



University College of North Denmark

AP Degree Program - Computer Science

DMAJ0914 – Group 57

Project Report

Group Members:

Miroslav Pakanec

Supervisors: Ronni Hansen

Submission date: Jun 6th 2016



AP Computer Science / DMAJ0914 – Group 57

Abstract:

This project consists of preparing, designing and implementing a web application for a transportation company. It consists of aspects from web development using open source technologies and database technologies courses.

In the project a new application is created from scratch. The system handles communication between customers, that would like to order a transportation, employees or transporters that will perform the transportation and managers of the company.

The software is a client-server web application, which uses multiple database technologies. The application was created using agile and SCRUM BUT development methodology.

Preface:

I am a forth semester student of the Computer Science course. I am 21 years old and I am coming from Slovakia.

First, I have had a difficulty to select a topic for the project, but when I heard about the opportunity to work for VIP transport, I snatched up to it immediately. It has been a wonderful experience to work with a real company. It was really about fulfilling customers requirements, rather than assuming that this is what customer might like. In addition, the software was developed with the mindset that it will be deployed, therefore I tried to avoid any mistakes and everything was developed with certain set of effort.

Problem statement:

Larger transport companies have a number of elements that have to be managed. Being precise and keeping track of transporters, customers, vehicles and fuel is essential to profit and reasonable management.

How can we design software that would help managers to keep track of these elements, easily collect data and produce understandable analysis in order to improve efficiency and reduce costs.

Table of contents

1	Introduction	7
2	Preliminary planning	8
2.1	Detailed problem statement.....	8
2.2	Requirements.....	8
2.2.1	Functional requirements.....	9
2.2.2	Non-functional requirements.....	9
2.3	Research.....	10
3	System development.....	10
4	Functional requirements	11
4.1	Use case model	11
4.2	Brief description	11
4.3	Domain model.....	12
5	Architecture	13
5.1	System architecture.....	13
6	Non-functional requirements	14
6.1	Security	14
6.1.1	Validation	14
6.1.2	Session management	15
6.1.3	Storing passwords.....	16
6.1.4	Cross site scripting	16
6.1.5	Cross site request forgery	17
6.2	Usability	18
6.2.1	Responsive design.....	18
6.2.2	Dynamic web application	19
6.2.3	Usability testing	20

7	Database technologies	21
7.1	Research.....	21
7.1.1	History of database technologies	22
7.1.2	Data.....	22
7.1.3	NoSQL databases	23
7.1.4	Relational and NoSQL database differences	25
7.1.5	CAP theorem	27
7.1.6	Polyglot persistence.....	29
7.2	Database implementation	29
7.2.1	Selecting a database	30
7.2.2	MySQL	31
7.2.3	MongoDB.....	36
8	Communication.....	44
8.1	HTTP.....	44
8.1.1	Structure of HTTP Transactions	44
8.1.2	Request Methods.....	45
8.1.3	Status codes.....	46
8.2	AJAX.....	46
8.3	AJAX implementation	47
8.4	Server response implementation	48
9	Conclusion	49
9.1	Project conclusion	49
9.2	Customer review	50
10	Bibliography.....	51

Figures and tables

Figure 3.1 - Story card	10
Figure 4.1 - Use case model	11
Figure 4.2 - Domain model	13
Figure 5.1 - System architecture	14
Figure 6.1 - Client-side validation of a date inputted by user	15
Figure 6.2 - Server-side validation of a date inputted by user	15
Figure 6.3 - Setting user's session after authentication	15
Figure 6.4 - Check if user is authorized to perform a request	16
Figure 6.5 - Example of invalidated user input - HTML and JavaScript	17
Figure 6.6 - Example of defense against XSS	17
Figure 6.7 - Example of CSRF in the application	17
Figure 6.8 - Sensitive data accessed by an attacker	17
Figure 6.9 - Web page displayed on iPhone 6 (resolution 375 x 667)	18
Figure 6.10 - Web page displayed on laptop (resolution 1600 x 900)	18
Figure 6.12 - Styled jQuery UI widget - date picker	19
Figure 6.11- Targeting an element with jQuery by its ID	19
Figure 7.1 - Database transaction states	26
Figure 7.2 - The CAP theorem	27
Figure 7.4 - The three guarantees of CAP theorem	28
Figure 7.3 - The CAP theorem	28
Figure 7.5 - Example of Consistency/ Partition tolerance	28
Figure 7.6 - Example of Availability/ Partition tolerance	29
Figure 7.7 - Polyglot persistence example	29
Figure 7.8- Referential integrity constraints displayed on the VIPTRANSPORT relational database schema	34
Figure 7.9 - Construction of connection object to MySQL database using mysqli library	34
Figure 7.10 - Example of prepare statement	35
Figure 7.11 - Example of binding parameter to prepare statement and its execution	35
Figure 7.12 - MySQL transaction 1	35
Figure 7.13 - MySQL transaction 2	35
Figure 7.14 - MySQL transaction 3	35
Figure 7.16 - Comparison of RDBMS and Document database concepts	36
Figure 7.17 - Successful start of MongoDB server	36
Figure 7.15 - MySQL transaction 4	36
Figure 7.18 - VIPTransport database stats	36
Figure 7.19 - Example of documents from collection 'notifications'	37
Figure 7.20 - Example of a document from collection 'transports'	37
Figure 7.21 - Example of returning collection object in DAL	38
Figure 7.22 - Getting notifications on a page load	39
Figure 7.23 - Getting notifications after user clicks 'Show more' button	39
Figure 7.24 - Example of an implementation of the find() method in PHP	40

Figure 7.25 - Example of aggregate operation in MongoDB	40
Figure 7.26 - Example of finding the number of unread notification using an aggregate search	40
Figure 7.27 - Example of aggregate implementation in PHP	41
Figure 7.28 - Example of map-reduce operation in MongoDB.....	41
Figure 7.29 - Example of querying documents by (transport) type and payment type	42
Figure 7.30 - Example of mapping and reducing documents from 'transports' collection	42
Figure 7.31 - Example of map-reduce implementation in PHP	43
Figure 7.32 - Aggregate compared to Map-reduce 1	43
Figure 7.33 - Aggregate compared to Map-reduce 2	43
Figure 8.1 - Example of request header when client requests statistics information	44
Figure 8.2 - Example of response header.....	45
Figure 8.3 - Commonly used AJAX options.....	47
Figure 8.4 - Example of AJAX implementation	47
Figure 8.5 - Example of Server response implementation	48
Table 2.1 - Prioritized list of non-functional requirements.....	9
Table 4.1 - Brief description of use cases.....	12
Table 6.1 - Usability test task list for different actors	20
Table 6.2 - Commonly mentioned system flaws and solutions.....	21
Table 7.1 - Example of SERVICE relation	32
Table 7.2 - VIPTRANSPORT database schema	33
Table 8.1 - Commonly used HTTP request methods.....	45
Table 8.2 - Possible response status codes	46

1 Introduction

The report focuses on creation and implementation of a software system for the company VIP transport. The system should handle communication between company managers, customer and employees. The report is divided into 7 parts:

1. Preliminary planning
2. System development
3. Functional requirements
4. Architecture
5. Non-functional requirements
6. Database technologies
7. Communication

Firstly, it was necessary to identify the problem in detail and find out, what customer's requirements are. The following was the research, which was necessary to fulfill these requirements. These concepts are explained in the preliminary planning section.

After the planning phase, the way how system will be developed needed to be identified. It was necessary to understand what would the satisfactory solution be for both customer and developer. In this phase, customer also provided story cards with detailed description of functionality.

The next phase was to create a functional requirements based on the story cards and discussion with the customer. In this section of the report, it is explained how the requirements were transferred into use case and domain model. System Architecture was created.

The fifth part focuses on non functional requirements. After they were briefly set in the preliminary planning phase, it was necessary to understand them in detail.

The section Database technologies was the next part of the project. This is a phase, which has been given a lot of research and focus. In fact it was one of the main parts of the project - understanding different available database technologies and trying to implement them if they are beneficial to the application. This section is divided into two main parts - research and implementation and it focuses on the process of selecting a database technology for the application. It is also a continuation to the previous section, because non-functional requirements such as performance, maintenance, availability, consistency and scalability are explained in detail.

The last section is communication. It describes how the client-server architecture works and how the components communicate with each other.

2 Preliminary planning

Preliminary investigation of the project consisted of several parts. First the problem had to be understood in detail and it was necessary to identify, what are the exact needs of the customer. This included functional and non-functional requirements. Since a lot of areas of the project required unknown technologies, it was necessary to do proper research. Sources of the research were carefully documented.

2.1 Detailed problem statement

Transportation companies have to keep track of number of things. Especially the ones that have multiple cars and several employees. There are two parts of the problem.

When companies grow, so does the number of customers they have to deal with. It can get difficult to manage all orders via email or messages. Therefore at a certain point, it is necessary for them to have a separate system that would handle these orders. With a simple functionality a large amount of workload can be released from manager himself and can be handled by a piece of software.

While the system should handle customer orders, there are other things that have to be managed. It is essential, that managers have an overview about all company's employees and cars. It must be clear, where each employee and car are at a given time and what route are they taking. This is necessary to avoid mistakes, such as confirming an order while cars or transporters are not available. On the other hand, a better understanding of how much orders the company is able to handle is given.

The final problem that was identified was handling payments, invoices and salaries. The manager should be easily informed on how many routes each employee made, how long has he been working (in this case travelling) and what was the distance of the transportation. These are all necessary variables for salary calculation.

2.2 Requirements

When the general problem was identified, it was time to set customer requirements. Understanding what customer needs are is essential for developing a useful product. While it is important to know what functionality customer desires, it is equally important to identify non-functional requirements and needs.

2.2.1 Functional requirements

During the discussion with the customer (the company owner), several requirements have been made, when it comes to systems functionality. Functional requirements include:

- Creating and updating orders
- User authentication and option to edit user profile
- Billing the company that the user is part of for transportation
- Confirming/ denying orders
- Viewing all employees
- Adding or updating a car
- Viewing services performed on a car
- Viewing all finished transportations
- Viewing all transportation passengers
- Calculating the revenue of the company
- Notifying managers about changes

2.2.2 Non-functional requirements

This type of requirements is essential for choosing the right technology for development. Some non-functional requirements contradict each other and it is necessary to understand where the company stands and how to draw a line in between them. A very clear example of this would be security and performance. Generally, more security means less performance and it is important to know what the company demands. It was necessary to answer questions such as: What sensitive information is going to be stored? How much data is going to be stored? Is data consistency necessary? How often does the system require maintenance? Is company expected to grow - is scalability a crucial factor?

After the dissolution a list of non-functional requirements was made. Requirements are split into two categories - must have (stared requirements that have the highest priority) and need to have. Non-functional requirements include:

Security*	<ul style="list-style-type: none"> • Confidential information accessible only by authorized users • All user passwords must be stored securely • Careful validation of user input • Authorized requests (avoiding Cross site request forgery)
Usability*	<ul style="list-style-type: none"> • Responsive design • Dynamic web application using java script
Performance*	<ul style="list-style-type: none"> • Fast database response time • Real-time statistics calculation
Scalability	<ul style="list-style-type: none"> • Number of database records may grow in the future
Maintenance	<ul style="list-style-type: none"> • System should allow easy code changes • Database schema might change
Consistency	<ul style="list-style-type: none"> • Some part of the system, such as payments and confirmations, require high data consistency

Table 2.1 - Prioritized list of non-functional requirements

2.3 Research

It was necessary to select the right technologies according to the requirements that have been set. First it was necessary to understand what different technologies exist and what are their strengths and weaknesses. This part was rather theoretical and required very wide research. When technologies were selected, it has taken several weeks to learn it, considering that almost whole stack was unknown. This applies to database technologies (MongoDB and MySQL) web development (PHP, JavaScript, JQuery, CSS and HTML) and communication (AJAX, JSON).

3 System development

Taking into consideration, that the system has been developed by an individual, it is rather difficult to label the process with a methodology. The process itself has been very agile - the plan has been created throughout the entire process of development.

The method can be characterized as SCRUM BUT, which means that some of the aspects of SCRUM were followed and some were not. The reason is that SCRUM is a method designed to manage team of developers, not individuals.

Beginning of the project was very SCRUM like. Customer has written story cards, which were then put into a product backlog. Then the development process eases from SCRUM practices. While there has been a product backlog, and story cards were labeled with high, medium and low priority, they really served as a guidance. On the weekly meeting with the customer, it has been said, what user stories to put into the sprint backlog, but the cards were neither prioritized not estimated.

ID	Story name	As a	I want to	So that	Acceptance criteria	Priority	Estimate
10	Edit cars	User (manager)	...be able access list of all cars. I will be able to edit the list (add/update /delete a car).	I can control the number and attributes of the cars I own.	All cars are shown. Buttons to edit or remove a car are next to each cars row. Button to add a car a visible. If car is to be removed confirmation is required.		

Figure 3.1 - Story card

During the development there were iterations very similar to sprints, but the process was not that well documented. In the beginning of the sprint, there was a meeting with the customer. It has started with the presentation, where the developer showed what is new in the system - the business value of the application. Afterwards the next sprint was planned according to customer's preferences and a simplified version of sprint backlog was created. Meetings happened regularly every ten days.

4 Functional requirements

4.1 Use case model

After discussion with the customer, requirements were analyzed and it was necessary to determine what different user types should be expected. Requirements were carefully considered and several use cases were created. In order to meet customers' requirements, it was necessary to have three different actors - customer, manager and transporter. Each actor has different privileges and can access different parts of the system.

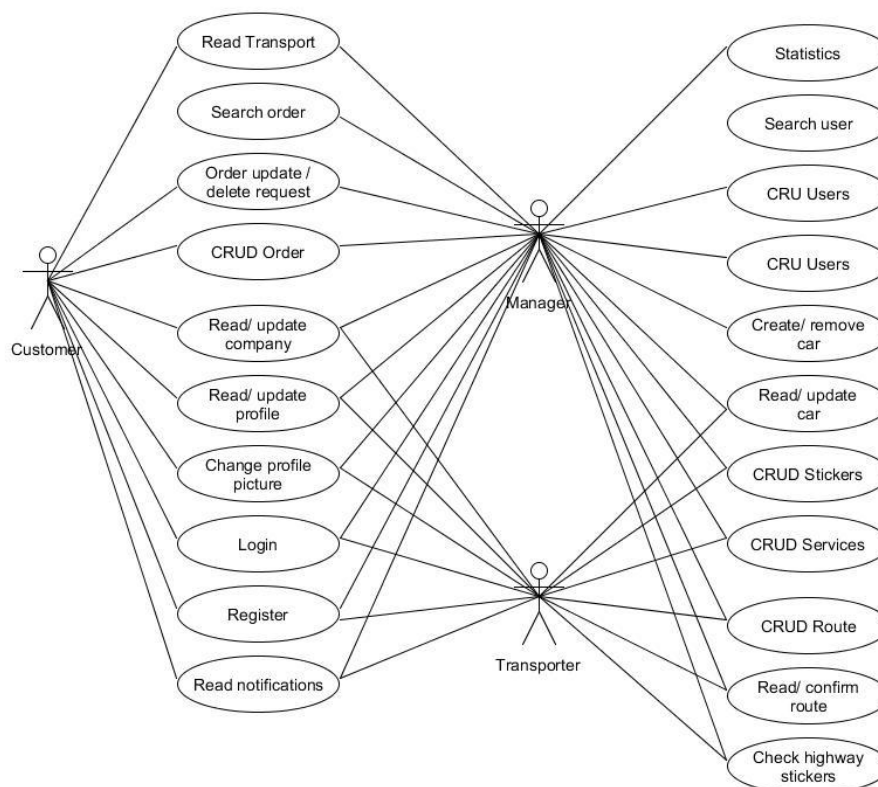


Figure 4.1 - Use case model

4.2 Brief description

Brief description of all requirements was created. This was enough to keep the development process on the right track, since the system was developed by one individual and communication with customer was frequent. Customer's requirements were also very broad, which gave a lot of freedom to the developer. In addition it was hard to prioritize these use cases. While transports and statistics have the highest priority, in order for them to be functional, other parts of the system have to be developed first. Due to this prioritizing was irrelevant.

UCN

Statistics	Get transport statistics based on selected option.
Read transport	Access list of transports (finished/ performed orders)
Search order	Help manage large sets of orders.
Order update/ delete request	If order is confirmed by manager, allow customer to send a request in a form of a message. Manager then either updates/ deletes the order or denies the request.
CRUD Order	CRUD order information in the application.
Read/ update company	User can add information about a company he is part of. User account will be associated with this company. Users order can be then paid by the company.
Read/ update profile	User can view and update his personal information and change password.
Change profile picture	User can upload and change his profile picture.
Login	To use the system, all users have to be authenticated.
Register	New users can be registered into the system.
Read notifications	Users are notified about changes that concern them and are able to view them.
Statistics	View transport statistics based on selected option, such as income per car, transporter or company.
Search user	Find users based on selected option, such as email, name, last name or type.
CRU users	Read and update user information or register new user.
Create/ remove car	Add a new car or delete existing car from the system.
Read/ update car	View all cars and update their attributes.
CRUD stickers	CRUD cars highway stickers in the system.
CRUD services	CRUD cars services in the system.
CRUD Route	Create route by confirming an order. View all current scheduled routes and update their information. Delete canceled routes.
Read/ confirm route	Read all details associated with a route (order and passenger details). If route was finished, confirm it and create finished transportation.
Check highway stickers	According to countries that route crosses, check if associated car has valid highway stickers.

Table 4.1 - Brief description of use cases

4.3 Domain model

Based on the requirements set by customer, domain model was created. Since there are three different actors, there has been a consideration of using inheritance. User would have three sub classes of customer, manager and transporter. It has been decided that only one class is going to be used. User types are going to be distinguished by the type attribute. The reason for this implementation is, that besides this attribute, classes would be similar and there was no need for further differentiation.

Transport is the central and most complex class. It takes information from all other classes - route, order, user, company and car. When it comes to privileges, user of type customer can only access order and company class. Manager can then confirm order and schedules a route, adding a user of type transporter and a car. Transporter cannot see orders or transports, while routes and cars are accessible to him.

UCN

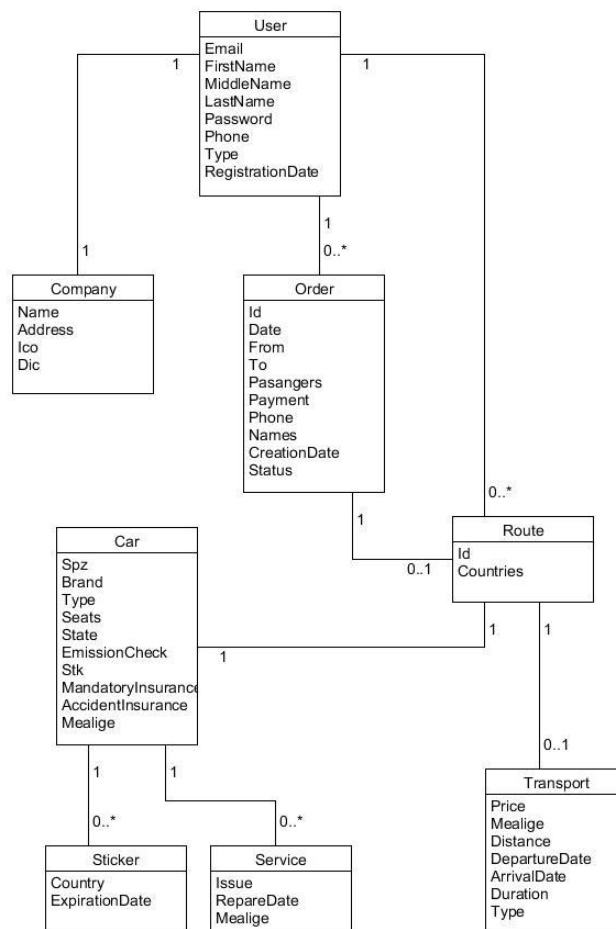


Figure 4.2 - Domain model

5 Architecture

5.1 System architecture

Architecture is a key part of the system that is going to be developed. It was decided that the application is going to have 3-tier architecture. This means that the system consists of three main components that communicate with each other by passing messages. Even though this components are running on the same physical machine, they are designed to run individual processes and therefore can be considered distributed. Each tier has a unique role in the system.

- presentation tier - accessed by user directly using a browser
- business tier - handles application logic and clients requests
- data tier - serves as data storage. In the application two different database technologies were used

Individual tiers are further distributed into layers. While a tier represents a physical machine a layer is a logical software component.

- presentation layer - holds presentation logic - how is data presented to client
- business layer - handles application logic, receives clients requests and returns responses
- data access layer - handles database connection, data mapping, creating and executing queries

Advantages:

- adaptable security
- scalability
- maintenance

Disadvantage:

- performance
- cost

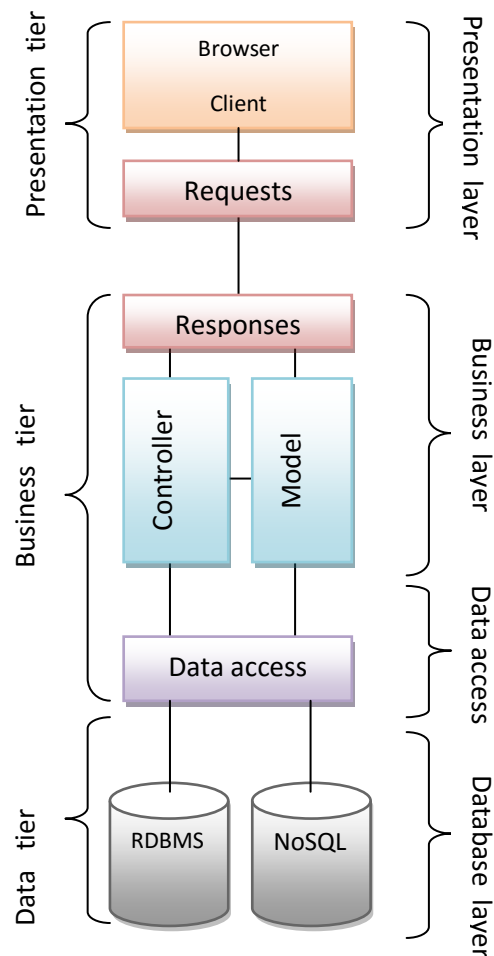


Figure 5.1 - System architecture

6 Non-functional requirements

In this section, non-functional requirements (set in 2.2.2) will be analyzed in detail.

6.1 Security

Security is one of the most important non-functional requirements of the system. In this section key security focuses will be discussed, as well as possible attacks and their avoidance.

6.1.1 Validation

While client-side validation is important and it can filter users input, it does not provide protection. It serves as better feedback to user - showing an error to user immediately improves usability and user can right away spot what is wrong.

Client-side validation by itself is dangerous for number of reasons and it is easy to bypass it and submit unwanted input to the server. (Long, 2008) It is important to keep in mind that the client cannot be trusted. In order to keep the application secured, server-side validation is necessary. This validation has to be at least equally strong or stronger, than the one on the client.

In the application, client-side validation is implemented using java-script. It is triggered by various events, most often when user leaves an input field (on blur) and afterwards when user submits the form by clicking on a button. On submit, all validations are triggered manually. Following method validates a date string on the client-side.

```
111     if(!value)
112         $(idErr).html('Cannot be empty').slideDown(500);
113     else if(!regex.test(value))
114         $(idErr).html('Incorrect format').slideDown(500);
115     else if(date == 'Invalid Date')
116         $(idErr).html('Invalid Date').slideDown(500);
117     else
118         $(idErr).html('').slideUp(500);
```

Figure 6.1 - Client-side validation of a date inputted by user

When data is submitted and received by server, it is validated again, before any other manipulations are performed. If the process fails, server returns 0 and notifies user. Following method validates a date string on the server-side and returns number of errors.

```
358     private function validateDateString($date, $format)
359     {
360         $d = DateTime::createFromFormat($format, $date);
361         if($d && $d->format($format) === $date)
362             return 0;
363
364         return 1;
365     }
```

Figure 6.2 - Server-side validation of a date inputted by user

6.1.2 Session management

A web session is a sequence of network HTTP request and response transactions associated to the same user. Modern and complex web applications require the retaining of information or status about each user for the duration of multiple requests. Therefore, sessions provide the ability to establish variables – such as access rights and localization settings – which will apply to each and every interaction a user has with the web application for the duration of the session (Siles, 2016).

In order to keep the authenticated state and track the user's progress within the web application, applications provide users with a session identifier. Application uses PHPs built in session management tools.

When a login form is submitted by a user, email and password are first validated and compared to data stored in a database. If inputted information match, session ID and other session variables, such as name or user type are set.

```
32     $_SESSION['email'] = $userModelObject->getEmail();
33     $_SESSION['fname'] = $userModelObject->getFname();
34     $_SESSION['mname'] = $userModelObject->getMname();
35     $_SESSION['lname'] = $userModelObject->getLname();
36     $_SESSION['phone'] = $userModelObject->getPhone();
37     $_SESSION['type'] = $userModelObject->getType();
```

Figure 6.3 - Setting user's session after authentication

When a client sends a request to the server, session variables are checked first. Session ID determines whether a user is logged in and allowed to use the system. Session type determines, whether a user has access to a specific part of the system. For example, if customer sends a request to delete a transport by modifying the URL, session variable type is checked. System then evaluates that user of type customer does not have access to this use case and the request is aborted.

```
if($orderState != 'Stand by' && $sessionType != 'manager')  
    $errorCounter += 1;
```

Figure 6.4 - Check if user is authorized to perform a request

Besides authentication on the server, session variables are also passed to clients when page is loaded. An example of usage of session on client is generating header buttons, or displaying welcome message with user's name when home page is entered.

6.1.3 Storing passwords

User authentication is crucial for our system. Users which did not pass the authentication are not able to use the system. Passwords are first hashed using PHP's cryptographic hash function and stored in the database. Passwords are hashed using PASSWORD_BCRYPT. BCRYPT Uses the CRYPT_BLOWFISH algorithm to create the hash and result is a 60 characters long string or false on failure.

6.1.4 Cross site scripting

Cross-site Scripting (XSS) refers to client-side code injection attack wherein an attacker can execute malicious scripts into a legitimate website or web application. XSS occurs when a web application makes use of invalidated user input within the output it generates. (acunetix).

As mentioned in section 6.1.1, users input is carefully validated on client side and server side. Although there has been a vulnerability found, specifically, when customer tries to update an order, that has already been confirmed. He is then informed, that the order cannot be changed and can send a message to the manager and explain what he would like to change. This input is validated only by length, which is about 200 characters. Request is then sent to the server and stored in MongoDB database and it can be read by any manager or customer himself.

Not validating user input is dangerous. Instead of expected message, attacker could enter some HTML or JavaScript code. Attackers code is stored in the database and waits there. Later when managers come to the websites and read notifications, code is downloaded to their browser and executed. This is cause by not properly implemented user input validation controls, together with bad output validation controls for data coming from the database.

Attacker really has a lot of option to cause harm. In order to prevent XSS attacks, htmlspecialchars() method can be used. This ensures that any HTML or JavaScript input will be escaped and ignored by browser.



Figure 6.5 - Example of invalidated user input - HTML and JavaScript

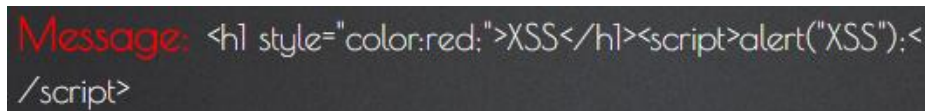


Figure 6.6 - Example of defense against XSS

6.1.5 Cross site request forgery

Cross-Site Request Forgery (CSRF) is an attack that forces an end user to execute unwanted actions on a web application in which they are currently authenticated. It is an attack that tricks the victim into submitting a malicious request. It inherits the identity and privileges of the victim to perform an undesired function on the victim's behalf (owasp, 2016).

Section 6.1.4 can be used as example. With a little bit of knowledge about how the application works attacker can capture, for instance, all orders. This can be considered a sensitive information for the company.

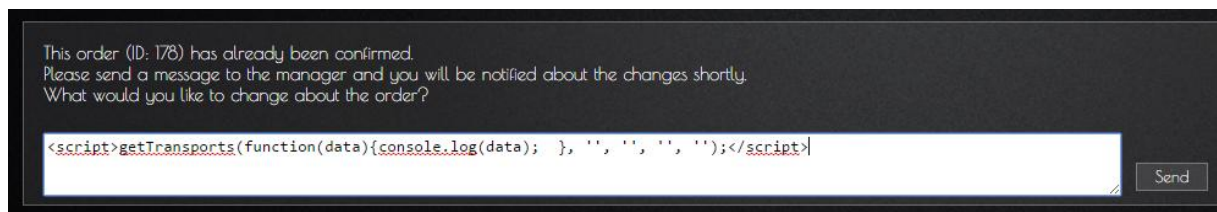


Figure 6.7 - Example of CSRF in the application

This message is sent to the server and stored in database. When manager opens notification page, script is downloaded and executed. In this example, data are just displayed in the console, but they can be sent elsewhere.

In order to prevent this, every time when a page is loaded, a token is generated (using md5 and a salt) and sent to the client. There it can be stored in a hidden field. When the request is sent, it is sent with this token, which is then compared on the server with the one previously generated.

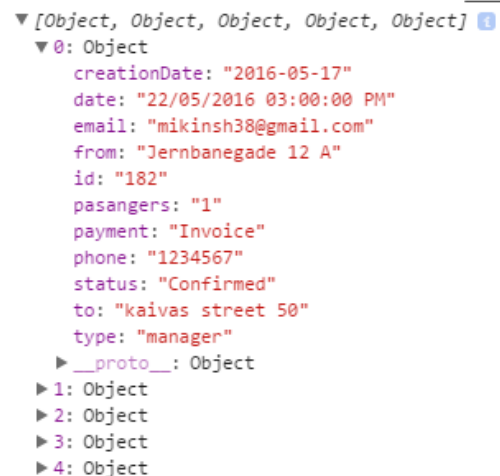


Figure 6.8 - Sensitive data accessed by an attacker

6.2 Usability

This non-functional requirements are crucial when developing an application. In fact, usability was prioritized as a must for the system. In this section, techniques and technologies used to achieve usability will be discussed, as well as detailed usability testing performed in the final phase of the development.

6.2.1 Responsive design

Responsive Web design is the approach that suggests that design and development should respond to the user's behavior and environment based on screen size, platform and orientation (Knight, 2011).

“Day by day, the number of devices, platforms, and browsers that need to work with your site grows. Responsive web design represents a fundamental shift in how we’ll build websites for the decade to come.” - Jeffrey Veen (Polacek).

In other words, the website should have the technology to automatically respond to the user's preferences. This would eliminate the need for a different design and development phase for each new gadget on the market (Knight, 2011).

While not all the pages in the application have responsive design yet implemented, here is an example of how it works. The statistics page adapts to different screen resolution on the laptop and mobile. This was implemented using Bootstrap framework and CSS.

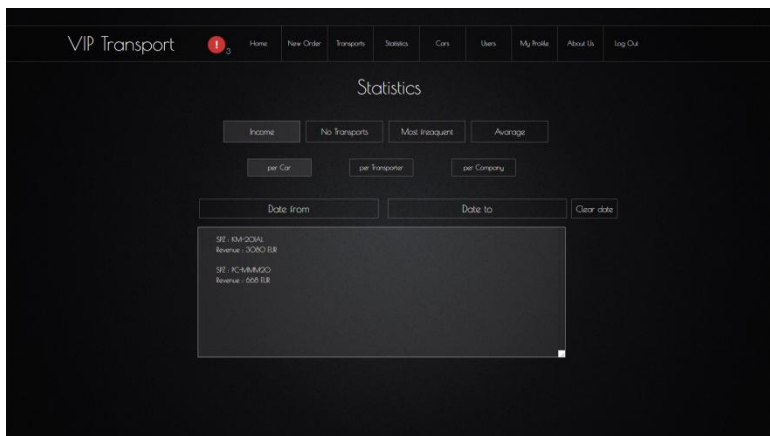


Figure 6.10 - Web page displayed on laptop (resolution 1600 x 900)



Figure 6.9 - Web page displayed on iPhone 6 (resolution 375 x 667)

6.2.2 Dynamic web application

For a web application, such as this one, page dynamics is a must. A large amount of logic was implemented on the client using JavaScript language and JavaScript libraries (jQuery and jQuery UI). It was necessary for variety of reasons, but overall it serves one main purpose - better user experience.

6.2.2.1 JavaScript

One of the reasons why JavaScript was implemented is response time. When some logic is implemented on the client it does not have to always communicate with the server and a user can receive instantaneous results, rather than waiting until the server responses.

Validation is a perfect example of how JavaScript was used. If user enters invalid value into the input field, error message is displayed automatically, without submitting the form or refreshing the page. JavaScript is also used for generating HTML elements. This was used specifically for generating tables. Another example of usage is communication with the server using AJAX, which will be explained in more detail in section 8.3.

6.2.2.2 JQuery

jQuery is a JavaScript library that allows web developers to add extra functionality to their websites. It is open source and it has become the most popular JavaScript library used in web development (techterms, 2013).

jQuery simplifies targeting HTML elements or communication with the server. It reduces the amount of code required for the job.

```
$('#dateToButton').on('click', function(){
    $('#datePickerAreaTo').slideToggle(300);
});

$('#dateClearButton').on('click', function(){
    $('#dateFromButton').val('Date from');
    $('#dateToButton').val('Date to');
    loadStatistics();
})
```

Figure 6.11- Targeting an element with jQuery by its ID

6.2.2.3 JQuery UI

jQuery UI is a set of user interface interactions, effects, widgets and themes built on top of the jQuery JavaScript Library. (jqueryui). An example of the GUI widget used in the application is a date picker, which was used on several places in the application and was styled with CSS. Animations were used as well. If, for example, user tries to submit a form while some error messages are visible, they will pulsate.

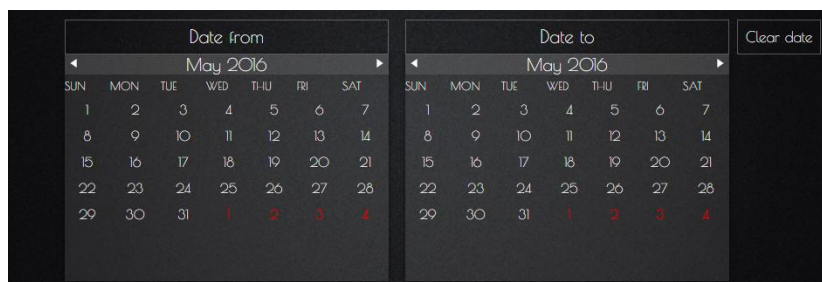


Figure 6.12 - Styled jQuery UI widget - date picker

6.2.3 Usability testing

The goal of usability testing is to identify any usability problems, collect quantitative data on participants' performance (e.g., time on task, error rates), as well as determine user satisfaction with the website. Put simply, if a website is difficult to navigate or does not clearly articulate a purpose, users will leave. Making it so they do not leave, makes testing websites a necessary task (Peacock, 2010).

The test was given to 3 candidates. Two different types of subjects were needed. Someone who would understand the system program-wise and would give constructive feedback to the developer and someone who has experience using such systems as a customer. For the testing, two computer science students (subject 1 and 2) and one person who often uses similar services (subject 3). Non explanation about the system was given to any subject. Testers played different actors:

- subject 1 - customer
- subject2 - transporter
- subject3 - customer and manager

Structure of the testing was very simple. First brief introduction to the system was given. Afterwards, subjects were given tasks (according to the actor). The following is shortened version of the task:

Manager	Transporter	Customer
Find transporter	Change cars status	Register and log in
Change transporter type	Add a service to a car	Change your phone number
Add new transporter	Add a highway sticker	Add a company
Change cars state	Find scheduled transport	Add profile picture
Confirm order	Find name of the passenger	Create new transport
Confirm route	Confirm transportation	Remove transports passenger
Find income per transporter	-	Change departure time
Find income per company	-	Find company email
-	-	Find out if transport was confirmed

Table 6.1 - Usability test task list for different actors

After subject finished his/her tasks, they were asked following questions:

- Has the system met your expectations?
- Does the system operate fast enough?
- How can be system improved?
- What do you not like about the system?
- Is the system overall easy to understand and navigate?
- What do you think about the design?

The following table shows commonly mentioned system flaws as well as possible solutions.

System flaws	Solution to the flaw
Icons are not self explanatory.	Changing icons or adding better description.
Response text is hard to read.	Changing the font-family. Changing font-color, so the text has bigger contrast with the background color.
American calendar is confusing.	Adding an option to choose between American and European calendar.
Notifications are not friendly.	Changing notification text, adding more visible option to click on the notifications details.
When address is inputted format is unclear.	Adding a placeholder or dividing the address input field into three - country, city and street.
Deleting passengers is unfriendly.	Adding a minus button to every passenger (as was implemented with cars services and stickers).
Message to the manager is unclear.	Adding a templates that customer could sent to the manager.

Table 6.2 - Commonly mentioned system flaws and solutions

There was also a lot of positive feedback to the system. All subjects reacted positively to system design, overall simplicity of the application and systems quick responses to user's actions. Subjects did not have a problem orienting in the system and all of the task were finished quickly without a need for help. Overall it has proven helpful and flaws and comments will be considered in the future development.

7 Database technologies

In this section database theory as well as implementation will be covered. Databases are very closely tight to non-functional requirements. When the right database is to be selected, consideration of non-functional requirements such as performance, scalability, availability, consistency and maintenance is a must.

7.1 Research

Before databases can be implemented, it is important to do proper research and select a technology that suits the needs. In this section, key elements that have to be considered when selecting a database technologies, will be discussed.

7.1.1 History of database technologies

For a long time Relational databases were an ultimate "go to" solution, when selecting a database technology for an application. Relational databases are very matured and reliable technology. They bring number of benefits, such as persistence of data, concurrency management using transactions and many others. Unfortunately, they also have their disadvantages. Impedance mismatch became obvious problem mainly because of the raise of object oriented programming. Another problem arose with horizontal scalability, which was a must after the rise of the internet and increased amount of traffic. The issue is, that SQL is designed to run on a single node system and does not work very well on large clusters (Fowler, Introduction to NoSQL • Martin Fowler, 2013).

These problems had inspired a new movement called NoSQL. This movement came with number of different database technologies, that also share some common characteristics (Fowler, Introduction to NoSQL • Martin Fowler, 2013).

7.1.2 Data

In this describes different types of data as well as the properties of data, such as volume, variety or velocity.

7.1.2.1 Data characteristics

7.1.2.1.1 Structured data

The information stored in relational databases is known as structured data, because it is represented in a strict format. Each record of the table follows the same format as every other record in the table (Ramez Elmasri, Chapter 12.1 - Structured, Semistructured, and Unstructured Data, 2011). When dealing with structured data, it is common to carefully design database schema. Therefore, nature of the data, that are going to be recorded, has to be known and understood. Structured data are easily stored, queried and analyzed (Beal).

7.1.2.1.2 Unstructured data

This type of data usually cannot be contained in a database. Unstructured data can be divided into two categories - textual, (Emails, Word documents, etc.) and non textual data (images, audio or video files, etc.) (Rouse, 2010).

7.1.2.1.3 Semi-structured data

While certain structure is present, this type of data lacks very rigid and strict data model structure. In addition not all data must follow the same structure. Semi-structure data also do not have a predefined schema - additional attributes may be introduced in some of the newer data items (Ramez Elmasri, Chapter 12.1 - Structured, Semistructured, and Unstructured Data, 2011). Typical for document, tree or graph databases.

7.1.2.2 Data volume

It is no surprise that as data storage has increased dramatically, large datasets simply became too unwieldy when stored in relational databases. In particular, query execution times increase as the size of tables and the number of joins grow. However, it is not always fault of the relational databases themselves. Rather, it has to do with the underlying data model (Sasaki, Graph Databases for Beginners: Why We Need NoSQL Databases, 2015).

7.1.2.3 Data velocity

Besides being big, today's data often changes very rapidly. Relational databases are not prepared to handle a sustained level of write loads and can crash during peak activity if not properly tuned (Sasaki, Graph Databases for Beginners: Why We Need NoSQL Databases, 2015).

The next question to ask is, whether the data model is likely to change and evolve or is most likely going to stay the same. Generally speaking, all the facts about the data model are not known at design time. Therefore some flexibility is needed. This presents many issues to the relational database management system (Dash, 2013).

7.1.2.4 Data variety

Today's data is far more varied than what relational databases were originally designed for. In fact, that is why many of today's RDBMS deployments have a number of nulls in their tables and null checks in their code – it is all to adjust to contemporary data variety (Sasaki, Graph Databases for Beginners: Why We Need NoSQL Databases, 2015).

On the other hand, NoSQL databases are designed from the bottom up to adjust for a wide diversity of data and flexibly address future data needs (Sasaki, Graph Databases for Beginners: Why We Need NoSQL Databases, 2015).

7.1.3 NoSQL databases

NoSQL means Not Only SQL, implying that when designing a software solution or product, there is more than one storage mechanism that could be used based on the needs (Sadalage, 2014).

7.1.3.1 Types of NoSQL databases

NoSQL databases can broadly be put into four categories.

7.1.3.1.1 Key-value stores

Key-value stores are the simplest NoSQL data stores to use. Since key-value stores always use primary-key access, they generally have great performance and can be easily scaled. Key-value stores are great for unstructured data and are typically used by web applications (online shopping carts, user session data) (Reinero, 2015).

7.1.3.1.2 Document databases

Document databases store and retrieve documents. One of the biggest advantages is that documents in the same collection, might be different. Documents store semi-structured data and use flexible JSON schema. Since schema is flexible, document databases have the possibility to store arrays (which would break the 1st normal form when using RDBMS) or polymorphic attributes (RDBMS would cause nulls).

While document databases might face redundancy in data, they are advantageous when it comes to performance.

Another advantage is that data is in format that the application expects and therefore developers avoid object relational independence mismatch.

Document databases are advantageous for reading heavy systems, because retrieving data is a question of single seek, rather than performing joins across multiple tables (Reinero, 2015).

7.1.3.1.3 Columns databases

These databases are designed for storing data tables as sections of columns of data, rather than as rows of data. While this simple description sounds like the inverse of a standard database, wide-column stores offer very high performance and a highly scalable architecture. Examples include: HBase, BigTable and HyperTable (planetcassandra).

7.1.3.1.4 Graph databases

These databases are designed for data whose relations are well represented as a graph and have elements which are interconnected, with an undetermined number of relations between them. Examples include: Neo4j and Titan (planetcassandra).

7.1.3.2 Aggregate data model

Looking at the types of NoSQL databases, there are big similarities between key-value, document and column-family types. All have fundamental unit of storage, which is a rich structure of closely related data: for key-value stores it's the value, for document stores it's the document, and for column-family stores it's the column family.

The rise of NoSQL databases has been driven primarily by the desire to store data effectively on large clusters. Relational databases were not designed with clusters in mind, which is why people have cast around for an alternative. Storing aggregates as fundamental units makes a lot of sense for running on a cluster. Aggregates make natural units for distribution strategies such as sharding, since there a large clump of data that is expected to be accessed together. An aggregate also makes a lot of sense to an application programmer (Fowler, AggregateOrientedDatabase, 2012).

There is a significant downside - the whole approach works really well when data access is aligned with the aggregates. The advantage of not using an aggregate structure in the database is that it allows to slice and dice the data different ways for different audiences (Fowler, AggregateOrientedDatabase, 2012).

7.1.4 Relational and NoSQL database differences

7.1.4.1 Nature of data

While Relational databases deal strictly with structured data, NoSQL databases are typical for semi-structured data. More about this topic in 7.1.2.1.

7.1.4.2 Database properties

The very crucial difference between these two types of databases arrives when it comes to consistency.

While RDBMS create a very safe environment when handling data, using ACID properties, it often comes with a cost. Achieving such write consistency requires sophisticated locking which is typically a heavyweight pattern for most use cases. It can be also unnecessarily pessimistic.

In the NoSQL world, ACID transactions are less fashionable as some databases have loosened the requirements for immediate consistency, data freshness and accuracy in order to gain other benefits, like scale and resilience.

Both consistency models come with advantages – and disadvantages – and neither is always a perfect fit (Sasaki, Graph Databases for Beginners: ACID vs. BASE Explained, 2015).

7.1.4.2.1 ACID properties

ACID stands for Atomicity, Consistency, Isolation, and Durability. These are the properties of a transaction (Tomar, 2011).

- **ATOMICITY:** The atomicity property identifies that the transaction is atomic. An Atomic transaction is either fully completed, or is not begun at all. If for any reason an error occurs and the transaction is unable to complete all of its steps, then the system is returned to the state it was in before the transaction was started.
- **CONSISTENCY:** A transaction enforces Consistency in the system state by ensuring that at the end of any transaction the system is in a valid state. If the transaction completes successfully, then all changes to the system have been properly made, and the system will be in a valid state. If any error occurs in a transaction, then any changes already made will be automatically rolled back.
- **ISOLATION:** When a transaction runs in Isolation, it appears to be the only action that the system is carrying out at one time. If there are two transactions that are both performing the same function and are running at the same time, transaction isolation will ensure that each transaction behaves as it has exclusive use of the system.
- **DURABILITY:** A transaction is Durable once it has been successfully completed and all of the changes it made to the system are permanent. There are safeguards that will prevent the loss of information, even in the case of system failure.

7.1.4.2.2 States of transaction

A database transaction can be in one of the following states(tutorialspoint):

- **Active** – In this state, the transaction is being executed. This is the initial state of every transaction.
- **Partially Committed** – When a transaction executes its final operation, it is said to be in a partially committed state.
- **Failed** – A transaction is said to be in a failed state if any of the checks made by the database recovery system fails. A failed transaction can no longer proceed further.
- **Aborted** – If any of the checks fails and the transaction has reached a failed state, then the recovery manager rolls back all its write operations on the database to bring the database back to its original state where it was prior to the execution of the transaction. Transactions in this state are called aborted. The database recovery module can select one of the two operations after a transaction aborts –

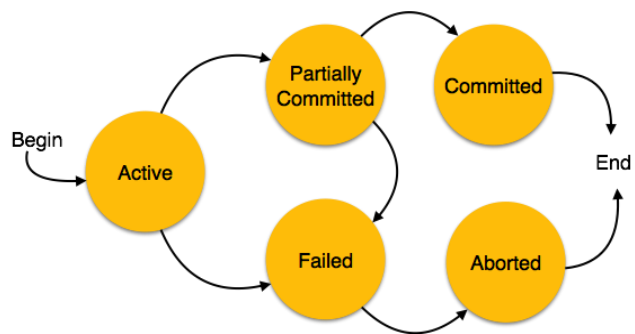


Figure 7.1 - Database transaction states

- Re-start the transaction
- Kill the transaction
- **Committed** – If a transaction executes all its operations successfully, it is said to be committed. All its effects are now permanently established on the database system.

7.1.4.2.3 BASE properties

- Basically available indicates that the system does guarantee availability, in terms of the CAP theorem (more about CAP theorem in section 7.1.5).
- Soft state indicates that the state of the system may change over time, even without input. This is because of the eventual consistency model.
- Eventual consistency indicates that the system will become consistent over time, given that the system does not receive input during that time.

(Rest, 2010)

A BASE data store values availability (since that is important for scale), but it does not offer guaranteed consistency of replicated data at write time. Overall, the BASE consistency model provides a less strict assurance than ACID - data will be consistent in the future (Sasaki, Graph Databases for Beginners: ACID vs. BASE Explained, 2015).

7.1.4.2.4 ACID compared to BASE

There is no right answer to whether your application needs an ACID versus BASE consistency.

Given BASE's loose consistency, it is important to be more knowledgeable and rigorous about consistent data if a BASE store is selected for an application.

On the other hand, planning around BASE limitations can sometimes be a major disadvantage when compared to the ACID transactions. A fully ACID database is the perfect fit for use cases where data reliability and consistency are essential (Sasaki, Graph Databases for Beginners: ACID vs. BASE Explained, 2015).

7.1.4.3 Scalability

Relational databases are designed to scale up, while NoSQL databases are designed to scale out (Perkins, Relational vs NoSQL Databases, 2014).

7.1.4.3.1 Vertical scaling

Vertical scaling, or scaling up, usually refers to adding more power (buying more expensive, robust server, better processor, RAM, etc.). It comes with a number of advantages - consuming less power, less cooling costs, less licensing costs and is generally easier to implement.

On the other hand, horizontal scaling costs much more and there is bigger risk of failure. There is also a limited upgradability in the future (Graf, 2013).

7.1.4.3.2 Horizontal scaling

Horizontal scaling, or scaling out, generally refers to adding more servers with less processors and RAM. This solution is much cheaper and the expenses are much more predictable. It is also easier to run fault-tolerance.

When compared with vertical scaling, utility costs and licensing fees are higher (Graf, 2013).

7.1.4.4 Impedence mismatch

The object-relational impedance mismatch is a set of conceptual and technical difficulties that are often encountered when a relational database management system (RDBMS) is being used by a program written in an object-oriented programming language or style, particularly when objects or class definitions are mapped in a straightforward way to database tables or relational schemas (wikipedia, 2016).

7.1.5 CAP theorem

The CAP Theorem states that, in a distributed system (a collection of interconnected nodes that share data), there can only be two out of the following three guarantees across a write/read pair: Consistency, Availability, and Partition Tolerance - one of them must be sacrificed (Greiner, 2014).

- Consistency - A read is guaranteed to return the most recent write for a given client.
- Availability - A non-failing node will return a reasonable response within a reasonable amount of time (no error or timeout).
- Partition Tolerance - The system will continue to function when network partitions occur.

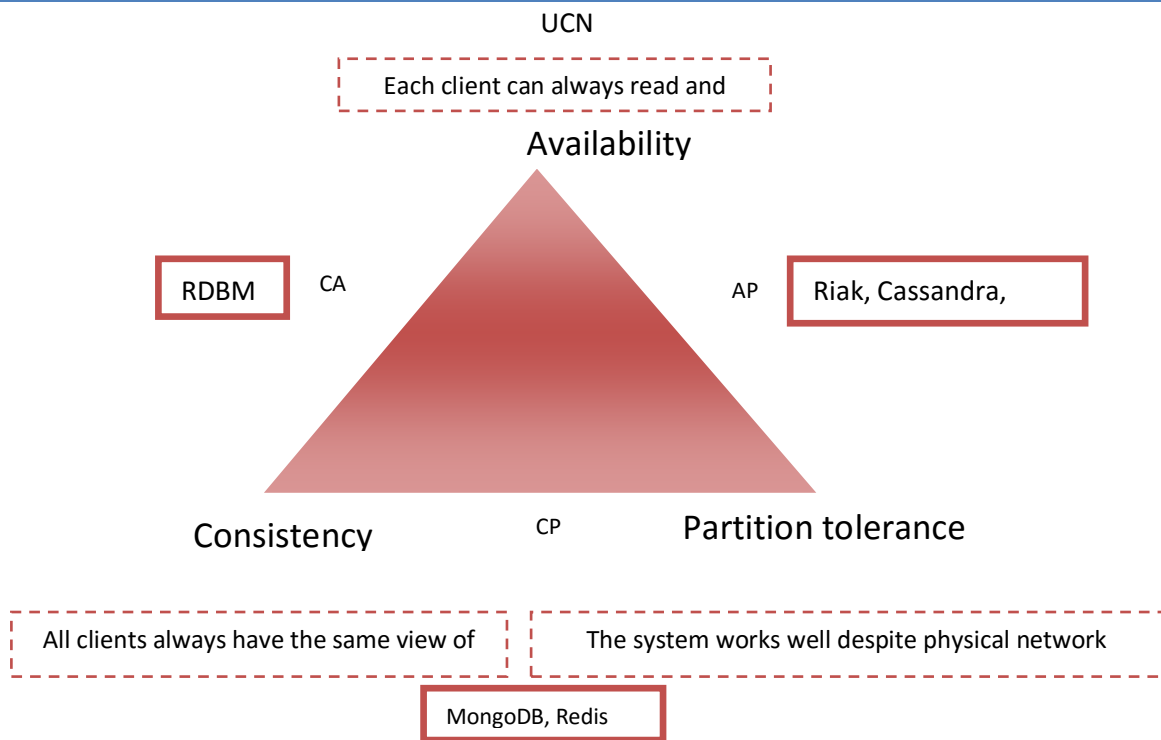


Figure 7.3 - The CAP theorem

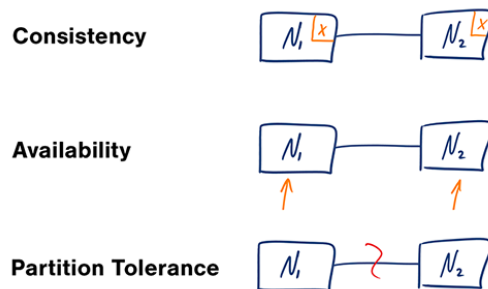


Figure 7.4 - The three guarantees of CAP theorem

Network failures happen to the system and nobody gets to choose when they occur. Given that networks are not completely reliable, partitions must be tolerated in a distributed system period. However, it is possible to choose what to do when a partition occurs. According to the CAP theorem, there two options: Consistency and Availability.

- CP - Consistency/Partition Tolerance - Wait for a response from the partitioned node which could result in a timeout error. The system can also choose to return an error, depending on the scenario desired. It is a good decision to choose Consistency over Availability when business requirements dictate atomic reads and writes.

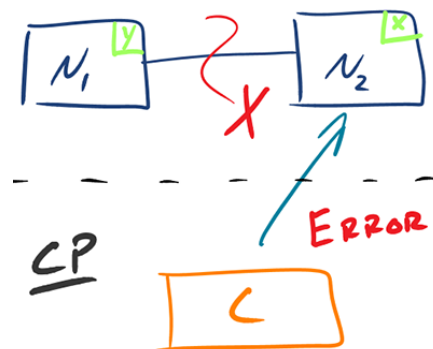


Figure 7.5 - Example of Consistency/ Partition tolerance

- AP - Availability/Partition Tolerance - Returning the most recent version of the data, which could be stale. This system state will also accept writes that can be processed later, when the partition is resolved. If business requirements allow some flexibility around when the data in the system synchronizes, choosing Availability over Consistency is the right decision. Availability is also a compelling option when the system needs to continue to function in spite of external errors (shopping carts, etc.)

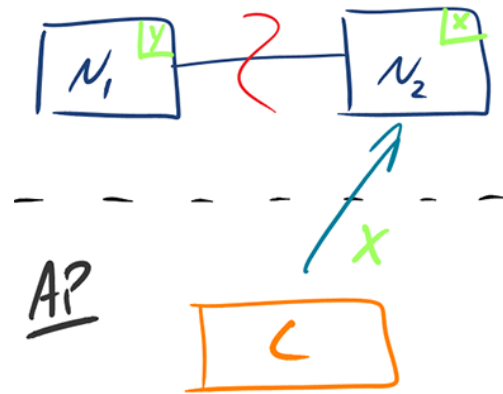


Figure 7.6 - Example of Availability/ Partition tolerance

Building distributed systems provides many advantages, but also adds complexity. Understanding the trade-offs available in the face of network errors, and choosing the right path is vital to the success of the application.

7.1.6 Polyglot persistence

The term Polyglot Persistence means, that when storing data, it is best to use multiple data storage technologies. Technologies are chosen based upon the way data is used by individual applications or components of a single application. It means picking the right tool for the right use case (Serra, 2015).

Increasingly we'll see such applications manage their own data using different technologies depending on how the data is used (Fowler, PolyglotPersistence, 2011).

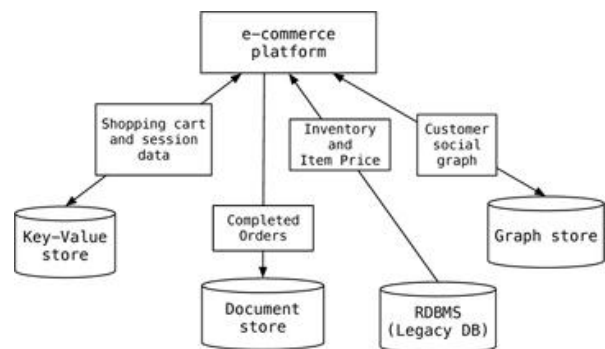


Figure 7.7 - Polyglot persistence example

Looking at a Polyglot Persistence example, an e-commerce platform will deal with many types of data (i.e. shopping cart, inventory, completed orders, etc). Instead of trying to store all this data in one database, which would require a lot of data conversion to make the format of the data all the same, store the data in the database best suited for that type of data (Serra, 2015).

7.2 Database implementation

Process of selecting the database (according to research in section 7.1) based on the customer's requirements as well as selected databases and their implementation will be discussed in this section.

7.2.1 Selecting a database

After discussion with a customer, it was decided that the application should store not only information about users (customers, employees), their companies, orders and transports, but also information about cars and services performed on the cars. Employees should also have a clear idea about what customers are doing via notifications.

The purpose of the application had to be kept in mind. Even though the system is able to store customer orders and transports, it is primarily made for the managers. The application should provide analysis of the transports. One of the specific requests from the customer was ability help to manage invoices - given a period of time, the system would calculate the revenue.

7.2.1.1 Nature of data

In order to select the right database system for an application, it is crucial to understand what data is going to be stored. It is necessary to understand the data characteristics, its estimated volume, velocity and variety.

Since VIP Transport is a very young company, it does not deal with a large number of customers. On average it has from 10-15 transports a week. Even when under the best circumstances, the company does not expect to grow rapidly in the following years. Smaller data volume plays in the favor of relational databases, but the outcome cannot be judged so easily. When it comes to notifications, number of records in the database could largely increase, since all employees have to be notified about the changes. NoSQL database should be considered in this case.

Considering that the number of users in total is not expected to be very large, neither is the number of changes. More important question is, whether the data model is likely to stay the same. Since VIP Transport is over a year old, customer had a very clear requirements about what data are necessary, when it comes to users, orders and cars. Because the data is known and structured, relational data model would fit. For transports themselves, customer requested two types - official and personal transports. Both kinds have different attributes, therefore more flexible schema could be a better option.

7.2.1.2 Required database properties

Knowing the data is essential, along with choosing database properties. The important question is: Does the database need ACID transactions?

ACID properties provide very save environment for cases where data consistency is necessary. An example of such case in the application would be when customers order is being confirmed, and a route is being created or when route is confirmed and transport is created. Both of these examples require multiple database calls that have to be processed atomically, in order to achieve consistency. Because of this reason relational database is a must, for certain parts of the system.

On the other hand, parts like the notification system do not require as strict consistency. Eventual consistency offered by NoSQL databases is a satisfactory solution.

7.2.1.3 Selected databases

Taking all previously mentioned factors into consideration, there are different parts of the application that have different needs. This indicates, that polyglot persistence is necessary.

Structured data that requires the usage of ACID transaction was stored in RDBMS (MySQL). Specifically information about users, companies, orders, routes and cars are a perfect fit for relational schema. Notifications and transports were stored using NoSQL technologies.

Both notifications and transports require flexible schema. Sometimes it is necessary to store arrays and polymorphic attributes. In addition, transports would require heavy reads, when it come to analysis. Because of these reasons it was decided that document NoSQL database (MongoDB) fits best as a solution.

7.2.2 MySQL

For storing structured data, that do not require flexible schema and on the other hand require ACID transactions, MySQL database technology was selected.

There are also other reasons, why MySQL is a reasonable pick. It works well with PHP and it is the most common stacks for web development (Lionite, 2014). Another reason to use MySQL as a database engine for a company like VIP Transport is that it is an open source technology and therefore very inexpensive. Because it is very popular, large amount of support is provided. It is also very compatible technology (Mack, 2014).

7.2.2.1 Relational Model Constraints and Relational Database Schemas

This section explains what relation schema and state are. It also describes how was relation database schema created, what are integrity constraints and how they are implemented.

7.2.2.1.1 Relation schema

Relation schema R denoted by $R(A_1, A_2, \dots, A_n)$, is made up of a relation name R and a list of attributes, A_1, A_2, \dots, A_n . Each attribute A_i is the name of a role played by some domain D in the relation schema R . A domain D is a set of atomic values - each value in the domain is indivisible (e.g. First_name - the set of character strings that represent first name of a person). D is called the domain of A_i and is denoted by $\text{dom}(A_i)$. A relation schema is used to describe a relation. R is called the name of this relation. The degree (or arity) of a relation is the number of attributes n of its relation schema (Ramez Elmasri, Chapter 3.1 - Relational Model Concepts).

The following is an example of relation schema. The name of the relation is SERVICE and it has a degree of 5.

SERVICE(Id: integer, Spz: string, Issue: string, Repare_date: date, Mealige: real)

7.2.2.1.2 Relation state

A relation (or relation state) r of the relation schema $R(A_1, A_2, \dots, A_n)$, also denoted by $r(R)$, is a set of n -tuples $r = \{t_1, t_2, \dots, t_m\}$. Each n -tuple is an ordered list of n values $t = \langle v_1, v_2, \dots, v_n \rangle$, where each value v_i , $1 \leq i \leq n$, is an element of $\text{dom}(A_i)$ or is a special NULL value. The i^{th} value in tuple t , which corresponds to the attribute A_i , is referred to as $t[A_i]$ (Ramez Elmasri, Chapter 3.1 - Relational Model Concepts).

The following is an example of SERVICE relation, which corresponds to SERVICE schema just specified. Each tuple of the relation represents particular service entity.

SERVICE				
Id	Spz	Issue	Repare_date	Mealige
219	BA-700PP	Broken wheel	2015-08-08	500.0
220	BA-700PP	Engine was not working	2016-05-02	550.0

Table 7.1 - Example of SERVICE relation

7.2.2.1.3 Relational database schema

A relational database schema S is a set of relation schemas $S = \{R_1, R_2, \dots, R_m\}$ and a set of integrity constraints IC . A relational database state DB of S is a set of relation states $DB = \{r_1, r_2, \dots, r_m\}$ such that each r_i is a state of R_i and such that the r_i relation states satisfy the integrity constraints specified in IC .

When we refer to a relational database, we implicitly include both its schema and its current state. A database state that does not obey all the integrity constraints is called an invalid state, and a state that satisfies all the constraints in the defined set of integrity constraints IC is called a valid state (Ramez Elmasri, Chapter 3.2 - Relational Model Constraints and Relational Database Schemas).

The following is an example of relational database schema called VIPTRANSPORT.

VIPTRANSPORT = {ROUTE, ROUTE_COUNTRY_CODE, CAR, ORDER, USER, SERVICE, STICKER, PASANGER_NAME, COMPANY}

UCN

ROUTE

<u>Id</u>	Order_id	Transporter_email	Car_spz
-----------	----------	-------------------	---------

ROUTE_COUNTRY_CODE

<u>Route_id</u>	<u>Country_code</u>
-----------------	---------------------

CAR

<u>Spz</u>	Brand	Type	Seats	State	Emission_check	Stk	Mandatory_insurance	Accident_insurance	Mealige
------------	-------	------	-------	-------	----------------	-----	---------------------	--------------------	---------

ORDER

<u>Id</u>	Email	DateTime	Departure_address	Arrival_address	Payment_type	Pasangers	Creation_date	Status
-----------	-------	----------	-------------------	-----------------	--------------	-----------	---------------	--------

USER

<u>Email</u>	First_name	Middle_name	Last_name	Password	Type	Email	Phone	Registration_date
--------------	------------	-------------	-----------	----------	------	-------	-------	-------------------

SERVICE

<u>Id</u>	Spz	Issue	Repare_date	Mealige
-----------	-----	-------	-------------	---------

STICKER

<u>Spz</u>	Country	Expiration_date
------------	---------	-----------------

PASANGER_NAME

<u>Order_id</u>	<u>Pasanger_name</u>
-----------------	----------------------

COMPANY

<u>User_email</u>	Company_name	Invoice_address	Ico	Dic
-------------------	--------------	-----------------	-----	-----

Table 7.2 - VIPTRANSPORT database schema

7.2.2.1.4 Integrity, referential integrity and foreign keys

The entity integrity constraint states that no primary key value can be NULL. This is because the primary key value is used to identify individual tuples in a relation (Ramez Elmasri, Chapter 3.2 - Relational Model Constraints and Relational Database Schemas).

The referential integrity constraint is specified between two relations and is used to maintain the consistency among tuples in the two relations. Informally, the referential integrity constraint states that a tuple in one relation that refers to another relation must refer to an existing tuple in that relation.

The conditions for a foreign key, given below, specify a referential integrity constraint between the two relation schemas R_1 and R_2 . A set of attributes FK in relation schema R_1 is a foreign key of R_1 that references relation R_2 if it satisfies the following rules:

- The attributes in FK have the same domain(s) as the primary key attributes PK of R_2
- A value of FK in a tuple t_1 of the current state $r_1(R_1)$ either occurs as a value of PK for some tuple t_2 in the current state $r_2(R_2)$ or is NULL ($t_1[FK] = t_2[PK]$).

Referential integrity constraints can be diagrammatically displayed by drawing a directed arc from each foreign key to the relation it references. The arrowhead may point to the primary key of the referenced relation.

The following example are referential integrity constraints displayed on the VIP TRANSPORT relational database schema.

UCN

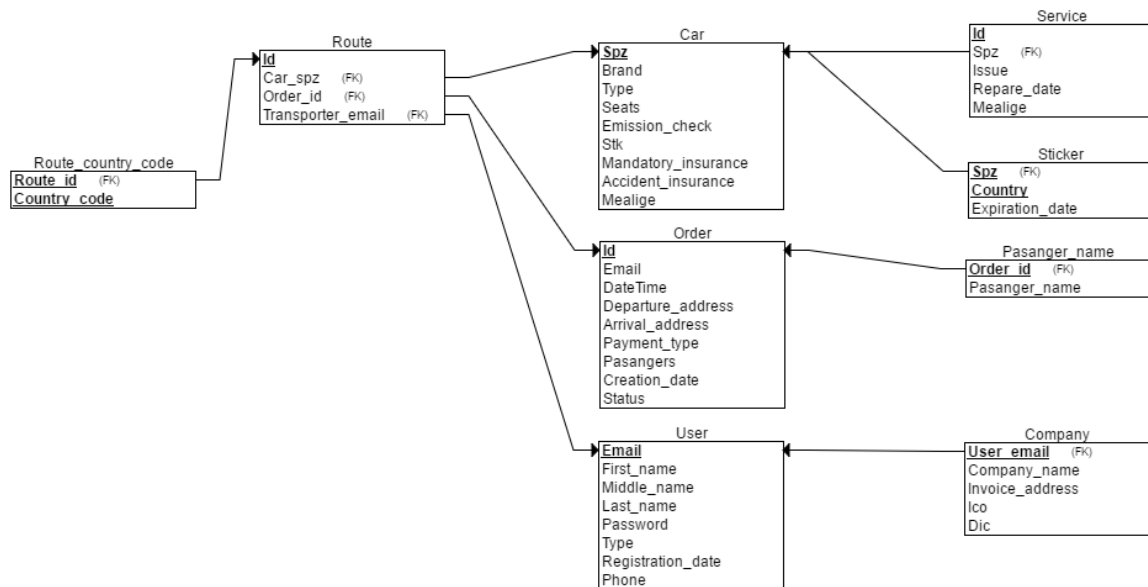


Figure 7.8- Referential integrity constraints displayed on the VIPTRANSPORT relational database schema

7.2.2.2 Data access layer

In this section, Data access layer (DAL) will be discussed in detail, showing how relational database is manipulated from the application. For number of purposes, mysqli extension has been used. mysqli is a library that allows accessing MySQL server. It has been used from establishing connections, to executing atomic transactions.

7.2.2.2.1 Connection

Connection to MySQL server was very straight forward. Mysqli connection object had to be created, having host, user, password and database name as a parameter for the constructor.

```

if(!DEFINED('DB_USER')) DEFINE ('DB_USER', 'root');
if(!DEFINED('DB_PASSWORD'))DEFINE ('DB_PASSWORD', '');
if(!DEFINED('DB_HOST'))DEFINE ('DB_HOST', 'localhost');
if(!DEFINED('DB_NAME'))DEFINE ('DB_NAME', 'VIPTransport');

$dbc = new mysqli(DB_HOST, DB_USER, DB_PASSWORD, DB_NAME);
  
```

Figure 7.9 - Construction of connection object to MySQL database using mysqli library

7.2.2.2.2 Security

When it comes to manipulation with databases, there is a huge possible threat called SQL injection. In order to avoid SQL injection attacks, input has been first validated, and in the DAL escaped - mysqli's `real_time_escape_string()` function has been used. Another security measure, was the usage of parameterized queries and prepare statements, as can be seen on the following example.

First, query and prepare statements are created. Later parameters are bound to the statement and the statement is executed.

```
$query = "INSERT INTO User (Email, First_name, Middle_name, Last_name, Password, Phone, Type, Registration_date)" .
        " VALUES (?, ?, ?, ?, ?, ?, ?, NOW())";
$stmt = mysqli_prepare($dbc, $query);
```

Figure 7.10 - Example of prepare statement

```
mysqli_stmt_bind_param($stmt, "sssssss", $email, $fname, $mname, $lname, $password, $phone, $type);
mysqli_stmt_execute($stmt);
```

Figure 7.11 - Example of binding parameter to prepare statement and its execution

7.2.2.2.3 Transactions

Transactions play a big role in relational databases. Their usage was one of the main reasons why RDBMS is used in the first place.

An example of a transaction in the system is a situation when an order is confirmed by a manager. In order to keep data consistent, several statements have to be executed atomically.

First, a database connection object is created and auto-commit is set to false. It is essential that all operations will be performed on this connection object.

```
$dbc = DatabaseConnection::openConnection();
$dbc->autocommit(false);
```

Figure 7.12 - MySQL transaction 1

Afterwards, three operations have to happen - Status attribute in the ORDER relation has to be updated to 'Confirmed', new ROUTE record has to be inserted and routes countries (if specified) have to be inserted into ROUTE_COUNTRY_CODE relation. Since for the last insertion, route ID is necessary, only two queries are yet created.

```
$orderWClause = "WHERE Id = ".$routeModelObject->getOrderId();
$orderQuery = "UPDATE transport_order SET Status = 'Confirmed' ".$orderWClause;
$routeQuery = $this->getRouteInsertQuery($dbc, $routeModelObject);
```

Figure 7.13 - MySQL transaction 2

Queries are then one-by-one executed. In between execution error check is performed on the connection object. If any error occurred, error message is returned and appended to errorString variable (otherwise NULL is returned). Also after the route is inserted, last inserted Id is obtained.

```
$dbc->query($orderQuery);
$errorString .= $dbc->error;
$dbc->query($routeQuery);
$routeId = $dbc->insert_id;
$errorString .= $dbc->error;
```

Figure 7.14 - MySQL transaction 3

After the last insertion, the length `errorString` variable is checked. If its empty all statements succeeded and transaction is committed. If any error occurred, transaction is rolled back.

7.2.3 MongoDB

MongoDB is a cross-platform, document oriented database that provides, high performance, high availability, and easy scalability. MongoDB works on concept of collection and document (tutorialspoint).

```
if(strlen($errorString) == 0){
    $dbc->commit();
    return 1;
}
else{
    $dbc->rollback();
    return 0;
}
```

Figure 7.15 - MySQL transaction 4

RDBMS	MongoDB
Database	Database
Table	Collection
Tuple/Row	Document
Column	Field
Table Join	Embedded Documents
Primary Key	Primary Key (Default key <code>_id</code> provided by MongoDB itself)

Figure 7.16 - Comparison of RDBMS and Document database concepts

7.2.3.1 Connection

After MongoDB is installed, in order connect to the database server, command 'mongod' has to be entered into the console. This will show 'waiting for connections' message on the console output and it indicates that the mongod.exe process is running successfully (tutorialspoint).

```
2016-05-22T18:25:24.286+0200 I NETWORK [initandlisten] waiting for connections on port 27017
```

Figure 7.17 - Successful start of MongoDB server

After the connection is created successfully, new MongoDB client has to be created in order to manipulate with the database. After the database is selected, its statistics can be shown using command `db.stats()`.

```
> db.stats()
{
  "db" : "VIPTransport",
  "collections" : 8,
  "objects" : 1487,
  "avgObjSize" : 257.3335574983188,
  "dataSize" : 382655,
  "storageSize" : 233472,
  "numExtents" : 0,
  "indexes" : 8,
  "indexSize" : 196608,
  "ok" : 1
}
```

Figure 7.18 - VIPTransport database stats

7.2.3.2 Schema

MongoDB uses flexible schema and a document is written in JSON format. This brings a number of advantages - two different documents in the same collection can have different set of attributes. This can be seen in the schema used for collection 'notifications' in the 'VIPTransport' database. Notice, that the second document has an additional attribute 'message'.

```

1 {
2   "_id" : ObjectId("57320cc35445811164001d0c"),
3   "text" : "User User(user@user.user) has created order at 12/05/2016 05:00:00 PM (ID:159)",
4   "reciever" : "man@man.man",
5   "action" : "created",
6   "type" : "order",
7   "read" : true,
8   "date" : "2016-05-10 18:30:59"
9 }
10 {
11   "_id" : ObjectId("573b72aa5445813c1c0079f3"),
12   "text" : "<strong><a href=\"notificationDetailPage.html?type=user&id=You\">You</a></strong> are registered!",
13   "reciever" : "miroslav_pakanec@gmail.com",
14   "action" : "register",
15   "type" : "user",
16   "message" : "Welcome on board Miroslav! Click <strong><a href=\"myTransportsPage.html\">HERE</a></strong> and
17   "read" : false,
18   "date" : "2016-05-17 21:36:10"
19 }

```

Figure 7.19 - Example of documents from collection 'notifications'

Another advantage of MongoDB is that it stores data as application expects. In other words, it naturally avoids impendence mismatch. An example is a transport document. It has a number of sub documents (route, order, company or employee) that mimic Model classes in the application.

```

3 {
4   "_id" : ObjectId("57383397539a55d1ec39986e"),
5   "price" : 100.0,
6   "mealige" : 800.0,
7   "distance" : 200.0,
8   "arrivalDatePickUp" : ISODate("2010-04-30T00:00:00.000+0000"),
9   "arrivalDateDestination" : ISODate("2010-05-30T00:00:00.000+0000"),
10  "duration" : "04:40:00",
11  "type" : "official",
12  "employee" : {
13    "email" : "man@man.man",
14    "type" : "manager"
15  },
16  "route" : {
17    "id" : NumberInt(58),
18    "orderId" : NumberInt(164),
19    "transporter" : "tran@tran.tran",
20    "car" : "KM-201AL",
21    "countries" : {
22      "country1" : "PT"
23    }
24  },
25  "order" : {
26    "email" : "user@user.user",
27    "date" : "13/05/2016 10:10:00 AM",
28    "from" : "Bratislava",
29    "to" : "Zilina",
30    "payment" : "Cash",
31    "phone" : "+421910245649",
32    "pasangers" : "1",
33    "names" : {
34      "name1" : "Miro P"
35    }
36  },
37  "company" : {
38    "name" : "UCN",
39    "address" : "Zilina",
40    "ico" : "12345678",
41    "dic" : "1234567890"
42  }
43 }

```

Figure 7.20 - Example of a document from collection 'transports'

7.2.3.3 Database Access Layer

Since the applications server-side language is PHP, in order to set up MongoDB, PHP-MongoDB driver had to be installed.

7.2.3.3.1 Database connection

Responsibility for connection to MongoDB has the DatabaseMongodbConnection class. Class can either return host connection string, database object or collection object according to what is needed. Here is an example of how collection object is returned.

```
5  Class DatabaseMongodbConnection{
6
7      public static function getCollection($database, $collection){
8
9          $host = 'mongodb://localhost:27017';
10
11         $mongo = new MongoClient($host);
12         $db = $mongo->$database;
13         $col = $db->$collection;
14
15         return $col;
16     }
```

Figure 7.21 - Example of returning collection object in DAL

7.2.3.3.2 Bulk write

The MongoDB\Driver\BulkWrite collects one or more write operations that should be sent to the server. After adding any number of insert, update, and delete operations, the collection may be executed.

Write operations may either be ordered (default) or unordered. Unordered operations are sent to the server in an arbitrary order where they may be executed in parallel. Any errors that occur are reported after all operations have been attempted (php).

When notifications are inserted or their read state is updated, unordered bulk write operation is used. This ensures multiple notifications are inserted/updated in parallel, which increases performance.

7.2.3.3.3 Finding records

Using method find() on a specific collections selects documents in a collection and returns a cursor to the selected documents. A cursor is a pointer to the result set of a query, not the result itself. Clients can iterate through a cursor to retrieve results (MongoDB).

Method find has two optional parameters - query and projection. Query are the Specifies selection criteria and projection specifies the fields to return in the matching documents. There are several cursor methods that modify its behavior, such as sort, limit or skip.

Method Sort() specifies the order in which the query returns matching documents. A parameter is expected in a form of a key-value pair. The key represents a documents field and the value can be either 1 or -1 representing the Ascending or Descending (MongoDb).

The `limit()` method limits the number of documents in the result set. Integer parameter is expected, representing the number of documents that want to be retreated.

The `skip()` method controls the starting point of the results set. Integer parameter is expected, representing the number of documents that are supposed to be skipped.

When notifications are fetched from the database, the `find` method is used. There are two ways how notifications can be queried - either a user can select all notifications or only unread notifications. In addition, notifications are queried by a receiver, for which session variable email is used. While parameter for projection is not specified, number of additional cursor methods were chained.

When notifications are retrieved, they are sorted by a date of creation, which ensures that the newest notifications are always on top. The amount (limit) of notifications fetched by one call is 10 and is not changing. When user enters the page, 10 notifications are fetched and the skip variable is 0. When user clicks on a button 'Show more' another 10 notifications are fetched and the skip variable increments by 10. This way we avoid heavy reads and notifications are stuck up on the client.

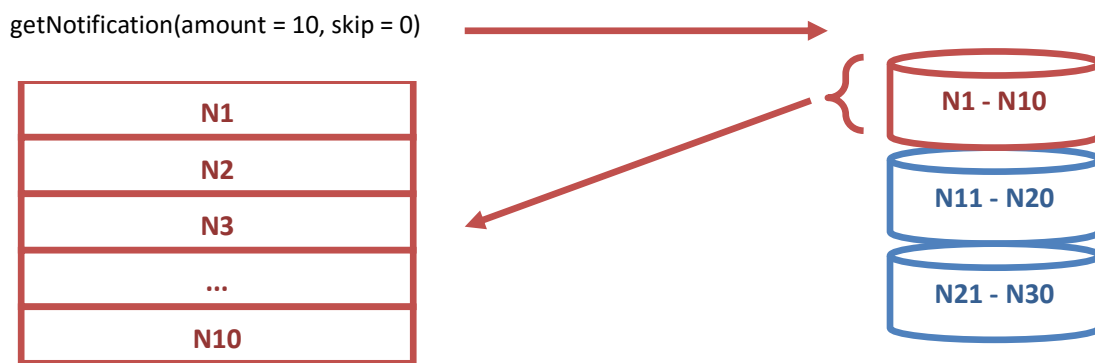


Figure 7.22 - Getting notifications on a page load

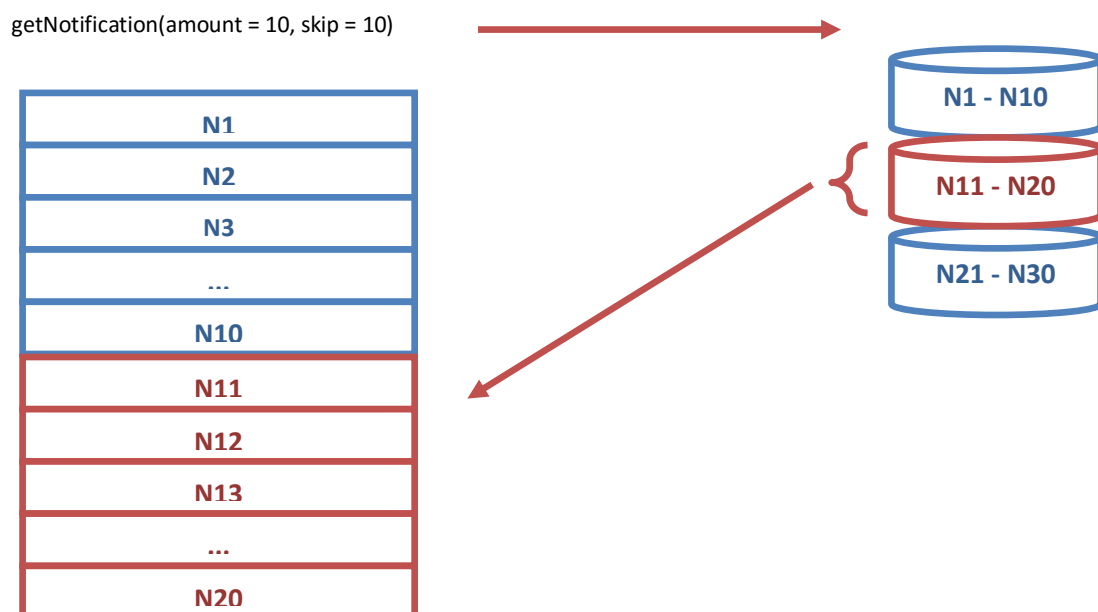


Figure 7.23 - Getting notifications after user clicks 'Show more' button

```
$cursor = $collection->find(array("reciever"=> $receiver),
                             array('limit' => $ammount,
                                    'skip' => $skip,
                                    'sort' => array('date' => -1)));
```

Figure 7.24 - Example of an implementation of the find() method in PHP

7.2.3.3.4 Aggregation

Aggregations are operations that process data records and return computed results. Running data aggregation on the mongod instance simplifies application code and limits resource requirements (MongoDB).

7.2.3.3.5 Aggregation Pipelines

Documents enter a multi-stage pipeline that transforms the documents into an aggregated result (MongoDB).

An example of usage of MongoDBs aggregation framework in the application is finding a total amount of unread notifications for specific user. While in the application a 'receiver' is represented by email and _id is an objectId, for the sake of example, in the figure 7.25 and 7.6, alphabetical and numeric values are used.

Collection
↓

```
db.notifications.aggregate([
  Match ➔ { $match: { "reciever": "A", "read": false}},
  Group ➔ { $group: { _id: "$reciever ", total: { $sum: 1}}}
]);
```

Figure 7.25 - Example of aggregate operation in MongoDB

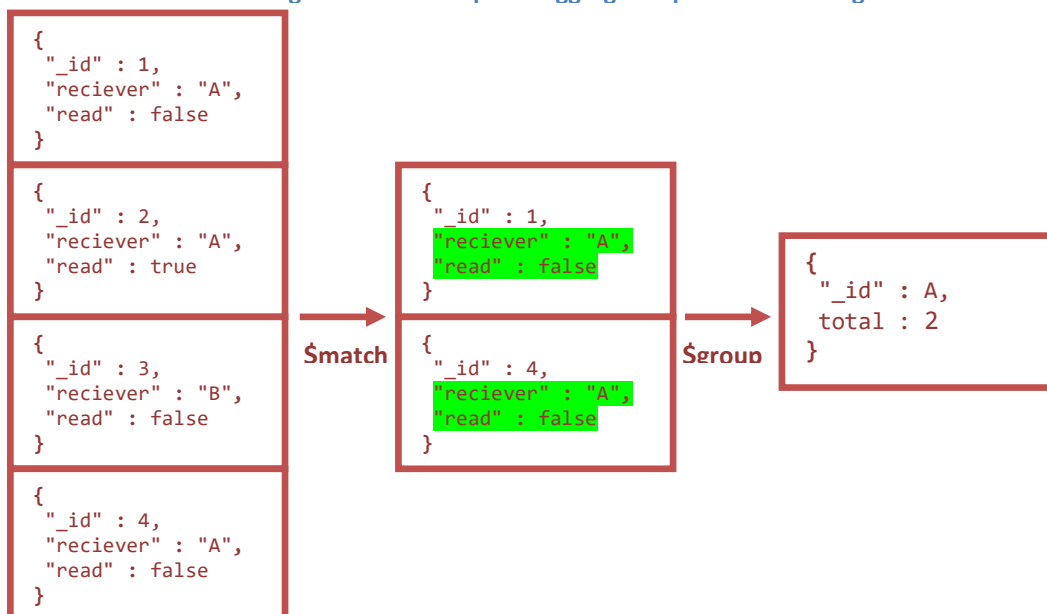


Figure 7.26 - Example of finding the number of unread notification using an aggregate search


```
$cursor = $collection->aggregate([
  [ '$match' => [ 'reciever' => $receiver, 'read' => false ] ],
  [ '$group' => [ '_id' => '$reciever', 'total' => [ '$sum' => 1 ] ] ]
]);
```

Figure 7.27 - Example of aggregate implementation in PHP

7.2.3.3.6 Map Reduce

MongoDB also provides map-reduce operations to perform aggregation. Map-reduce operations have two phases: a map stage that processes each document and emits one or more objects for each input document, and reduce phase that combines the output of the map operation. Like other aggregation operations, map-reduce can specify a query condition to select the input documents as well as sort and limit the results (MongoDB).

Map-reduce uses custom JavaScript functions to perform the map and reduce operations. While the custom JavaScript provide great flexibility compared to the aggregation pipeline, in general, map-reduce is less efficient and more complex than the aggregation pipeline.

An example of usage of map-reduce operation in the application, is analysis of a transport collection. So far there are implemented solutions for calculating the revenue per car, transporter and company. If company is selected, there are additional options added that the user can select from - a payment type. That can either be invoice, cash, credit card or all of these options together. In this specific case, MongoDB also emits and returns all known information about the company - name, address, registration number and tax number.

```
db.transports.mapReduce
(
  function(){ emit( {
    name: this.company.name,
    address: this.company.address,
    registrationNumber: this.company.ico,
    taxNumber: this.company.dic
  },
    this.price); },
  function(key, values){
    return Array.sum( values)
  },
  {
    query: { "type": "official",
      "order.payment" : "Cash",
      "arrivalDateDestination" : {
        $gte: new ISODate("2010-04-01T00:0:31Z"),
        $lt: new ISODate("2016-06-01T00:0:31Z")
      }
    },
    out: "total_price_per_company"
  }
);
```

Figure 7.28 - Example of map-reduce operation in MongoDB

UCN

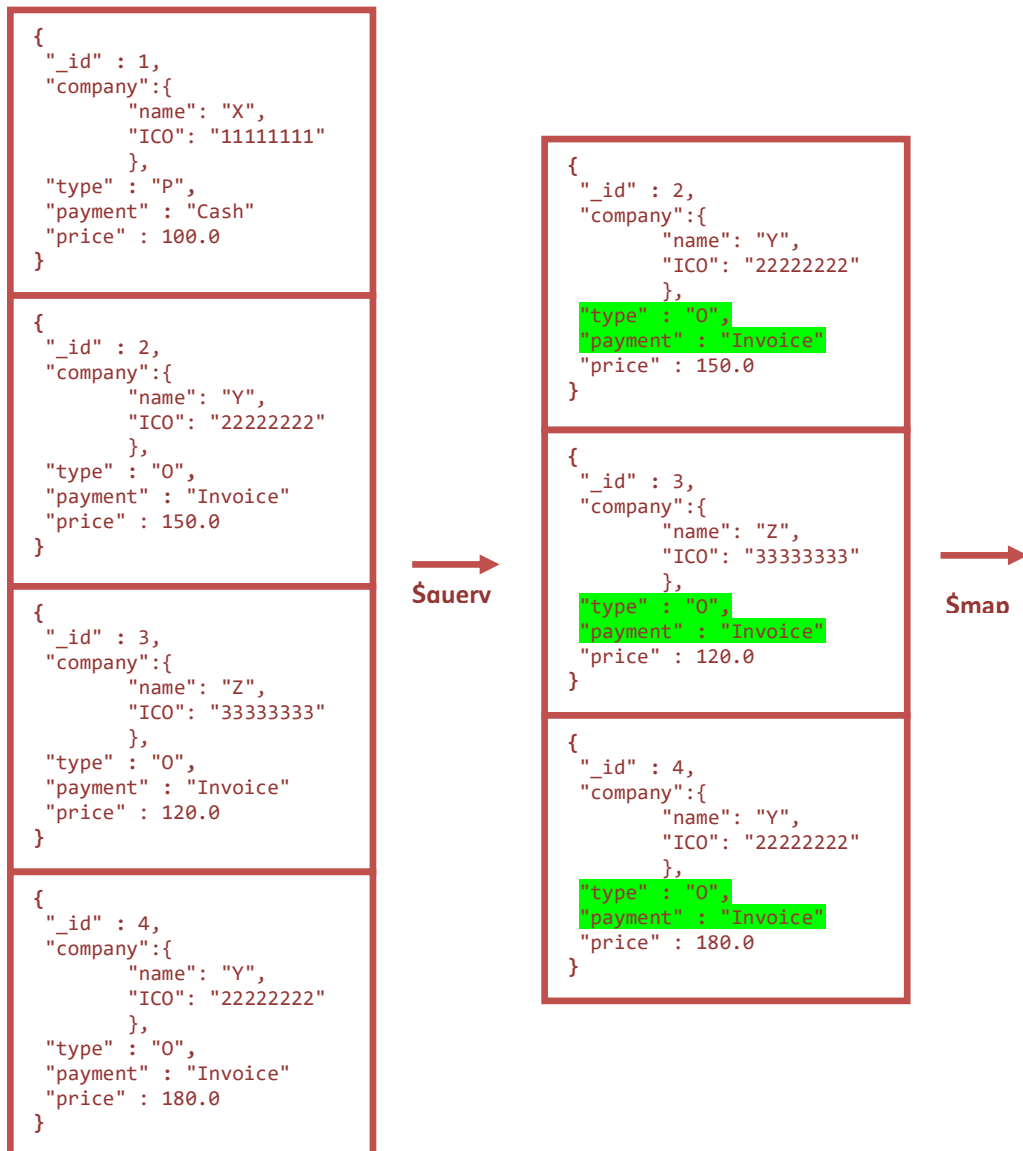


Figure 7.29 - Example of querying documents by (transport) type and payment type

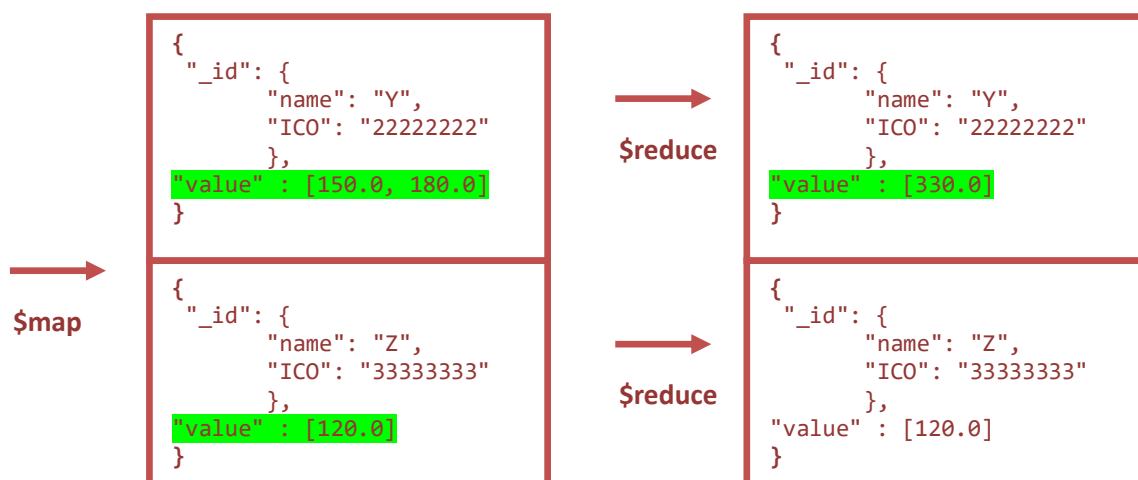


Figure 7.30 - Example of mapping and reducing documents from 'transports' collection

```

$statistics = $db->command(array(
    "mapreduce" => $mapReduce['mapreduce'],
    "map" => new MongoDB\BSON\Javascript($mapReduce['map']),
    "reduce" => new MongoDB\BSON\Javascript($mapReduce['reduce']),
    "query" => $mapReduce['query'],
    "out" => 'map_reduce_collection'));

$collection = DatabaseMongodbConnection::getCollection($db, "map_reduce_collection");
$cursor = $collection->find();

```

Figure 7.31 - Example of map-reduce implementation in PHP

7.2.3.3.7 Map reduce compared to Aggregation pipelines

Functionality-wise, Aggregation is equivalent to map-reduce but, on paper, it promises to be much faster. The following is the performance of two equivalent queries, one performed using Aggregation framework and one using map-reduce by Luca Marturana. Visualization of MongoDBs activity is done using Sysdig Cloud.

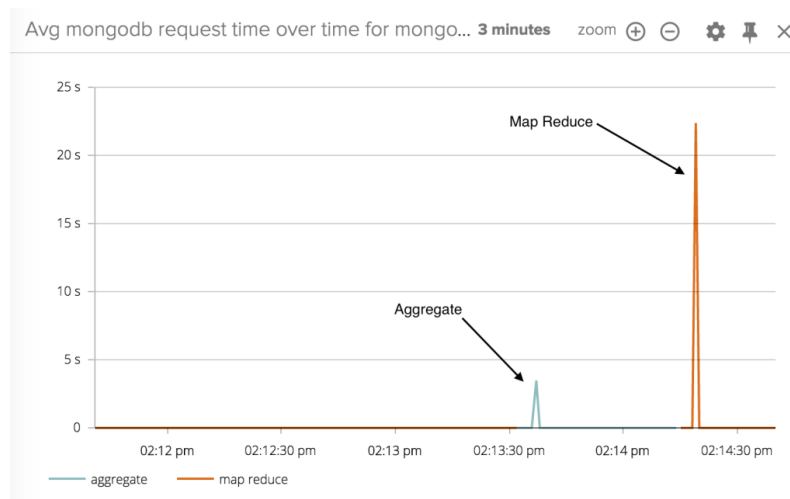


Figure 7.32 - Aggregate compared to Map-reduce 1

Correlating MongoDB activity with CPU usage.

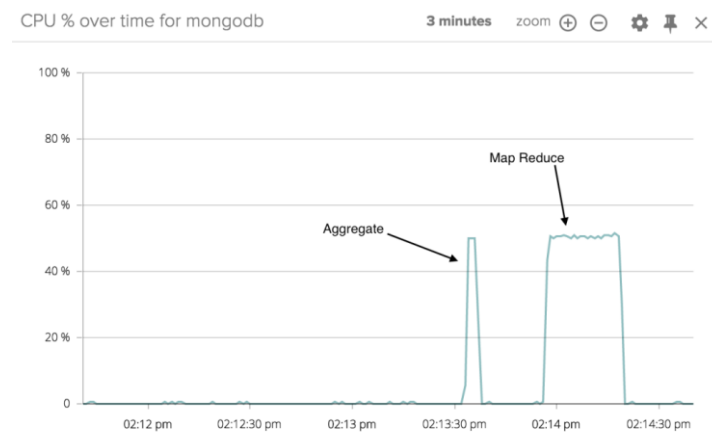


Figure 7.33 - Aggregate compared to Map-reduce 2

Aggregate is about 6x faster than map-reduce. The obvious conclusion is: if you are sending map-reduce queries to your Mongo backend and are concerned about performance, you should try switching to the Aggregation framework as soon as possible (Marturana, 2015).

8 Communication

Since the application is a three tier system, client and the server are present on a separate tiers. These tiers can be a separate physical machines and there has to be a way how they can communicate with each other. This communication was implemented in a way, that a client (presentation tier) sends requests to the server (business tier). In this section, HTTP requests will be discussed, as well as their implementation.

8.1 HTTP

HTTP stands for Hypertext Transfer Protocol. It's the network protocol used to deliver virtually all files and other data (collectively called resources) on the World Wide Web, whether they're HTML files, image files, query results, or anything else (Marshall, 2012).

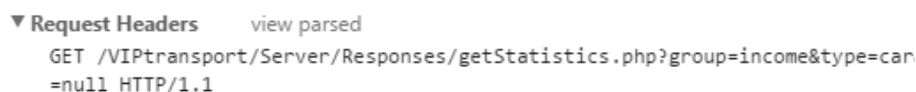
8.1.1 Structure of HTTP Transactions

Like most network protocols, HTTP uses the client-server model: An HTTP client opens a connection and sends a request message to an HTTP server; the server then returns a response message, usually containing the resource that was requested. After delivering the response, the server closes the connection (making HTTP a stateless protocol, i.e. not maintaining any connection information between transactions).

The format of the request and response messages are similar, both kinds of messages consist of: an initial line, zero or more header lines, a blank line and an optional message body (e.g. a file, or query data, or query output) (Marshall, 2012).

8.1.1.1 Initial Request Line

The initial line is different for the request than for the response. A request line has three parts, separated by spaces: a method name, the local path of the requested resource, and the version of HTTP being used (Marshall, 2012).



```
▼ Request Headers    view parsed
GET /VIPtransport/Server/Responses/getStatistics.php?group=income&type=can
=null HTTP/1.1
```

Figure 8.1 - Example of request header when client requests statistics information

8.1.1.2 Initial Response Line

The initial response line, called the status line, also has three parts separated by spaces: the HTTP version, a response status code that gives the result of the request, and an English reason phrase describing the status code (Marshall, 2012).

▼ Response Headers view parsed
HTTP/1.1 200 OK

Figure 8.2 - Example of response header

8.1.2 Request Methods

The request method indicates the method to be performed on the resource identified by the given Request-URI. The method is case-sensitive and should always be mentioned in uppercase (tutorialspoint).

S.N.	Method and Description
1	<p>GET</p> <p>The GET method is used to retrieve information from the given server using a given URI. Requests using GET should only retrieve data and should have no other effect on the data.</p>
2	<p>HEAD</p> <p>Same as GET, but it transfers the status line and the header section only.</p>
3	<p>POST</p> <p>A POST request is used to send data to the server, for example, customer information, file upload, etc. using HTML forms.</p>
4	<p>PUT</p> <p>Replaces all the current representations of the target resource with the uploaded content.</p>
5	<p>DELETE</p> <p>Removes all the current representations of the target resource given by URI.</p>

Table 8.1 - Commonly used HTTP request methods

8.1.3 Status codes

The Status-Code element in a server response, is a 3-digit integer where the first digit of the Status-Code defines the class of response and the last two digits do not have any categorization role. There are 5 values for the first digit (tutorialspoint):

S.N.	Code and Description
1	1xx: Informational The request has been received and the process is continuing.
2	2xx: Success The action was successfully received, understood, and accepted.
3	3xx: Redirection Further action must be taken in order to complete the request.
4	4xx: Client Error The request contains incorrect syntax or cannot be fulfilled.
5	5xx: Server Error The server failed to fulfill an apparently valid request.

Table 8.2 - Possible response status codes

8.2 AJAX

AJAX stands for Asynchronous JavaScript and XML. AJAX is a new technique for creating better, faster, and more interactive web applications with the help of XML, HTML, CSS, and JavaScript. Ajax uses XHTML for content, CSS for presentation, along with Document Object Model and JavaScript for dynamic content display.

Conventional web applications transmit information to and from the server using synchronous requests. It means you fill out a form, hit submit, and get directed to a new page with new information from the server. With AJAX, when you hit submit, JavaScript will make a request to the server, interpret the results, and update the current screen. In the purest sense, the user would never know that anything was even transmitted to the server and can continue to the application while the client program requests information from the server in the background.

XML is commonly used as the format for receiving server data, although any format, such as JSON or even plain text, can be used (tutorialspoint).

8.3 AJAX implementation

The jQuery's \$.ajax() function is used to perform an asynchronous HTTP request. The signatures of this function are shown below:

```
$.ajax(url[, options])
$.ajax([options])
```

The url parameter is a string containing the URL you want to reach with the Ajax call, while options is an object literal containing the configuration for the Ajax request.

In its first form, this function performs an Ajax request using the url parameter and the options specified in options. In the second form, the URL is specified in the options parameter, or can be omitted in which case the request is made to the current page.

There are a lot of different options you can specify to bend \$.ajax() to your need. In the list below you can find option that were commonly used in the application (Rosa, 2015).

Data	The data to send to the server when performing the Ajax request
dataType	The type of data expected back from the server
Error	A function to be called if the request fails
Password	A password to be used with XMLHttpRequest in response to an HTTP access authentication request
Success	A function to be called if the request succeeds
Url	A string containing the URL to which the request is sent

Figure 8.3 - Commonly used AJAX options

Here is an example of how ajax was used in the application. Function getFinishedTransports() is expected to return all transports that have been finished in the past. Since the function is used to get a resource, GET method was used. URL also had to be specified. File getFinishedTransports.php return data in a JSON format, therefore a data type option had to be specified. If the requests succeeds, handleData() function (specified in the parameter) is called. In case of an error, user is notified.

```
function getFinishedTransports(handleData){
    $.ajax({
        type: 'GET',
        url: '../Server/Responses/getFinishedTransports.php',
        dataType: 'json',
        data: '',
        success: function(response){
            handleData(response);
        },
        error: function(response){
            alert('We are sorry, transports could not be obtained');
        }
    });
}
```

Figure 8.4 - Example of AJAX implementation

8.4 Server response implementation

If the HTTP request was successfully received by the server, it sends the response back to the client. Here is an example of server response, when client requests statistics data.

```
<?php

require $_SERVER['DOCUMENT_ROOT'].'/VIPTransport/Server/Controller/statisticsController.php';

if(isset($_GET['group'], $_GET['type'], $_GET['query'])) {

    $statisticsControllerObject = new StatisticsController();
    $response = $statisticsControllerObject->getStatistics($_GET['group'], $_GET['type'], $_GET['query']);
    echo $response;
}

?>
```

Figure 8.5 - Example of Server response implementation

9 Conclusion

9.1 Project conclusion

Finally, all customer's requirements were successfully met. The web application is fully operational and helps all, customers, managers and transporters to keep track of their transportations.

While all functional requirements that were set at the beginning of the project were met, it was also necessary to give a significant amount of focus to the non-functional ones. It was enriching to see how big impact they might have on the whole user experience. Therefore usability was one of the biggest priorities when system was developed. On the other hand nothing is perfect and the final usability testing showed, that there are things to improve.

Besides usability, security is a very key element, especially when developing a web application. Number of different attacks can occur, against which the application has to defend. Finding a security hole in the application was a huge wakeup call of how dangerous the web can be.

One of the positive sites of the development was proper research before making final decisions when it comes to selecting the technology. As it has been said, there are many good technologies out there and it is the job of the developer to select what suits the application the best. In addition it is more and more common to select multiple technologies that serve different purposes in the application.

One of the biggest learning point was working for a real company. Communication with the customer really showed some of the necessities of SCRUM. On the other hand, some of the principles had to be adjusted because of the fact that the application was developed by an individual. It has been a great experience, to discuss the development with a real customer, who has a clear picture in mind of what to expect from the application, functionality-wise.

9.2 Customer review



VIP transport s.r.o.
Gemerská 4, 010 08, Žilina
IČO: 46 304 959
info@viptransport.sk

Application review

Planning

At the beginning we have had a meeting, where we discussed about how the company works and what would facilitate my work as well as collaboration with clients. I have set the requirements from the user perspective, which then Miroslav independently transformed into requirement specifications.

Application

The application satisfies all requirements, which were set at the beginning of our collaboration.

Collaboration

I am absolutely satisfied with collaboration with Miroslav. We have used a modified version of SCRUM development, but instead of daily stand-up meetings, we have had a synchronized online meeting every week. The length of an iteration was approximately two weeks. Miroslav had always delivered required functionality in expected time and quality.

Usage

I plan to test the application on selected clients. After fixing possible mistakes, I plan to incorporate the application in daily usage.

Acquisition

The application will spare manual work and help avoid mistakes, which accrue in current manual data processing. The application also fully automates the process from ordering to invoicing.

In Žilina, 1.6.2016

Katarína Ščambová, CEO of VIP transport

VIP transport s.r.o.
Gemerská 4, 010 08 Žilina
IČO: 46 304 959 DIČ: 2023323611
www.viptransport.sk

10 Bibliography

- acunetix. (n.d.). *Cross-site Scripting (XSS) Attack*. Retrieved from acunetix:
<http://www.acunetix.com/websitesecurity/cross-site-scripting>
- Beal, V. (n.d.). *structured data*. Retrieved from webopedia:
www.webopedia.com/TERM/S/structured_data.html
- Dash, J. (2013, September 18). *RDBMS vs. NoSQL: How do you pick?* Retrieved from zdnet:
<http://www.zdnet.com/article/rdbms-vs-nosql-how-do-you-pick/>
- Fowler, M. (2011, November 11). *PolyglotPersistence*. Retrieved from martinowler:
<http://martinfowler.com/bliki/PolyglotPersistence.html>
- Fowler, M. (2012, January 19). *AggregateOrientedDatabase*. Retrieved from martinowler:
<http://martinfowler.com/bliki/AggregateOrientedDatabase.html>
- Fowler, M. (2013, February 19). *Introduction to NoSQL • Martin Fowler*. Retrieved from youtube:
https://www.youtube.com/watch?v=ql_g07C_Q5I
- Graf, B. (2013, May 17). *Scalability: Scale-up or Scale-out, What it is and Why You Should Care*. Retrieved from vtation: <http://www.vtation.com/scalability-scale-up-scale-out-care/>
- Greiner, R. (2014, August 14). *CAP Theorem: Revisited*. Retrieved from robertgreiner:
<http://robertgreiner.com/2014/08/cap-theorem-revisited/>
- jqueryui. (n.d.). *jqueryui*. Retrieved from jqueryui home page: <https://jqueryui.com/>
- Knight, K. (2011, January 12). *Responsive Web Design: What It Is and How To Use It*. Retrieved from smashingmagazine: <https://www.smashingmagazine.com/2011/01/guidelines-for-responsive-web-design/>
- Lionite. (2014, March 27). *Using PHP with MySQL - the right way*. Retrieved from binpress:
<https://www.binpress.com/tutorial/using-php-with-mysql-the-right-way/17>
- Long, N. (2008, October 2). *JavaScript: client-side vs. server-side validation*. Retrieved from stakoverflow: <http://stackoverflow.com/questions/162159/javascript-client-side-vs-server-side-validation>
- Mack, J. (2014, March 28). *Five Advantages & Disadvantages Of MySQL*. Retrieved from datarealm:
<https://www.datarealm.com/blog/five-advantages-disadvantages-of-mysql/>
- Marshall, J. (2012, December 10). *HTTP Made Really Easy*. Retrieved from jmarshall:
<https://www.jmarshall.com/easy/http>
- Maturana, L. (2015, March 12). *MongoDB Showdown: Aggregate vs Map-Reduce*. Retrieved from The Official Sysdig Blog: <https://sysdig.com/blog/mongodb-showdown-aggregate-vs-map-reduce/>

- MongoDB. (n.d.). *Aggregation Introduction*. Retrieved from MongoDB documentation:
<https://docs.mongodb.com/v3.0/core/aggregation-introduction/>
- MongoDb. (n.d.). *cursor.sort()*. Retrieved from MongoDB documentation:
<https://docs.mongodb.com/v3.0/reference/method/cursor.sort/#cursor.sort>
- MongoDB. (n.d.). *db.collection.find()*. Retrieved from mongoDB documentation:
<https://docs.mongodb.com/v3.0/reference/method/db.collection.find/>
- owasp. (2016, May 22). *Cross-Site Request Forgery (CSRF)*. Retrieved from owasp :
[https://www.owasp.org/index.php/Cross-Site_Request_Forgery_\(CSRF\)](https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF))
- Peacock, M. (2010, April 13). *The What, Why and How of Usability Testing*. Retrieved from cmswire:
<http://www.cmswire.com/cms/web-engagement/the-what-why-and-how-of-usability-testing-007152.php>
- Perkins, A. (2014, Jun 29). *Atomic vs. Eventual Transactions*. Retrieved from youtube:
<https://www.youtube.com/watch?v=nVWieRqU6HE>
- Perkins, A. (2014, Jun 29). *Data Structures*. Retrieved from youtube:
https://www.youtube.com/watch?v=I_jZzT5gSEc
- Perkins, A. (2014, Jun 22). *Relational vs NoSQL Databases*. Retrieved from youtube:
<https://www.youtube.com/watch?v=XPqrY7YEs0A>
- php. (n.d.). *The MongoDB\Driver\BulkWrite class*. Retrieved from php documentation:
<http://php.net/manual/en/class.mongodb-driver-bulkwrite.php>
- planetcassandra. (n.d.). *planetcassandra*. Retrieved from planetcassandra:
<http://www.planetcassandra.org/what-is-nosql/>
- Polacek, J. (n.d.). *What The Heck Is Responsive Web Design?* Retrieved from johnpolacek:
<http://johnpolacek.github.io/scrolldeck.js/decks/responsive/>
- Ramez Elmasri, S. B. (2011). Chapter 12.1 - Structured, Semistructured, and Unstructured Data. In N. Elmasri, *Fundamentals of database systems*. Pearson Education.
- Ramez Elmasri, S. B. (n.d.). Chapter 3.1 - Relational Model Concepts. In S. B. Ramez Elmasri, *Fundamentals of database systems*. Pearson Education.
- Ramez Elmasri, S. B. (n.d.). Chapter 3.2 - Relational Model Constraints and Relational Database Schemas. In S. B. Ramez Elmasri, *Fundamentals of database systems*. Pearson Education.
- Reinero, B. (2015, July 22). *Polyglot Persistence: Hybrid Apps with MongoDB and RDBMS*. Retrieved from youtube: https://www.youtube.com/watch?v=X1yNTq_R2JA
- Rest, N. v. (2010, August 1). *Explanation of BASE terminology*. Retrieved from stackoverflow:
<http://stackoverflow.com/questions/3342497/explanation-of-base-terminology>

- Rosa, A. D. (2015, August 26). *How to Use jQuery's \$.ajax() Function*. Retrieved from sitepoint: <https://www.sitepoint.com/use-jquery-ajax-function/>
- Rouse, M. (2010, April). *unstructured data*. Retrieved from searchbusinessanalytics.
- Sadalage, P. (2014, October 1). *NoSQL Databases: An Overview*. Retrieved from thoughtworks: <https://www.thoughtworks.com/insights/blog/nosql-databases-overview>
- Sasaki, B. M. (2015, September 4). *Graph Databases for Beginners: ACID vs. BASE Explained*. Retrieved from Neo4j: <http://neo4j.com/blog/acid-vs-base-consistency-models-explained/>
- Sasaki, B. M. (2015, August 28). *Graph Databases for Beginners: Why We Need NoSQL Databases*. Retrieved from Neo4j: <http://neo4j.com/blog/why-nosql-databases/>
- Serra, J. (2015, July 1). *What is Polyglot Persistence?* Retrieved from jameserra: <http://www.jameserra.com/archive/2015/07/what-is-polyglot-persistence/>
- Siles, R. (2016, August 1). *Session Management Cheat Sheet*. Retrieved from owasp: https://www.owasp.org/index.php/Session_Management_Cheat_Sheet
- Subieta, K. (2008, January 15). *Impedance mismatch*. Retrieved from ipipan: http://www.ipipan.waw.pl/~subieta/SBA_SBQL/Topics/ImpedanceMismatch.html
- techterms. (2013, March 1). *jQuery*. Retrieved from techterms: <http://techterms.com/definition/jquery>
- Tomar, N. (2011, July 20). *SQL Server: ACID Properties*. Retrieved from c-sharpcorner: <http://www.c-sharpcorner.com/blogs/sql-server-acid-properties1>
- tutorialspoint. (n.d.). *DBMS - Transaction*. Retrieved from tutorialspoint: http://www.tutorialspoint.com/dbms/dbms_transaction.htm
- tutorialspoint. (n.d.). *HTTP - Methods*. Retrieved from tutorialspoint: http://www.tutorialspoint.com/http/http_methods.htm
- tutorialspoint. (n.d.). *HTTP - Status Codes*. Retrieved from tutorialspoint: http://www.tutorialspoint.com/http/http_status_codes.htm
- tutorialspoint. (n.d.). *MongoDB - Environment*. Retrieved from tutorialspoint: http://www.tutorialspoint.com/mongodb/mongodb_environment.htm
- tutorialspoint. (n.d.). *MongoDB - Overview*. Retrieved from tutorialspoint: http://www.tutorialspoint.com/mongodb/mongodb_overview.htm
- tutorialspoint. (n.d.). *What is AJAX?* Retrieved from tutorialspoint: http://www.tutorialspoint.com/ajax/what_is_ajax.htm
- wikipedia. (2016, May 14). *Object-relational impedance mismatch*. Retrieved from wikipedia: https://en.wikipedia.org/wiki/Object-relational_impedance_mismatch