International University of Sarajevo

Faculty of Engineering and Natural Sciences

Department of Computer Science and Engineering

The development of web frameworks with an example framework made in JAVA programming language

by

Zaid Zerdo

Supervisor

Associate Professor Jasminka Hasić Telalović

Sarajevo, 2016

APPROVAL PAGE

I certify that I have supervised and read this study and that in my opinion, it confirms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a thesis for the degree of Bachelor of Science in Computer Science and Engineering.

_____

Supervisor

I certify that I have read this study and that in my opinion, it confirms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a thesis for the degree of Bachelor of Science in Computer Science and Engineering.

_____

Examiner

I certify that I have read this study and that in my opinion, it confirms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a thesis for the degree of Bachelor of Science in Computer Science and Engineering.

_____

Examiner

This dissertation was submitted to the Department of Computer Science and Engineering, and is accepted as a partial fulfillment of the requirements for the degree of Bachelor of Science in Computer Science and Engineering.

_____

Program Coordinator

This dissertation was submitted to the Faculty of Engineering and Natural Science and is accepted as a partial fulfillment of the requirements for the degree of Bachelor of Science in Computer Science and Engineering.

_____

Dean, Faculty of Engineering and Natural Sciences

DECLARATION

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

……..

Signature

……..

Date

COPYRIGHT PAGE

International University of Sarajevo

Declaration of Copyright and Affirmation of Fair Use of Unpublished Research

Copyright © by Zaid Zerdo. All rights reserved.

The development of web frameworks with an example framework made in JAVA programming language

No parts of this unpublished research may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording or otherwise without prior written permission of the copyright holder and IUS Library.

Affirmed by: Zaid Zerdo

....

Signature Date

TABLE OF CONTENTS

ABSTRACT

Web application frameworks have risen in popularity in the last decade. Such applications enable easier development of web applications, because frameworks come with a large built-in set of tools that are essential to web application development, ex. handling clients with ease, MVC pattern support, JSON support, database connection handler, HTTP request support, etc. In order to demonstrate the development of such tools, an example framework, written in the JAVA programming language from scratch with only two external libraries used (JSON and SQL related), is constructed and example code is given throughout the chapters. The framework is constructed at the level of socket programming, i.e. raw socket manipulation is used to make the server, acquire and handle clients, etc. The constructed framework is just a prototype and points of improvement are also given. A prototype web application, alongside a RESTful service, is made on the foundations of the prototype framework.

Keywords: web frameworks, framework construction and implementation, JAVA implementation, HTTP requests, client handling threads, MVC design pattern, JSON, SQL, REST, front-end technologies

1. INTRODUCTION

The simplest definition of a web framework is that it is a collection of tools that helps in the development and construction of quality websites, which is something that has become quite difficult as many papers have identified [2]. Depending on the framework, besides the development of websites, various web APIs can be made on that framework, as well as web libraries and web resources.

Throughout the years, a multitude of web frameworks in various languages have been written, ex. ASP .NET framework written in C#, Spring framework written in JAVA and AngularJS in Javascript. A single programming language has seen the development of many web frameworks, such as Spring and PLAY, both of which were written in JAVA.

Developing a web framework is a long-term process, which does not end. Discontinuing support of a web framework, i.e. stopping its development, is problematic for the framework. In order for a framework to be popular and contemporary, it has to constantly evolve. Having an enormous user base is crucial, as well, since a community keeps the framework alive.

This thesis will attempt to go through the process of web framework development. The web framework was developed in JAVA 8, using only a few external libraries and a few ideas from the three-tiered architecture [6][7][8]. Such a thing is usually seen as “reinventing the wheel”, but understanding the key concepts of a web framework only enhances the understanding of other frameworks, since almost all web frameworks have a set of characteristics that they share, such as HTTP request parsing, client handling and so on.

1.1. A POINT ON EXCEPTION HANDLING

Handling exceptions has become a vital part of long-term development of any application, since such a thing brings ease in debugging. Investment in exception handling code is an investment in stability, as well.

Concepts such as error logging and detailed exception handling are not in the scope of this thesis and all exception handling code will be shortened in order for the code to be easily readable. The catch blocks should contain stability-inducing code, as well as logger code which will log all the warnings and errors that occur in the execution of an application like this.

2. FRAMEWORK CONSTRUCTION

Framework development is a long and tedious process and its development is outlined in many papers and books [3]. Many aspects have to be developed at the same time and testing can be quite difficult. Identifying the starting point of development is key. The starting point of this thesis is the client handling part of the application.

2.1. CLIENT HANDLING

The core aspect of the framework must be client handling. Such a process involves the acceptance of a client, which connects to a raw socket object, and then the server allocates CPU time for the client to be handled. Thinking about the client while developing the server [4] can be quite difficult, but is essential since the whole point is to connect the two segments.

Multithreading is the natural way to go here, since serial execution in such an application where there are many isolated parts does not make much sense. The multithreading structure in this thesis is outlined in the following subchapters.

2.1.1. SERVER THREAD

The server thread is a thread that accepts clients connecting to its socket, with the right port value, and then rerouting the execution to a different thread. This type of thread should be separated from the main thread of the program, especially if there is any GUI involved [9].

```
private class ServerThread extends Thread {

        @Override
        public void run() {
            while (true) {
                 Socket client = server.accept();

                 clientQueue.add(new HTTPClient(client));
            }
        }
}
```

Code 1 – A single thread is made, which accepts client sockets and then puts the clients into a queue, which will then handle the client independently from this thread. This thread will continue its work immediately after creating the HTTP client. It is important to note that the client queue is shared between a large number of threads, which is fine since the data structure used is thread safe, i.e. has built-in code to handle concurrency.

```
public Server(int portNumber) throws IOException {
    server = new ServerSocket(portNumber);
}

public void startServer() {
    serverThread = new ServerThread();

    for (int i = 0; i < listenerThreadNumber; i++) {
          clientHandlers.add(new HTTPClientHandler(i, clientQueue, this));
    }

    serverThread.start();
}
```

Code 2 – The encapsulating class which hides all the details of the server should contain methods like these, as well the thread class from the previous code snippet.

When it comes to exception handling, not many exceptions can pop-up in this thread. Most exceptions, such as inability to create the server socket and security exceptions would occur before starting the server.

It is only natural to encapsulate the server into a separate class and add methods that are crucial to the execution of the server thread, such as a method that starts the server shown in Code 2 – The encapsulating class which hides all the details of the server should contain methods like these, as well the thread class from the previous code snippet. shown above.

2.1.2. HANDLING CLIENTS

```
Routes routes = server.getRoutes();

while (true) {
        HTTPClient client = clientQueue.take();

        String requestString = client.readFromClient();

        Request request = new Request(requestString, client.getIP());

        if (request.isValid()) {
                String filename = routes.getFilenameForRequest(request);
                ControllerHeader controller = routes.getControllerForRequest(request);

                if (filename != null) {
                        client.sendToClient(new File(filename), false);
                } else if (controller != null) {
                        View view = Invoker.invoke(controller, request);

                        boolean loggedIn = Sessions.getEmailFromIP(client.getIP()) != null;
                        int type = Sessions.getTypeFromIP(client.getIP());
                        client.sendToClient(Parser.parseView(view, loggedIn, type), true);
                }
        }

        client.closeConnection();

}
```

Code 3 – The code above represents a client handler, i.e. a thread that has been started in order to handle clients. Concepts such as requests, controllers, views, sessions and invokers will be covered in later chapters, but their usage is shown in the code.

Handling a client should not be a complex process since client handling is a stateless process. Such a process does not preserve any data between client sessions. The process is quite simple, the client asks the server (sends a request) and the server replies with data. No data is saved anywhere, except for logging, perhaps.

The stateless process can be seen in Code 3 and goes as follows:

- The handler thread repeats forever a process of taking clients from the queue. It is important to note that the take method is a blocking method (will wait until a client is found) since the data structure is a linked blocking queue.
- When a client is found, its request string is read and then a request object is made. Everything is high-level here and will be explained later.
- If the request is valid then a process of finding the right response for the client starts.
    - If the client requests a file, such as a script file or a style sheet, the server will open a stream in order for it to send the file to the client.
    - If the request the client made fits with a certain controller (MVC model), then the controller is called and the controller gets the responsibility of serving the client accordingly. Before giving up control to the controller, the handler also checks whether there is an opened session and whether the client is logged in.
- Finally, the client socket must be closed in order to signal the client that the request has been server and that it received everything that it asked for. Internet browsers wait for this signal before loading up the page.

2.1.3. PRODUCER-CONSUMER METHODOLOGY

This type of methodology presents a way of making sure that the overall server architecture is stable, i.e. works as intended. It fits the need since the architecture has clear consumers (client handlers) and things to consume (client requests). It can be debated that the server is actually the only producer here since it produces the clients, which will then send the requests that need to be consumed. Both aspects of the framework have been explained before. In this chapter the methodology implementation will be explained.

The queue of clients has been mentioned before. It is a linked blocking queue of HTTP clients that have been produced in the server code as shown in Code 1. Client handlers do not need a queue, since they all operate in parallel. A simple list, such as an

array list, can be used to store the client handlers for future reference. Again, the natural structure of the producer-consumer methodology in this case might be debatable, but the overall concept has been shown to work without hiccups.

Due to the structure of the producer-consumer implementation, it is quite easy to change the number client handlers, since it does not impact the overall execution. By default, I have used four client handlers, but the number can be changed by the user of the framework easily.

2.2. HTTP REQUESTS

In general HTTP request is a formal request sent by the client to the server asking for a certain resource or data. HTTP requests have been standardized throughout the past decades. Those standards have been partially followed in the development of this framework.

The most obvious descriptor of HTTP requests is the type of request. The following types are supported by this framework:

- POST – request type that is usually associated with data sending (posting), done by the client. The data is contained in the request body and needs to be parsed differently than the usual request arguments. Such requests are used when hiding sensitive information, which can be shown in other request types, and is used when editing or updating existing resources or adding new ones.
- GET – request type that is associated with pure resource requests, i.e. requesting a resource or a representation of a given route. This type of request is NOT used when a change in the resource is expected as the result of this request.
- RESOURCES – an informal request type used in this project in order to indicate file resources that are present on the server and the client requests them. Files like this include images, script files, style sheets, etc.

There are many other request types that are standardized, like PUT, DELETE and UPDATE, that are not present in this framework. The given three (one of which is not standardized) proved to be enough for the framework to function adequately.

POST and GET have many small differences, ex. only one of them can be bookmarked by the browser, that are not mentioned in the bullet points above. Both

```
String[] lines = data.split("\n");

if (lines.length < 5) {
        isValid = false;
        return;
}

String[] basicInfo = lines[0].split(" ");

if (basicInfo[0].equals("GET")) {
        type = GET;
} else if (basicInfo[0].equals("POST")) {
        type = POST;
} else {
        type = NONE;
        isValid = false;
}
```

Code 5 – Code that splits the request string into lines and then figures out what is the request type, either GET, POST or something else. Another implementation includes an array that has all the types and then the code would contain a loop that iterates through the types.

```
if (basicInfo.length >= 3) {
        route = basicInfo[1];

        for (int i = 2; i < basicInfo.length - 1; i++) {
                route += " " + basicInfo[i];
        }

        if (route.contains("?")) {
                int start = route.indexOf("?") + 1;
                String[] data = route.substring(start).split("&");

                for (String d : data) {
                        String[] split = d.split("=");
                        if (split.length == 1) {
                                getData.put(Parser.parseHTMLString(split[0]), "");
                        } else {
                                getData.put(Parser.parseHTMLString(split[0]),
                                        Parser.parseHTMLString(split[1]));
                        }
                }

                route = route.substring(0, start - 2);
        }
}
```

Code 4 – Represents a way to acquire all the arguments from the request.

POST and GET data is stored in maps, in which keys are the parameter names and the map values are the parameter value/data.

```
if (!lines[lines.length - 1].equals("POSTDATA")) {
        String[] keyPairs = lines[lines.length - 1].split("&");
        for (String kp : keyPairs) {
                String[] splitted = kp.split("=");

                if (splitted.length == 1) {
                        postData.put(Parser.parseHTMLString(splitted[0]), "");
                } else {
                        postData.put(Parser.parseHTMLString(splitted[0]),
                                        Parser.parseHTMLString(splitted[1]));
                }
        }
}
```

Code 6 – Represents a way to parse the POST data and put them in a map.

2.3. CONTROLLERS AND THE ROUTE INVOKER

The most important information present in the request is the request route. Routes can range from a sub-site from the main website or a resource present on the server. Every request route must be handled, even routes that are invalid.

```
public class ExampleController extends Controller {

        public View routeIndex(Request request) {
                return new View("src/examples/index.html");
        }

        public View routeUserLogin(Request request) {
                return new View("src/examples/login.html");
        }

        public View routeUserLoginConnection(Request request) {
                String mail = request.getPostData("email");
                String password = request.getPostData("password");

                System.out.println(mail);
                System.out.println(password);

                System.out.println(ExampleModel.isExistingInDatabase(mail, password));

                return new View("src/examples/login2.html", request.getAllPostData());
        }
}
```

Code 7 – An example of how a controller might work. The class contains action controller methods, one rerouting to the index page, one to the login page and one manipulating request data in order to reroute to the page once the user is logged in.

To handle the requested route an entity called a controller is needed. Every route is bound to a single controller, which will be called when the requested route is called.

As seen in Code 6 controllers are quite important and should be kept short as it is a good practice [5]. The example does not show how the controllers are called and invoked. That is done by the route invoker and the controller superclass.

```
public static class ControllerHeader {
        private Class<? extends Controller> controllerClass;
        private String methodName;

        public ControllerHeader(Class<? extends Controller> controller, String method) {
                controllerClass = controller;
                methodName = method;
        }

        public Class<?> getControllerClass() {
                return controllerClass;
        }

        public String getControllerMethodName() {
                return methodName;
        }
}
```

Code 9 – The controller header only contains the actual controller and method name.

```
public static View invoke(ControllerHeader controller, Request request) {
        Class<?> controllerClass = controller.getControllerClass();
        Object controllerObject = controllerClass.newInstance();
        String controllerName = controller.getControllerMethodName();
        Method controllerMethod = controllerClass.getMethod(controllerName, Request.class);

        controllerMethod.setAccessible(true);
        return (View) controllerMethod.invoke(controllerObject, request);
}
```

Code 8 – Invoker which invokes the controller for the given request. In order to invoke a certain method from a certain class the built-in reflection JAVA API must be used. Before this method is called the appropriate controller header is found for the needed request. Besides that, exception handling code is omitted.

```
public ControllerHeader getControllerForRequest(Request request) {
        return controllerRoutes.get(request);
}
```

Code 10 – Through the magic of maps, finding the right controller for a given request is done in only one line of code. Of course, it is encouraged to encapsulate it in an instance method.

2.4. SERVER-SIDE PSEUDO-LANGUAGE PARSER

Web frameworks usually come with a server-side language written in files that are front-end oriented. One example is the use of the Scala language in the JAVA's PLAY web framework. Even PHP shares some similarities to that, since that language is implanted into HTML files/code, which later will be translated to HTML anyway.

```
<html>
    <head>
        #- +src/view/includes.html -#
        <title>Astronomia!</title>

    </head>

    <body>
        #- +src/view/navbar.html -#

        <div class="panel panel-default">
            <div class="panel-body">
                <h2>News</h2>
            </div>
        </div>

        <div class="panel panel-default">
            <div class="panel-body">
                <h3>#- @title -#</h3>
                <img src="#- @image -#">
                #- @body -#
            </div>
        </div>
...
```

Code 11 – The bolded lines of code represent the server-side language present in front-end HTML files in the web framework demonstrated in this paper. Keep in mind that the bolded lines of code will be replaced by the appropriate HTML lines of code once the parser parses the file.

With the growth of web pages that are not 100% static, i.e. always displayed the same data, the need for dynamic sky-rocketed. In order to accomplish this, server-side code was added to front-end data, code which will be translated to the according front-end code once the parser parses the server code.

2.4.1. "COMPILER" DEVELOPMENT AND THE LIMITATIONS

Designing a compiler for that language is needed, since such a language must support conditional statements, loops, etc. Compiler development is a thesis for itself

and will not be covered in this one. Instead of developing a full-fledged compiler for this thesis, a pseudo-compiler, more precisely a simple parser, “compiles” and translates the server code into front-end code.

The limitations of designing an alternative to a compiler is the problem of the absence of parse trees and statement grammar evaluation. As such, the statements written will be quite rigid, not versatile at all, meaning that they must follow a strict pattern of spaces, letters and so on. The parser follows more precisely the designing structure of a pattern recognition application than a compiler.

2.4.2. SUPPORTED COMMANDS

The pseudo-compiler supports the following commands:

- Conditional “if” statements – statements that check whether a certain condition is satisfied before executing the code that is present in the body of the statement.
- Include “+” statements – statements that include all the lines of code from a different specified file and substitutes the include statement with it.
- Include data “@” statements - statements that are substituted with data that is provided beforehand.
- Loop “repeat” statements - statements that are repeated through a set of data given beforehand.
- Session “loggedin” conditional variables – used in conjunction with “if” statements to check whether the user is logged in.

Again, since the compiler is substituted with a simple pattern matching application all of the mentioned commands and variables are quite inflexible, ex. the repeat statement does not support iteration through a given integer value and the conditional statements only support boolean variables as conditions.

The actual parser code is a bit too huge to fit here, containing a few hundred lines of code of checks and raw text parsing code. A few examples are provided on the next page on how the parser works.

```
<div class="panel panel-default">
            <h3>Comments</h3>
            #- repeat comments -#
            <div class='row'>
                <div class='col-sm-8'>
                    <div class='panel panel-white post panel-shadow'>
                        <div class='post-heading'>
                            <div class='pull-left image'>
                                <img src='#- *comments userimg -#' class='img-circle avatar'
                                          alt='user profile image'>
                            </div>
                            <div class='pull-left meta'>
                                <div class='title h5'>
                                    <a href='#'><b>#- *comments user -#</b></a>
                                    made a post.
                                </div>
                                <h6 class='text-muted time'>At #- *comments time -#</h6>
                            </div>
                        </div>
                        <div class='post-description'>
                                <p>#- *comments comment -#</p>
                           </div>
                    </div>
                </div>
            </div>
            #- endrep comments -#
</div>
```

Code 13 – Example of the repeating statement that loops through all the comments supplied to the parser beforehand by the client handler's controller.

```
private static void writeToClient(String line, BufferedWriter wr, View view, int lineId) {
        if (line.contains("#-") && line.contains("-#")) {
                int start = line.indexOf("#-");
                int end = line.indexOf("-#");
                String before = line.substring(0, start);
                String after = line.substring(end + 2);
                String code = line.substring(start + 3, end - 1);
                String parsed = parse(code, view, lineId);
                if (!deleteBecauseIf && !deleteBecauseEmptyRepeat) {
                        wr.write(before + parsed + after);
                        wr.newLine();
                }
        } else if (!deleteBecauseIf && !deleteBecauseEmptyRepeat) {
                wr.write(line);
                wr.newLine();
        }
}
```

Code 12 – The parser goes line by line looking for the server-code indicator, the hash signs, before starting to parse the code. As indicated by the blue variables, i.e. instance variables, the process is not stateless between line processing, meaning that information must be kept between the lines, ex. must be known whether the code is inside an if or a repeat statement.

2.5. SESSIONS IMPLEMENTATION

Sessions are specially stored data that are unique to every user and should, ideally, last as long the user is logged in.

```
#- if loggedin -#
        <div class="panel panel-default">
            <div class="panel-body">
                <h3>Comment on the news</h3>

                <form role="form" method="post" action="/comment/add">
                    <div class="form-group" hidden>
                        <input type="number" class="form-control" name="id" value="#- @id -#"
                                readonly>
                    </div>
                    <div class="form-group">
                        <textarea class="form-control" rows="5" id="comment"
                                name="comment"></textarea>
                    </div>
                    <p>
                        <button type="submit" class="btn btn-default">Post comment</button>
                    </p>
                </form>
            </div>
        </div>
#- endif -#
```

Code 14 – Example of a session used to control whether to display code displayable only to users that are logged in, i.e. have their IP logged into the session variables.

Session implementation varies drastically between frameworks. The biggest issue in session development is the security, i.e. preventing someone from forging a session and thus accessing private data like they are someone else. Some frameworks implement sessions by using hashes that cannot be decrypted in order to solve that problem.

The example framework implements sessions bound to IP addresses of users. That means that every user that logs in has their IP address logged into the session. The drawback of this method is that only one user can be logged in from a certain IP address, which sometimes can even be a positive thing.

Only two things are kept in sessions, which is another drawback, email and account type (admin or regular user) per IP address. This type of data storing is, again, perfect for maps, having keys the IP address, while the values are the emails and type.

```
public class Sessions {
	private static HashMap<String, String> ipEmailMap = new HashMap<>();
	private static HashMap<String, Integer> ipTypeMap = new HashMap<>();

	public static void addIPSession(String ip, String email, int type) {
		ipEmailMap.put(ip, email);
		ipTypeMap.put(ip, type);
	}

	public static boolean hasSession(String ip) {
		return ipEmailMap.containsKey(ip);
	}

	public static String getEmailFromIP(String ip) {
		return ipEmailMap.get(ip);
	}

	public static int getTypeFromIP(String ip) {
		if (ipTypeMap.containsKey(ip)) {
			return ipTypeMap.get(ip);
		} else {
			return -1;
		}
	}

	public static void removeSession(String ip) {
		ipEmailMap.remove(ip);
		ipTypeMap.remove(ip);
	}
}
```

Code 15 – Helper class that deals with all sessions. Note that the class deals with static methods and variables, making it impossible to use multiple sessions at once. A better implementation might be using the singleton pattern. It might be better to use a pair-like data structure in order to avoid storing the IP address twice, since two maps are used instead of one.

2.6. MVC SUPPORT

The MVC design pattern has grown popularity in the past few years, not only in web application development, but also in other types of application development, such as mobile applications. The hierarchy of this particular design pattern segments the project into several coherent wholes [1].

Three parts of the MVC pattern include:

- Model – part of the design pattern that handles data, especially concerning the data acquired from the database. Models are handled by controllers and in the end displayed by using views.

- View – part that deals with end-user front-end output representation. Views are manipulated by controllers and given model data to display data.
- Controller – part of the pattern that handles input and deals with both models and views. Typically, controllers are bound to request routes.

2.6.1. VIEWS

```
private String filename;
private String jsonString;
private HashMap<String, String> allPostData;
private JSONObject jsonPostData;
private boolean isJson;

public View(String filename) {
        this.filename = filename;
        isJson = false;
}

public View(String filename, HashMap<String, String> allPostData) {
        this(filename);
        this.allPostData = allPostData;
}

public View(String filename, JSONObject jsonPostData) {
        this(filename);
        this.jsonPostData = jsonPostData;
}

public View(JSONObject json) {
        isJson = true;

        jsonString = json.toString();
}

public View(int answer) {
        isJson = true;

        if (answer == JSON_ACCEPT) {
                jsonString = "{\"accept\": true}";
        } else if (answer == JSON_DECLINE) {
                jsonString = "{\"accept\": false}";
        }
}
```

Code 16 – Main view class, alongside with its attributes and constructors, which show that views are quite versatile and can be constructed from many different input data.

Valid views can be either filenames or JSON strings, while also containing POST data that might be used in the parser to inject data, as described in previous chapters. Note that it's filenames, and not the whole file as a string, since HTML data, and other front-end data, is contained in those files. The parser will handle everything related to that.

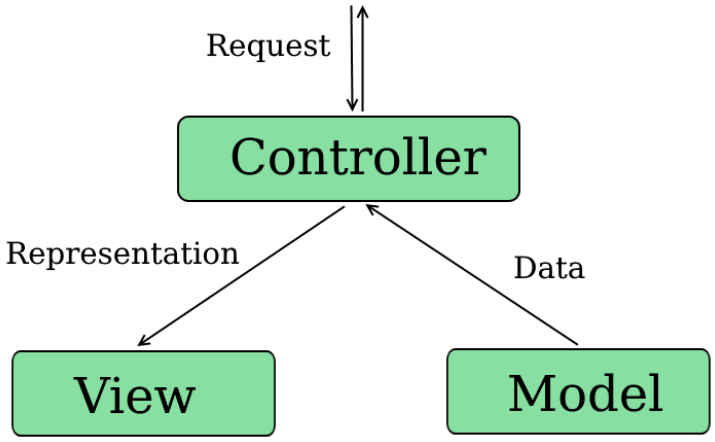

Figure 1 – The MVC model represented in a graph

2.6.2. MODELS AND CONTROLLERS

Both of these parts of the MVC were or will be explained in other chapters in more detail. Controllers have been explained in a chapter before, with the explanation of the super class, as well as with example controller/action methods. Models are closely connected with the SQL database and will be explained chapter 2.8.

2.7. JSON

```
$.ajax({
	type: "POST",
	url: "/register",
	data: {email: mail, username: username, pass1: pass1,
		pass2: pass2, check: true}, cache: false
}).done(function(result) {
	var json = $.parseJSON(result);
	if (!json["usernameexists"] && !json["mailexists"]) {
		form.submit();
	} else {
		if (json["usernameexists"]) {
			$("#errors").html("Username is already taken.");
		} else if (json["mailexists"]) {
			 $("#errors").html("Email is already registered.");
		}
		$("#errordil").show();
	}
});
```

Code 17 – Example JSON usage in Javascript AJAX, without exception handling code.

JSON became quite essential in the past few years in web application development due to the ease of use of a format as JSON. Implementing a JSON parser is a tedious process of satisfying all the criteria needed, tag detection and text parsing. Instead, I opted to import an existing JSON library that deals with all the JSON objects and arrays that are needed.

```
public String getPostValue(String key) {
        if (allPostData != null && allPostData.containsKey(key)) {
                return allPostData.get(key);
        }

        if (jsonPostData != null && jsonPostData.has(key)) {
                return jsonPostData.get(key).toString();
        }

        return "";
}
```

Code 18 – An example where JSON is useful. Post data can also be given in the form of a JSON, which is quite similar to maps, anyway.

2.8. SQL DATABASE CONNECTION

Writing a JAVA-SQL connector from scratch is also a difficult task, similar to making a JSON parser, so, again, I opted to use an external library. I used the mysql-connector-java-5.1.39 library, which handles all the SQL related problems.

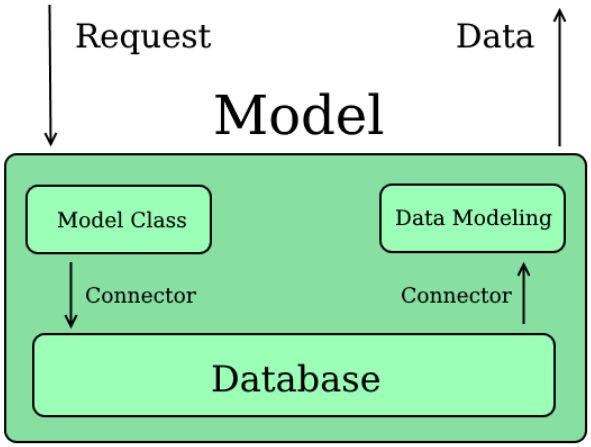

Figure 2 – Representation of a model in the MVC model

In the framework designed according to this thesis, models are the sole connection to the database. The connection is extremely lite-core. No ORM is used, only raw SQL statements are supported, as shown in the few code snippets.

```
public abstract class Model {

        protected static void saveToDatabase(String sqlStatement) {
                Server server = Server.getServerInstance();

                server.executeSQLInsertionStatementOnDatabase(sqlStatement);
        }

        protected static ResultSet loadFromDatabase(String sqlStatement) {
                Server server = Server.getServerInstance();

                return server.executeSQLQuerryStatementOnDatabase(sqlStatement);
        }

}
```

Code 20 – Any model must inherit this super class, and as such can use these protected methods directly. Obviously, the complexity of the database-connection is quite low. The two insertion and query methods are contained in the server and database class.

```
private Connection connection;

public Database(String url, String user, String password) {
                url = "jdbc:mysql://" + url;
                connection = DriverManager.getConnection(url, user, password);
}

public void executeInsertionStatement(String sqlStatement) {
                Statement statement = connection.createStatement();
                statement.setEscapeProcessing(true);
                statement.executeUpdate(sqlStatement);
                statement.close();
}

public ResultSet executeQuerryStatement(String sqlStatement) {
                Statement statement = connection.createStatement();
                statement.setEscapeProcessing(true);
                ResultSet result = statement.executeQuery(sqlStatement);

                return result;
}
```

Code 19 – The low-level code of SQL execution from the class that deals with databases.

3. POINTS OF IMPROVEMENT

The given web framework discussed in this thesis is just a prototype and in this chapter a few points of improvement will be discussed.

3.1. DYNAMIC CLIENT HANDLING

Handling client is essential to the performance and stability of the web application. Naturally, one PC, even with multiple threads, is not even close to fulfilling the need of multiple clients in popular web applications. A similar approach can be used as the one used here on a single PC.

Instead of having a single thread used for client accepting and multiple threads for client handling, a single machine can be used for client accepting and multiple machines, with multiple threads, can do client handling. So, instead of using threads on a single machine, use multiple machines in a single system.

Besides that, a dynamic system can be made which adds and removes machines to the system to balance to load and reduce the electricity bills as a result.

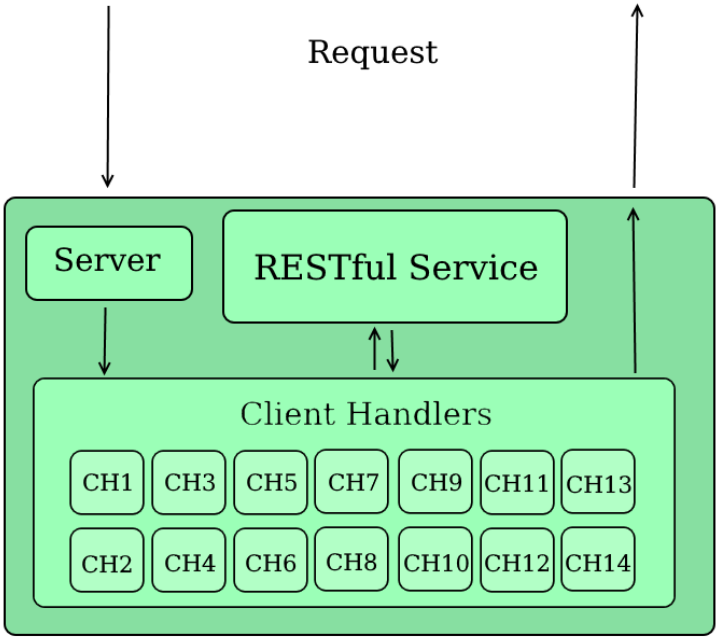

Figure 3 – A dynamic system of client handlers

3.2. A FULL-FLEDGED COMPILER

This is the obvious first thing to improve the server-side language used. Constructing a compiler is tedious process, but the fruits of a full-fledged compiler are many. Support for the following commands should be given:

- Variables and data acquisition
- Extended conditional statement support
- More flexible loop statements
- Integrated session support
- More built-in methods

3.3. BETTER REQUEST TYPE SUPPORT

There might be little value in implementing additional request types, such as UPDATE, DELETE, CONNECT, OPTIONS and PUT, but it only makes the framework more standardized.

3.4. BETTER SESSION SUPPORT

Currently, sessions only support email and type account storing. Not only that, but the sessions are quite distant from the framework user, i.e. sessions are built-in and not visible to the framework user. As such, the user cannot store any information into sessions. This is an important limitation that needs to be addressed.

Session security is also lacking since session keys are bound to IP addresses. That represents a “budget” implementation. A simple addition would be to also take the user’s browser and more information to make it more unique for one IP address.

4. EXAMPLES

4.1. CONTROLLERS

```
public View routeUserManagement(Request request) {
	String mail = Sessions.getEmailFromIP(request.getClientIP());
	int type = Sessions.getTypeFromIP(request.getClientIP());

	if (mail != null && type == Server.ADMIN_TYPE) {
		RestCall call = new RestCall(RestService.IP, "/users/all",
					RestService.PORT);
		HashMap<String, String> userData =
			UserModel.getAllUsersAsMapExceptUser(call.getJSONArray(),
		mail);
		return new View("src/view/userman.html", userData);
	} else {
		return new View("src/view/denied.html");
	}
}
```

Code 21 – Example of a controller method that routes an administrator to a site in which users are managed. The example also shows sessions, routing to views, passing data into views.

```
public MiserableView routeSpecificNews(MiserableRequest request) {
	String id = request.getGetData("id");

	RestCall call1 = new RestCall(RestService.IP, "/news/?id=" + id,
			RestService.PORT);
	JSONObject json1 = call1.getJSONObject();

	RestCall call2 = new RestCall(RestService.IP, "/comments/?id=" + id,
			RestService.PORT);
	JSONObject json2 = call2.getJSONObject();

	json2.put("id", json1.get("id"));
	json2.put("title", json1.get("title"));
	json2.put("image", json1.get("image"));
	json2.put("body", json1.get("body"));

	return new MiserableView("src/view/news.html", json2);
}
```

Code 22 – Example of a controller that routes a specific new article.

```
public View routeUserComment(Request request) {
        String ip = request.getClientIP();
        String mail = Sessions.getEmailFromIP(ip);
        String id = request.getPostData("id");
        if (mail != null) {
                RestCall call = new RestCall(RestService.IP, "/user/id/?mail=" + mail,
                                        RestService.PORT);
                JSONObject json = call.getJSONObject();
                String userId = String.valueOf(json.getInt("id"));

                String comment = request.getPostData("comment");

                new RestCall(RestService.IP,
                                        "/comments/add/?userid=" + userId +
                                        "&comment=" + comment + "&newsid=" + id,
                                RestService.PORT);

                RestCall call1 = new RestCall(RestService.IP, "/news/?id=" + id,
                                RestService.PORT);
                JSONObject json1 = call1.getJSONObject();
                RestCall call2 = new RestCall(RestService.IP, "/comments/?id=" + id,
                                        RestService.PORT);
                JSONObject json2 = call2.getJSONObject();

                json2.put("id", json1.get("id"));
                json2.put("title", json1.get("title"));
                json2.put("image", json1.get("image"));
                json2.put("body", json1.get("body"));
                return new View("src/view/news.html", json2);
        }
        return new View("src/view/denied.html");
}
```

Code 23 – Example of a controller method that handles user commenting on a news article. As much as four REST calls are made in a single method in order to handle data. It can also be seen that JSON is heavily used to manipulate data.

4.2. MODELS

```
public static void newNews(String title2, String image2, String body2) {
        saveToDatabase(String.format("INSERT INTO
                news(id, title, image, body)
                VALUES(null, '%s', '%s', '%s')", title2, image2, body2));
}
```

Code 24 – One of the static methods of the model that handles news articles. The code shows the method that saves to the database from the model's superclass. The null value for ID is used because auto-increment is used in the database. Even though an ORM is not used, doing like this is safe from SQL injections since the SQL statement is escaped in the method that saves to the database.

```
public static boolean isMailExistingInDatabase(String mail) {
        ResultSet set = loadFromDatabase(String.format("SELECT id FROM users
                WHERE email = '%s'", mail));
        return set.next();
}
```

Code 25 – A boolean method that checks whether an email exists in the database, which is used when checking the given mail during registration. The database method returns a set which needs to be iterated through. It is not needed to iterate through the whole thing, since the set is either empty or has only one element inside. The method "next" returns a boolean value indicating whether there are more elements to be iterated through.

4.3. VIEWS

```
    #- +src/view/navbar.html -#

    <div class="panel panel-default">
        <div class="panel-body">
            <h2>Login</h2>
        </div>
    </div>

        <div class="panel panel-default">
        <form role="form" method="post" action="/login">
            <div class="form-group">
                <label for="email">Email:</label>
                <input type="email" class="form-control" id="email" name="mail"
                                placeholder="Enter email">
            </div>
            <div class="form-group">
                <label for="pwd">Password:</label>
                <input type="password" class="form-control" id="pass" name="pass"
                                placeholder="Enter password">
            </div>

            <div class="alert alert-danger" id="errors" hidden>
                <strong>Error!</strong> Invalid email/password combination.
            </div>

            <button type="submit" class="btn btn-default">Submit</button>
        </form>

    </div>

    #- +src/view/footer.html -#
```

Code 26 – Front-end HTML code with the Bootstrap library. Server code to include the footer and header is also there. The code handles the login site, sending the form data to the /login route.

4.4. RESTFUL SERVICE

The best example of the framework is the construction of a separate REST API that is connected to the main application, i.e. two applications working together [10]. In theory, it should be quite easy to connect an Android application to the REST service.

```
public View routeAddNews(Request request) {
        String title = request.getGetData("title");
        String image = request.getGetData("image");
        String body = request.getGetData("body");

        try {
                NewsModel.newNews(title, image, body);
                return new View(View.JSON_ACCEPT);
        } catch (Exception e) {
                e.printStackTrace();
                return new View(View.JSON_DECLINE);
        }
}
```

Code 28 – Adding an item is quite easy with the segmented hierarchy present in the framework. The given code demonstrates a REST controller method that adds news articles from the given request.

```
public View routeTypeUser(Request request) {
        String mail = request.getGetData("mail");

        int type = UserModel.getUserType(mail);

        JSONObject json = new JSONObject();
        try {
                json.put("type", type);
        } catch (JSONException e) {
                System.err.println("Nonexpected internal JSON error.");
                e.printStackTrace();
                return new MiserableView(MiserableView.JSON_DECLINE);
        }

        return new View(json);
}
```

Code 27 – This REST controller method demonstrates how a REST method can return data to the caller in the form of a JSON object. The method is a bit too complex for a method that simply returns the type of user of the given mail. That is partially explained by the fact that the request is first routed to the REST service, which then connects to the database, acquires the data, constructs a JSON object and then sends the data to the request caller.

5. CONCLUSION

The development of web application frameworks, as well as web technology in general, will evolve rapidly and more contemporary ways to address problems described in this thesis will be found and implemented. One example is cloud computing, which will help solve the problem of client handling and server architecture. To a degree, such technology has already started going into mainstream web application development, one example being the Microsoft Azure service.

Difficulty that arises from web framework development will still be present. There are too many moving parts that need to be addressed, which need to work simultaneously. Another problem is that there are many part types, ex. a server-side language compiler is much different to the client handling methodology. Not only does the development require many different fields of computer science working together, but an advance in one of those fields requires an update to the framework in order for it to remain relevant.

REFERENCES

1. Reenskaug, Trygve, and James O. Coplien. "The DCI architecture: A new vision of object-oriented programming." An article starting a new blog:(14pp) http://www. artima. com/articles/dci_vision. html (2009).
2. Rosen, Deborah E., and Elizabeth Purinton. "Website design: Viewing the web as a cognitive landscape." Journal of Business Research 57.7 (2004): 787-794.
3. Fayad, Mohamed, Douglas C. Schmidt, and Ralph E. Johnson. Building application frameworks: object-oriented foundations of framework design. 1999.
4. Climushyn, Mel. "Web Application Architecture from 10,000 Feet, Part 1 – Client-Side vs. Server-Side," Atomic Object, 6 4 2015. [Online]. Available:

https://spin.atomicobject.com/2015/04/06/web-app-client-side-server-side/. [Accessed 9 7 2016].

5. Reehal J., "ASP.NET MVC Controller Best Practices – Skinny Controllers," Arrange Act Assert, 24 9 2009. [Online]. Available: https://web.archive.org/web/20150516021854/http://www.arrangeactassert.com/asp-net-mvc-controller-best-practices-%E2%80%93-skinny-controllers/. [Accessed 3 7 2016].
6. "Using Client Access Express in a three tier environment," IBM, [Online]. Available: http://publib.boulder.ibm.com/html/as400/v5r1/ic2933/index.htm?info/rzaii/rzaiithreetier.htm. [Accessed 5 9 2016].
7. "Three-Tiered Distribution," Microsoft, [Online]. Available: https://msdn.microsoft.com/en-us/library/ff647546.aspx. [Accessed 2 9 2016].
8. "Understanding the Three-Tier Architecture," Oracle, [Online]. Available: http://docs.oracle.com/cd/B25221_05/web.1013/b13593/undtldev010.htm. [Accessed 6 9 2016].
9. "main and the GUI Event Dispatch Thread," Leepoint, 2007. [Online]. Available: http://www.leepoint.net/JavaBasics/gui/gui-commentary/guicom-main-thread.html. [Accessed 5 9 2016].
10. Fielding, Roy Thomas. "Chapter 5: Representational State Transfer (REST)". Architectural Styles and the Design of Network-based Software Architectures (Ph.D.). University of California, Irvine. This chapter introduced the Representational State Transfer (REST) architectural style for distributed hypermedia systems. REST provides a set of architectural constraints that, when applied as a whole, emphasizes scalability of component interactions, generality of interfaces, independent deployment of components, and intermediary components to reduce interaction latency, enforce security, and encapsulate legacy systems. 2000.