Tommi Pälviö

Designing and implementing a web application for survey management

Metropolia University of Applied Sciences

Bachelor of Engineering

Information and Communications Technology

Thesis

11 April 2017

Metropolia

Author Title Number of Pages Date	Tommi Pälviö Designing and implementing a web application for survey management 51 pages 11 April 2018
Degree	Bachelor of Engineering
Degree Programme	Information and Communications Technology
Professional Major	Software Engineering
Instructors	Peter Hjort, Senior Lecturer

This thesis covers the process of designing and implementing a web application for managing e-form based surveys. This thesis focuses on the implementation of the application's server side, and the testing of the application.

The goal was to create a user-friendly application matching the requirements of the clients using modern web technologies. The design was based on a special type of surveys that utilize multidimensional questions and emphasize the answerer background in the result analysis. The application was developed on Node platform utilizing currently relevant frameworks.

Due to difficulties caused by the initial technology stack, two versions of the application were created. The first version was finished and delivered to the clients to serve as a test run for the concept, while the second version with a different technology stack was developed in the background. The specifications for the system underwent changes during the development, and the second version of the application was also used to alter certain aspects of the design to better fit the changed use case.

In the end the final version of the application was delivered to the clients. While the clients found general usability of the application slightly lacking, the survey management system was implemented successfully to meet the initial requirements.

Keywords	Node.js, Koa.js, Mocha, Sails.js, web development

Metropolia

Tekijä Otsikko Sivumäärä Aika	Tommi Pälviö Web-sovellus sähköisten kyselyiden hallintaan 51 sivua 11.4.2018
Tutkinto	Insinööri (AMK)
Tutkinto-ohjelma	Tieto- ja viestintätekniikka
Ammatillinen pääaine	Software Engineering
Ohjaaja	Lehtori Peter Hjort

Insinöörityön tarkoituksena oli kehittää web-sovellus sähköisiin lomakkeisiin pohjautuvien kyselyiden hallintaan. Insinöörityössä keskityttiin erityisesti sovelluksen palvelinpuolen suunnitteluun, tekniseen toteutukseen ja testaukseen.

Työn tavoitteena oli luoda asiakkaan vaatimuksia vastaava sovellus hyödyntäen ajankohtaisia web-teknologiota. Sovelluksen suunnittelun pohjana olivat asiakkaan käyttämät erikoiskyselyt, jotka perustuvat moniulotteisiin kysymyksiin ja joiden tuloksia analysoidaan vastaajien taustatietoja painottaen. Sovellus kehitettiin Node-alustalle hyödyntäen alustalle saatavilla olevia web-kehitykseen suunniteltuja kehyksiä.

Alkuperäisen kehyspinon aiheuttamien ongelmien vuoksi sovelluksesta kehitettiin kaksi versiota. Ensimmäinen versio kehitettiin loppuun käyttäen alkuperäisiä kehyksiä, ja se toimi asiakkaalla testikäytössä, kunnes taustalla kehitetty uudistettuun kehyspinoon pohjautuva versio saatiin valmiiksi. Sovellukselle asetetut vaatimukset muuttuivat huomattavasti kehitysprosessin aikana, ja sovelluksen toisen version suunnittelua muutettiin paremmin vastaamaan sovelluksen uudistunutta käyttötarkoitusta.

Lopullisen version valmistuttua se toimitettiin asiakkaalle. Lopullisen version kokonaiskäytettävyys ei vastannut täysin asiakkaan odotuksia, mutta itse kyselyiden hallintaominaisuus toteutettiin onnistuneesti vastaamaan asiakkaan vaatimuksia.

Avainsanat	Node.js, Koa.js, Mocha, Sails.js, web-kehitys

Metropolia

Contents

List of Abbreviations

1 Introduction 1

2 Theoretical background 1

2.1 MongoDB database 2
2.2 MariaDB 4
2.3 JavaScript programming language 6
2.4 Node.js platform 8
2.5 Frameworks 12
2.5.1 Sails framework 12
2.5.2 Koa framework 14
2.6 Mocha testing framework 16

3 System design 16

3.1 Overview of the application 16
3.2 Project QAEMP 17
3.3 Initial technologies 18
3.4 Changing the stack 18
3.5 Database solution 19
3.6 Modeling 20
3.6.1 Ownership of records 21
3.6.2 Templates 22
3.6.3 Survey sharing 22

4 Technical implementation 23

4.1 Sails stack 23
4.1.1 Policies 24
4.2 Koa stack 25
4.2.1 Server module 25
4.2.2 Middleware 25
4.2.3 Controllers 30
4.2.4 Loading modules 33
4.2.5 Model implementation 34

Metropolia

4.3 Exporting result data 38
4.4 Aggregating the result data 40
4.5 Localization 41
4.6 Testing 43
4.6.1 Manual testing 43
4.6.2 Model tests 44
4.6.3 Controller tests 45
5 Development 46
5.1 Methodology 46
5.2 Difficulties during the development 47
5.3 Continued development 48
6 Synopsis 48
References 50

List of Abbreviations

RDBMS — Relational database management system. A program or a collection of programs that manage a relational database.

JSON — JavaScript object notation. Data format for representing data with objects consisting of key-value pairs.

SQL — Structured query language. Language used by most relational databases for defining database structures and forming queries.

HTTP — Hypertext transfer protocol. Networking protocol used by browsers and web servers for communication over the World Wide Web.

API — Application programming interface. A set of instructions or tools that function as an interface for other applications.

NPM — Node package manager. Tool for managing dependencies of a project, and distributing and installing Node modules.

SPA — Single page application. A web application that dynamically modifies the contents of a page instead of loading entirely new pages from a server.

1 Introduction

The purpose of this thesis was to design and implement a user-friendly web application for managing and running e-form based surveys using modern web technologies. The project was ordered by Katriina Schrey-Niemenmaa and Markku Karhu, who both work as lecturers at Metropolia UAS. The application was designed to be used as an electronical tool in an ongoing project QAEMP. Publicly available survey management tools did not provide the support for result analysis in the way that was needed in project QAEMP, which is why the clients wanted to develop a tool that was designed specifically for the needs of the project.

There are two theses written about this project. This thesis focuses on the techniques and technologies used to develop the server side of the application as well as the testing of the application, and the other one, written by Miika Ahonen, focuses on the frontend development and technologies. This thesis will describe the design and development processes of the project and the technical solutions used to create the application.

The application was developed on Node platform, and currently popular web development frameworks were utilized in the development. This thesis covers the usage of the main frameworks, Sails and Koa, used in the backend development. A major part of the design process of the application was to create a model system to define a structure for the data going through the application, and the modelling process is one of main topics of this thesis. This thesis also describes how Mocha testing framework was utilized as a tool in the development to create a more maintainable code base.

2 Theoretical background

This section gives an overview of all the essential technologies used in the project and describes how these technologies are currently relevant. The reasons why specific technologies were chosen for the project is covered in chapter three.

2.1 MongoDB database

MongoDB is a document-oriented database software and one of the more popular NoSQL database solutions. There are multiple design philosophies that separate MongoDB from traditional RDBMS (relational database management system). Most notably, MongoDB saves data as self-contained documents instead rows across tables. MongoDB has a heavy emphasis on speed, scalability and usability, and it is generally a more specialized solution than RDBMS. Feature wise the major difference between MongoDB and traditional a RDBMS is the lack of support for transactions for MongoDB, which makes RDBMS the optimal solution in cases where data consistency is a priority. Leaving out the element providing data consistency was a deliberate design decision to streamline the implementation, and MongoDB compensates the lack of data consistency with other features like native support for clustering. [1, 2-3.]

The emphasis on clustering is what makes MongoDB highly scalable. Clustering is effective with MongoDB because it stores data as self-contained documents that exist entirely in one database. With relational data, clustering is more complicated, because it can never be guaranteed that the whole document with all its relations reside on one server. Because of the complexity, RDBMS solutions do not natively support clustering. The benefit of clustering is that it makes so called horizontal scaling possible, meaning that the performance of a system, like a database server, can be increased by adding more individual machines to work on the same task that is being split between the machines. The alternative method of scaling is vertical scaling, which means increasing the performance of a system by upgrading the hardware of the machines running it. Horizontal scaling is in most cases considered to be the preferable solution, meaning that it is often more cost effective to have a large number of low-end machines working together instead of a single or a few really expensive ones. [1, 6.]

MongoDB uses BSON (binary JSON) as its data format. BSON is a schemaless data format that resembles JSON (JavaScript object notation). The data format effectively describes the contents of a document, so the structure of document does not need to be specified beforehand. [1, 3-4.] Because of the schemaless data format, any stored document can be modified or replaced independently from other documents [1, 3]. The main benefit from using BSON instead of JSON is the traversing speed, which results in faster query and indexing operations. Another benefit is that the conversion to any programming language's native data format is faster with BSON. The only notable downside in

using BSON is that it requires slightly more disk space than standard JSON, which is considered an acceptable trade-off for the faster database operations. [1, 10.]

MongoDB supports dynamic queries that do not need to be prepared in advance. MongoDB queries work the same way as SQL queries by simply specifying the parts of the document the query should be based on, and the database management system will handle processing the query. [1, 11.]

Aggregation framework is one way to quickly construct and run complex queries with MongoDB. The framework is a pipeline-based tool that allows for creating queries in smaller pieces using its own set of operators. [1, 14.] In an aggregation, documents go through a multi-stage pipeline that transforms the documents into result objects as can be seen in image 1. The pipeline consists of stages that provide operations such as document filtering and transformation. Each stage can additionally contain operators to further modify the passing document. An operator can, for example, calculate the average value or concatenate a string based on documents' fields. Aggregation pipeline is based on native MongoDB operations and it is the preferred method for data aggregation with MongoDB. [2.]

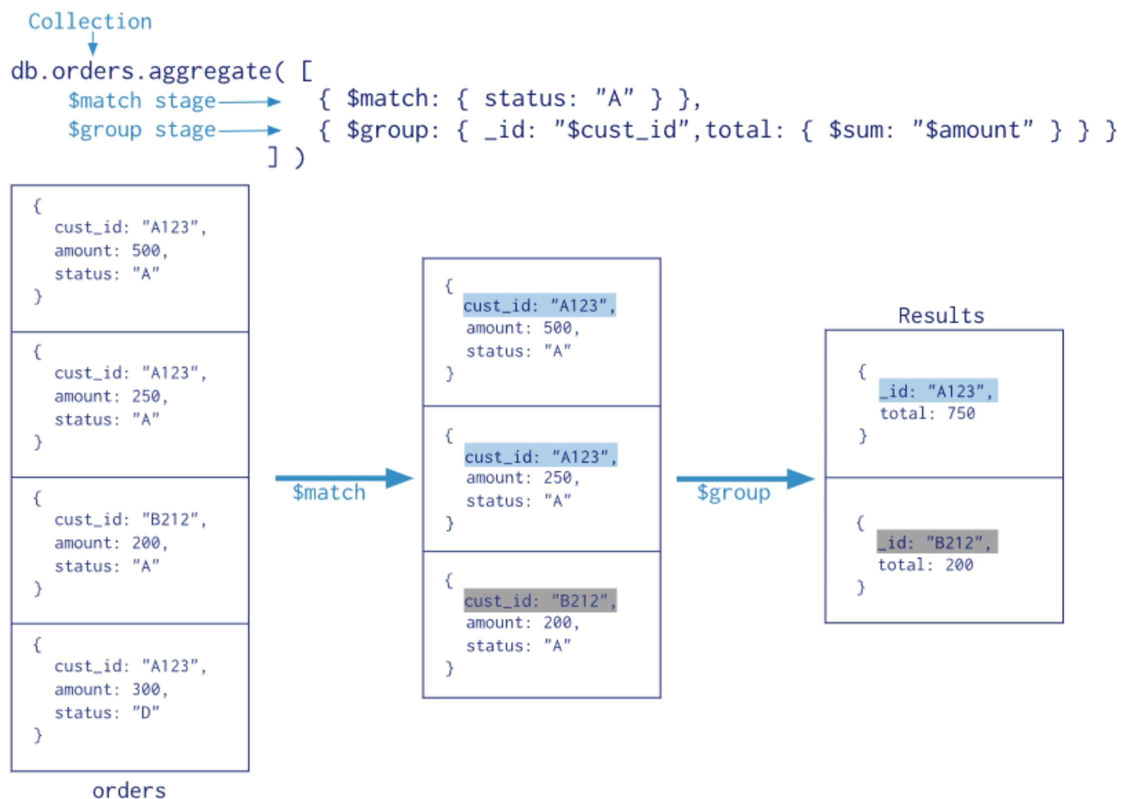

Image 1. An example of a MongoDB aggregation [2].

Indexing is a feature that can be used to speed up queries and optimize the performance of read operations. The keys of documents in a collection can be indexed, and after indexing a key, MongoDB doesn't need to go through every record in the database to find the relevant documents if queried by the index. Indexing can also be used to enforce the uniqueness of a key, which is why by default every document in MongoDB is indexed on the _id key. [1, 11-12.] The downside of using indexes is that they will make the write operations slower, because the indexes need to be updated and validated with every write operation. Adding too many indexes to a collection can severely hinder the write performance. [1, 264.]

2.2 MariaDB

MariaDB is relational database management system. RDBMS refers to a program or a collection of programs that manage a relational database. The purpose of an RDBMS is to guarantee that the data gets stored in organized manner, restrict the access to the

data, and provide operations like data retrieval and management. [3, 23-24]. Other popular database management systems include Oracle, PostgreSQL and SQLite. MariaDB is based on the SQL like most RDBMS programs. [3, 24; 4, 11.]

In relational databases the data is stored inside tables, and a single database can contain multiple tables. Tables consist of one or more named columns that contain a singular piece of data relating to a single record. The data of a table is represented in two dimensions with rows and columns, and a row is the smallest unit of information a table can store. The columns of a table will define the structure of the table and dictate how data is stored. Each column is given a datatype that defines what kind of data can be stored in the column. [4, 14; 3, 23-24.]

In the relational model that is the foundation for all relational databases, every record inside a table is given a unique identifier also called a primary key. The primary key is the safest way of accessing a specific record in the database, because it will always point to the same record even if the particular record has been modified. The primary key is what allows a table to contain references to other tables. A column can be used to store a foreign key, which is a primary key pointing to a record stored in another table. This is safe because primary keys are guaranteed to be unique within a table. The references can be used to split the stored data into separate logical structures with each structure having its own table. [4, 33-34.]

SQL (structured query language) is the language used to manage most relational databases, and it is considered to have three main purposes. Firstly, SQL is used in creating a new database and defining a database structure by creating the tables and columns. Secondly, the language is used to control database security and access. Thirdly, SQL is used to define queries, which is the most common use case for the language. Queries are processed by the database, and commonly queries are either meant to retrieve some data from the database or to manipulate the data inside the database. Queries consist of multiple different components that make it possible to customize a query to a great detail to for example retrieve a very specific piece of information from a database. [4, 11-12.]

In SQL a transaction refers to a group of operations considered to form a logical unit of work. In order for a group to be considered a transaction, it must follow the ACID principle that dictates that a transaction needs to guarantee the consistency of the database during and at the end of a transaction. The major implications of the principle are that in

case one of the operations forming a transaction fails to execute, the changes made by all the other operations will be reversed, so that either all of the operations succeed or not a single one will. Transactions are the main tool used to maintain data integrity during write operations, and they are most commonly used to group multiple write operations when the data is being written to multiple different tables and the writes are related to each other. [4, 303.]

2.3 JavaScript programming language

JavaScript is a prototype-based programming language released in 1995. JavaScript is best known for being the language used in web development to define how web pages react to dynamic events such as mouse clicks. JavaScript is a flexible language and it supports object oriented, imperative and functional programming styles. [5.]

JavaScript was originally developed to enhance the user experience of web sites, but today the language is also widely used outside of web development, and it is possible to use the language to build, for example, a desktop widget or a server side of an application. [6, 2-3; 7, 1-3.] Due to being designed for a specific use case, the feature set of JavaScript was originally somewhat limited in comparison to many other programming languages, but active development of the language and the release of a JavaScript based Node platform have made JavaScript a popular language especially in backend programming. [5.]

JavaScript is based on ECMAScript standard and the word JavaScript is commonly used to refer to one implementation of the standard. In order for an implementation to be considered valid, it needs to support all the features and the syntax defined by the standard. In addition to the required features stated by the standard, developers are free to add their own features to their implementation, also called an engine. The open nature of engines is expected to increase the innovativeness of the developers, but on the contrary developers are also forced to take engines designed by other developers in to consideration. [7, 1-2.] Widely used implementations of the standard include SpiderMonkey developed by Mozilla, and Google’s V8 that is used by Google Chrome browser and Node platform [5]. ECMAScript standard is still under active development and the latest version, ECMAScript 8, was released in 2017 [8].

ECMAScript 2016 or ES5 was a major update to the language and it established the language as more than just a scripting language by introducing a number of new features. One of the major changes was the introduction of classes. Previously the only way to imitate classes was to use prototype-based class-like structures. New class syntax was accompanied by static methods and sub-classing which make writing classes with the language far more accessible than before. The standard also added “let” and “const” keywords for declaring mutable and read only variables. Other major features added in the version were promises and generators. [9.]

A promise is placeholder for value that may not be known when the promise is initialized but might be available in the future. A promise allows asynchronous methods to return values such as synchronous methods by immediately returning a promise that will resolve to a value at some point in the future. A promise has three states; pending, rejected and fulfilled. A promise that has yet to be resolved or rejected is pending. Promises are associated with handlers that will get executed once the asynchronous function returns a value or throws an error. Listing 1 shows how a promise object is constructed and executed by providing the constructor with two handler functions to run depending on whether the inner function returns a value or throws an error. [10.]

```
let myFirstPromise = new Promise((resolve, reject) => {
  setTimeout(function(){
    resolve("Success!");
  }, 250);
});

myFirstPromise.then((successMessage) => {
   console.log(successMessage);
}).catch(error => console.log(error));
```

Listing 1. Defining and executing a simple promise [10].

Generators are special functions that can be used to define iterative algorithms. A generator function is able to maintain its own state and it functions as factory for iterators. Iterators in JavaScript are objects that provide a next method that returns the next value in a sequence. Generator functions are defined using the syntax seen in listing 2. The internal state of a generator can be modified with the next method by passing it a parameter that will be considered as the result of the latest yield expression. [11.]

```
function* idMaker() {
  var index = 0;
  while(true)
    yield index++;
}
```

```
var gen = idMaker();

console.log(gen.next().value); // 0
console.log(gen.next().value); // 1
console.log(gen.next().value); // 2
// ...
```

Listing 2. An example of a simple generator function [11].

A major use case for generators is to represent complex sequences. Generators are effective in representing even infinite sequences without any kind of overhead, because generators compute the next value to be yielded only on demand. Generators can also be used to execute asynchronous operations in a synchronous way, because the value of an asynchronous function can be yielded without providing the function a callback. [11; 9.]

ECMAScript 8 added a new way of executing asynchronous code by introducing async functions. Inside an async function an await expression can be used to halt the execution of the function and wait for the specified promise to resolve or reject. Listing 3 demonstrates the usage of an async function. [12.]

```
async function asyncFunction() {
  var result = await new Promise(resolve => {
    setTimeout(() => { resolve('resolved'); }, 2000);
  console.log(result);
}

asyncFunction(); // expected output: "resolved"
```

Listing 3. An example of an async function.

Async functions were designed to simplify using promises in a synchronous fashion. Like generator functions, async functions can be used as an alternative way of running promises. [12.]

2.4 Node.js platform

Node.js is a popular JavaScript runtime that is widely used as a server-side platform to run JavaScript based web applications. The runtime is based on Google's V8 JavaScript engine, and it utilizes non-blocking event-driven architecture with the single threaded programming model. Node is compatible with most of the new features introduced in the latest version of ECMAScript, and additionally the platform contains built-in libraries

called core modules. [13, 1; 9.] Node.js is different from other web development platforms for it can create its own server to handle requests and serve static files, and therefore programs written for the platform are not necessarily dependant on a separate server layer such as the Apache server [14].

One reason for the high popularity of Node is its capability to handle requests quickly, which has made it a preferable platform for fast and scalable web applications. The architecture the platform is based on makes it possible to run operations asynchronously so that slower operations do not significantly impair the performance of the whole application. [13, 1; 15.] When running operations asynchronously functions are usually provided with another function as a callback that will be executed after the original function has been successfully executed. With a callback the platform does not need to wait for the original function to finish, and instead it is able to freely run other operations and return to execute the callback function after original function has been executed. [13, 1.]

Node is event driven meaning that the platform will wait for a new event, and then handle the event as instructed. The event driven nature of the platform is built around so-called event loop that is responsible for listening for new events and scheduling operations. [16; 9.] The event loop also makes it possible for the platform to perform non-blocking operations despite it being based on the single threaded programming model [17]. The event loop utilizes the system kernel by delegating operations to the kernel as much as possible, and when an operation is executed, the kernel will notify Node to que an appropriate callback to be executed. When running, Node will continuously process the event loop that consists of several different phases that can be seen in image 2. [16; 17.]

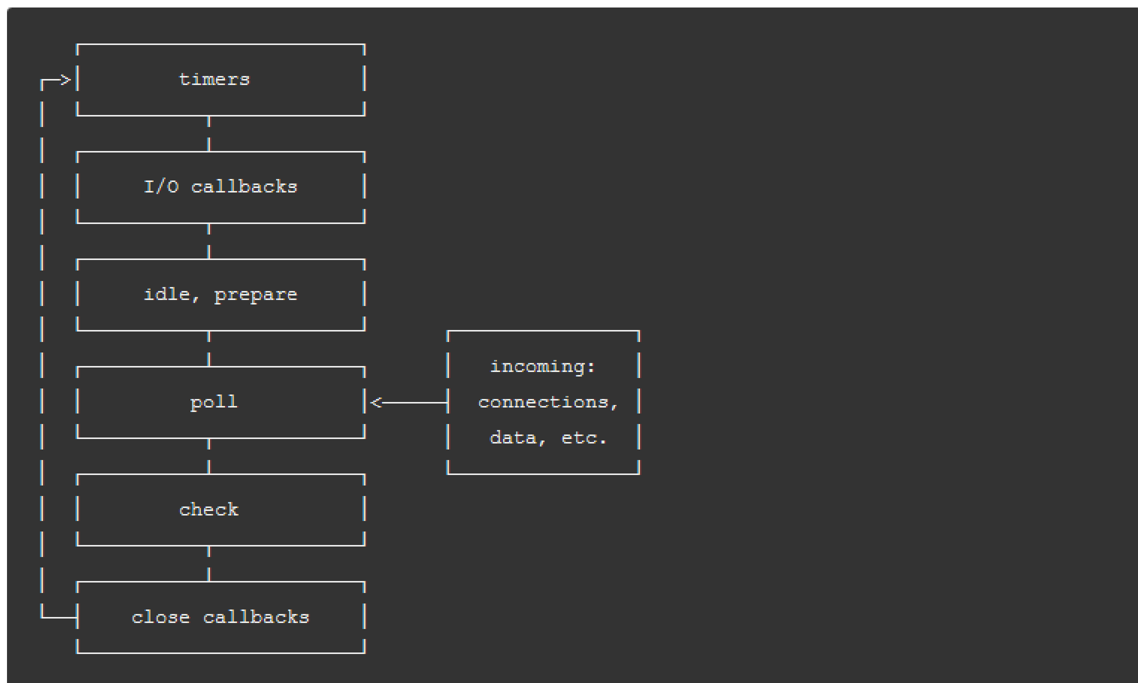

Image 2. Different phases of the event loop [17].

Every phase of the event loop has its own queue of callbacks functioning with the first in first out principle, and every phase contains operations specific to the particular phase. Timers phase is used to execute callbacks set by JavaScript functions setTimeout and setInterval. The next phase, I/O callbacks, is the phase that executes callbacks of returning I/O operations, so if for example a non-blocking network operation is returning, its callback will be triggered in this phase. New I/O operations are retrieved by the poll phase, followed by check phase that allows executing callbacks scheduled by setImmediate function immediately after the poll phase. The last phase is used to callbacks related to closing a handle or a socket. When entering a phase, the event loop will execute callbacks from that phase's queue until there are no callbacks in the queue or the maximum number of callbacks has been executed, before moving to the next phase. Every time after finishing the event loop, the platform will check whether it is waiting for any I/O operations or timers, and if not, it will perform a clean shutdown. [9; 17.]

Traditional webservers, like Apache, either create a new process to handle a request or pass the request to an idle thread. The handling of a request usually involves the application executing some time consuming operation such as accessing a networked resource or some data on a disk, and a thread or process will have to wait until all the operations related to a request are completed before the request can be consider handled, and the server is able to terminate the process or delegate a new task to the thread.

Spawning and maintaining threads and processes are heavy operations for the server's memory and CPU. Multiple simultaneous requests will cause the server to spawn multiple threads or processes to match the number of incoming requests, which may temporarily hinder the performance of the server. Instead of having multiple threads or processes, node-based servers handle every request using only one thread while utilizing asynchronous execution and are capable of handling more requests per second than traditional webservers as can be seen in image 3. [18, 23-26.] Node can also made to utilize multiple CPU cores by creating multiple processes. These processes can function as separate servers or as child processes of the main server. [14.]

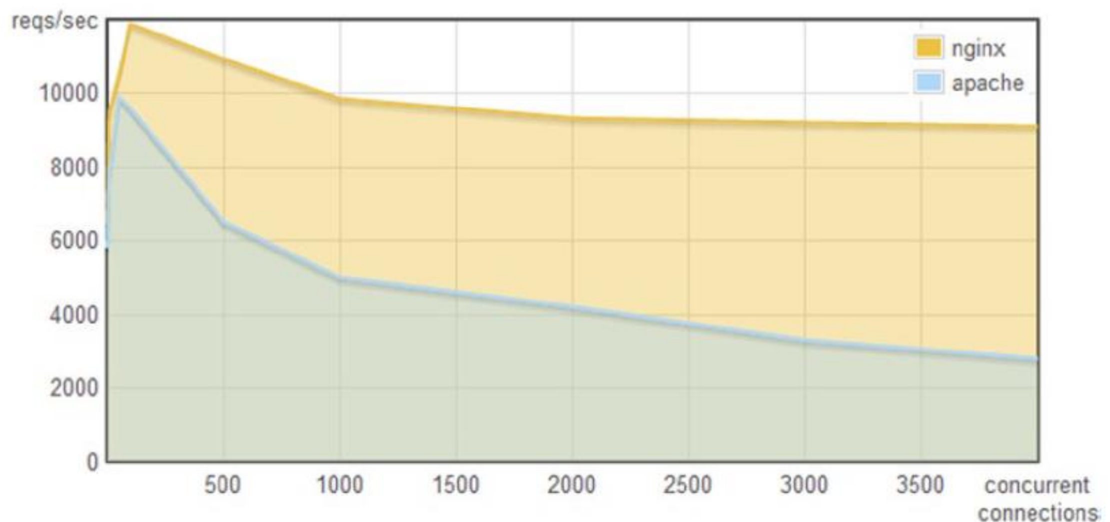

Image 3. Performance comparison between Apache and Node based server [18, 26].

Node platform supports modularity and it's common that software written on the platform utilize third-party modules and frameworks. Modules are an essential part of developing on the platform, because JavaScript is a relatively simple language and it does not include its own standard library unlike many other programming languages. Module management is commonly handled by NPM (Node package manager) package manager that allows adding modules written by other developers from the central NPM registry to your own application and publishing your own modules for others to use. Despite being a separate project from Node.js, due to its popularity and general acceptance, the platforms installing package has included the package manager since version 0.6.3. [13, 9-11; 9.]

Node platform also brings certain other benefits especially to web development. A major advantage is that all modern web browsers are capable of executing JavaScript, which

means that components, such as libraries and classes, can technically be shared between the front and backend of an application. Being able to use JavaScript as a programming language in both the server and browser parts of an application also means that developers do not necessarily have to simultaneously assume multiple languages. The platform also currently has a rich ecosystem and there is a large number of third-party modules available through the platform's package manager. [19.]

2.5 Frameworks

Instead of just adding additional functionality to an application like modules and libraries, frameworks generally function as a foundation for an application. There are different kind of frameworks and some of them only provide a layout for the developers to build upon, whereas so called full-stack frameworks define entirely the technical implementation of an application. [9.]

In web development, frameworks can be split into frontend and backend frameworks. Backend framework is a component of the server side that is usually involved in the process of request handling. Frontend framework is part of browser side that usually affects how a web page is rendered and how it reacts to user interaction. [9.]

Most backend frameworks of Node.js platform conform to MVC (model-view-controller) pattern. MVC model consists of three components; model, view and controller. Model includes all logic and operations related to data. View is responsible for representing the data. Controller accepts user input and transforms the input into commands for views and models. Another typical feature in backend frameworks of the platform is the use of middleware. Middleware are functions usually designed to process requests and responses, and they are responsible for connecting the different components of a software. [20.]

2.5.1 Sails framework

Sails is a MVC style backend framework for Node.js platform. Sails is built upon Express framework and it consists of multiple smaller modules. Sails contains large amount of functionality in itself, and it defines strict conventions for the development. Due to the enforced guidelines, the development with Sails is straightforward and the framework is considered to be highly user friendly. Waterline library is a major feature of Sails, and it

makes it possible to utilize multiple different database solutions with a Sails application by supporting relations between models stored in different types of databases. [9; 20]. With Sails models are defined as separate modules by specifying the attributes and model relations using the syntax seen in listing 4. The system allows for extensive customization of a model, and the database library will enforce the created schema, and update or create a database table accordingly to match the schema. Relations between models are created simply by defining the target model and the attribute of the model containing the foreign key. [9; 21.]

```
// api/models/User.js
{
  attributes: {
    name: { type: 'string', required: true, },
    age: { type: 'number', defaultsTo: 18 },
    pet: {
      collection:'pet',
      via: 'owner'
    }
  },
}
```

Listing 4. An example of a model definition used in Sails framework.

Waterline has its own query language that is used to transform the queries into the native query language of the target database. This allows the queries to be written in the same way regardless of the database type the target model is stored in. [22.]

```
const results = await Model.find({
  name: 'mary',
  age: 50
});
```

Figure 1. An example of a Waterline query.

Sails uses controller modules to define how requests are handled. Request and response objects are available in the controller actions, and those can be used to extract data from the request and modify the response by assigning it a payload. Listing 5 shows an example of a simple Sails controller. Sails contains its own mechanism for routing, and it will generate automatically routes when new controllers are added. [9; 23.]

```
// file /api/controllers/HelloController.js
module.exports = {
  // automatically generated route will equal to /hello/world
  world: function (req, res) {
    return res.send("Hello world!");
  }
}
```

Listing 5. A simple Sails controller module with one action.

Sails uses its own special type of middleware called policies. Policies are written in separate modules and a policy configuration file is used to define which policies are executed before each route. The common use case for policies is to restrict the access certain routes from unauthenticated users. [24.]

2.5.2 Koa framework

Koa is backend framework based on Express developed by the same people, and it uses JavaScript generators to define middleware functions [9; 25]. Koa is minimalistic framework that is essentially a middleware-based HTTP (hypertext transfer protocol) server library. Generator syntax allows the code to be written in almost synchronous style which makes writing callback dependent middleware simpler and the code more readable. Suggested use cases for Koa include lightweight web applications and HTTP APIs. [9.]

Setting up a Koa project is simple and only requires the Koa module to be installed [9]. Koa application is essentially just a special object that contains an array of middleware functions that will be executed upon receiving a request in a stack-like manner. Listing 6 illustrates how a Koa application with a simple middleware is defined and started. Koa does contain methods for some common tasks like redirections, but it does not bundle any middleware within its core, so to add more complex functionality to a Koa application, like an API to define and respond to different kinds of HTTP requests, additional middleware adding modules need to be installed. [9, 25.] Typical workflow with Koa application requires planning the middleware needed in the application beforehand and researching the publicly available modules. The developers maintain a list of compatible third-party modules in documentation of the framework. [9.]

```
const Koa = require('koa');
const app = new Koa();

app.use(async ctx => {
  ctx.body = 'Hello World';
});

app.listen(3000);
```

Listing 6. Simplistic Koa application with one middleware [25].

Koa's approach to middleware utilizes asynchronous functions to run middleware in two parts, and the control will flow downstream and then back upstream. When a middleware

function invokes next(), it will suspend and yield control to the next middleware function registered with the app. Once there is no more middleware to run downstream, the middleware stack will unwind, and each middleware will execute its upstream part. [25.]

Koa uses so called context to encapsulate node's request and response objects. Context also implements the essential methods used in HTTP server development, so that common server functionality does not need to be implemented using middleware. A context is created every time a request is received and is referenced usually with ctx identifier in middleware. Context can be extended by adding additional properties to the app.context prototype that is used for creating new context objects. For convenience, many methods of the context will simply delegate the task to the request or response objects it encapsulates, and the methods are otherwise identical. [25.]

Koa-router is popular third-party routing library for Koa. Koa-router has a chainable API, and like Express, it is based on HTTP verbs. Listing 7 demonstrates how to set up a simple Koa application with koa-router enabled. [9.]

```
var Koa = require('koa');
var Router = require('koa-router');

var app = new Koa();
var router = new Router();

router.get('/', (ctx, next) => {
  // ctx.router available
});

app
  .use(router.routes())
  .use(router.allowedMethods());
```

Listing 7. Setting up a simple Koa application with koa-router [26].

Originally the major strength of Koa was it utilizing the generator syntax that came with ECMAScript 2015, but today the newer features of the language standard are more widely used by the node community. Current advantages of Koa include its streamlined nature, and the availability of high quality third-party modules. From a developer's perspective Koa also has an efficient syntax and it is highly flexible and can be configured to match projects with specific requirements. On the other hand, the high level of configurability may also alienate some developers and produce less reusable code. [9.]

2.6 Mocha testing framework

Mocha is an open source testing framework that can run on both Node platform and browser, and it provides the base for running tests. The framework does not include its own assertion tools, and it works with any third-party assertion library or Node's built-in assertion module. The framework supports testing of asynchronous code by providing both callback and promise based test bases. [27.]

Mocha tests consist of two kinds of code blocks. The so called describe blocks are used to wrap the smaller "it" blocks that contain the runnable test code and related assertions. Describe blocks can be nested to group test blocks. Listing 8 shows a simple test written with mocha and node's assertion module. [27.]

```
var assert = require('assert');
describe('Array', function() {
  describe('#indexOf()', function() {
    it('should return -1 when the value is not present', function() {
      assert.equal([1,2,3].indexOf(4), -1);
    });
  });
});
```

Listing 8. An example of a mocha test [27].

In addition to test blocks, Mocha provides hooks that can be used to run code before or after tests. Hooks are designed for setting up preconditions for tests and cleaning up after tests. [27.]

3 System design

3.1 Overview of the application

The application is an e-form based survey management tool, for managing, running and analysing results of surveys through a web client. The application is designed specifically for the surveys of the self-evaluation system used in project QAEMP, but it can be used to run practically any kinds of surveys.

The flow of the application is such that users are expected to login or register to the application in order to gain access to its features. In addition to managing and answering

surveys, users are able to manage groups and evaluations that are used in specifying a target group for a survey. The application offers a graphical representation of the results of survey, and additionally the results can be downloaded as a separate file.

Each user of the application is attached to a group or groups that represent the user's background, like for example whether a user is a student or a teacher. Answers to a survey are grouped based on the answerers' groups, so that the answers immediately reflect the discrepancies between each group's answers. The target group of a survey is defined by so called evaluation group that consists of multiple user groups. Evaluation groups can be used multiple times, making it simple to run multiple surveys with the same target group.

To make recreating a survey simple, surveys can be saved as templates for future use. When creating a new survey, user can select any existing template as foundation for the survey so that all the questions of the template will be immediately available in the new survey. All the questions attached to templates are also separately available for importing when editing an existing survey.

Answering to a survey can be done in two ways. Users that have registered to the application can see and access all ongoing surveys matching their groups. Surveys can also be distributed by sharing a survey specific link that can be used to quickly answer the survey without registering.

3.2 Project QAEMP

The goal of the project QAEMP is to explore a practical approach to quality assurance through continuous self-evaluation and cross sparring. The self-evaluation system used in the project is based on a specific set of questions and the statistics gained by comparing the answers of different answerer groups.

The questions of the self-evaluation system are multidimensional, meaning that the questions are evaluated using multiple sub questions that are also called criteria. A typical question of the system would be to evaluate the urgency and importance of an issue. In this case the issue is the question, and the urgency and importance are criteria used to evaluate the issue.

3.3 Initial technologies

Node was selected as the platform for the application, because everyone in the developer team had previously worked with it, and it was suited for the goal of building a modern scalable web application with a large number of third-party modules available.

It was decided early on that a framework was going to be used as a foundation for the application to speed up the development, because time and manpower for the project were limited. To maximize the development speed, Sails was selected as the main framework for the application, because it provided a set of clear guidelines for development, and all the functionality needed to start designing the actual application logic.

3.4 Changing the stack

Later in the development it became clear that Sails and the Waterline library bundled with it were the cause for the major problems the team was dealing with. Particularly the major feature of the library allowing models to be saved in different databases didn't work with way the models of the application were designed. Later in the development, the feature was fixed by an update, but the same update in turn introduced new issues to existing features. A lot of time was used to create workarounds for the Waterline related issues, and in the end, it was decided that the initial product would be finished using the original stack and Sails, but once delivered to the client, another version of the application would be developed with new frameworks for the backend implementations and database operations. The initial version became a test run for the concept, and any features needing to be reworked would get an overhaul in the next version which would replace the original one in production once finished. After delivering the initial version by the deadline, the schedule to develop the next one was more open-ended.

For the final version Koa was selected as the backend framework, because Koa was highly configurable and didn't enforce any strict conventions like Sails that had previously added a limiting effect to the development. Adopting Koa meant redesigning the entire server component, but this way it could be customized for the application's needs.

No framework or library was adopted to replace Waterline, because it was decided to maintain the existing database solution, and there wasn't replacement that would have offered the same kind of support for utilizing different kinds of databases at the same

time. Instead of database library, it was decided that Waterline like functionality would be built by ourselves on top of plain database drivers.

Due to the high amount of functionality in the front end, React was adopted to the project. The goal with React was to simplify and increase the maintainability of views that had large amount of scripting attached. The idea of adding SPA (single page application) like functionality to application was considered but dropped in favour of preserving the existing backend logic.

The major task in changing the backend solution was to design the server component and the model system, which were this time not provided directly by the frameworks. Once the foundation was implemented, changing the old code base to use the new system was fairly straightforward, and for example most of the controller actions only needed to be converted from using generators to use async functions.

3.5 Database solution

The application uses both MariaDB and MongoDB databases to store data. The database solution was designed utilizing the features of Waterline database library that was bundled with the Sails framework used to implement the initial version of the application. The model structure of application is relational, so a relational database was selected as the foundation for the application. However, in the modelling phase it was discovered that especially the Answer model would benefit from being stored to a document-based database, and MongoDB was adopted to the project.

Answer model was going to be a target for complex queries, and the aggregation framework of MongoDB was a fitting tool for the purpose. At this point it was also clear that some of the fields in Answer model were going to be stored as JSON. Storing data in JSON format to a relational database is possible, but it not supported to run complex queries based on fields of that type, which is another reason why MongoDB was selected as the database for the model. Modelling wise document-based format also allowed the model to remain more open-ended and expandable.

3.6 Modeling

The modeling started by defining the minimal components needed to create a tool for survey management. This included the models for containing information about the users, surveys, questions, and answers. The system was extended by adding the group and evaluation models to fulfill the requirement for more detailed result analysis, and a surveySession model to combine the answers of a single user. The models and their respective relations can be seen image 4.

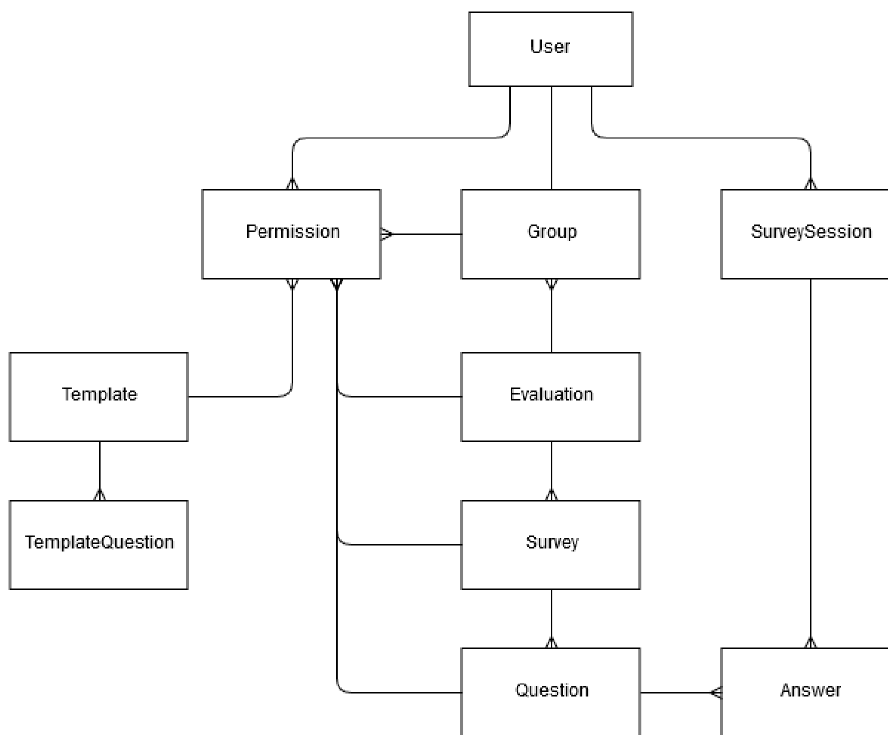

Image 4. The models of the application and their respective relations.

Originally the relation between user and a group was immutable, so once chosen, users were not able to change their respective group. This was later found to be too restrictive, and convention was loosened up so that users could swap their group between the ones they had access through the permission table.

The evaluation model is what can be considered the target group of a survey, and it contains one or more groups. The evaluation attached to a survey specifies which groups are able to participate in the survey. The evaluation model was added so that target groups could be easily shared between surveys and recreating a survey with the same target group was simple.

The major question in designing the question model was how to implement the support for multiple sub-questions. It was evaluated whether the question criteria should have its own model, but to keep the questions self-contained it was decided to implement the criteria as a JSON field of the question model. The criteria field is expected to be an array of criteria objects, and a particular structure that can be seen in listing 9, was designed for the criteria objects.

```
{
  "title": "Importance",
  "description": "",
  "type": "scale",
  "options": { "min":0,"max":5 }
}
```

Listing 9. An example of a criteria object.

The keys “type” and “options” of a criteria object are used to define what kind of question the criteria represent. The two criteria types currently supported are linear scale and multiple choice. In the client side, the criteria type defines the type of input element rendered for the question, and the options contain the information used to customize the input. For a scale type, the options contain the minimum and maximum values for the input, and for a select type, the options contain the selectable choices, and their respective values. These criteria types were selected, because the questions of the client’s evaluation system can be expressed using these two types. The criteria structure was something that we wanted to leave as flexible as possible, so that additional types of criteria could be easily added later on.

3.6.1 Ownership of records

In the beginning, to restrict access to the instances of models that users can interact with, these models contained an owner attribute defining the one user allowed to modify a specific record. The owner attribute was commonly set when a record of a managed model, like a survey or a group, was created. But sense the application was originally designed to be used on organizational level, it meant that multiple users would need to access the same records. The owner attribute was removed in favor of a new Permission model used to create ownership defining relations between the user model and any of managed models. Permission model contains the ids of the user and the target instance, and name of the target model. With the Permission model, multiple users can have access to the same managed record. The Permission model was further extended by adding a new attribute defining the allowed operations. With predetermined values for each

type of permission, the attribute is used to define whether a user has the permissions to view, edit or delete a specific record.

The tradeoff that comes with the high customizability is that checking the ownership of managed record always involves an additional database query. This also means that retrieving records from the database based on permissions requires the queries going through an extra table. Creating new records also becomes more complicated as it must be guaranteed that no instance of a managed model gets saved without also saving at least one permission record pointing to it. Preventing the database from ending up in an inconsistent state was achieved by wrapping the two insertion operations inside a transaction, meaning that if either of the operations fails, the database will be reverted back to its original state.

3.6.2 Templates

To address the requirement for being able to quickly recreate surveys, two additional models were implemented. The idea is that the new models work as counterparts for the survey and question models, so that a survey can be alternatively saved with its questions as a template and template questions. In the application's current state, templates are considered as public property, and anyone creating a survey has the option to choose an existing template to use as the foundation for the new survey.

3.6.3 Survey sharing

For easier distribution of surveys, a system for sharing them was developed. The system is based on a new link model built around the idea of a unique key used to access a particular survey. The model is also used to specify which group of the related survey's evaluation the users accessing the survey this way will represent. Alternatively the decision can also be left for the users. A survey can have multiple link objects for different groups at the same time, so that the distribution can be customized to a greater detail.

4 Technical implementation

4.1 Sails stack

The models are implemented using the standard model system of Sails. The system only allows for creating static methods for the models, because instances of a model are plain JSON objects. Due to these limitations, very little logic was placed inside the models, and consequently the application logic started to stack up in the controllers. The primary function for a model is to define schema for the data encapsulated by the model.

The controller actions were implemented originally using callback based structures, but the more complex actions began quickly to suffer from impractical structuring and low readability due to the large amounts of callbacks caused by asynchronous operations. To reduce the number of callbacks, generators were taken into use.

To simplify the usage of generators, a utility function for fully executing a generator was created. The function is based on the concept that every yield statement of the provided generator includes a promise that resolves to the target value or rejects with an error. The so called generate function is provided with a generator, and when invoked, it will return a new function. The returned function will in turn return a promise resolving to the last operation of the generator that is usually used to gather all the results accumulated inside the generator. This makes it possible to place all the asynchronous operations inside a generator and reduce the overall number of callbacks. Listing 10 illustrates a controller action utilizing the generate function to handle two asynchronous operations without resorting to multiple callbacks.

```
let findAnswers = utils.generate(function* () {
  let answers = yield Answer.find({ surveyId: surveyId });
  let sessions = yield SurveySession.find({ survey: surveyId });
  return Promise.resolve({ answers, sessions });
});

findAnswers().then(results => res.status(200)
  .json({ answers: results.answers, sessions: results.sessions }))
  .catch(err => res.serverError(err));
```

Listing 10. An example of a controller action utilizing a JavaScript generator to handle asynchronous operations.

The structure also provides more ways to handle errors, as any error can be caught and handled inside the generator if needed. If an error is not handled inside the generator,

the handling is done by the outer catch handler, but this will also crash the generator. Compared to the callback-based structure, this is more flexible, as a callback-based chain of asynchronous operations will always get broken in case of even one operation resulting in a failure.

4.1.1 Policies

The policy system of Sails was used to create rules for controlling which routes users could access. The top-level policy checks whether the request is made by an authenticated user, and the policy is used with all the routes except for the ones used for registering and logging in to the application. The policy will redirect the request the login action in case the request doesn't contain a valid session. Unlike the top-level policy, the rest of the policies will throw an error if the policy does not grant the user access to the route it's protecting. The policies were designed to throw, because if a policy fails to grant access, it means that the application has ended up in a failure state as the user interface should never allow users to access routes they have no permission to access.

Because the routes were designed to contain the model name, record id and the type of action whenever possible, it was possible to create policies that test the route with regular expression and check the related permission if the route contains these three parameters. For example, a request made to route /survey/5/edit, will trigger the canEdit policy that verifies that the current user has an edit permission to the survey record with id 5. The policies checking view, edit and delete permissions could be attached to all the routes, because the policies themselves are able to determine from the route if they need to fully execute, and if not, they can simply yield the control to the next policy. In addition to checking the permission, a common task for the policies is to verify that the record the request is related to actually exist in the database.

Route specific policies were written for actions that need special validation and not just a permission check. These routes include answering a survey and accessing survey results. The logic in these policies is not reusable and could have been implemented as part of the related controller actions, but it was decided that the all access control related logic should be kept in policy modules.

4.2 Koa stack

4.2.1 Server module

The Koa version of the application is built around a server module that is used to set up and launch the application. The launching happens in multiple faces. First a new Koa application is created, and a reference is stored to a variable, and then all the middleware functions, both third-party and self-written, are registered with the application. The ordering in which the middleware are registered is important, because the middleware stack will get executed in the registering order and some of the functions are dependent of each other, so it's important that the core features like request parsing and sessions are enabled before trying to access them in another middleware. After registering the middleware, controllers are loaded using a custom module explained in more detail in the loading modules section, and in the end the Koa application is used the start a HTTP server. At this point the application is fully running and listening for requests.

Many middleware functions can be configured by passing them parameters, and a configuration module was created for storing middleware and application related settings. Many configuration variables like database authentication related information are stored as environment variables of the system, and the configuration module can be used to easily access them inside the application. The most important environment variable defines what environment the application is being run on, and it will customize the behaviour of the application, and the way it is being set up. For example to better suit development, certain middleware are provided with different set of configurations, and some middleware are entirely disabled.

4.2.2 Middleware

The application uses multiple third-party middleware functions to establish core functionality for the application. Essential middleware application include for example setting up session support and request body parsing. Koa-session is middleware providing support for cookie-based sessions that can be used to store information. Because sessions retain data between requests, they are used in the authentication system of the application. Once a user has been successfully authenticated, the user information is saved to a cookie with a limited lifespan, so that the user can be identified with every request. Koa-

body is a body parser middleware used to make the request body and its contents available in the context as an object, simplifying the process of working with the data attached to a request.

In addition to the body part, a request can also transfer data via the URL, and additionally the format of the received data varies depending on how the data was sent, which is why an additional middleware was written for request parsing to supplement the body parser. The middleware aggregates all the available parameters of a request to a single object and exposes it to the context object. This way the all request parsing is done early on in the upstream of the middleware stack.

Koa-csrf is middleware used to protect the application from cross-site request forgery. The middleware generates a unique token that is used to verify that a request actually originated from an authenticated user. If no valid token is found, the request is automatically rejected. The middleware works so that a token is attached to context object of every GET type request, and every POST type request is expected to contain a valid token as a part of its payload. Actions returning a view will have the token saved somewhere in the view, usually as a meta element or as an input as part of a form element. Forms will automatically send the token in the submit process, and any AJAX request sent by the frontend scripting will have token separately attached to the payload by the script.

So called responder middleware was designed to control the flow of the request handling, and it is responsible for initializing every new request and response in addition to managing the error handling of the application. The middleware’s upstream component is used to setup the context object upon receiving a new request by binding additional functions used in request handling to the context object. The received request is further analysed, and in certain scenarios the middleware can simply terminate the processing of the request and bypass the rest of the middleware stack by immediately redirecting the request. The request analysis is used to handle for example the base URL of the application. If the request contains a valid session, the request is redirected to a profile action and otherwise to the login action. Finally the responder verifies that the request points to a valid route or resource and yields the control the next middleware.

Originally the request analysis was also used to explore the possibility of matching the requested URL against static view files of the application, so that in case of a match, the responder could instantly return the view and consider the request handled. The system

would have allowed for an alternative way of designing the routing of the application as well as building views by requesting and combining partial view components. This functionality was however dropped in favour of maintaining a more traditional MVC style approach to the backend. Rendering plain views would have required major modifications to the backend logic, because the template engine used to render all of the application's views expects to receive the data used to populate the view at the time of rendering. A solution would have been either to have the needed data stored to a session beforehand or redesigning the views so that the frontend scripts would populate the view with data after the initial rendering.

The downstream component of the responder is used to setup the response. If the context object is flagged for returning a view, the responder will render the requested view using the provided data. If the response is returning only data, the data will be transformed from JSON to a string format for sending. The data will also go through sanitation process that will iterate all the data objects and delete fields with predefined keys containing sensitive data, like for example the password field of a user object.

The responder also contains the outermost try catch block that will catch any unhandled errors occurring during the response handling. To keep the application logic simple the approach taken to the error handling is to let the application crash in most cases instead of trying to return its state back to normal and focus on reporting the error. In a case of an unhandled error, the catch clause seen in listing 11 only responds with an appropriate view explaining the error. If the error is a generic one, a standard error view is returned based on the current environment the application is running on. If the environment is set to production, a more cleaned up version of the view is retuned.

```
} catch (err) {
  Logger.error(err.stack);
  ctx.status = err.status ? err.status : 500;

  //Handle any special errors here
  if (err instanceof SurveyNotActiveError) {
    ctx.view('error-surveynotactive.ejs', { survey: err.survey, moment });
  }
  else responder.view(config.env === 'production' ?
          'error-production.ejs' : 'error.ejs', { error: err });

  responder.render(ctx);
}
```

Listing 11. Catch clause logic of the responder middleware.

To customize the error handling process, separate error types, like the one in listing 12, were created. The errors are accessible anywhere in the application and can be thrown in special cases instead of a generic error. Custom error classes are used to store additional information about the cause of the error, and the responder is set up so that it can handle certain error types differently, so that for example a more detailed error view can be returned. Another reason for adding custom error types was to store status codes and use the errors to attach the appropriate code to the response in the catch clause, instead of always resorting to internal server error.

```
class NotFoundError extends Error {
  constructor(msg, model, query){
    super(msg);
    this.model = model;
    this.query = query;
    this.status = 404;
  }
}
```

Listing 12. Defining a custom error by extending the base error class.

Koa doesn’t contain a way to directly serve static files, so koa-static, a file serving middleware, was used to expose a project folder containing the frontend assets. The management of assets was implemented using Webpack middleware. Webpack is a module bundler that was used for code optimization and dependency management for frontend assets like scripts and stylesheets. The middleware is also used to convert the contents of JavaScript files to a format that is supported by current browser versions, making it possible to use the latest syntax of the language, like async functions, even if the syntax isn’t yet widely supported by browsers.

Due the application being multi paged, we used Webpack to build view-based script bundles that contain all the code needed in the scripting of a given view. Listing 13 shows how the script bundle for profile view is constructed by defining a new entry point for the profile.js file that is used to build a dependency graph starting from the specified file based on require statements, and finally all the dependencies are outputted to a single bundle file.

```
const webpackConfig = {
  entry: {
    'js/profile': path.join(config.appRoot, 'client', 'scripts', 'pages',
          'profile.js')
  }
}
```

Listing 13. Defining an entry point in Webpack configuration.

The resulting bundle is saved to /client/dist asset folder as defined in the configuration as seen in listing 14. After the file extension is added to the file name, the bundle is accessible at /js/profile.js. With bundles, every view only requires one script tag to load all the scripts effectively simplifying the script management and preventing errors caused by importing scripts in an incorrect order.

```
output: {
  path: path.join(config.appRoot, 'client', 'dist'),
  publicPath: '/',
  filename: '[name].js'
}
```

Listing 14. Output configuration for Webpack.

In addition to bundling scripts, Webpack was also used to manage stylesheets. To avoid having to use multiple imports, the middleware was used to create a single bundle containing all the stylesheets of the application, including the ones provided by vendors. Because stylesheets are not dependent on each other the way scripts are, the dependencies had to be artificially created for the Webpack to be able to construct a bundle. Dependencies were defined in an index file consisting of import statements, and the file was given as the top-level entry point to Webpack.

Because Webpack is by default only capable of processing JavaScript, additional loaders had to be used for handing stylesheets. Listing 15 illustrates the configuration used to test files using a regular expression to define if they are stylesheets, and the process of using a loader to convert all stylesheets to the CSS format that is the standard stylesheet language. The folders containing stylesheets provided by vendors are defined separately in the configuration, because there was no reason to expose the whole vendor folder to the style loader. Due to the conversion to CSS, it was possible to use a CSS extension language SASS for more effective styling. Assets in formats that are not supported by Webpack or its standard loaders, like images and fonts, are simply saved to the exposed asset folder that makes them accessible to the frontend.

```
{
  test: /\.s?css$/, //.css, .scss
  use: ["style-loader", "css-loader", {
    loader: "sass-loader", // compiles Sass to CSS
    options: {
      outputStyle: 'compressed',
      ext: 'css',
      includePaths: [
        path.join(config.appRoot, 'node_modules', 'motion-ui', 'src'),
        path.join(config.appRoot, 'node_modules', 'flatpickr', 'dist')
      ]
    }
```

```
  }
}
```

Listing 15. Setting up a style loader with Webpack.

Authenticator is a module containing multiple middleware functions for managing access to the application's routes. The module is not loaded during the starting of the application, and instead the middleware functions are separately attached to specific routes inside the controllers. The middleware are based on the policies of the previous version, and offer the same kind of functionality.

4.2.3 Controllers

To ensure more maintainable and clear code structure, the controller related code was split into separate controller modules with each module loosely encapsulating the actions related to a single data model of the application. For example, the user controller contains all the actions related to the user model, such as signing up a new user. Controllers are implemented as static classes and controller actions are async class methods with the standard middleware function signature. Despite the controller actions being middleware functions, they may only contain an upstream component, because actions are the last functions to execute in the middleware stack.

Each controller has its own koa-router object and the getter method implemented by every controller is used to bind all the actions to the object and return it. An example of a router method with two actions is illustrated in listing 16. Because the controller class is static, initializing the router object happens when the router method is invoked for the first time. Every router is given a prefix according to the model it relates to in order to create a sensible URL scheme. The example router is from user controller, so the prefix for the route in this case is "user". Each attached action is given the request method it responds to, a name, a route and the middleware functions to run. In this example the router of the first action is provided only with the main middleware function, but the second action also utilizes a middleware function of the Authenticator class to verify that the current user is authenticated. The first action of the example also shows the syntax for defining named route parameters as part of the route definition.

```
static get router(){
  if(router.stack.length > 0) return router;

  router.prefix('/user');
```

```
  router.get('changelocalization', '/locale/change/:locale',
    this.changeLocale);
  router.get('logout', '/logout', Authenticator.login, this.logout);

  return router;
}
```

Listing 16. An example of a controller's router method with two action.

User controller's action for changing localization is seen in listing 17. The action is a simple one but displays the common characteristics of an action. Almost every action is dependent on utilizing some information passed along with the request. In this action the locale value is extracted from the context's params property. Because the property contains all the data of the request, the locale value could have been passed as a named route parameter, a part of form data or a JSON payload, and it would still be available in the same way. Usually actions write or read some data either from the database or the cookie-based session as a part of their request handling logic. In this case the new locale value is written to the session. Finally, every action invokes a property of the context to define the response. In this example the response is set to redirect back to the route where the request originated from. Other common options are to setup the response to return some data or a view.

```
static async changeLocale(ctx, next) {
  const locale = ctx.params.locale;
  if (!Translator.supportedLocales.includes(locale))
    throw new Error(`'${locale}' is not a supported locale!`);

  ctx.session.locale = locale;

  return ctx.redirect('back', '/');
}
```

Listing 17. A middleware function used as a controller action.

The application is multi paged, but it also contains functionality similar to a SPA (single page application), and this is reflected on the controllers. Technically the controllers have three kinds of actions; the ones returning a view or data, and the ones accepting data. The simplest type of actions simply renders and return a view, and these actions usually consist of running some database queries to retrieve the data needed in rendering the view. Views are rendered using the view function attached to the context by the responder middleware. The view function takes the path to the view file and an object containing the data used in the rendering as parameters, as demonstrated in listing 18. In addition to any separately passed data, the view function will always append the context to the data object being passed to the view.

```
return ctx.view('/user/profile.ejs', { user: userObject });
```

Listing 18. Return statement of a controller action returning a view.

The object passed to view is accessible in the view and is used to populate the view with information. The view function is designed to lookup the file extension of the given view file and use an appropriate template engine to render it. Though the application is setup to support a couple of different template engines, EJS is the one used with all the default views. EJS files use the standard HTML syntax, but a set of special tags can be used to embed JavaScript snippets in to the document. When the file is rendered by the server, the JavaScript snippets will be executed to construct the final view. The snippets are primarily used to inject data to the HTML document using the object passed to the view, or to manipulate the structure of the document with control structures by hiding a certain element if a condition is not met or by looping over an array to create multiple HTML elements based on the contents. Listing 19 demonstrates how JavaScript snippets can be used to manipulate the document based on whether the session of the given context contains a user object.

```
<% if (ctx.session.user) { %>
  <p>Hello <%= ctx.session.user.username %></p>
<% } else { %>
  <p>No one is logged in</p>
<% } %>
```

Listing 19. An example of using EJS to manipulate a HTML document.

Actions returning data are similar to the ones returning a view, and the logic of these actions is also about gathering the needed data and appending the data to the response. The only thing needed to do with data before sending is to convert it the string format, which is automatically handled by the responder middleware. These actions are exclusively accessed by the frontend scripts to fill or update some data needed in the view.

The actions accepting data are implemented in two ways depending on how the data is being sent. Actions designed for handling AJAX requests usually validate and process the submitted data and send a confirmation that the data was valid and received successfully. These actions are fairly simple, unless the data requires complicated processing. Due to the large amount functionality implemented in frontend and the dynamic nature of the used data, AJAX requests are generally the preferred way of transferring data to the backend. The alternative way is to use the HTML form component to submit the data. The data is accessible in the same way as with AJAX requests, but in this case the action is expected to return a view. This way relies on a standard feature of the

HTML, and is quick to setup, but adding features like handing a failed submission without losing the data, requires writing additional logic to both the action and the returned view. Despite the drawback, the method is suited for submitting data with simple structure, like the login information that consists of just two fields; username and password.

4.2.4 Loading modules

Controllers, models and other modules that exist in bulks with a common namespace, are required using so called requireNamespace function. The function was created to require multiple modules at once, and to avoid the Node platform's circular dependency problem caused by multiple modules requiring each other. The function is provided with the folder name containing the modules and the wanted namespace, defined by the export statement of a module. For example, with all the controller modules, the controller class is exported with the namespace "controller" using statement seen in listing 20.

```
Class AnswerController { … }

exports.controller = AnswerController;
```

Listing 20. Exporting a controller class.

Using the logic seen in listing 21, the function will generate a collection of modules contained in the module directory. The walk function is a generator that uses node's file operations to iterate all the files of the given directory and any subdirectories, while yielding the paths to these files. The generator is iterated to require modules, and all the modules matching the given namespace are stored to an object. Finally, the object containing all the required modules is used to extract and return the all contents with the given namespace.

```
const iterator = walk(directory);
let result = iterator.next();
while (!result.done) {
  if(result.value.endsWith('.ts')) {
    result = iterator.next();
    continue;
  }
  let module = require(result.value);
  if (module [namespace]) {
    //store potentially incomplete reference containing the namespace here
    modules[path.basename(result.value, '.js')] = module;
  }
  result = iterator.next();
}

const moduleNames = Object.keys(modules);
for (let moduleName of moduleNames) {
```

```
  modules[moduleName] = modules[moduleName][namespace];
}
return modules;
```

Listing 21. Requiring all the modules with a given namespace using a generator function to lookup file paths.

The requireNamespace function is used in conjunction with an index file used to load all the modules stored in the same directory. If provided with path to a folder, Node's require statement will always look for an index.js file. In addition to simply require multiple modules at once, Index files can be used to customize the way the modules are loaded.

For example, each controller module has its own router, and every controller must be separately attached to the Koa application, so to guarantee that controllers are loaded correctly in the main server module, an index file seen in listing 22 was created.

```
const controllers = Utils.requireNamespace('controllers', 'controller');
exports.load = function(app){
  if(!(app instanceof Koa)) throw new Error();
  for (let [name, controller] of Utils.iterateObject(controllers)){
    app.use(controller.router.routes());
    app.use(controller.router.allowedMethods());
  }
}
```

Listing 22. The index file for loading controller modules.

The particular index file used to require controllers doesn't actually return the controllers themselves, but a load function that must be provided with a reference to the main Koa application, so that all the controllers can be loaded and correctly attached to the Koa application.

4.2.5 Model implementation

The goal for the model implementation was to create a set of features similar to those of Waterline library bundled with Sails in order to reduce the overall changes needed in the code base during the changing of the framework stack. Also, many features of the library, like the querying system, were powerful tools in general, and it was desired to have something similar in the new version. The new model system was implemented using the class system of JavaScript, and by making the classes globally available in the application. The system is built around inheritance and the class hierarchy can be seen image 5. The Model class is used as an interface to define stump implementation for all

the required model methods. The MariaModel and MongoModel classes are used to attach database methods to the lower level classes. ManagedModel class implements the methods for tying a class to the permission system of the application. The classes in the bottom of the hierarchy represent the actual models.

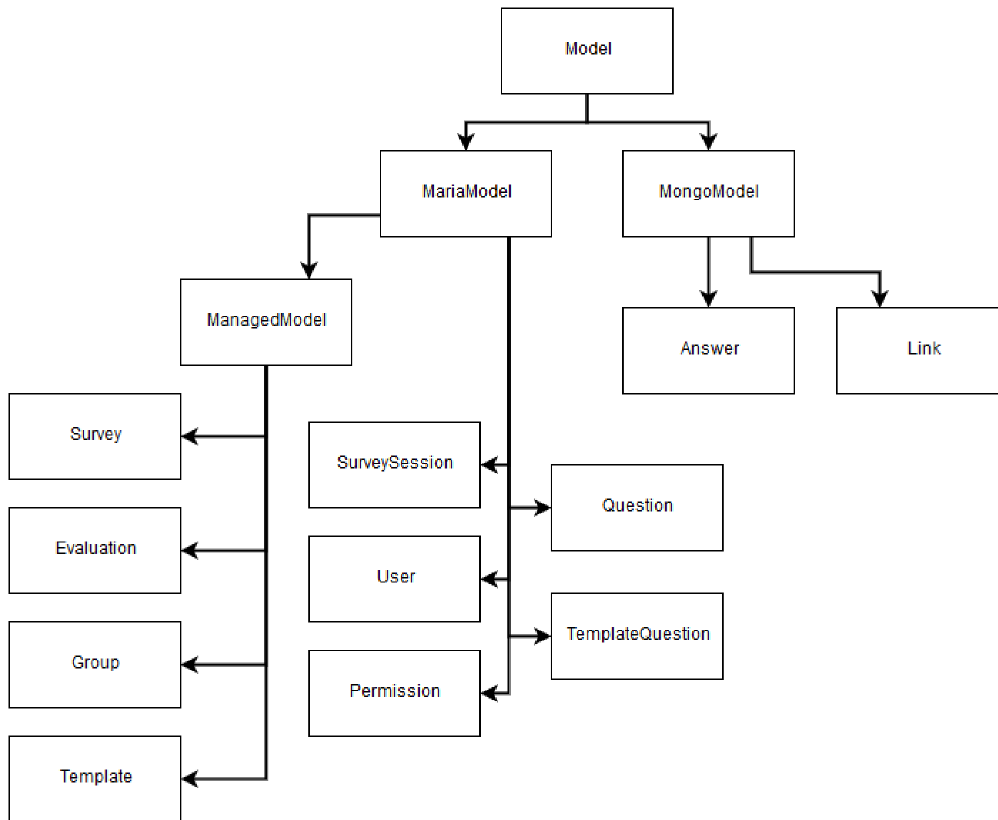

Image 5. The class hierarchy.

The MariaModel and MongoModel classes implement the static database methods, so that all the lower level classes have access to the database operations through inheritance. Due to inheritance, the lower level classes only need to override the variable defining the database table or collection containing the objects of this type for the database methods to work. The query operation seen in listing 23 will target the user table with provided query, because the find method of MariaModel is invoked through the User class.

```
await User.find({ username: "test" });
```

Listing 23. Invoking the static find method of MariaModel through the User class.

Because of the open-ended nature of JavaScript classes, running any insertion operations against the Maria database requires ensuring that the objects are in a format that

matches the table structure. For this purpose, every class is required to implement serialize method for extracting all the fields specified the table structure from an object. Though it's not required, the same convention is used with the classes stored in the MongoDB to force a structure for the collections. When data is queried from the database the data is deserialized from rows and documents into instances of the classes. An important part of the deserialization is also to process certain attributes. For example, all the JSON fields stored to the Maria database are considered plain strings by default, and those need to be parsed, and likewise the date strings are turned into proper Date objects for easier handling.

All the database operations are managed by the database classes that were created to wrap the database drivers, and to define the methods for the database operations. Picture 6 shows the hierarchy of the database classes. The base Database class is used like an interface to define the mandatory database actions like selecting, updating and deleting records. The actual database classes implement the methods specified by the base class, and any additional methods for exposing database specific features, like transactions. The database classes contain a pool of connections that is used to check whether a new connection needs to be created or if there already is one available when a connection is requested to run a database operation. The methods of both database classes accept queries in JSON format resembling the basic queries of Waterline library.

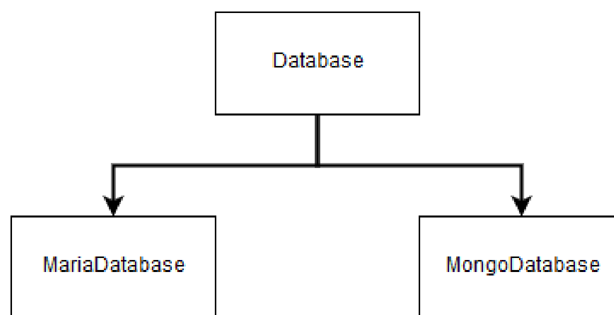

Image 6. The hierarchy of the database classes.

The implementation of the MongoDatabase class wrapping the MongoDB driver was straightforward, because the driver uses JSON based queries by default, and very little processing is needed to handle and validate the queries. In addition to the database operations, the class exposes the aggregation feature of the MongoDB database.

The design of the MariaDatabase class is more complex and it required creating an inner SQLQuery class for expressing queries. The inner class contains the tools for constructing queries and converting them to the SQL format. The inner class was designed to be flexible, and queries can be defined using both SQL and JSON formats, but to create more complex queries, SQL needs to be used. The inner class contains chainable methods for constructing complex queries, and an example of the query creation can be seen in listing 24.

```
const query = new SQLQuery("user");
query.select("SELECT * FROM user");
  .where({ id: 1 })
  .or()
  .where("user.group IN (1, 2, 3)");
```

Listing 24. Building a query with the SQLQuery class.

When a query is ready to be executed, the state of query object is converted to a line of SQL and passed to the database driver. Listing 25 shows the select method of the MariaDatabase class that utilizes the inner class to construct a query, and then passes the query object to the execute function that validates the query, converts it to SQL and further passes it the database driver.

```
async select(table, where, ...params) {
  const query = new SQLQuery(table).where(where).values(...params);
  const [results] = await this.execute(query);
  return results[0];
}
```

Listing 25. Select method of the MariaDatabase class used to run select type queries.

Extending the ManagedModel will attach any class to the permission system of the application. The save method of the class guarantees that any new instance of the class is also saved with at least one permission object, or else the method will throw an error. The save method also uses a transaction to wrap all the database operations. In addition to the special save method, the class provides methods for managing permissions by easily adding and removing permission objects from a managed object. Listing 26 shows how a user is granted view and edit permissions to a new evaluation object.

```
let evaluation = new Evaluation()
await evaluation.allow(ctx.session.user, "view", "edit").save();
```

Listing 26. Adding permissions to a new Evaluation object.

Models loaded using an index file that makes the classes globally available in the application by requiring each of the classes and injecting them to the global object of Node. This way the models need to be required only once in the server module when starting the application. A global flag is also set up during the loading, so that trying to load the models for a second time will result in an error.

4.3 Exporting result data

One requirement for the application was the feature to export result data. The requirement for the export format was that it needed to be supported by Microsoft Excel program. In favor of customization, the default Excel workbook format XLSX was selected over CSV (comma-separated values) as the export format. The approach to expressing the results was to create a single row per answering session, so that all the answers of a session would be clearly grouped along with the name of the group the answerer represents.

The feature was implemented as a static method of the Answer class utilizing exceljs library for managing Excel worksheets. The method is provided with the id of the survey containing the desired results. First the id is used to retrieve the survey and all the related objects. As a preliminary action all the answer and question objects are processed to guarantee that all the fields of type JSON are valid and parsed to the correct format. Constructing the worksheet begins by defining the columns. Each column consists of a header that will be visible in the resulting file, and a unique key that is used to insert data to the column. Every row contains a column for group, and rest of the columns are created by iterating the questions. Every question criteria will have its own column, and each question will have a column for evidence. Unique column keys are created by combining the question id and criteria title or column type. If the user info field of the survey contains additional questions, extra columns will be added accordingly.

An overview of filling the worksheet can be seen in listing 27. The filling happens by iterating the answering sessions with each session having its own row. With a session selected, a new iteration over the predefined columns is started. The type of the column indicated by its key is used to extract the relevant data from the session or its answers and append the data to the new row. In case the required data is not available, e.g. a session does not contain an answer for a particular criterion, the column will be left empty

for the current row. After all the columns have been iterated, the row is considered complete and it is appended to the worksheet.

```
sessions.forEach(session=>{
  const row = {};
  columns.forEach(column => {
    //Column for userGroup
    if(column.key == 'userGroup') { … }

    //Column for evidence
    if(column.key.includes('-evidence')) { … }

    //Column for userInfo
    if(column.key.includes('-userInfo')) { … }

    //Column for scale/select value
    …
  });
  worksheet.addRow(row);
});
```

Listing 27. An overview of the process of filling a worksheet with data.

After all the sessions are processed, some styling is added to the column fields of the worksheet. An example of a resulting worksheet with nine sets of answers for a question with three criteria can be seen in image 7. Finally, the worksheet is ready to be written to a file. After a successful write, the method returns the name of the created file.

	A	B	C	D	
1	**Group**	**Question 1 - Importance**	**Question 1**	**Question 1**	**Question 1 - Evidence**
2	Control group 3		2	3	Having been a gymnast, they
3	Control group 1	4	2	2	
4	Control group 3	3	1	2	Before scorpions, persimmon
5	Control group 3	5	2	1	Some joyous prunes are thou
6	Control group 3	4	5	2	A thrifty alligator's ant comes
7	Control group 1	3	0	4	A grape is a funny scorpion.
8	Control group 1	4	3	4	Before octopus, kangaroos we
9	Control group 3				Extending this logic, a success
10					

Image 7. A worksheet filled with test data created by the application.

The controller action responsible for exporting data is used to invoke the method and return the created file as a response. Koa lacks a built-in way of sending files, and koa-send library is used to define the file as the payload for the response.

4.4 Aggregating the result data

The set of data used by the client side to draw graph and represent the results of a survey is constructed using MongoDB aggregations. Sending the entire result set and processing it separately would cause serious overhead to the application in case of a larger result set. The required information includes the average values of answers, and the count how many times each specific answer was selected by the users. Listing 28 shows the aggregation for calculating the average results grouped by the answerer group. The match stage of the aggregation is used filter out questions that are not part of the target survey. The group stage groups the answers using the unique combination of question id, criteria title and group id to guarantee that there will be exactly one result object for every question criteria per answerer group. The additional operators of the group stage will calculate the average value of the answers. This requires two operators, because the answer's value is saved differently depending on the criteria type. The third operator of the group stage creates a set from criteria objects attached to answers.

```
await Answer.aggregate([
  { "$match": { "survey": survey } },
  { "$group": {
    _id: {
      question: "$question",
      criteriaTitle: "$criteriaTitle",
      type: "$type",
      userGroup: "$userGroup"
    },
    select: { $avg: "$values.select.value" },
    scale: { $avg: "$values.scale.value" },
    criteria: { $addToSet: "$criteria" }
  }}
]);
```

Listing 28. Aggregation for calculating the average results of a survey.

The results of the aggregation will produce an array of objects like the one seen in listing 29. The results are mapped to transform them into Answer objects for more convenient handling.

```
{
  _id: {
    question: 2,
    criteriaTitle: 'Current status',
    type: 'scale',
    userGroup: 2
  },
  select: null,
  scale: 2.75,
  criteria: [ [Object] ]
}
```

Listing 29. A result object produced by an aggregation.

Each result object's set containing the different versions of criteria is used to find the latest version there is an answer for. This is done because a survey and its questions can be modified anytime by the survey's owner, so to guarantee that the result set will reflect the answers in a meaningful way, the latest version must be extracted from the aggregation results. The criteria are used by the client side to display the basic criteria information. This doesn't however guarantee that all the answers would abide the numerical boundaries defined by the selected criteria, meaning that if the boundaries are modified, the answers not matching the new criteria version won't be discarded, nor will the aggregation operation ignore those answers when building the result set. This was a deliberate design choice to prevent any kind of data loss resulting from simply modifying questions and should the questions of a survey undergo such drastic changes that the existing answers should no longer be a part of the result set, the survey should be recreated.

4.5 Localization

The support for multiple locales was one of the lesser requirements for the application, and the localization was implemented at the end of the migration to the Koa based stack. This chapter explains the methodology used to create the localization system, and the technical implementation of the system is explained in more detail in the thesis of Miika Ahonen.

The challenge in implementing the localization was that both the frontend and backend needed the means to translate text. Views are rendered by the server, and all the content including the localized texts will be set at the time of rendering. The browser side scripting is used to create new content on top of the views, and the new content needs to be localized as well. Many third-party libraries for localization are designed to be used with either with the server side or the browser side of the application, and we refrained from using two different libraries for a singular problem. One technique to solve the problem would been to utilize the fact that views can contain scripts in addition to the HTML content, so the scripts could have been placed directly into the views and manipulated just like other content during rendering, meaning that a single library for localization would have sufficed. Due to the large quantity and complexity of the scripts, it would have been

impossible to implement the frontend functionality using only views in this case. The solution also would not have allowed for reusing the scripts, and the scripts not going through the process of converting the code into a browser compatible version would have limited the available features of the scripting language. The possibility of using the server side to supply the client-side scripts with translations was also explored, but it would have required redesigning parts of the frontend logic, which we wanted to avoid keeping the changes to the existing code minimal at this stage of development.

In the end it was decided to create a new module usable by both ends of the application to handle the localization. There are some differences in the JavaScript code designed for the browser and the Node environment. The main difference is the export system of node used to specify the parts of a file to expose when a module is required, whereas the browser doesn't support the export expression at all. To get around the problem, the type of exports can have tested, and a new object can be created for the browser to use. Another issue is that unlike Node, the browser will export all the contents of a module using the global scope and not just the part defined with the previously added export statement. The browser can be enforced to return only the exports by for example placing the contents of the module inside a closure, like the one seen in listing 30 that accepts an exports object as a parameter. As long as these two issues are addressed, a module can be required on the Node platform and imported with Webpack in client-side scripts. The latest features of JavaScript can be used, sense the code gets converted to a browser compatible format.

```
(function (exports) {
  // contents of the module
  class ClassToExport { … }

  // defining the exports
  exports.ClassToExport = ClassToExport;
}) (typeof exports === "undefined" ? {} : exports);
```

Listing 30. Structure of a module usable by the Node platform and client-side scripts.

The localization module is used in the Responder middleware to inject it to views, so the module is available in the views at the time of rendering by default. User's preferred locale is stored to the current session, but the module has fallback language if the session is not available. The session is accessible only in the server side, but the value of the locale is always stored in a meta element in the views, so that the client-side scripts can access the value.

4.6 Testing

Testing the code was a major part of the development process. In addition to the mandatory manual testing, simple test cases were written occasionally to help getting a new feature to work in a correct way, or in cases where a feature could not be efficiently tested manually. Only later in the development during the migration to the final technology stack it was decided to create conventions for the testing with the goal of improving the overall test coverage and the maintainability of the code base.

The tests can be divided into controller and model related tests. The model tests are used to test the methods of data models with the aim to have any method more complex than a getter or setter tested. Controller related tests are focused specifically on covering controller actions manipulating data or containing some error prone logic. Ideally controller actions wouldn’t contain logic of their own, and instead would only utilize methods of the model or service classes, but because this condition was not met, it was decided to also design tests for controllers.

4.6.1 Manual testing

There was large amount of manual testing involved in the development. To have a constant flow of test data, and to make the locally run version of the application resemble the production environment more closely, a script for populating the databases with random data was created.

The script was implemented as a mocha test, because it made it possible to split the process into logical blocks using test cases provided by the framework. Each block is used to generate objects of one type, or to create relations between different kinds of objects. The benefit in isolating different operations to their own test cases, is that mocha can be used to run only a selected set of tests, so that for example only specific types of objects can be generated. Not all of the data is randomized, and instead of using randomly generated questions, the script uses questions from an actual use case provided by the clients to make the generated data seem more authentic. The script was made flexible by utilizing multiple customizable variables to control the population process, so the quantity of the generated data or the frequency used to create connections between records can be quickly modified. Listing 31 shows the test block used to generate Evaluation objects.

```
  it('Should create evaluations', async function() {
    const users = await User.where();
    for(let i = 0; i < evaluationLimit; i++){
      const name = `Evaluation ${i+1}`;
      Logger.info(`Adding evaluation ${name}`);
      await new Evaluation({name})
           .allow(Utils.selectRandom(users), Permission.ADMIN).save();
    }
  });
```

Listing 31. The test block used to generate Evaluation objects in the population script.

In addition to generating dummy data to test the local version of the application with, the script was used to test how the application performed when provided with large sets of data. The effects could be observed in the request handling performance as well as in the performance of the user interface, and how efficiently it managed to display all the given information.

4.6.2 Model tests

Listing 32 shows a part of the MariaModel class test that has the same structure used in all model related tests. There is a describe block for every class method that contains all the tests related to that method. If the tests require some kind of one time set up, like inserting some data to the database prior to the tests, the before hook is used to handle that. An instance of the target class accessible in the test blocks is defined inside the outer describe block, so that the beforeEach hook can be used to instantiate a new object with default state for every test. The beforeEach hook is also used to set up any required conditions shared by multiple tests.

```
describe("MariaModel", async function() {
  /**
   * @type {MariaModel}
   */
  let target;
  before(async function() { … });
  beforeEach(async function() { … });

  describe("save", async function() { … });
  describe("find", async function() { … });
});
```

Listing 32. An example of a model test structure.

The goal with the tests is to test the standard behavior of a method, and if a method has clear fail states, these states are tested as well. Listing 33 gives an overview of the tests written for the find method of MariaModel class. These tests focus on trying to provoke the different return types of the method.

```
describe("find", async function () {
  it("find should return a record matching the given query",
    async function (){ … });

  it("find should return null if no records matched the given query",
    async function () {
      const actual = await User.find({ id: -1 });
      assert.isNull(actual);
  });

  it("find should return null if more than one record matched the query",
    async function () { … });
});
```

Listing 33. The describe block for find method containing three tests.

Well maintained tests are a useful tool in development for detecting bugs in the application logic and observing how changes to the code affect the whole application. Tests also work as a good foundation for writing new code. The behavior of a function can be examined closely even without attaching it to the application by simply matching it against a test case. Compared to manual testing, unit tests can be easily rerun multiple times, and with unit tests it's easier to create any preconditions for the tests.

The benefits of the tests were slightly hindered by the fact that that the classes of the application have dependencies between each other due to the use inheritance, making it in many cases impossible to design fully independent tests. This creates a situation where tests that are not related directly to the problem will also start to fail in case there is a bug in the code, making it more difficult to determine where the error originates from. In this case the way to find the source of error is to look for the lowest common component in the class hierarchy among the failing tests. The way to prevent the problem would be to design a specific order for running the tests, so that the tests of the lower level components would be run first.

4.6.3 Controller tests

Changing the backend to a Koa based one, meant that each controller action had to be rewritten. Despite some minor chances to the logic of the actions, the action signatures stayed the same to keep frontend code compatible with the redesigned backend. Instead of trying to attach the two parts of the application while the backend was still in development, the actions were developed testing them using Postman ADE (API development environment) and mocha tests.

The mocha tests were written using chai-http plugin for Chai assertion library that makes it possible to send customized HTTP requests for the application to handle, and create assertions based on the response. The structure of the tests is essentially the same as in model related tests, and the hooks are used to setup the preconditions for the tests, but the the setup also includes starting the application. Listing 34 shows a test written for the login action.

```
it('login with incorrect credentials should fail', async function () {
  let res = await agent.post('/user/login')
    .send({ 'username': user.username, 'password': 'notValid' });
  expect(res).to.have.status(200);
  expect(res).not.to.have.cookie('koa:sess');
  expect(res).to.be.html;
});
```

Listing 34. A mocha test for a controller action.

The agent variable in the example contains a chai request agent used to maintain the cookies between requests if needed. In the example test the request contains invalid data and in this case the action is expected to handle the related error. Sense the action should fully process the request, a response with status code 200 is expected in the first assertion. The second assertion verifies that the action failed by checking that the response doesn’t contain the cookie created if the authentication was successful. The last assertion checks the content type of the response and verifies that the action returned a view.

5 Development

5.1 Methodology

The development process happened in close cooperation with the clients, and there were meetings about the project every couple of weeks. The meetings were used to inform the clients about the progress made in the development, and because the initial design and requirements for application were relatively loose, time was also spent on polishing the design together with the clients. Since the clients were actively involved in the development process, it was phased so that there were milestones set between every meeting with the goal of having something new to show the clients.

In the early phases of the development certain features were prioritized to create an early working version of the application, so that it would be possible for the clients to actually test the application and better assess the current direction of the development. The feedback from the clients was used throughout the development process to shape application. The production environment for the application was also set up yearly in the development, so that a working version was always available for the clients.

5.2 Difficulties during the development

A major problem during the development was the loosely defined use case for the application. Originally the application was designed to be used at an organizational level, and the features were designed accordingly to be stricter, and the application was practically built around the idea of groups that would separate users from each other. This meant that features of the application were dependent on users from different groups interacting with each other, and it was expected that every user would register to the application and select a group for themselves. For example a single user wasn't able to run a survey with an evaluation consisting of multiple groups, unless other users had first created the groups and allowed the groups to be used as part of an evaluation. During the development the focus however shifted towards the application being a more generic survey management tool, which meant that a single user needed to be able to use all the features of the application without relying on other users. This was a major change in the application design as the group centred design was dissolved. In addition to redesigning existing features, the change also required creating entirely new ones to make the flow of the application match the new use case.

The loose requirements combined with the continuous feedback also presented its own challenges to development, for the specifications for the features could quickly change, and new features were requested often by the client. An influx of changes to the design was counterproductive, and it would have been beneficial to specify the requirements for the application more thoroughly in the beginning of the development.

Another thing that had a negative impact on the development was that the actual testing conventions were adopted so late, and for a long time there was no way of easily checking how a change to the code would affect the entire application. This allowed multiple bugs to get passed the initial manual testing, which was inconvenient, because fixing problems in the code got often more complicated when it was done afterwards. The lack

tests was also emphasized by fact that the fact multiple people often had to work on the same features, and it was sometimes difficult to determine whether a new feature designed by someone else in the developer team was working in the intended way.

5.3 Continued development

Sense the application underwent large changes at end of the development, a logical next step for the development would be to go through the flow of the application with the clients in detail and assess how it could be improved now that it has been in use. From the client's perspective, the usability was a major concern throughout the development, and it was gradually improved by simplifying certain features of the application. Because the group centric design was dropped in the end, the usage could be further streamlined based on the fact the last user tests implied that the users were more likely to distribute the surveys using the sharing feature than forcing all the participants to register to the application.

From the technical standpoint, the process of transferring the logic from the controller modules to the models should be the next priority. Placing the logic to the models makes it possible to the cover most of the application's logic with lower level tests that are easier to setup and maintain than controller tests. Eventually the change would increase the maintainability of the application.

6 Synopsis

The goal of the thesis was to implement a user-friendly application for managing and running e-form based surveys. The application was designed and implemented fully by the developer team utilizing the Node platform and currently popular web-development frameworks available for the platform. Two version of the application were created using different technologies to achieve the best result from a technical standpoint.

The application was successfully created in the given period of time, and the survey management part was built successfully to meet the requirements of the client, and the application can be effectively used to create and distribute surveys with multidimensional questions. The application was taken in to use by the client and will work as platform for the coming QAEMP surveys.

The way the users and answerer groups are linked together underwent multiple design changes during the development, and in the end, it was not highly approved by the clients, because of its complexity. The concept of answerer groups was tied so tightly to the application design, that despite it being heavily simplified, it makes the flow much more complex compared to other available survey management tools. The goal of having the application easy to use, was not met in this regard, and should the application be further developed, the enhancement of the usability should one of the priorities.

The development process suffered mostly from vague specifications and lack of common development conventions, and in my opinion this project highlighted the importance of having a clear design to follow at the beginning of the development, instead of slowly outlining the design while the product is being developed at the same time.

References

1 Plugge E, Hawkins TS, Membrey P. The Definitive Guide to MongoDB. Apress; 2015.

2 Aggregation [online]. MongoDB.
URL: https://docs.mongodb.com/manual/aggregation/. Accessed 15 March 2018.

3 Sriparasa SS. Building a Web Application with PHP and MariaDB: A Reference Guide. Packt Publishing; 2014.

4 Wilton P, Colby JW. Beginning SQL. Wrox; 2005.

5 About JavaScript [online]. MDN web docs; 3 February 2018.
URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/About_JavaScript. Accessed 2 March 2018.

6 White A, Horn S, Orchard LM. JavaScript programmer's reference. Wrox; 2009.

7 Daggett ME, Fielding J. Expert JavaScript. Springer; 2013.

8 ECMAScript 2017 Language Specification [Online]. Ecma International; June 2017.
URL: https://www.ecma-international.org/ecma-262/8.0/. Accessed 15 March 2018.

9 Young A, Cantelon, M, Meck B. Node.js in action, second edition. Manning Publications; 2017.

10 Promise [online]. MDN web docs; 9 February 2018.
URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise. Accessed 18 March 2018.

11 Iterators and generators [online]. MDN web docs; 31 March 2018.
URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Iterators_and_Generators. Accessed 18 March 2018.

12 Async functions [online]. MDN web docs; 10 April 2018.
URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/ Statements/async_function. Acessed 23 March 2018.

13 Gackenheimer C, Ashworth T, Jonge A. Node.js Recipes. Springer; 2013.

14 Brown E. Web Development with Node and Express. Sebastopol, O’Reilly; 2014.

15 Overview of Blocking vs Non-Blocking [online]. Node.js.
URL: https://nodejs.org/en/docs/guides/blocking-vs-non-blocking/. Accessed 2 March 2018.

16 Trevor N. Understanding the Node.js Event Loop [online]. The NodeSource Blog; 20 January 2015.
URL: https://nodesource.com/blog/understanding-the-nodejs-event-loop/. Accessed 3 April 2018.

17 The Node.js Event Loop, Timers, and process.nextTick() [online]. Node.js.
URL: https://nodejs.org/en/docs/guides/event-loop-timers-and-nexttick/. Accessed 3 April 2018.

18 Syed BA, Bean M. Beginning Node.js. Apress; 2014.

19 Mardan A. Full Stack JavaScript. Apress; 2015.

20 Prokopiev A. Which of the Node.JS Frameworks to Choose and Why? [online]. Artjoker.
URL: https://artjoker.net/blog/which-of-the-nodejs-frameworks-to-choose-and-why/. Accessed 4 April 2018.

21 Waterline: SQL/noSQL Data Mapper [online]. Sails.
URL: https://next.sailsjs.com/documentation/concepts/models-and-orm/. Accessed 5 March 2018.

22 Waterline query language [online]. Sails.
URL: https://next.sailsjs.com/documentation/concepts/models-and-orm/query-language. Accessed 5 March 2018.

23 Actions and controllers [online]. Sails.
URL: https://next.sailsjs.com/documentation/concepts/actions-and-controllers/. Accessed 5 March 2018.

24 Policies [online]. Sails.
URL: https://next.sailsjs.com/documentation/concepts/policies. Accessed 5 March 2018.

25 Koa, next generation web framework for node.js [online]. Koa.
URL: http://koajs.com/. Accessed 5 March 2018.

26 Mingoia A. Koa-router [online]. Github.
URL: https://github.com/alexmingoia/koa-router. Accessed 5 March 2018.

27 Mocha [online]. Mochajs; 8 April 2018.
URL: https://mochajs.org/. Accessed 6 March 2018.