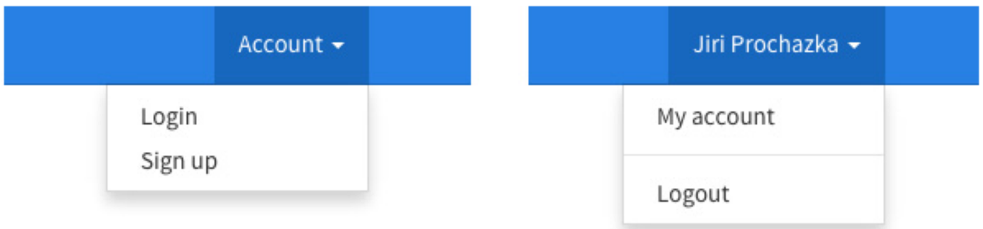
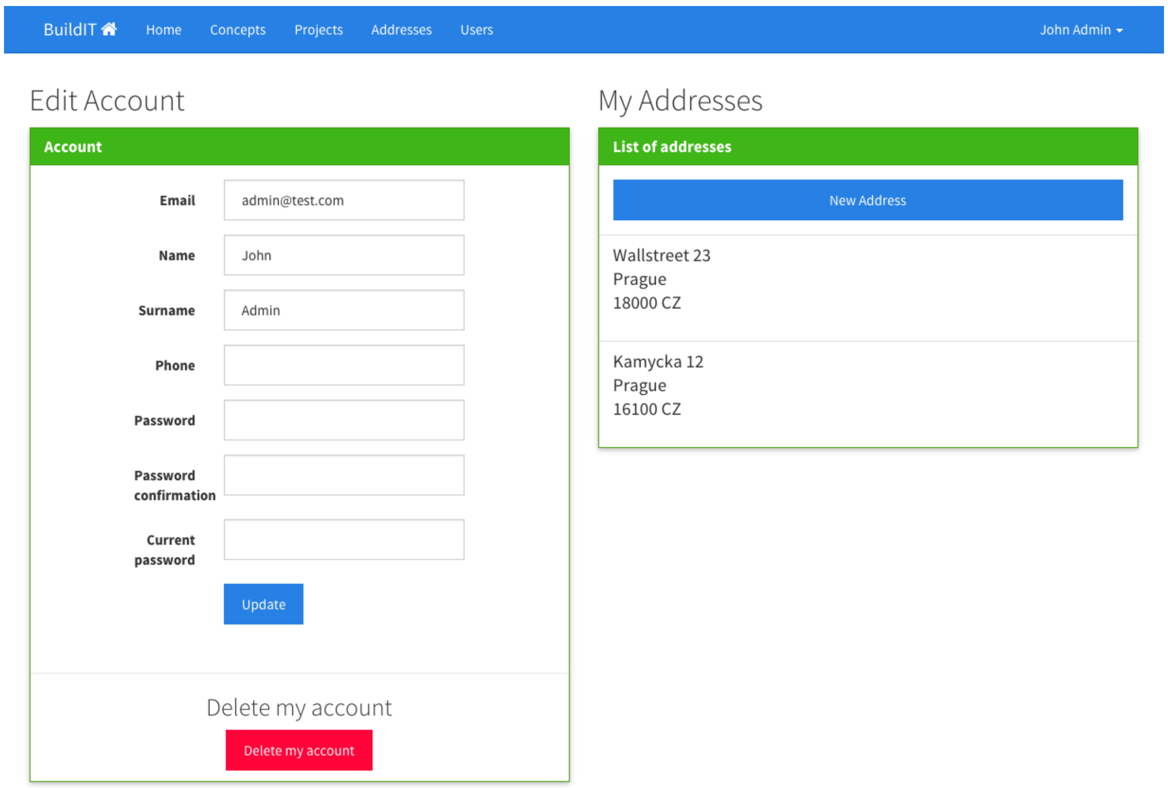


Figure 20 - UI Account dropdown menu

Source: Own

My Account page was generated by Devise gem and it serves for editing user's attributes. By default it contains only email field, fields for changing password and link for account delete. This form was again redesigned in similar way as the sign up form with addition of delete account action and user's addresses management. It was divided into two panels where first contains account fields and delete action and second contains list of user's addresses and action to create a new one. Individual addresses were made as links to edit form of them. Implementation of the page is displayed on Figure 21.



Edit Account

Account

Email	admin@test.com
Name	John
Surname	Admin
Phone	
Password	
Password confirmation	
Current password	

Update

Delete my account

Delete my account

My Addresses

List of addresses

New Address

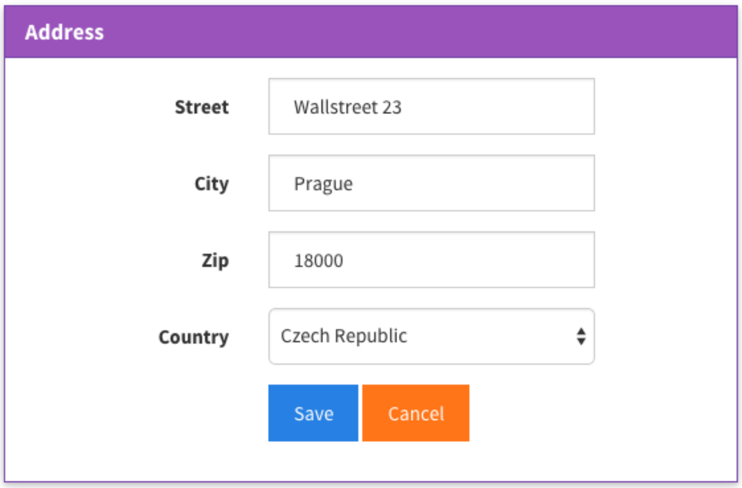
Wallstreet 23
Prague
18000 CZ

Kamycka 12
Prague
16100 CZ

Figure 21 - My Account page

Source: Own

Next set of pages to implement was the Address model, since the account page links to its forms. The account page has a list of addresses with links to edit form and direct link for creating a new address. These two forms use a same form partial, which is then rendered inside of the actions' views (new and edit). The form was redesigned similarly as other forms to keep consistency in the application. It was also enhanced by a list for the country attribute. An external gem called "country_select" was used to prepopulate the select field. The helper method of the gem took preferred countries and preselected country as input arguments. The show view and action of address were removed, since there was no need for them. The address form appearance is shown on Figure 22.


Address

Street	Wallstreet 23
City	Prague
Zip	18000
Country	Czech Republic

Figure 22 - Address form

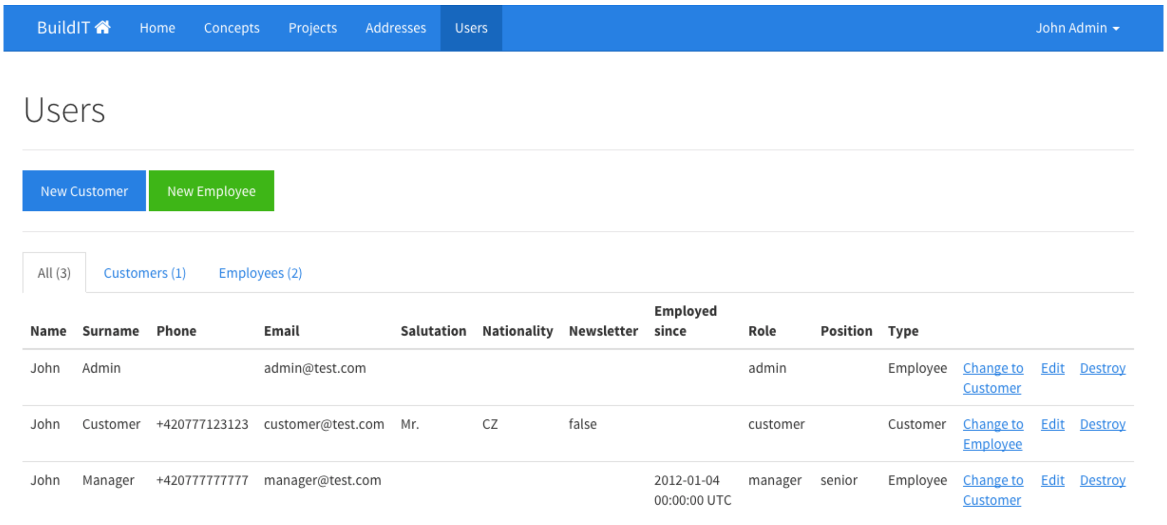
Source: Own

The index view and action were kept in order to list and manage all addresses at one place. Access to this list will be restricted when authorization rules apply. The view was arranged as a table containing all attributes in columns and actions at the end. Since the list of all addresses is considered as one of the main modules the link for it was added to the navbar. The AddressesController had to be modified to work as desired. The show action was removed from the controller as mention earlier. Next modification was about creating a new address. New address has to be assigned to some user to be semantically correct. This assignment was done within the create action in the AddressController, where the currently logged user is assigned to the newly created address. At last, redirected addresses were changed to user's account page.

After implementation of this module, there is one more involved in accounts administrations - User. User module will be accessible only to application administrators. It should allow modify all users as well as assigning Employee or Customer type. At this point, there are two separate controllers for customers and employees. It was decided to create additional controller for users in general in order to list them at one place. A generator had been run and created UsersController with one index action. The index view was made with Bootstrap's tabs where administrator can switch between three bookmarks - all, customers and employees. Index view files of employees and customers were prefixed with "_" symbol, which indicates a partial view. Thanks to this change, the index view of users can render these partials in corresponding tabs. The last index partial for users in general was created in the same way to render three index partials on the page. This unification made the structure well arranged as the indexes stayed in their respective folders.

Content of the tabs is loaded all at once and clicking on individual tabs is showing and hiding particular parts of the overall content. At the controller there are three instance variables initialized and assigned with collection of all records of customers, employees and users in general. The action links and attributes are dependent on the object type. A method was created to help this decision process in UsersHelper file.

The edit form is sufficient for both entities and there was no need for show actions, therefore they were removed. The lists were extended with action for entity type change for each record. This action changes customer to employee and vice versa. It was separated from the form due to security issues - sending important attribute is better to implement as standalone action, where authorization can be done easier and more robust. The page containing lists of users is shown on Figure 23.



Users

Name	Surname	Phone	Email	Salutation	Nationality	Newsletter	Employed since	Role	Position	Type			
John	Admin		admin@test.com					admin		Employee	Change to Customer	Edit	Destroy
John	Customer	+420777123123	customer@test.com	Mr.	CZ	false		customer		Customer	Change to Employee	Edit	Destroy
John	Manager	+420777777777	manager@test.com				2012-01-04 00:00:00 UTC	manager	senior	Employee	Change to Customer	Edit	Destroy

Figure 23 - Users' page

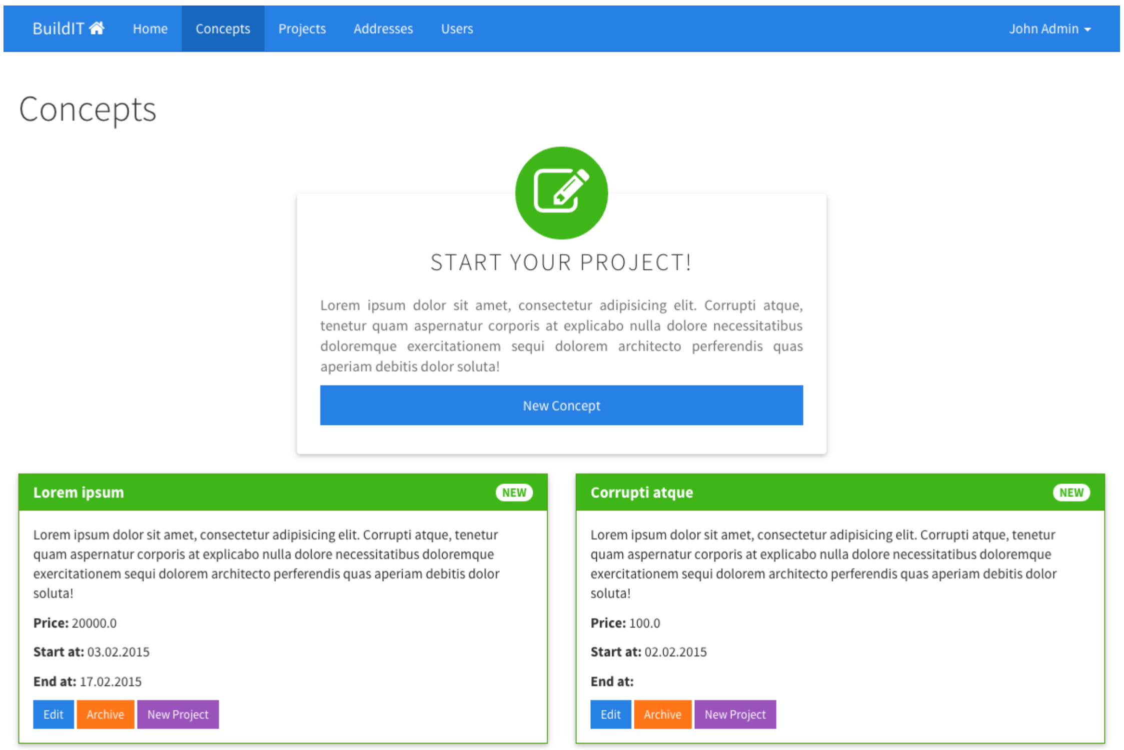
Source: Own

Once the lists of users were created it was necessary to modify forms for updating and creating records. The design was again similar like previous forms, furthermore, partials were used to simplify the structure and decrease redundancy. For example fields for address were stored within a partial file nested in address views and was used several times - edit and new forms of customer and employee, account update form, etc.

4.3.8 Application core

The next section was to implement the core modules, which makes the application functional and productive. The development covered user administrative part, logging, basic design and other modules for application proper working. The following development and implementation will be dealing with the application's core functionality with Project as a main entity.

The first module to implement was the Concept. Concept should be the first entity to create when user wants to setup a project. The assignment defined some attributes and associations, which had been created by the scaffold generator earlier. The index page was redesigned to include a short promo text with action for creating a new concept and a complete list of user's concepts organized within two boxes per row. Each box contains all attributes and action buttons to edit, archive or create a project from it. The whole page with example data is shown on Figure 24.



Concepts

START YOUR PROJECT!

Lorem ipsum dolor sit amet, consectetur adipisicing elit. Corrupti atque, tenetur quam aspernatur corporis at explicabo nulla dolore necessitatibus doloremque exercitationem sequi dolorem architecto perferendis quas aperiam debitis dolor soluta!

Lorem ipsum

Lorem ipsum dolor sit amet, consectetur adipisicing elit. Corrupti atque, tenetur quam aspernatur corporis at explicabo nulla dolore necessitatibus doloremque exercitationem sequi dolorem architecto perferendis quas aperiam debitis dolor soluta!

Price: 20000.0

Start at: 03.02.2015

End at: 17.02.2015

Corrupti atque

Lorem ipsum dolor sit amet, consectetur adipisicing elit. Corrupti atque, tenetur quam aspernatur corporis at explicabo nulla dolore necessitatibus doloremque exercitationem sequi dolorem architecto perferendis quas aperiam debitis dolor soluta!

Price: 100.0

Start at: 02.02.2015

End at:

Figure 24 - Concepts' page

Source: Own

Authorization for using these actions will be restricted later during the development. Triggering new or edit action should bring a form for the concept in a new modal window. To meet this requirement the original form was enclosed within a modal (see Figure 25), which was separated as a partial view. Edit action was more complicated to code, since it would be inefficient to render modal with an edit form for each concept listed. It was implemented as an asynchronous (AJAX-like) call with concept's id as an argument to the concepts controller. The controller then returns concept object and renders respective JavaScript view, which contains the same modal partial but with prepopulated fields from the object. The next edit action overwrites the last edit form modal, thus there is only need for one modal for editing all concept objects.

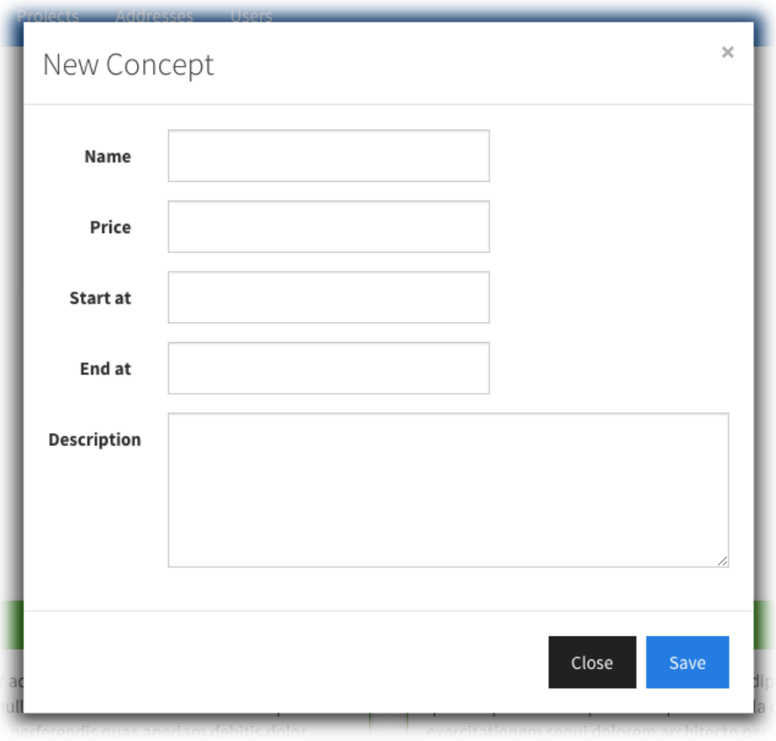


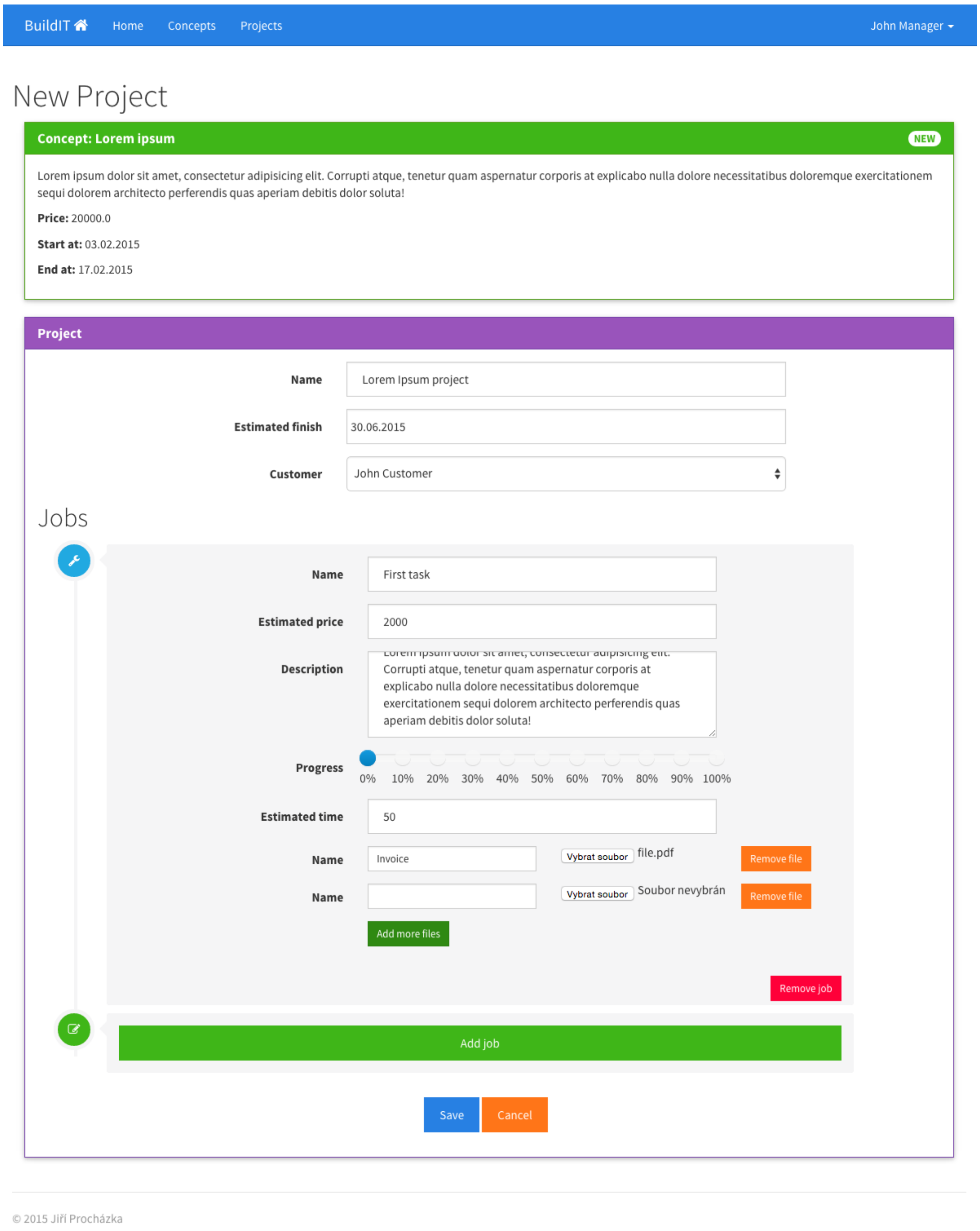
Figure 25 - Concept modal form

Source: Own

A concept is a starting point for any project; therefore each box from the concepts list has to contain action to create a new project on bases of the particular concept. The box was supplied with action to new project with the concept's id as an argument. Before a project is created from a concept, there has to be a check if the concept has no project assigned. By the requirements, every concept can have only one project. To ensure this rule, a validation was created within project model as a private instance method.

The next part of the development was the project module, beginning with forms. Form for creating a new project should contain the concept as a guideline. The project has a few attributes and several jobs. The form had to be done accordingly, with concept, fields for project and then possibility of adding multiple jobs within. As was already done before with users and addresses, the project was set to accept nested attributes for jobs. However, it was more complicated in this case, since the number of jobs can be variable. Basically it was necessary to implement project form with dynamic jobs' forms. A gem called Cocoon (https://github.com/nathanvda/cocoon) was included in the application to solve this issue. Cocoon provides helper methods for dynamical adding and removing nested forms - in this case job's attributes. The gem is mainly based on jQuery and it is well integrated with Rails. A few bits of code were necessary to implement to make it work as desired (permit attributes, modify models, create form partials, etc.). The form was used for creation and modification of project and its jobs. The design was made as was in other parts of the application using panel boxes.

The forms for jobs were designed as a timeline from the very first at the top to the latest at the bottom ended with action to add a new job form. This design with forms inside allows using it for action new as well as for edit action. The blueprint of the timeline with basic CSS file had been obtained from http://bootsnipp.com/snippets/featured/single-column-timeline-dotted and was modified to suit the needs of this web application. The individual fields of job's form are mostly text fields with one text area for description. There is one extra attribute progress, which has an integer value and should show percentage of the job completed. This input was implemented as drag slider with 10% step from 0 to 100 with a helpful tool of bootstrap plugin bootstrap-slider (https://github.com/seiyria/bootstrap-slider). A few modifications were necessary to match the desired design. The complete form is presented on Figure 26.



New Project

Concept: Lorem ipsum NEW

Lorem ipsum dolor sit amet, consectetur adipisicing elit. Corrupti atque, tenetur quam aspernatur corporis at explicabo nulla dolore necessitatibus doloremque exercitationem sequi dolorem architecto perferendis quas aperiam debitis dolor soluta!

Price: 20000.0

Start at: 03.02.2015

End at: 17.02.2015

Project

Name	Lorem Ipsum project
Estimated finish	30.06.2015
Customer	John Customer

Jobs

Name	First task
Estimated price	2000
Description	Lorem ipsum dolor sit amet, consectetur adipisicing elit. Corrupti atque, tenetur quam aspernatur corporis at explicabo nulla dolore necessitatibus doloremque exercitationem sequi dolorem architecto perferendis quas aperiam debitis dolor soluta!
Progress	0% 10% 20% 30% 40% 50% 60% 70% 80% 90% 100%
Estimated time	50
Name	Invoice — Vybrat soubor file.pdf — Remove file
Name	Vybrat soubor Soubor nevybrán — Remove file

Figure 26 - Project's form

Source: Own

4.3.9 Paperclip

According to the assignment, each job can have zero or more attachments. Attachments contains only name attribute and were designed to include file uploaded by user. Uploading files to the application was done by another gem called Paperclip.

Paperclip needs an image processor installed on the system (particularly ImageMagick) to work. It can be easily installed by command `brew install imagemagick` on OS X if the processor is not present. After bundling the Gemfile, the application was prepared for including files. Paperclip uses keyword `has_attached_file` with a name passed as a symbol argument within a model definition to include file.

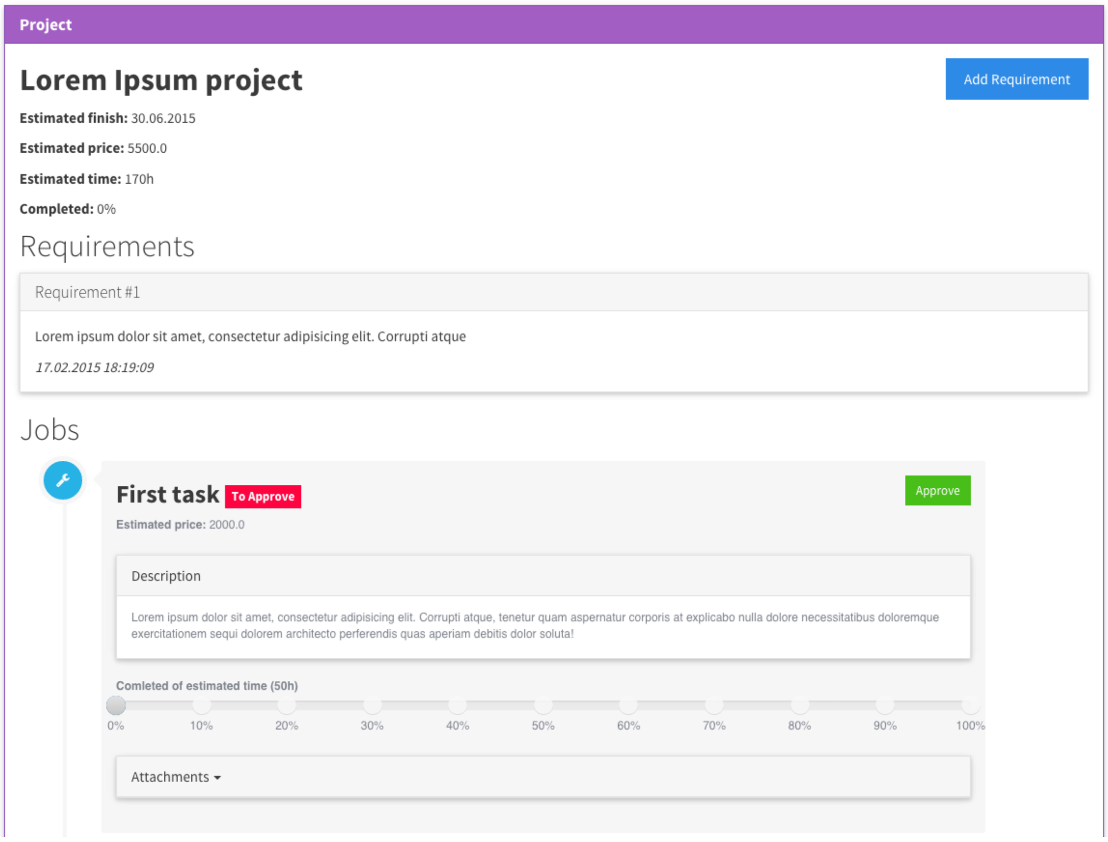
Incorporating Paperclip directly within the job model would not work for variable number of files linked. Therefore, it was decided to use Attachment model as a solution for multiple files linked to one job. Attachment model was equipped with Paperclip file, thus an instance of Attachment equals to one file; and Job instance can have multiple Attachments. The model had to be extended with validations for content type to prevent various security issues. Restrictions for the content types were made to pdf, doc, docx, xlsx, jpeg, png and gif files. Afterwards, a migration had to be created to include Paperclip attribute to the Attachment model. Attachments can be added to any Job instance, which is created or updated as nested object of Project. Partial with Job form was extended with another nested form for attachments.

It was required to dynamically add optional number of attachments to any job. Problem of dynamic nested forms was already solved with the Project and its Jobs; therefore the same gem (Cocoon) was used to implement Attachments. There were few obstacles to handle this two level nesting, which were solved by defining custom classes of HTML elements, supplying additional parameters to Cocoon methods and permitting another level of nested attributes within the project controller. The attachment's form for adding files was done as simple text field with name attribute and file field for browsing and selecting file from file system. Modifications of already created attachments were disabled in order to simplify the workflow. Project's edit form contains form for adding new attachment(s) and list of files already saved with action icon to delete the attachment asynchronously. Since the Attachments are created and showed from Project's form there was no need for its respective basic views and controller actions.

Project's show page was implemented similarly to project's form to maintain consistency. Nevertheless, there were some changes to make, in order to offer a clear overview of the project and all of its parts. It is also a place were customers can approve individual jobs, thus giving them confirmation for realization. Jobs will be labelled as "To approve" until customer use the "Approve" action as can be seen on Figure 27.

When this action is triggered, it sets a date and time to `confirmed_at` attribute of the particular job. This date is then showed as label next to the name.

Since project's show page is mainly for the needs of the customer, it also includes an action for adding requirement to the project. The action opens a new modal window with form for requirement, which after submitting closes and refreshes the page, thus residing on the same page with the project overview. Each requirement appears on the project overview grouped in Bootstrap's accordion, to capture all of the important information at one page.


Project

Lorem Ipsum project

Estimated finish: 30.06.2015

Estimated price: 5500.0

Estimated time: 170h

Completed: 0%

Requirements

Requirement #1

Lorem ipsum dolor sit amet, consectetur adipisicing elit. Corrupti atque

17.02.2015 18:19:09

Jobs

First task To Approve

Estimated price: 2000.0

Description

Lorem ipsum dolor sit amet, consectetur adipisicing elit. Corrupti atque, tenetur quam aspernatur corporis at explicabo nulla dolore necessitatibus doloremque exercitationem sequi dolorem architecto perferendis quas aperiam debitis dolor soluta!

Comleted of estimated time (50h)

Figure 27 - Project's show cut

Source: Own

Lastly in the project module, a listing of all projects was made. In order to make it comprehensible for both customers and employees, the projects were divided into three groups. The projects are displayed and separated within Bootstrap's tabs as Active, Completed and Archived - the division criteria is self-explanatory. Individual projects are captured with basic attributes in panels, similarly as concepts' list. Needed collection of projects is retrieved via controller, where desired query is performed according to the active tab. Design of the projects page is displayed on Figure 28.

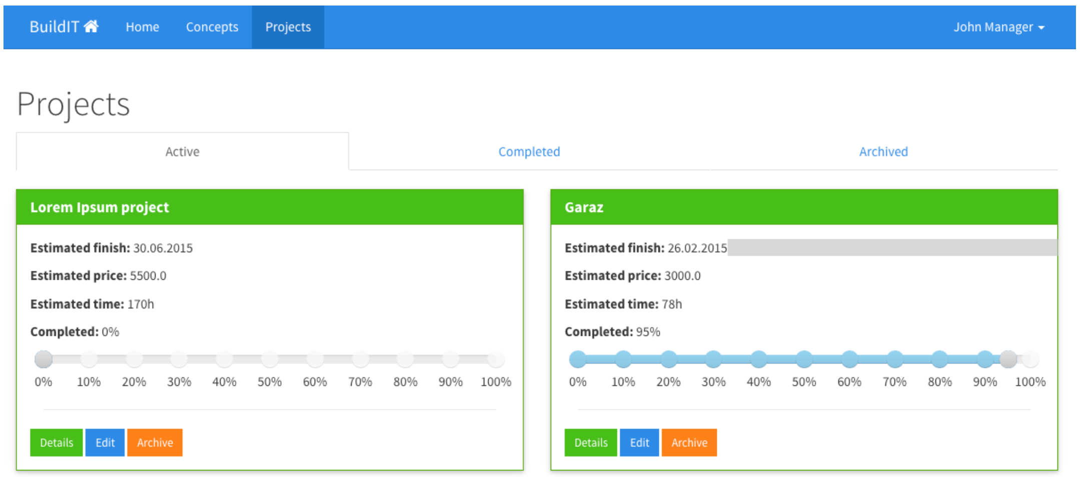


Figure 28 - Projects page

Source: Own

4.3.10 Datepicker

A JavaScript Datepicker plugin was added to the application for better UX. It is more comfortable for a user to pick a date from a pop-up menu instead of filling the date manually. There are many plugins providing this functionality. A Datepicker called bootstrap-datepicker (https://github.com/eternicode/bootstrap-datepicker) was chosen for this application. It has easy initialization, where only few parameters were set for desired functionality. This initialization code was put to application.js, therefore the plugin's functionality is available across the whole application. When a class "datepicker" is assigned to any input field, a calendar for date pick appears after clicking in the field.

4.3.11 Authorization

The next part will cover the implementation of authorization rules to all users of the application. A support gem called Cancancan, which is basically a support library for user's authorization to access resources, was added to the application. It simply defines a class "Abilities" with list of users' roles and their permissions on various resources. This gem is dependent on Devise gem, which solves the users' authentication and restricts the access to signed users only. The application had to be equipped with `before_action :authenticate_user!` line in all controllers, which are subjects for signed users only. The only controller, which was omitted, was the HomeController that should remain accessible to everybody. Inheritance was used to simplify this task by adding the code within the ApplicationController from which other controllers are inheriting. The HomeController was included with exception `skip_before_action :authenticate_user!` to leave this controller without the necessity of login. Thanks to this, unlogged users will not be able to see other pages than the home page and login/sign up forms.

After running bundle install, the Cancancan's generator was run to create Ability class - `rails g cancan:ability`. The ability class contains list of authorization rules for different types of users. It was decided to use the role-based authorization. It was necessary to add new attribute to User model (create migration, add column, run migration), where the role will be stored in a string. According to the assignment, three basic roles were defined - customer, manager and administrator. Thanks to this type of authorization, more roles can be easily added in the future. The defined roles are inheriting the authorization rules and adding more with every higher level. This behaviour was implemented in compliance with the gem's documentation. User model was added with array of strings called ROLES, where the roles are listed and ordered ascending with higher permissions - `ROLES = %w[customer manager admin]`. The model was also extended with instance method, which takes role as a symbol in arguments and returns Boolean whether the particular user has permissions of the passed role (User with admin role has also permissions of manager and customer). The method is simply comparing index values of the roles within the defined array. Consequently, in the Ability class conditions for every role where defined, where individual permissions were stated with the inheritance in mind.

These authorization rules were implemented as defined in Table 8. It describes application's modules and authorized actions according to user's role. It was essential to add method call (`load_and_authorize_resource`) within every controller to incorporate all authorization rules. It was added to all controllers except home and devise, which should be available for all users and guests. This method also creates and assigns relevant instance variables to individual actions with permissions rules applied (e.g. user can see only concepts created by him). Once the rules were coded, it was necessary to set a default role to newly registered users to "customer". Since these operations were done, the application access had been restricted and began to throw exception when user tried to access resource without necessary permissions. The thrown exceptions were handled within ApplicationController to friendlier version as redirect with alert message.

	Customer	**Manager**	**Admin**
Home	all	all	all
Concepts	create, update*, list*	create, update, list	create, update, list
Projects	show*	create, update*	all
Requirements	create*	-	all
Attachments	download*	create, remove	create, remove
Addresses	create*, update*	create*, update*	all
Users	register*, update*	create, update	create, update, change type

(*) Action can be performed only on objects assigned to the user

Table 8 - Authorization rules

Source: Own

The next part was to hide buttons and actions to which user doesn't have permissions and causes the exceptions. Cancancan gem provides helper methods for views as well that solves this problem. The method simply returns Boolean value, whether the current user is authorized for the action passed or not. Example view injection asking for authorization rules about creating a new requirement can be represented as `<% if can? :create, Requirement %>`. Many parts of various views were conditioned within similar blocks.

4.3.12 Pagination

As the application grows bigger in the number of records, there is a necessity for performance improvements. Particularly, retrieval of large number of records from database might be very time consuming and expensive operation for the server. It also might take a lot of time to render many records at the client side. To prevent these potential performance issues, the records listing had been equipped with limit and offset for the database. This means, that only certain amount of records will be requested and send back from the server. This application restricted the number of records on 6 at one page, due to performance and design reasons.

This technique is called pagination, since the whole collection of records is separated to smaller parts on more pages with links on these pages. Gem called will_paginate (https://github.com/mislav/will_paginate) was used in this application to implement pagination. A paginate method (provided by the gem) was added to controllers' actions where a query for many records was present. An example of a query for listing concepts was done as follows - `@concepts = @concepts.paginate(page: params[:page]).order('id DESC')`. This query reuses already assigned instance variables by Cancancan gem, which lists only records selected according to defined permissions. The respective view was then supplied with helper method for rendering links with pages - `<%= will_paginate @projects %>`. A global variable for the maximum number of records per one page was defined within application.rb file as `WillPaginate.per_page = 6`.

Additionally, the application and the paginate gem were extended with another gem called will_pagiante-bootstrap (https://github.com/bootstrap-ruby/will_paginate-bootstrap), which modifies the page links to match the Bootstrap CSS styles. The modification was done by adding extra parameter to the helper method - `renderer: BootstrapPagination::Rails`. The pagination was implemented in this way in all relevant controllers. An example of the resulting view for pagination links is shown on Figure 29.

Figure 29 - Pagination links view

Source: Own

4.3.13 Tests

Tests are vital for every application, especially for complex ones. The developed web application is fairly simple and small at this point, however future development might extend it. Therefore, tests were created for this application as well to ensure correct functionality with future development. It is good practise to include all model and controller tests. In this case, many of the tests were particularly focused on authorization of users. The possibility of user seeing records, which belongs to someone else, might decrease credibility of the application and the company as well.

The tests were created according to the theoretical basics described earlier in this thesis, beginning with model (unit) tests. It is best practise to test all of the models functions. This includes validations, callbacks and other instance or class methods.

Scaffold generators were used during the development of this application; therefore folder test/models in application root already contains relevant test files for all models. The generators also created fixtures, which serves as testing data and are used for filling the test database before every testing. Most of the fixtures were modified or added with more records and relations between them.

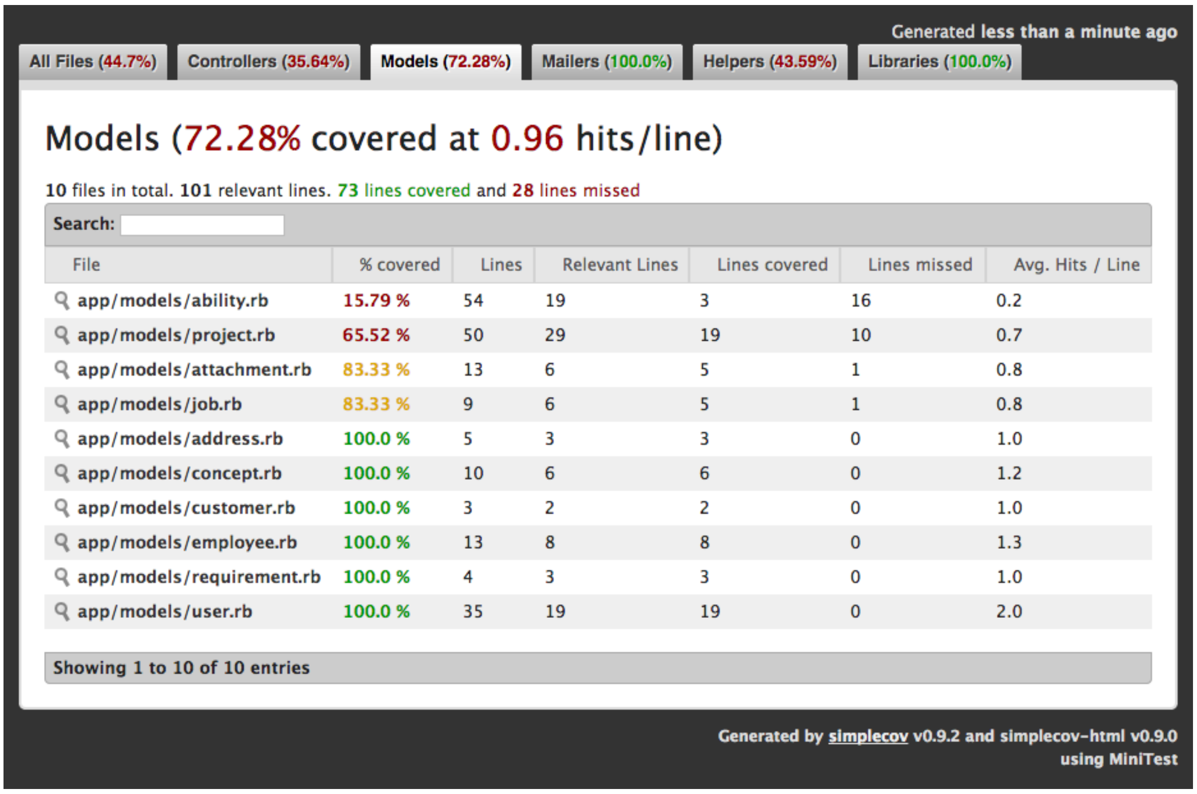
Controllers' tests were made to test all of the actions, routes and parameters. Scaffold generators again generated the test files with default tests for basic CRUD actions. However, almost all of these tests failed due to unknown method error of authenticate. This was cause by gem Devise, which had been added for authenticating users. Devise documentation refers to this issue with easy solution - the test_helper.rb file within its class definition was extended with inclusion of Devise module for tests - `include Devise::TestHelpers`. Once this module is included user can be signed in by simple command before protected actions. This can be done in a setup block within the controller's test class. Any created user can be signed in within the test scope, which provided a space for authorization testing as well. User fixtures contain users with all three types of roles. Testing permissions for different resources was done by signing different users and testing results of various actions.

During the creation of tests, few extra fixtures were added to cover all possible scenarios and states of the application. After creating all necessary tests covering models and controllers, there were some actions, which were not tested yet. It was the Devise login and sign in. To ensure these actions work properly as well, there was created an integration test with complete flow of user's sign in process. With this last test, the application could be stated as well tested with 278 asserts within 117 tests.

4.3.14 Test coverage

The application coverage by tests should be as high as possible. To ensure that all of the important parts of the application are tested a gem for measuring the coverage was installed for the test environment. There are many gems and libraries suitable for this task, for this application a gem called SimpleCov (https://github.com/colszowka/simplecov) was used since it is one of the most popular and easy to use. A line `gem 'simplecov', require: false` was added to Gemfile within test group of gems. Once the bundle install command had been run the coverage measurement was ready to work. In order to measure coverage of all tests it had to be triggered right at the beginning of test_helper.rb file by calling commands `require 'simplecov'` and `SimpleCov.start 'rails'`.

After this setting, the coverage report is generated when any of the tests are executed. The report is created in html file with simple and clear design describing all of the important information. It shows coverage percentage of every relevant application file and after clicking on it, detailed preview of the file with covered and uncovered lines highlighted. Thanks to this report it is very easy to determine which part of the application needs more tests. An example screenshot of the coverage report is shown on Figure 30, where only few of the model tests were created. This example was representative to show various states of files. Tests for this application were already created and by measuring with this tool, it scored 99.53% of coverage by tests. There were only two lines missed, which were insignificant and too complicated to test, thus left untested. The result generated by Simplecov is captured in Appendix 10.7.



Models (72.28% covered at 0.96 hits/line)

10 files in total. 101 relevant lines. 73 lines covered and 28 lines missed

File	% covered	Lines	Relevant Lines	Lines covered	Lines missed	Avg. Hits / Line
app/models/ability.rb	15.79 %	54	19	3	16	0.2
app/models/project.rb	65.52 %	50	29	19	10	0.7
app/models/attachment.rb	83.33 %	13	6	5	1	0.8
app/models/job.rb	83.33 %	9	6	5	1	0.8
app/models/address.rb	100.0 %	5	3	3	0	1.0
app/models/concept.rb	100.0 %	10	6	6	0	1.2
app/models/customer.rb	100.0 %	3	2	2	0	1.0
app/models/employee.rb	100.0 %	13	8	8	0	1.3
app/models/requirement.rb	100.0 %	4	3	3	0	1.0
app/models/user.rb	100.0 %	35	19	19	0	2.0

Showing 1 to 10 of 10 entries

Generated by simplecov v0.9.2 and simplecov-html v0.9.0
using MiniTest

Figure 30 - Coverage report from SimpleCov

Source: Own

4.4 Heroku deployment

Once the application is ready for testing by users of the company, it has to be published on the Internet as a running application. Since the initial testing is typically done by few users, mainly from the assigning company, it is not necessary to rent or buy some servers, but rather use some free of charge solution. The so far developed version of this web application was deployed on cloud application platform called Heroku (www.heroku.com). Heroku web pages offer detailed documentation for usage. Platform usage is conditional on creating account, where user can set parameters of the services.

After creating the account, it was possible to create a new cloud application. Heroku uses Git repositories for application deployment. In order to deploy on the server it was necessary to install Heroku Toolbelt. This tool logs in the user, thus links the account with the development machine. It is available for all major platforms, for this thesis the Mac OS X version was used. After the package installation a command `heroku login` was executed in shell for developer's authentication. Next part should be initialization of Git repository, but since the application has been checked to Git index during the development, this part is mostly done. A new Heroku remote repository was linked with the local repository to enable deployment to the server by executing `heroku git:remote -a buildit-test` from the application folder. The parameter "buildit-test" is the name of the application created on Heroku website. When the remote is ready, the application can be deployed via command `git push heroku master`, which deploys the master branch. The shell obtains remote server output with various information and installation notes. Once the application is deployed, it is also necessary to create and migrate the database. This can be done via Heroku Toolbelt by command `heroku run rake db:migrate`. The basic Rails commands works on Heroku servers by prepending `heroku run` keywords.

At this point, the application runs on the server, however without loading any assets. This is a well known issue, which can be easily fixed by adding `config.serve_static_assets = true` within the block in production.rb file. Before the deployment all assets have to be precompiled by command `RAILS_ENV=production bundle exec rake assets:precompile`. After commit and push, the application can run in the same way as on the local machine. By default, Heroku gave this app URL as follows - https://buildit-test.herokuapp.com.

4.5 Refactoring with MongoDB

The partial goal of practical part was to implement the same web application with the usage of document oriented database, particularly MongoDB. The application had been developed in MVC style, therefore it was possible to refactor the model part and keep the rest almost unchanged. The main change was to rewrite models and database queries. There also might be some dependencies on pg gem, which shall be resolved. The Mongoid gem mapper was chosen for this application since it is one of the most popular.

The MongoDB implementation had to begun with installation of the database itself on the local machine. MongoDB is available for all major operating systems. There are few ways of the installation on OS X, where using homebrew is the easiest one. Similarly like PostrgreSQL, MongoDB installation was done via executing `brew install mongodb` in shell. After installation database service daemon had been run, which made the database available. Calling `mongo` command from shell, which should open the JavaScript console of the database, can test success of these operations.

Once the environment had been prepared, the implementation can move to the application. Since this refactoring should not affect the original development, a new Git branch had to be made. It was created by `git checkout -b mongodb`, which moved the current checked index to the new branch called mongodb.

4.5.1 Mongoid

As already mentioned, Mongoid data mapper was used for linking Ruby objects to persistent documents stored in MongoDB. Adding new gem within the Gemfile can do the integration to Rails application. However, the application has already several extra gems, which should be compatible with mongoid gem.

The first step of the integration was to add the mongoid gem in the Gemfile (`gem 'mongoid', "~> 4.0.0"`). After bundling the gems, generator for creating basic configuration file for MongoDB and Mongoid was executed - `rails g mongoid:config`. After these operations, the application was capable of storing and retrieving data from MongoDB alongside of PostgreSQL. The next step was to get rid of ActiveRecrod (PostgreSQL ORM) and solve errors connected with dependencies and other issues. Before removing pg gem, there were some gems considered as dependent on pg - Devise, Cancancan, WillPaginate and Paperclip. As further development showed, only the last two gems had to be substituted with mongoid compatible version.

The application.rb file had to be modified to remove require 'rails/all' line (because it contains ActiveRecord) and require 'action_controller/railtie', 'action_mailer/railtie' and 'rails/test_unit/railtie' as the Mongoid documentation mentioned. Devise initializer had also been changed in section where ORM is defined - active_record was substituted by mongoid. After this, the pg gem was removed, the server successfully started and home page was loaded.

The next part of the migration was to convert all models on Mongoid. Mongoid model has to contain include statement of `Mongoid::Document` and its fields are stated within the model class, which no longer inherits from ActiveRecord. Additionally, model were also included with `Mongoid::Timestamps` module to automatically manage timestamps as ActiveRecord does. All models were modified according to this.

A few methods had to be changed, due to different definition in Mongoid gem - e.g. method for calculating average of attributes was changed from `average()` to `avg()`. Inheritance of User model was done seamlessly with children attributes defined within its classes. Collections of subclasses contain only attributes that belong to it (on the database level) in contrast to PostgreSQL, where all possible attributes were stored in one parent table. The type attribute, which distinguishes subclasses according to classes, was no longer needed to manually define - Mongoid handles the inheritance automatically and creates `_type` attribute on background.

Since Paperclip gem was substituted with Mongoid compatible version, the Attachment model was also slightly modified to work with the new gem. The Attachment class was additionally included with `Mongoid::Paperclip` module and the method defining the file field was changed to `has_mongoid_attached_file :file`. The rest of the model remained unchanged.

One of the last modifications was removal of unused files, which remained from PostgreSQL implementation. MongoDB does not work with any kinds of migrations and changes can be done dynamically. Thanks to this the whole folder containing migration files and schema.rb file were removed without causing any failure.

After these modifications had been implemented, the application was completely functional and from the user's point of view basically unchanged. The only visible change can be seen in the address line, where previously called integer IDs of objects changed to longer hexadecimal form.

MongoDB by extension Mongoid offers embedding documents as an alternative association type. Embedded relations can be used for one to one or one to many objects where the child objects are stored within the parent object. Simply put, the parent document contains the embedded documents similarly like attributes. Embedding significantly improves efficiency, since the database is queried once for the document, which contains its children. However, the children objects cannot be retrieved without referencing the parent object first. Therefore, embedding should be used only where this restriction makes no obstacles. In case of this web application embedding was considered for relations project - jobs and job - attachments. These associations could be refactored using embedding. A figurative diagram on Figure 31 was created to demonstrate an example of document embedding within the application. Mongoid provides methods `embeds_many`, `embeds_one` and `embedded_in` for declaration of these relations. Nevertheless, at this point of development process it was still unclear whether the children objects might be used independently on the parents in the future. Since there are no performance issues and most likely won't be in the nearest future, it was decided to keep the relationships without using embedding.

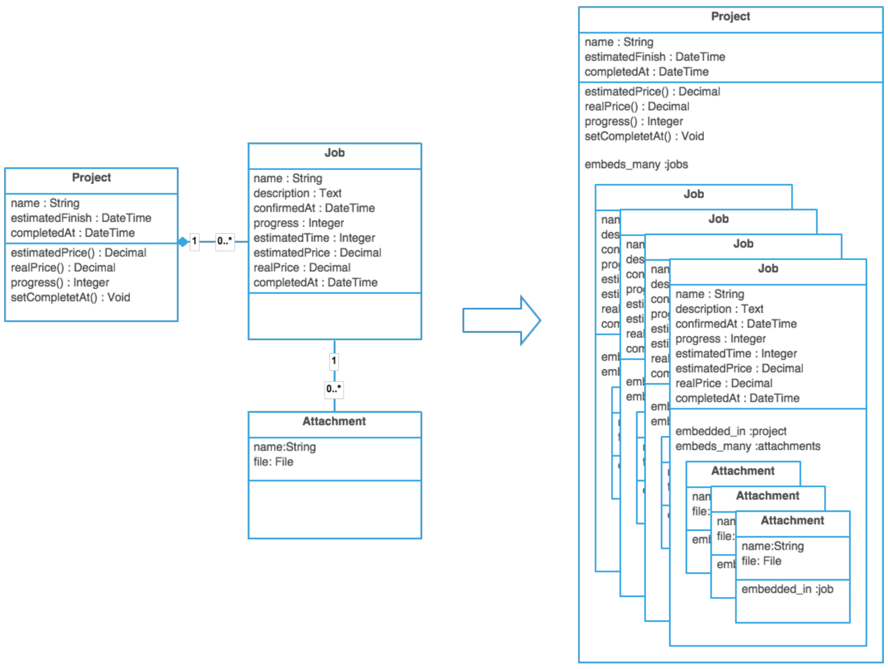


Figure 31 - Alternative documents structure - Embedding

Source: Own

4.5.2 Tests

The migration to a different database caused the tests' failure since the previous testing data (saved as fixtures) is not supported in Mongoid. A gem called factory_girl_rails (https://github.com/thoughtbot/factory_girl_rails) was used as replacement for fixtures. It behaves in a different way than fixtures, which are loaded to database without any model interactions. Factories on the other side are created in the same way as regular instances of any model, therefore all validations and callbacks are triggered during the creation of an object. FactoryGirl gem allows defining factories for all application's models, which basically serves as drafts for objects. An example definition of a project's factory can be as follows:

```
FactoryGirl.define do
     factory :project_one, class: Project do
       name "Lorem Ipsum project"
       estimated_finish "2015-06-30 00:00:00"
       archived false
       created_at "2015-02-17 17:31:09"
       updated_at "2015-02-17 17:31:09"
       association :concept, factory: :concept_one, strategy: :build
       association :customer, factory: :customer1, strategy: :build
       association :employee, factory: :employee1, strategy: :build
     end
end
```

The code defines a factory with name `project_one` (which has to be unique) of class project with all mandatory attributes. Associations of this object with other models are defined as references to other factories, which have to be declared as well. The optional parameter `strategy: :build` indicates, that during the object creation the referenced objects are only build and not saved. This feature enabled better control of assigning relations between objects - referenced objects are saved only when desired.

At the beginning of any test, it is necessary to delete all test collections to ensure there are no identical objects before creating new ones. This is because the database is persistent and all objects remains within the collections. Populating the test database with documents can be done by calling `FactoryGirl.create(:name)` method with a name of predefined factory as an argument.

Once the factories had been defined, the tests were modified to work with new form of test data. Majority of the tests' asserts were left unchanged and adjustments were done only for the test data definition and declaration. This is in correspondence to changing the data model part and keeping the application logic uniform. The application coverage by tests was again measured by SimpleCov gem with more than 99% of coverage reported.

4.6 OpenShift deployment

OpenShift is online cloud platform for deployment of applications written in Java, Ruby, PHP, Node.js, Python or Perl by RedHat. It similarly like Heroku provides various plans beginning with free accounts, which can be scaled up to desired performance for additional charges. Again the only requirement for using the free plan is to create an account. After registration and activation process the administration prompts user with a guide for creation of the first application. The first step is to choose the type of application with several predefined options, including Ruby on Rails 4 that was chosen. The next step contains a form for public URL, link for GitHub repository and other settings, which were mainly for charged plans.

Once the application is created it is setup with Ruby 2.0 and MySQL 5.1 by default. Modules, such as database, are called cartridges and represent a managed runtimes for the application. Since the BuildIT application after the refactoring process depends on MongoDB, a new cartridge with MongoDB 2.4 was added. Free account offers hosting for three Gears, which basically designate an application with cartridges. OpenShift offers a command line tool in a form of gem, which allows managing applications from local machine. This tool was installed by `gem install rhc` and configured by `rhc setup`, where credentials were inputted and account was linked with local machine. OpenShift automatically generates a new Git repository, where its hooks are included for application deployment with Git's push. This repository was cloned on the local machine and extended with the BuildIT application code. Once the changes are committed to OpenShit's repository it can be pushed, which triggers the deploy action. The first deploy was not successful, because current configuration of this cloud platform does not allow Rails 4.2, therefore a necessary downgrade was conducted to older Rails 4.1.9. It didn't affect any of the application's functionality.

Lastly, to interact with MongoDB, it was necessary to add OpenShift's cartridge credentials and other values to mongoid.yml configuration file for production environment. After these changes, deploy was successful and the application was running on URL (http://buildit-prochazka.rhcloud.com/) assigned during the new application registration process.

OpenShift also offers standard SSH access to the application structure with all of its cartridges. This is an advantage over previous Heroku platform, which does not offer this functionality. Nevertheless, the SSH access is very limited, without root privileges.

5 Evaluation of results and recommendation

The practical part provided description of the whole development process with all obstacles that occurred. It captured the whole process with description of tools and utilities used. The beginning contains formalisation of the assignment after receiving demands from the company, which wouldn't be possible without experiences of the developer. The design of the application was also conducted by the developer, which prolonged the development process. Even with a single developer, the application was created approximately within two weeks thanks to application of various technologies.

Ruby on Rails confirmed its status as very productive and fast prototyping framework. Work with it was efficient and readable, with minimum configuration needed. Ruby itself has very simple syntax, which emphasises the ease of using Ruby on Rails. Since Ruby is interpreted language, Rails can offer a console interface, which helps the development in a way of testing methods and manipulation with data. The MVC pattern, which Rails framework incorporates, is the most popular for its structure and independence between individual parts. This was confirmed in the practical part, when all of the models were switched from PostgreSQL database to MongoDB. This transition did not require any additional changes to views or controllers, thus proving the independence and high abstraction of database layer. The framework also supports running several databases at the same time, which gives a space for various migrations or using advantages of different databases' concepts.

The application was developed with PostgreSQL at first and then refactored with MongoDB, thus using different databases within the same application. This enabled to compare both databases in the terms of implementation difficulty and performance. These databases are incorporating different concepts and technologies. It is difficult to compare it, since it differs a lot in certain areas. However, this thesis allowed observing particular advantages and disadvantages of both. PostgreSQL is considered as rather adult and mature database, with transactions support, which is very useful for financial records. It is sometimes too strict when it comes to saving data with various encoding and formats. PostgreSQL is more than 30 years older than MongoDB, therefore it provides wider support, comprehensive documentation and countless number of solved issues. However, in comparison to MongoDB it has several downsides. MongoDB has built-in scaling and sharding options that work out of the box, which PostgreSQL lacks.

If the database designer follows rules of normalization, relational databases tend to have queries with many joins, which rapidly slow down the execution time when the database contains numerous records. MongoDB provides document embedding, which is basically storing documents within one parent, thus receives all desired data from a single object with one query. Fixed schema of relational databases versus dynamic fields of MongoDB is a controversial topic. It is sometimes desired and considered as an advantage, yet sometimes it can be a security and consistency risk. From the point of the development in the practical part of this thesis (and development in general), dynamic fields instead of fixed schema are definitely an advantage, which speeds up the process. However, the production point of view might be the opposite. Regarding the performance comparison in the application, there were no observable differences between PostgreSQL and MongoDB. The application is too small and simple, running in one server environment for such a comparison. Practical experiences showed that even within larger applications it is sometimes impossible to evaluate, which database is better. Both databases have some advantages and disadvantages, where the balance in between is dependent on data model and application requirements.

PostgreSQL also has a potential competitor in a form of popular MySQL database. MySQL also uses SQL as its query language and most of its functions are the same. PostgreSQL is considered as more reliable, faster and better in general, thus more suitable for large data storage with complex queries. Therefore, it is better to use PostgreSQL for more complicated applications while MySQL can be sufficient for smaller applications. This rule does not apply to all cases, since some problems can be solved by using MySQL or PostgreSQL and vice versa.

The practical part also showed usefulness of various tools used during the development. Particularly, Git version control system helped to prevent data loss, enabled distribution of the source code across multiple machines and allowed to track and revert changes in history. Git branching feature turned out to be very useful, while developing the application with two different databases - it enabled to simultaneously edit the code in separate branches without interfering each other.

Usage of Bootstrap framework in the practical part helped to create consistent and responsive design of the application. Since it allows slight modifications of colours and appearance, the elements were customized to be distinguishable from other applications based on Bootstrap. UI elements and layouts of Bootstrap again significantly speeded up the development - it was not necessary to create elements and styles from scratch. A few third party snippets based on Bootstrap were also used in the application to improve user experience.

Lastly, the practical part included deployment to two different cloud platforms - Heroku and OpenShift. Both platforms were fairly easy to use and deploy. They provide similar functionality within the free plans. There were some minor differences of their services, but generally insignificant. Nevertheless, each platform has its main disadvantage that was noticed during the deployment. Heroku doesn't provide access to its filesystem, thus denying storage of any binary files there. As a substitution, Heroku offers linkage to other cloud services (such is Dropbox, Amazon S3, etc.) for the storage. It was considered as its biggest disadvantage, yet it is solvable. OpenShift, on the other hand, offers around 1 GB of file system space. However, the service, its web pages and the whole administration was very slow and often terminating due to some errors. Nevertheless, considering that both of these cloud platforms are free of charge, the quality is more than sufficient.

The web application development was done with Ruby on Rails, which satisfied all needs required by the assigning company and the developer. However, it is not the only option for creating web application. The obvious alternative is PHP with some framework such is Laravel, Phalcon, Symfony2 or Nette. There would be a condition for PHP in version 5 and later, where it was enriched by object-oriented functionalities. PHP's frameworks provide similar functions as Ruby on Rails with the MVC pattern as well. Majority of small web applications were created in PHP, since it is easy to use and widespread. Almost all of hosting servers provide PHP support, which makes the development and consequent deployment even reachable. Ruby on Rails is based on Ruby, which as a purely object-oriented language while PHP was only extended with the object properties. Ruby is also general purpose programming language whilst PHP was made specifically for web pages. This gives Ruby on Rails more options for solving any issue.

For absolute beginner, it is probably better to use PHP with a framework. However, for experienced developer its better to use Ruby on Rails with all its features and possibilities, especially for projects, where the development will continue and more functionalities with expanding complexity might be added in the future. Web development can be also done with other new trending technologies; such is NodeJS and AngularJS. However, these technologies are not as mature as Ruby on Rails, provides limited support and its future can’t be predicted.

6 Conclusion

Based on the theoretical part and author's own experiences in field, the practical part was created to describe the whole process of a web application development assigned by a company from building industry. It covers formalisation of the assignment, development of working application with Ruby on Rails framework with PostgreSQL database and Bootstrap framework for layout design. The following part describes the process of switching PostgreSQL for MongoDB database and the refactoring connected with it. Finished application in two versions (according to database) was deployed to free cloud platforms for public display and evaluation of the assigning company.

The main output of this thesis is the custom tailored web application for the company from building industry, which can suit as a tradable product in the future. It also gives a described example of development process that can suit as manual for creating other web applications. Using SQL and NoSQL databases showed the flexibility of Ruby on Rails and the incorporated the MVC pattern. The thesis also theoretically and practically introduced various tools and utilities for web application development. The deployment and usage of two free cloud platforms gave an example how to publicly display and operate a web application without any costs.

The future of web applications will probably increase on its popularity since the tools and ways of development are becoming more productive, efficient and easy to use. An obvious benefit of web application is mainly the platform independence, which have enlarged its usage to mobile devices, tablets, etc. Technological progress enables to reduce costs on computer equipment, as well as server hosting or cloud platform rental. Thanks to that, the web application development has a very promising future.

7 Bibliography

BADAWY, A. Aly Badawy. *Using Postgres' Hstore datatype in Rails* [online]. 29. 7. 2014 [cit. 2014-11-2]. Available at WWW: <https://alybadawy.com/post/using-postgres-hstore-datatype-with-ruby-on-rails>

BANKER, K. *MongoDB in Action*. 1st Edition. Shelter Island: Manning, 2012. 287 p. ISBN: 9781935182870.

BASU, S. *Ruby on Rails Study Guide: The History of Rails* [online]. 2013 [cit. 2014-08-04]. Available at WWW: <http://code.tutsplus.com/articles/ruby-on-rails-study-guide-the-history-%20of-rails--net-29439>

Bootstrap. *About Bootstrap* [online]. 2014 [cit. 2014-12-27]. Available at WWW: <http://getbootstrap.com/about/>

Bootstrap. *CSS* [online]. 2014 [cit. 2014-12-31]. Available at WWW: <http://getbootstrap.com/css>

Bootstrap. *Getting started* [online]. 2014 [cit. 2014-12-31]. Available at WWW: <http://getbootstrap.com/getting-started/>

CHACON, S. a B. STRAUB. *Pro Git*. 2nd Edition. New York: Apress, 2014. 456 p. ISBN: 978-1484200773.

CHODOROW, K. Snail in a Turtleneck. *History of MongoDB* [online]. 2010 [cit. 2014-11-15]. Available at WWW: <http://www.kchodorow.com/blog/2010/08/23/history-of-mongodb/>

CHODOROW, K. *MongoDB: The Definitive Guide*. 2nd Edition. Sebastopol: O'Reilly Media, 2013. 409 p. ISBN: 978-1-449-34468-9.

FLANAGAN, D. a Y. MATSUMOTO. *The Ruby Programming Language*. 1st edition. Sebastopol: O'Reilly Media, Inc., 2008. 429 p. ISBN 978-0-596-51617-8.

GAMBLE, A., C. CARNEIRO a R. A. BARAZI. *Beginning Rails 4*. 3rd Edition. Berkley: Apress, 2013. 303 p. ISBN 978-1-4302-6034-9.

GOYVAERTS, J. Regural-Expressions.info. *Ruby Regexp class* [online]. 2013 [cit. 2014-08-09]. Available at WWW: <http://www.regular-expressions.info/ruby.html>

HANSSON, D. H. Rails 4.0. *Riding Rails* [online]. 2013 [cit. 2014-08-04]. Available at WWW: <http://weblog.rubyonrails.org/2013/6/25/Rails-4-0-final/>

HANSSON, D. H. Rails 4.1.0. *Riding Rails* [online]. 2014 [cit. 2014-08-04]. Available at WWW: <http://weblog.rubyonrails.org/2014/4/8/Rails-4-1/>

LOELIGER, J. a M. MCCULLOUGH. *Version Control with Git*. 2nd Edition. Sebastopol: O'Reilly Media, Inc., 2012. 434 p. ISBN: 978-1-449-31638-9.

MAEDA, S. Ruby-doc.org. *The Ruby Language FAQ* [online]. 2002 [cit. 2014-06-23]. Available at WWW: <http://www.ruby-doc.org/docs/ruby-doc-bundle/FAQ/FAQ.html>

MATTHEW, N. a R. STONES. *Beginning Databases with PostgreSQL*. 2nd Edition. Berkley: Apress, 2005. 637 p. ISBN: 1-59059-478-9.

MONGODB, I. MongoDB Manual. *MongoDB CRUD Introduction* [online]. 2014 [cit. 2014-11-23]. Available at WWW: <http://docs.mongodb.org/manual/core/crud-introduction/>

PostgreSQL. *High Availability, Load Balancing, and Replication* [online]. 2014 [cit. 2014-11-8]. Available at WWW: <http://www.postgresql.org/docs/9.3/static/high-availability.html>

PostgreSQL. *PostgreSQL: About* [online]. 2014 [cit. 2014-11-1]. Available at WWW: <http://www.postgresql.org/about/>

REGE, G. *Ruby and MongoDB Web Development Beginner's Guide*. 1st Edition. Birmingham: Packt Pub., 2012. 310 p. ISBN: 978-1-84951-502-3.

RUBY, S., D. THOMAS a D. H. HANSSON. *Agile Web Development with Rails 4*. 1st Edition. Raleigh: Pragmatic Bookshelf, 2013. 434 p. ISBN 978-1-937785-56-7.

SANDERSON, A. Monkey and Crow. *Tagging With ActiveRecord and Postgres* [online]. 2. 5. 2013 [cit. 2014-11-2]. Available at WWW: <http://monkeyandcrow.com/blog/tagging_with_active_record_and_postgres/>

SEGUIN, W. a M. PAPIS. RVM: Ruby Version Manager. *RVM - Documentation* [online]. 2014 [cit. 2014-08-09]. Available at WWW: <https://rvm.io>

SIEGER, N. Nick Sieger. *RubyConf: History of Ruby* [online]. 2006 [cit. 2014-6-24]. Available at WWW: <http://blog.nicksieger.com/articles/2006/10/20/rubyconf-history-of-ruby/>

SPURLOCK, J. *Bootstrap*. 1st Edition. Sebastopol: O'Reilly Media, 2013. 109 p. ISBN: 978-1-449-34391-0.

TATE, B. *Seven Languages in Seven Weeks*. 1st Edition. Raleigh: The Pragmatic Bookshelf, 2010. 317 p. ISBN 978-19-3435-659-3.

THOMAS, D. a A. HUNT. Programming Ruby: The Pragmatic Programmer's Guide. *Programming Ruby The Pragmatic Programmer's Guide* [online]. 2001 [cit. 2014-06-24]. Available at WWW: <http://ruby-doc.com/docs/ProgrammingRuby/html/preface.html>

THOMAS, D., C. FOWLER a A. HUNT. *Programming Ruby 1.9 & 2.0*. 3rd Edition. Raleigh: The Pragmatic Bookshelf, 2013. 863 p. ISBN 978-1-93778-549-9.

YONG, M. K. Mkyong. *Postgresql Point-In-Time Recovery* [online]. 30. 8. 2012 [cit. 2014-11-8]. Available at WWW: <http://www.mkyong.com/database/postgresql-point-in-time-recovery-incremental-backup/>

8 Figures

9 Tables

10 Appendix

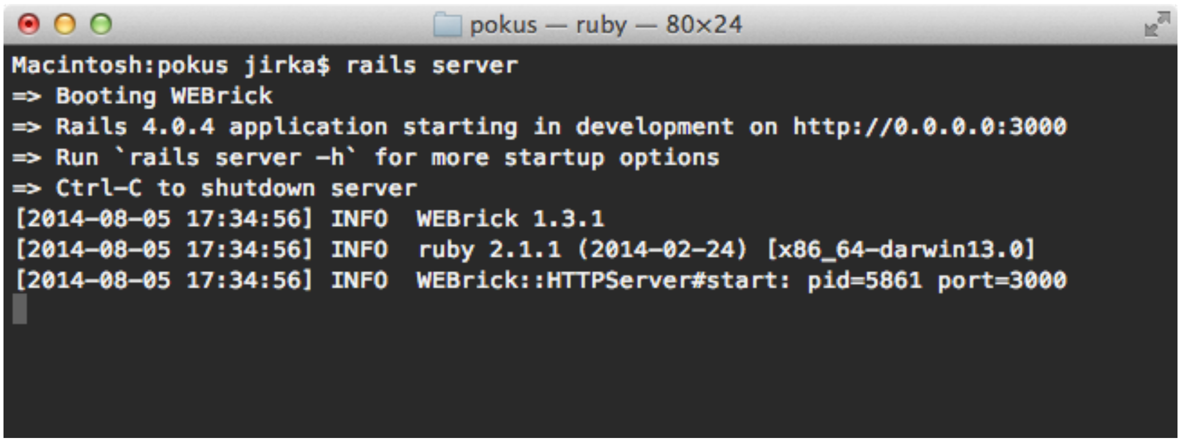
10.1 Lecture's controller

```
class LecturesController < ApplicationController
  before_action :set_lecture, only: [:show, :edit, :update, :destroy]

  # GET /lectures
  # GET /lectures.json
  def index
    @lectures = Lecture.all
  end

  # GET /lectures/1
  # GET /lectures/1.json
  def show
  end

  # GET /lectures/new
  def new
    @lecture = Lecture.new
  end

  # GET /lectures/1/edit
  def edit
  end

  # POST /lectures
  # POST /lectures.json
  def create
    @lecture = Lecture.new(lecture_params)

    respond_to do |format|
      if @lecture.save
        format.html { redirect_to @lecture, notice: 'Lecture was successfully
created.' }
        format.json { render action: 'show', status: :created, location: @lecture
}
      else
        format.html { render action: 'new' }
        format.json { render json: @lecture.errors, status: :unprocessable_entity
}
      end
    end
  end

  # PATCH/PUT /lectures/1
  # PATCH/PUT /lectures/1.json
  def update
    respond_to do |format|
      if @lecture.update(lecture_params)
        format.html { redirect_to @lecture, notice: 'Lecture was successfully
updated.' }
        format.json { head :no_content }
      else
        format.html { render action: 'edit' }
        format.json { render json: @lecture.errors, status: :unprocessable_entity
}
      end
    end
  end

  # DELETE /lectures/1
  # DELETE /lectures/1.json
  def destroy
    @lecture.destroy
    respond_to do |format|
      format.html { redirect_to lectures_url }
      format.json { head :no_content }
    end
  end

  private
    # Use callbacks to share common setup or constraints between actions.
    def set_lecture
      @lecture = Lecture.find(params[:id])
    end

    # Never trust parameters from the scary internet, only allow the white list
through.
    def lecture_params
      params.require(:lecture).permit(:name)
    end
end
```

10.2 Lectures list view

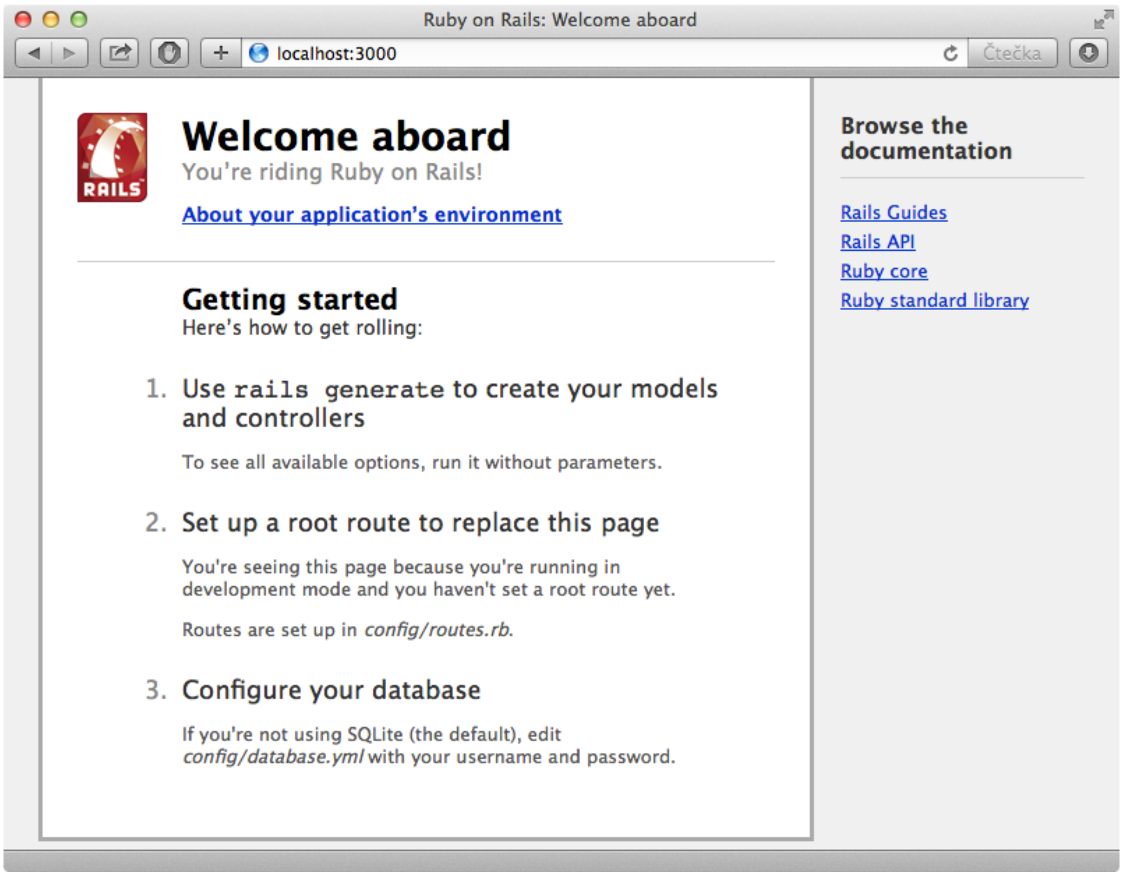
```
<h1>Listing lectures</h1>

<table>
  <thead>
    <tr>
      <th>Name</th>
      <th></th>
      <th></th>
      <th></th>
    </tr>
  </thead>
  <tbody>
    <% @lectures.each do |lecture| %>
      <tr>
        <td><%= lecture.name %></td>
        <td><%= link_to 'Show', lecture %></td>
        <td><%= link_to 'Edit', edit_lecture_path(lecture) %></td>
        <td><%= link_to 'Destroy', lecture, method: :delete, data: { confirm:
'Are you sure?' } %></td>
      </tr>
    <% end %>
  </tbody>
</table>
<br>

<%= link_to 'New Lecture', new_lecture_path %>
```

10.3 Start of WEBrick web server



```
Macintosh:pokus jirka$ rails server
=> Booting WEBrick
=> Rails 4.0.4 application starting in development on http://0.0.0.0:3000
=> Run `rails server -h` for more startup options
=> Ctrl-C to shutdown server
[2014-08-05 17:34:56] INFO  WEBrick 1.3.1
[2014-08-05 17:34:56] INFO  ruby 2.1.1 (2014-02-24) [x86_64-darwin13.0]
[2014-08-05 17:34:56] INFO  WEBrick::HTTPServer#start: pid=5861 port=3000
```

10.4 Welcome page of Rails



Welcome aboard
You're riding Ruby on Rails!
About your application's environment

Getting started
Here's how to get rolling:

1. Use `rails generate` to create your models and controllers

   To see all available options, run it without parameters.

2. Set up a root route to replace this page

   You're seeing this page because you're running in development mode and you haven't set a root route yet.

   Routes are set up in *config/routes.rb*.

3. Configure your database

   If you're not using SQLite (the default), edit *config/database.yml* with your username and password.

Browse the documentation

Rails Guides
Rails API
Ruby core
Ruby standard library

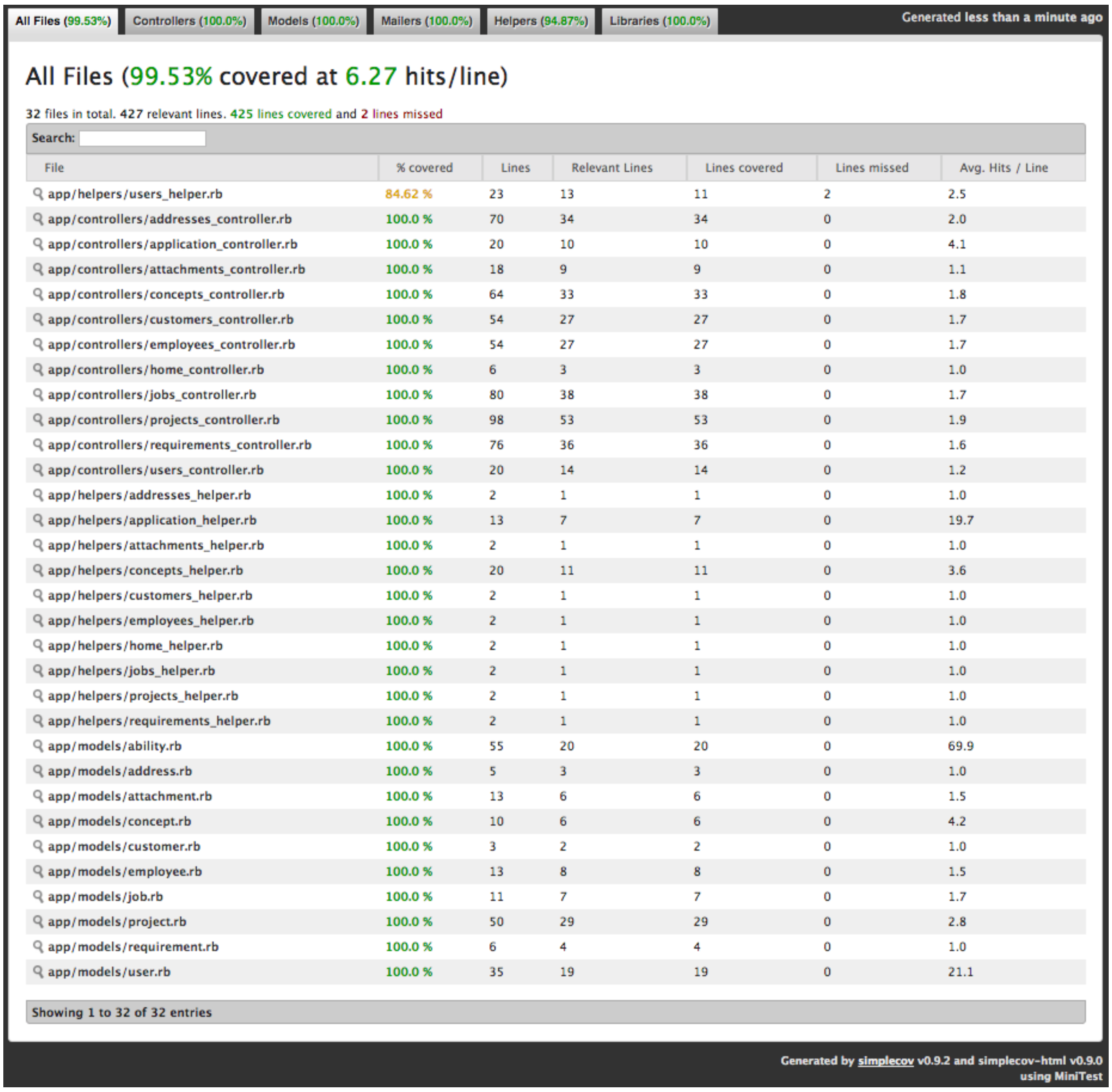
10.5 Migration file with up and down methods

```
class AddNoteToLecture < ActiveRecord::Migration
  def up
    add_column :lectures, :note, :string
  end

  def down
    remove_column :lectures, :note
  end
end
```

10.6 Migration file with change method

```
class AddNoteToLecture < ActiveRecord::Migration
  def change
     add_column :lectures, :note, :string
  end
end
```

10.7 Test coverage result by Simplecov



All Files (99.53%) | Controllers (100.0%) | Models (100.0%) | Mailers (100.0%) | Helpers (94.87%) | Libraries (100.0%)

Generated less than a minute ago

All Files (99.53% covered at 6.27 hits/line)

32 files in total. 427 relevant lines. 425 lines covered and 2 lines missed

Search:

File	% covered	Lines	Relevant Lines	Lines covered	Lines missed	Avg. Hits / Line
app/helpers/users_helper.rb	84.62 %	23	13	11	2	2.5
app/controllers/addresses_controller.rb	100.0 %	70	34	34	0	2.0
app/controllers/application_controller.rb	100.0 %	20	10	10	0	4.1
app/controllers/attachments_controller.rb	100.0 %	18	9	9	0	1.1
app/controllers/concepts_controller.rb	100.0 %	64	33	33	0	1.8
app/controllers/customers_controller.rb	100.0 %	54	27	27	0	1.7
app/controllers/employees_controller.rb	100.0 %	54	27	27	0	1.7
app/controllers/home_controller.rb	100.0 %	6	3	3	0	1.0
app/controllers/jobs_controller.rb	100.0 %	80	38	38	0	1.7
app/controllers/projects_controller.rb	100.0 %	98	53	53	0	1.9
app/controllers/requirements_controller.rb	100.0 %	76	36	36	0	1.6
app/controllers/users_controller.rb	100.0 %	20	14	14	0	1.2
app/helpers/addresses_helper.rb	100.0 %	2	1	1	0	1.0
app/helpers/application_helper.rb	100.0 %	13	7	7	0	19.7
app/helpers/attachments_helper.rb	100.0 %	2	1	1	0	1.0
app/helpers/concepts_helper.rb	100.0 %	20	11	11	0	3.6
app/helpers/customers_helper.rb	100.0 %	2	1	1	0	1.0
app/helpers/employees_helper.rb	100.0 %	2	1	1	0	1.0
app/helpers/home_helper.rb	100.0 %	2	1	1	0	1.0
app/helpers/jobs_helper.rb	100.0 %	2	1	1	0	1.0
app/helpers/projects_helper.rb	100.0 %	2	1	1	0	1.0
app/helpers/requirements_helper.rb	100.0 %	2	1	1	0	1.0
app/models/ability.rb	100.0 %	55	20	20	0	69.9
app/models/address.rb	100.0 %	5	3	3	0	1.0
app/models/attachment.rb	100.0 %	13	6	6	0	1.5
app/models/concept.rb	100.0 %	10	6	6	0	4.2
app/models/customer.rb	100.0 %	3	2	2	0	1.0
app/models/employee.rb	100.0 %	13	8	8	0	1.5
app/models/job.rb	100.0 %	11	7	7	0	1.7
app/models/project.rb	100.0 %	50	29	29	0	2.8
app/models/requirement.rb	100.0 %	6	4	4	0	1.0
app/models/user.rb	100.0 %	35	19	19	0	21.1

Showing 1 to 32 of 32 entries

Generated by simplecov v0.9.2 and simplecov-html v0.9.0
using MiniTest