sortit: Web Card Sorting Backend

Haris Ljajić

Haris Ljajić

sortit: Web Card Sorting Backend

Master's Thesis

to achieve the university degree of

Diplom-Ingenieur

Master's Degree Programme: Computer Science

submitted to

Graz University of Technology

Supervisor

Ao.Univ.-Prof. Dr. Keith Andrews
Institute for Information Systems and Computer Media (IICM)

Graz, 23 Aug 2016

Haris Ljajić

sortit: Card Sorting Web Anwendung Backend

Masterarbeit

für den akademischen Grad

Diplom-Ingenieur

Masterstudium: Informatik

an der

Technischen Universität Graz

Begutachter

Ao.Univ.-Prof. Dr. Keith Andrews
Institut für Informationssysteme und Computer Medien (IIICM)

Graz, 23. August 2016

Diese Arbeit ist in englischer Sprache verfasst.

Statutory Declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used.

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Date/Datum	Signature/Unterschrift

Abstract

Card Sorting is a collaborative method in user experience design for understanding how people think about content and categories. It is a technique used to help design and develop an information architectures, such as website navigation paths, workflows, or menu structures. Traditionally, card sorting is performed face-to-face with printed cards and a facilitator. More recently, web-based card sorting applications have become available, allowing card sorting studies to be run online with remote test users.

`Sortit` is a new, web-based card sorting application, which takes advantage of one of the most recent advances in modern web development: full-stack web application frameworks. Full-stack frameworks employ JavaScript at all application levels on both client and server. `Sortit` makes use of the Meteor framework together with other popular technologies, including React and Redux.

Kurzfassung

Das Card Sorting Verfahren wird im User-Experience-Design eingesetzt um ein besseres Verständnis für die Denkweisen der Nutzer über Inhalte und Kategorien zu bekommen. Diese Technik wird angewendet um Informationsarchitekturen sowie Website-Navigationspfade, Workflows oder Menüstrukturen zu entwerfen und zu entwickeln. Card Sorting wird in der Regel von Testpersonen und dazugehörigem Moderator, von Angesicht zu Angesicht, mittels bedruckter Karten durchgeführt. Durch neueste webbasierte Anwendungen können Online Card Sorting-Studien mit Teilnehmern durchgeführt werden.

`Sortit` ist eine neue, webbasierte Card-Sorting Anwendung, die den Vorteil der jüngsten Fortschritte der modernen Web-Entwicklung nutzt: full-stack web application frameworks. Full-Stack-Frameworks verwenden JavaScript auf allen Anwendungsebenen, Client sowie Server. `Sortit` nutzt das Meteor Framework zusammen mit anderen gängigen Technologien, einschließlich React und Redux.

Contents

List of Figures

List of Listings

Acknowledgements

I especially wish to thank my advisor, Keith Andrews, for his immediate attention to my questions and endless hours of toil in correcting draft versions of this thesis.

Special mention goes to my sister-in-law Sanja Ibraimović Ljajić, and my brother Emil Ljajić, for converting my barely legible sketches into beautiful vector diagrams.

I dedicate this thesis to my parents.

Haris Ljajić
Graz, Austria, August 2016

Credits

I would like to thank the following individuals for permission to use their material:

- The thesis was written using Keith Andrews' skeleton thesis [Andrews, 2012].
- The diagrams used in Figures 3.1, 3.2, 3.3, 3.4, 3.5, 4.1, 5.1, 5.2, 5.3 and 5.5 were drawn by Sanja Ibraimović Ljajić and Emil Ljajić.

Chapter 1

Introduction

This thesis describes the `Sortit` card sorting web application, and in particular its backend. Chapter 2, describes card sorting, and examines currently existing card sorting web applications. Chapter 3 surveys some of the more popular full-stack JavaScript frameworks. A short review of the React [Facebook, 2013] UI development framework, and its advantages over the traditional UI frameworks is provided in Chapter 4.

The second part of the thesis describes the technical implementation of this work. Chapter 5 discusses technical issues involved in the development of the `Sortit` web application. In particular, the chapter documents the requirements, design, and implementation phases of the project.

Chapter 2

Card Sorting

Card Sorting is a collaborative method from the field of user experience used to understand how people think about content and categories. It helps design and develop information architectures, such as workflows, menu structures, or website navigation paths. The goal is to create a "usable design" – a logical and user-friendly structure, which is easy to understand and navigate by end users. [Spencer, 2009]

Traditionally, card sorting is performed face-to-face with printed cards, a facilitator, and one user at a time. In preparation, the project manager first identifies key concepts and writes them on index cards. Each test user arranges the cards into groups (and sometimes subgroups) according to the perceived relationships between the concepts. This technique is called *open* card sorting, and is illustrated in Figure 2.1.

In order to process the results of a card sorting study, it is important to understand the organising principle (or *mindset*) each participant employed when grouping the cards and labelling the groups. When sorting grocery items, for instance, one sorter might have supermarket shelf positions in mind, whereas another might group the items according to country of origin, or which kind of meal each item may be used for. In a face-to-face sort, the facilitator can help ascertain each sorter's mindset. In an online sort, each sorter can be asked to describe their grouping strategy in a dialog box at the end of their sort. For analysis of card sorting results, it is important to first group the sorts by mindset, so that all sorts with, say, supermarket shelf, as the organising principle are analysed together. Mixing sorts from different mindsets does not produce meaningful results.

For a statistical analysis to be feasible, it is typically necessary to have several dozen sorts in a mindset. The first step is to standardise group names. All groups are assigned standardised names from a list of standard groups created for each mindset. This eliminates inconsistencies where groups contain the same or similar cards, but have been given different names. Standardised sorts are then submitted to a series of analytic methods to derive best a fit grouping.

Alternatively, cards can be sorted into predetermined groups. This variant is called *closed* card sorting and is illustrated in Figure 2.2. Closed card sorting is typically used to evaluate how well content fits into a given set of category names. However, other techniques are often better at evaluating how well a given information hierarchy works. For example, simply mocking up the tree structure in a web browser and asking users to locate certain items (a technique called tree testing) produces better results than closed card sorting in practice, especially since multiple levels of a hierarchy can be evaluated at once.

There is also a combination of the two card sorting methods, known as *hybrid* card sorting. Predetermined groups are set initially, but if a participant cannot find a suitable group for a card, they are allowed to create their own group during the sorting session. Like closed card sorting, hybrid card sorting is also not very useful in practice. Unless otherwise stated, in this thesis, the term card sorting always refers to open card sorting.

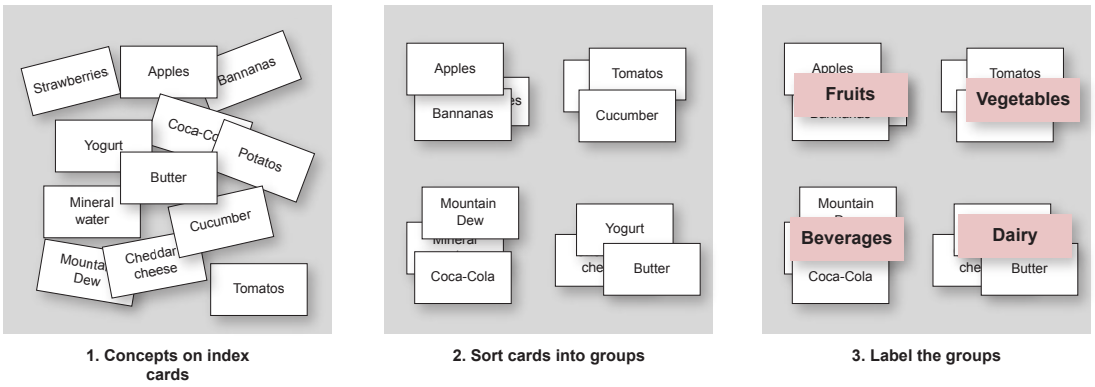

Figure 2.1: Open card sorting. Each user first sorts cards into groups, then labels the groups. [Adapted from the original by Spencer [2009]. Redrawn by Sanja Ibraimović Ljajić and used with kind permission.]

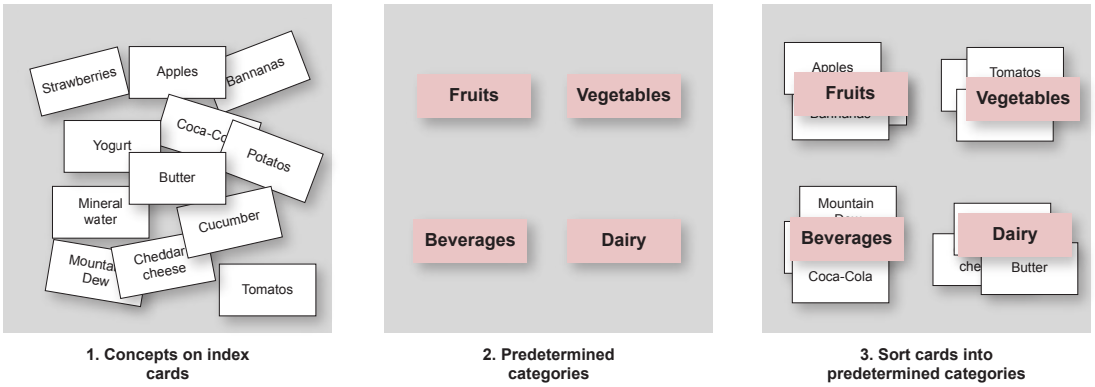

Figure 2.2: Closed card sorting. Cards are sorted into predetermined categories. [Adapted from the original by Spencer [2009]. Redrawn by Sanja Ibraimović Ljajić and used with kind permission.]

A number of web-based card sorting solutions already exist. These are mostly commercial services, requiring a fee or monthly subscription for all but the most limited of card sorting exercises. Three such web-based card sorting applications are described in the following sections.

2.1 SimpleCardSort

SimpleCardSort [SCS, 2016] is a commercial, web-based card sorting application, offering two subscription packages: pro and basic. The following features are available:

- Unlimited participants.
- Unlimited projects.
- Unlimited cards with optional tooltip descriptions.
- Open, closed, and hybrid sort types.
- Group merging.
- Customisable messaging.
- Results analysis – including similarity matrix and tree view.
- Redirect to a custom URL after sort.
- Subgroups (pro).
- Replaying participants moves step-by-step – participant sort replay (pro).
- Ordering cards within a group – in-group ordering (pro).
- Using a custom logo (pro).

The user interface for card sorting managers is shown in Figure 2.3. The page features options for configuring name, type, and options related to subgroups and in-group ordering. Cards are added by inputting into a provided text box, one per line. Also, some UI customisation options are available. These options allow the configuration of greeting messages, instructional texts, card display order, etc.

The card sorting user interface is illustrated in Figure 2.4. A group can be created by either dropping a card onto the workspace, clicking on a card in the card list, or clicking on the `Create New Group` button in the header. There is no underlying grid system for positioning and arranging groups, which means that groups can be freely moved around across the workspace. Card sorting is performed by dragging a card from one group, and dropping it onto another.

The analysis interface is shown in Figure 2.5. The interface features an abundance of options for calculating the metrics relevant for card sorting analysis. The analysis options include:

- Participant summary.
- Participant sort replay.
- Raw data ($participant \times card$).
- Group summary.
- $Groups \times Card$.

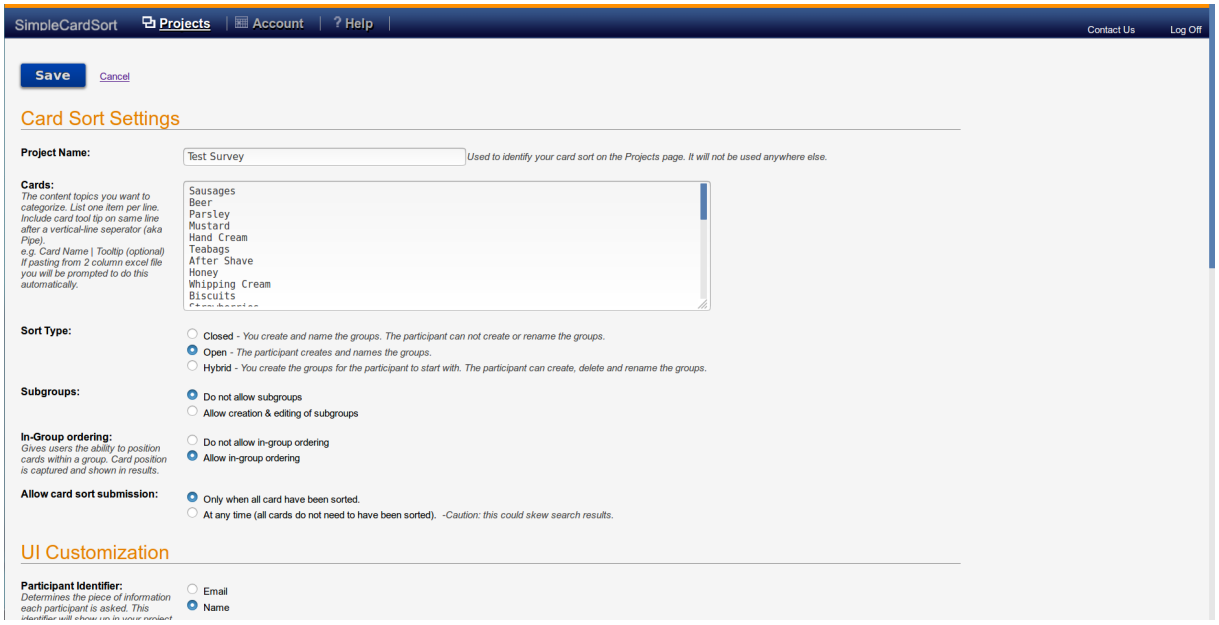

Figure 2.3: The SimpleCardSort user interface for managers. [Screenshot taken by the author of the thesis.]

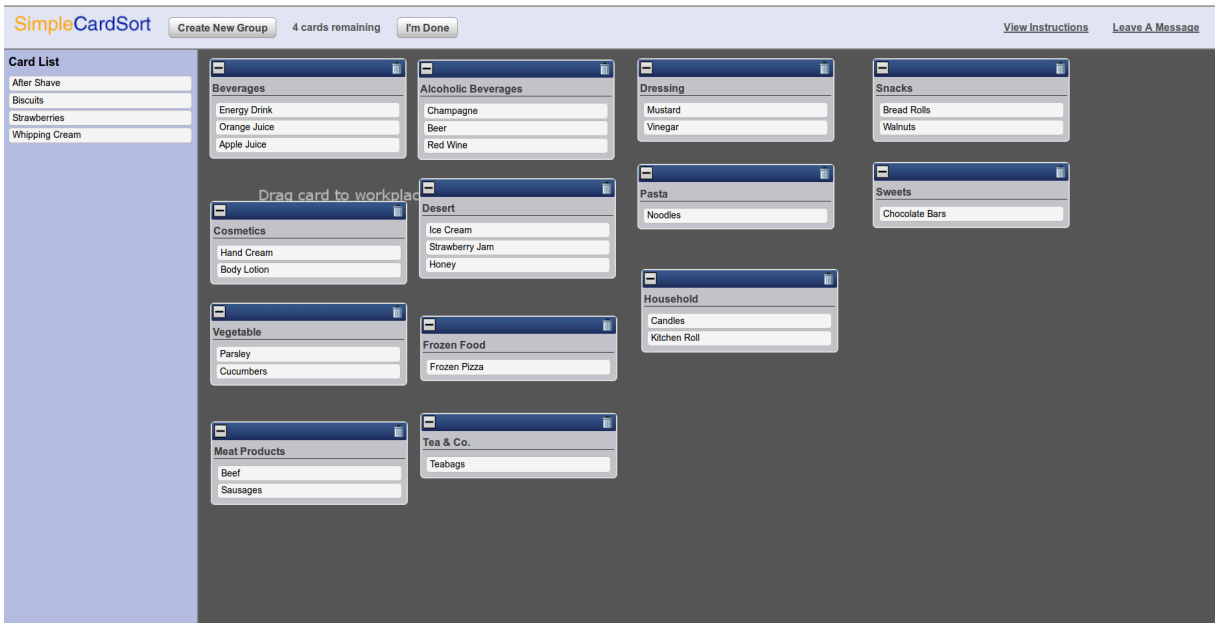

Figure 2.4: The SimpleCardSort card sorting interface. [Screenshot taken by the author of the thesis.]

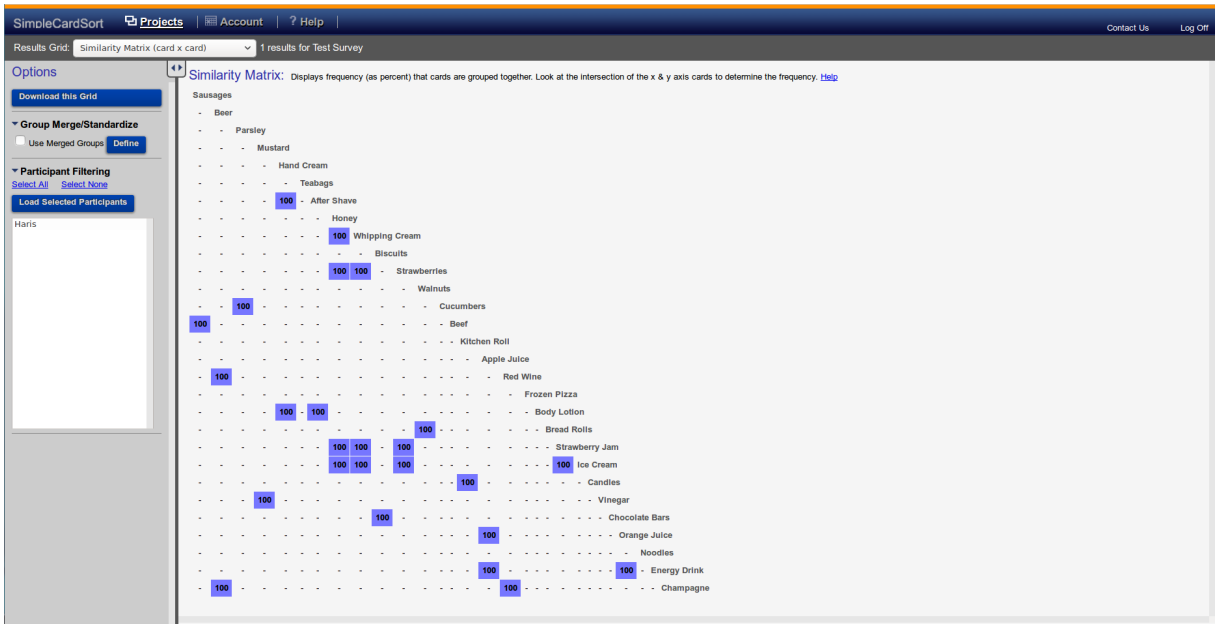

Figure 2.5: The SimpleCardSort analysis user interface showing similarity matrix of a sort. [Screenshot taken by the author of the thesis.]

- Card summary.
- Similarity matrix (*card* × *card*).
- Tree (dendrogram).
- Solution (best fit grouping).

2.2 usabiliTEST

usabiliTEST [usabiliTEST, 2016] is a commercial, web-based card sorting tool which supports open, closed, and hybrid card sorting projects. usabiliTEST provides a number of built-in analytic tools:

- Display raw numbers.
- Convert to percentages.
- Group percentages.
- Create a distance matrix.
- Plot multi-dimensional scaling (MDS) analysis of the distance matrix.

The card sorting manager user interface is shown in Figure 2.6. The interface is divided into three sections:

1. Project details section, where card sorting type, name, description, logo, language, and privacy settings can be configured.
2. Section for adding cards, which allows cards to be added individually, or in bulk.

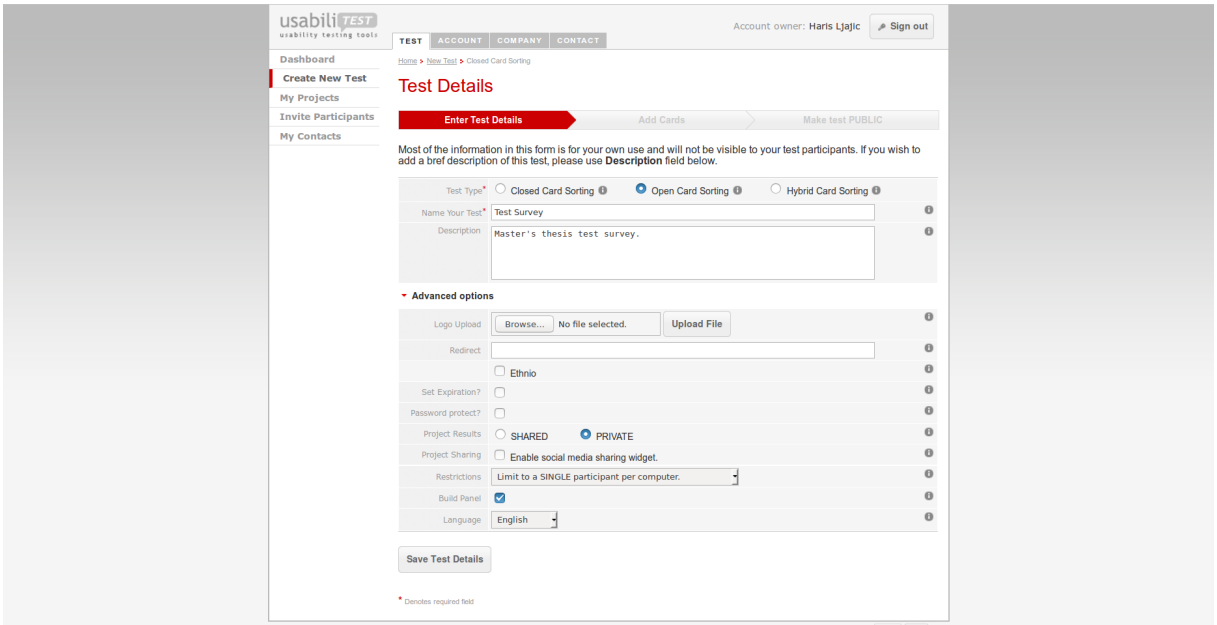

Figure 2.6: The usabiliTEST card sorting manager user interface. [Screenshot taken by the author of the thesis.]

3. Project overview section, which provides an overview of the project settings with the project URL, but also allows settings to be edited.

The user interface for card sorting managers also features a project dashboard, where various project statistics such as view count, number of times participated, and average completion rate are displayed.

The sorting interface is shown in Figure 2.7. Groups are created by dropping a card onto the workspace. A mild grid system is in place, which helps arrange newly created groups, but groups can also be freely positioned across the workspace. Dropping a card onto a group adds that card to the group.

2.3 OptimalSort

OptimalSort [Optimal, 2016] is a commercial, web-based card sorting application. Apart from the standard card sorting features and data analysis tools, it also has a very modern user interface, as illustrated in Figure 2.9.

The card sorting manager user interface is shown in Figure 2.8. Like usabiliTEST, OptimalSort allows for configuring various project details, such as sort name, type, language, URL, etc. Cards are added individually or imported in bulk. OptimalSort provides support for generating printable cards as `PDF` for use in a face-to-face (moderated) card sort. Each card has a barcode for convenient data entry back into OptimalSort. There are also options for configuring greeting or instructional messages. In addition, OptimalSort provides a convenient interface to setup questionnaires.

The sorting user interface is shown in Figure 2.9. Groups are created by dropping a card onto the workspace, and cannot be freely positioned, since the grid system enforces arranging groups into columns.

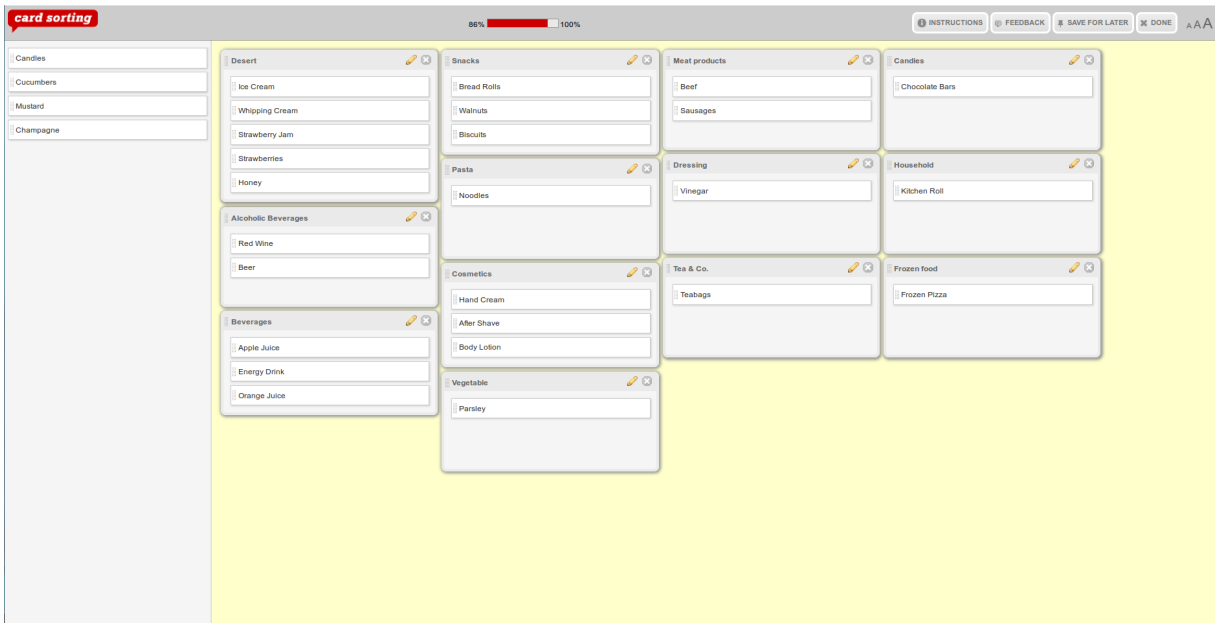

Figure 2.7: The usabiliTEST card sorting interface. [Screenshot taken by the author of the thesis.]

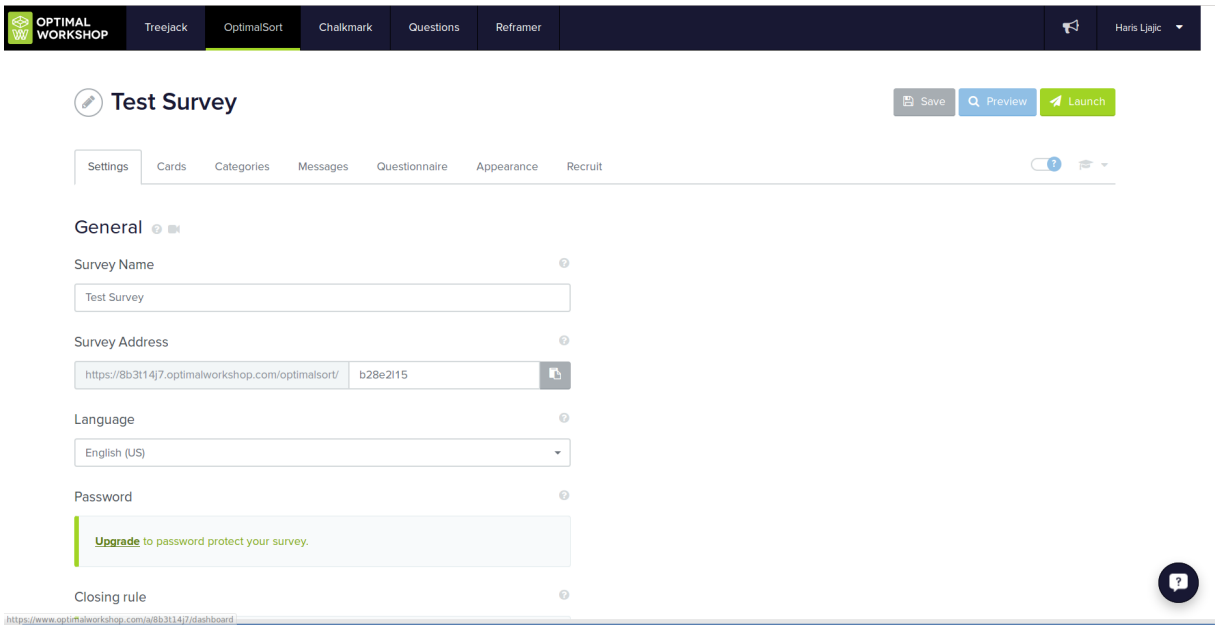

Figure 2.8: OptimalSort online card sorting manager user interface. [Screenshot taken by the author of the thesis.]

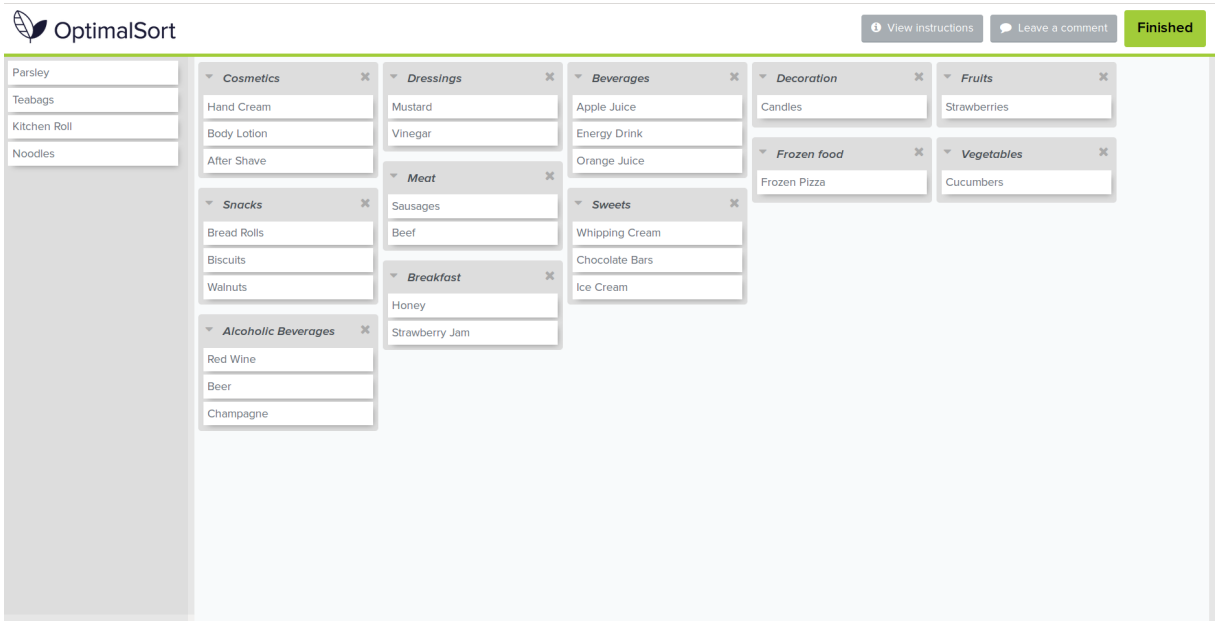

Figure 2.9: OptimalSort online card sorting interface. [Screenshot taken by the author of the thesis.]

Chapter 3

Full-Stack Frameworks

A full-stack framework, better known as a web application framework, is a set of software tools optimised to work together, in order to support the development of dynamic web pages, web applications, or web services. One of the main responsibilities of a web framework is to enable the fast development of functioning web applications. To achieve this, it is important to minimise the amount of overhead. Web frameworks provide core functionality common to a certain segment of web applications, typically including user session management, data persistence, and templating.

With the introduction of Node [NF, 2009], JavaScript programmers suddenly ceased to be frontend-only web developers. It is now possible to build servers and databases in JavaScript, in addition to user interfaces. Indeed, developers can now build entire web applications using only JavaScript, a trend often called full-stack JavaScript [Bretz, 2014].

3.1 Server-Side JavaScript and Node

Before delving into the specifics of each of the web application frameworks, it is important to understand how and why enabling JavaScript on the server-side made such an impact on the web development world. Node was originally written in 2009 by Ryan Dahl, who wanted to find an easier way for the browser to track file upload progress [Harris, 2013].

Node is an asynchronous, event-driven JavaScript runtime, designed to build scalable network applications. Node can handle many connections simultaneously, and does so by firing callbacks when each one is established. If there is no work to be done, Node sleeps. This is contrary to the today's more common concurrency model, where OS threads are utilised. Thread-based networking is known to be difficult to use, while also relatively inefficient. Node is free from those concerns. Almost no function in Node directly performs I/O, so a process never blocks. Since nothing blocks, it is possible to develop fast and scalable applications in Node.

Node's ideas are traceable to systems like Ruby's EventMachine or Python's Twisted, but Node takes the event model even further. An event loop is presented as a runtime construct instead of as a library. Unlike other systems, where there is always a blocking call to start the event-loop, Node does not employ such a concept. Node simply enters the event loop after executing the input script, and exits the event loop when there are no more callbacks to perform. This behaviour is like browser-based JavaScript – the event loop is hidden from the user. HTTP is a first class citizen in Node, designed with streaming and low latency in mind, making Node well suited for the foundation of a web library or framework [NF, 2009].

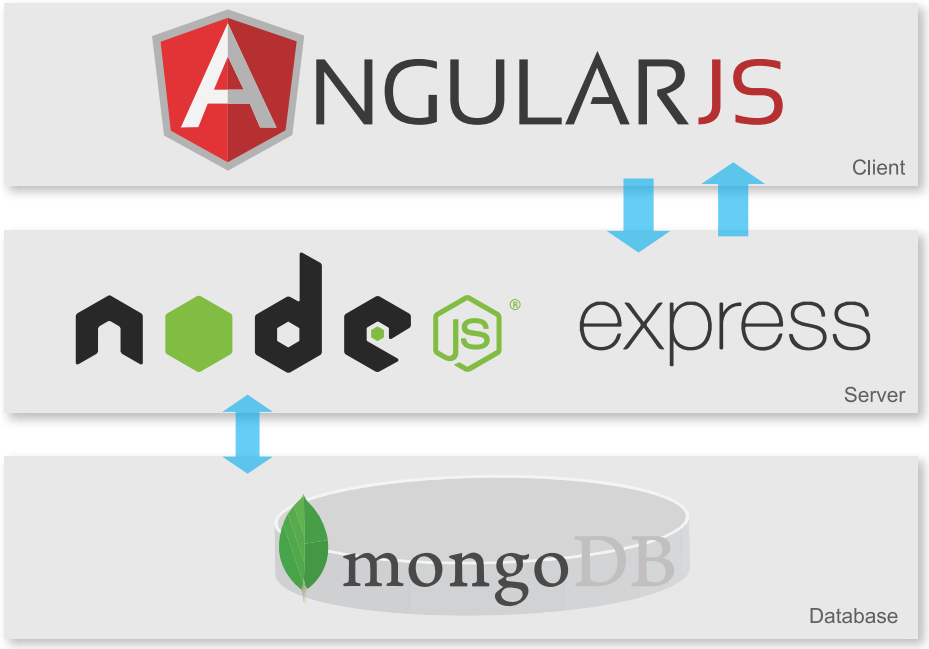

Figure 3.1: The technology stack of the MEAN [Haviv, 2014] framework. [Adapted from the original by Silva [2014]. Redrawn by Sanja Ibraimović Ljajić and used with kind permission.]

3.2 MEAN

A combination of four popular JavaScript technologies: MongoDB [2009], NF [2010], Google [2010], and NF [2009] has become known as the "MEAN Stack", which is illustrated in Figure 3.1. MEAN is licensed under the MIT [OSI, 1999] licence.

3.2.1 MongoDB

MongoDB [MongoDB, 2009] is a non-relational (NoSQL), schema-less, document-oriented, open source database, written in the C++ programming language. Being a document-oriented database, MongoDB operates with collections in a format similar to JSON. This means that many applications now can model and store data in a natural way, and in spite of their complex, nested hierarchies remain indexable and retrievable.

A MongoDB database can contain multiple `Collections`, which consist of field-value tuples. Values themselves can be comprised of further field-value pairs. `Collections` are saved in BSON format, a close relative of JSON format, thereby providing support for all JavaScript data types and being an ideal database for any Node platform. Due to its document-oriented, schema-less nature, each document (record in a `Collection`) can have different fields, and each field can have a different data type. In a relational database, such as MySQL, where every record must conform to a strict schema, this would not be possible.

The fact that it is written in C++, already qualifies MongoDB as a performant database. In addition, NoSQL databases generally perform better than their relational counterparts, since there are no relations among the `Collections`, needing to be taken into account during operations. Fields are indexable for better performance, and are easily replicable to allow for better backup strategies [MongoDB, 2009].

3.2.2 Express

Express [NF, 2010] is a "fast, unopinionated, minimalist web framework for Node.js" [NF, 2010]. Express is a server-side web application framework for the Node platform. It facilitates the design and development of modern web applications by adding utilities to the Node. Any serious application based on Node, requires the same boilerplate code for tasks such as:

- Parsing HTTP request bodies.
- Parsing cookies.
- Managing sessions.
- Organising routes.
- Determining proper response headers based on data types.
- Handling errors.
- Extracting URL parameters (e.g. `/project/abcd`).

Express solves these and many other problems by providing ways to reuse code elegantly and establishing a Model-View-Controller (MVC)-like structure for a web application [Mardan, 2014].

3.2.3 AngularJS

AngularJS [Google, 2010] is a client-side JavaScript web framework providing Model-View-Controller and Model-View-ViewModel (MVVM) architectures for creating single-page web applications. It is an open-source framework developed by Google.

AngularJS has a built-in two-way data binding and its own HTML-based templating language. It also has a feature called "directives" which support extending HTML with new attributes, and even new elements. This improves readability on the one side, but on the other side leads to automatic creation of reusable components. AngularJS is also highly testable, which is of great importance when developing large-scale client-side projects.

AngularJS makes large chunks of often written code unnecessary. Those familiar with jQuery would be astonished at how little code is required to accomplish basic actions. Through data binding and directives, some of the most time-consuming tasks, such as DOM selections, DOM manipulation and event handling, are almost completely automated. Moreover, by utilising the Dependency Injection design pattern [Jenkov, 2015; Google, 2016] the code can be even further reduced and more importantly testing and refactoring simplified.

3.2.4 Node

Node is the backbone of every full-stack JavaScript framework, and is discussed previously in Section 3.1

3.2.5 Getting Started with MEAN

Installing MEAN is relatively straightforward. There is an option to download a ZIP archive from the official website [Haviv, 2014], or to simply clone the entire GitHub repository [Daig and Cohen, 2016]. Since MEAN was initially merely a bundle of technologies, users needed an efficient means of scaffolding their applications. In their efforts to find a solution, MEAN contributors decided to use Yeoman

generators to provide a powerful, easy to maintain, and open solution for scaffolding applications. As a result, the MEAN.JS Yeoman generator was created, which makes development, build process, and deployment of production level MEAN applications easier.

Beside the application generator, MEAN also provides sub-generators to help avoid having to repeat structured code and thereby speed up development. These sub-generators are:

- CRUD Module Sub-Generator. The module helps create a new CRUD [Wikipedia, 2016] module.
- AngularJS Module Sub-Generator. The AngularJS module sub-generator is used to avoid the often redundant task of creating a new AngularJS module structure. It creates the proper skeleton and initialisation file.
- AngularJS Route Sub-Generator. Modules often require routes. The AngularJS Route Sub-Generator helps set up a routes.js file for each module, with views, controllers, and their corresponding routes.
- AngularJS Controller Sub-generator. Given the module name, the AngularJS Controller sub-generator creates a new AngularJS controller in the specified module's `controllers` folder.
- AngularJS View Sub-Generator. Once a controller is in place, adding views make it useful. The AngularJS view sub-generator facilitates this process, creating a new view in the specified module's `views` folder, and allowing for adding a route definition for it.
- AngularJS Service Sub-Generator. Given a service folder, this sub-generator creates a new AngularJS service.
- AngularJS Directive Sub-Generator. This sub-generator creates a new AngularJS directive in the specified module's `directives` folder.
- AngularJS Filter Sub-Generator. The AngularJS filter sub-generator creates a new AngularJS filter and put it in a specified module's `filters` folder.
- AngularJS Config Sub-Generator. This sub-generator creates a new AngularJS config section in a specified module's `config` folder.
- AngularJS Test Sub-Generator. A MEAN application is pre-bundled with the Jasmine testing framework and Karma test runner. In order to test AngularJS controllers, Karma needs a test file to run the tests upon. The AngularJS test sub-generator helps create such a test file.
- Express Model Sub-Generator. Even though a single Express model in a MEAN application is probably sufficient, a sub-generator is provided to help creating it.
- Express Controller Sub-Generator. To avoid the redundancy of creating empty Express controllers, the Express controller sub-generator is provided.
- Express Routes Sub-Generator. In order for controller methods to be accessible by the outside world, a routing file within the app/routes is required. Express routes sub-generator helps create an empty one.
- Express Test Sub-Generator. MEAN application will also come pre-bundled with Mocha testing framework. Express test sub-generator will help provide tests Mocha test runner will use for running tests.

Using any of the provided generators is as simple as executing a command with the generator's name and the name of what is to be created.

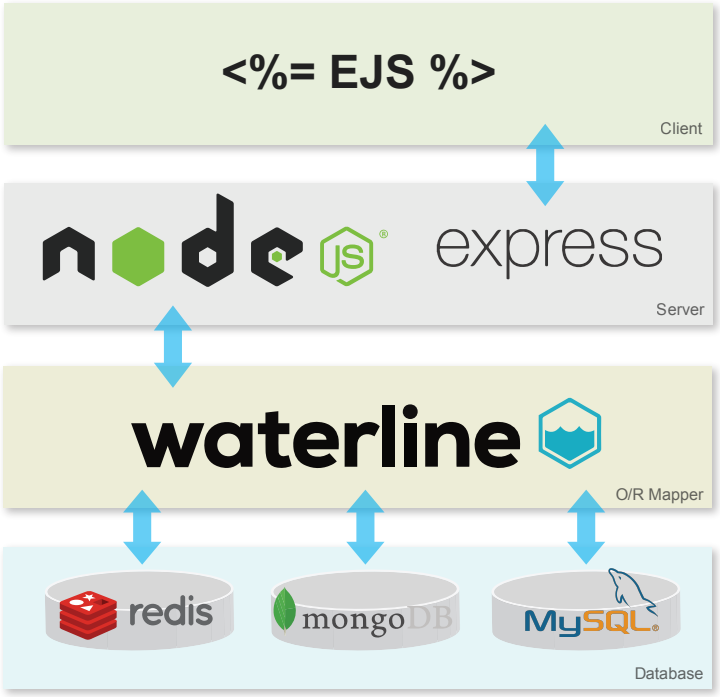

Figure 3.2: The technology stack of the Sails [McNeil, 2012] framework. [Drawn by Emil Ljajić and used with kind permission.]

3.2.6 Conclusion

The MEAN stack is a well-designed framework for starting full-stack web application development in JavaScript. It provides many facilities to avoid wasting time writing boilerplate code. Also, it is very flexible in terms of using other packages from the npm registry, since the framework itself does not enforce any constraints.

Judging from GitHub statistics (number of times the project was starred) and Stack Overflow mentions (number of times MEAN was tagged in questions), it seems that the MEAN stack does not generate much interaction in the web developer community, which might be a problem for someone just starting with web application development. Another downside of the MEAN framework is the fact that even though it is fairly simple to set up, it does not come pre-bundled with the technologies a developer is going to be working with, i.e. the technologies of the MEAN acronym: they must be installed separately and prior to setting up the MEAN workspace.

3.3 Sails

"Sails is the most popular MVC framework for Node.js. It is designed for building practical, production-ready Node.js apps in a matter of weeks – not months." [McNeil, 2012] Sails emulates the familiar MVC pattern of more traditional web development frameworks, as illustrated in Figure 3.2, enriched with support for some of the modern application requirements. It adds data-driven APIs along with a scalable, service-oriented architecture. Sails is especially tailored for building chat or real-time dashboard applications, as well as multiplayer games, but is suitable for use in any web application project – top to bottom. Like MEAN, Sails is also licensed under the MIT [OSI, 1999] licence.

Sails is developed with speed in mind – speed in creating server-side JavaScript applications. This is

particularly interesting to startup developers, who are often in need of means to test their ideas quickly, without compromising quality. Nonetheless, its robust service-oriented architecture and many different types of components allow for a very clean separation of responsibilities, which even makes building enterprise-level applications possible. Furthermore, the fact it is written in JavaScript permits another benefit – the ability to share code between the client and the server. Implementing data validation on the client and the server with the same rules in Sails becomes a matter of copying and pasting the relevant piece of source code.

One very important aspect of Sails is that it wraps a stack of loosely coupled components. Almost every aspect of the system is customisable. Most of its core components can be adapted without compromising the overall framework's stability. For getting the job done as quickly as possible, Sails will provide the logistics in terms of robust built-in components. However, should there be a need of a fully tailored web development environment, Sails will not stand in the way either. Under the hood, Sails does not differ much from the MEAN stack. It uses Node and Express. There is, however, one important distinction pertaining to the persistence layer – the Waterline ORM.

3.3.1 Waterline ORM and the Model Layer

The creator of Sails, Mike McNeill, did not only create Sails. He also wrote Waterline [Balderdash, 2012], a tiny database-agnostic, object-relational mapper (ORM), which is directly used by Sails. Waterline provides an abstraction layer and a common query interface to any database in use. This not only means that a web application built in Sails can have a wide variety of databases to choose from, but also working with any of them would, in a technical sense, be the same. This is feature is huge, and something otherwise only seen in enterprise-level server-side frameworks, such as SAP Hybris [SAP, 2012].

Sails data models work the same as in any other MVC framework. A model of type `Object` is created, and used to describe the attributes. The only difference is that the attribute values are not native JavaScript data types, but strings which are mapped to different database systems by adapters. As a result, no additional annotation is required by the ORM.

Apart from providing validation tools for the models, Waterline also enables the implementation of life-cycle callbacks – functions which are executed automatically at a certain point in time, yet another feature typical of enterprise-level frameworks [Fomin II, 2015].

3.3.2 The Controller and the Routes

Every model in Sails is automatically equipped with an already functioning JSON-based REST API. This means that anyone working with Sails can now stop worrying about building up a REST library, or laboriously implementing all of the CRUD functions and their corresponding HTTP methods. The magic lies within the "Blueprints", Sails' way of quickly generating routes and actions based on the application design. Together, blueprint routes and blueprint actions constitute the blueprint API, the built-in logic that powers the RESTful JSON API, which is automatically provided whenever a model and controller are created.

3.3.3 The Views Engine

Analogues to other MVC frameworks, Sails views are markup templates which are compiled into HTML pages on the server. Sails comes pre-configured to use Embedded JavaScript templates [Bitovi, 2014] (EJS) as its view engine. The choice of EJS came as a result of its highly conventional syntax, similar to some of the most often used templating languages, such as php, asp, jsp, and so on. However, should some other view engine be more preferable, swapping out EJS is relatively straightforward.

Sails provides two options for bootstrapping single-page applications: either using a single view, or using a single HTML page from the assets folder. Each option has its advantages and disadvantages,

and the choice between them depends on the exact needs of the application. The former option enables the web application to be almost completely decoupled from the server by passing the data it requires directly into the HTML that is rendered on the client. The latter provides better ways of distributing the web application content geographically closer to its users, since everything residing in the `assets` folder can be moved to a CDN [McNeil, 2012].

3.3.4 Getting Started with Sails

Assuming that Node and the npm package manager are installed, installing Sails is a fairly simple process. Executing the following command in the console

```
$ npm -g install sails
```

will install Sails on all three major operating systems.

Creating a new Sails project is similarly straightforward, and merely requires the following command:

```
$ sails new <project-name>
```

In case Node and npm are not installed prior to Sails installation, detailed instructions on how to install them are provided on the Sails website [McNeil, 2012].

3.3.5 Conclusion

The Sails framework provides a very robust platform for fast-paced full-stack application development, but it does not stop there. By adhering to the MVC-pattern principles, building enterprise-sized applications is also possible, and the fact that these applications can be written entirely in JavaScript, means that Sails brings something new to the table. There is considerably less context-shifting compared to traditional MVC frameworks, and it is possible to write code more consistently.

The powerful Waterline ORM, provides a simple data access layer, regardless of the database in use. Apart from the plethora of community projects, Sails also officially supports and ships adapters for MySQL, MongoDB, PostgreSQL, Redis, and local disk. In addition, Sails introduces a new way of building relational data models, aimed at making data modelling more practical. Beside the usual one-to-many and many-to-many relations, it is possible to assign multiple aliases per model.

As regard community interest, Sails is definitely in a better position than the MEAN stack. However, even though the promise of “one framework to rule them all” might be alluring, it is not always realistic. In that sense, Sails can be extended with other frontend technologies, such as Angular, Backbone, iOS, Android, or Windows, for additional functionality.

3.4 DerbyJS

DerbyJS [Derby, 2011] is an MIT-licensed [OSI, 1999], MVC framework based on Node and Express for building real-time, collaborative applications. It is illustrated in Figure 3.3. DerbyJS aims to replace currently popular libraries such as Rails, Django, and Backbone, where inconsistencies can occur when adding dynamic features to applications. Server code renders different initial states, whereas jQuery through selectors and callbacks hopelessly tries to make sense of the DOM and user events afterwards. This means that adding new features usually involves changing both frontend and backend, typically in different languages. DerbyJS identifies and mitigates these issues.

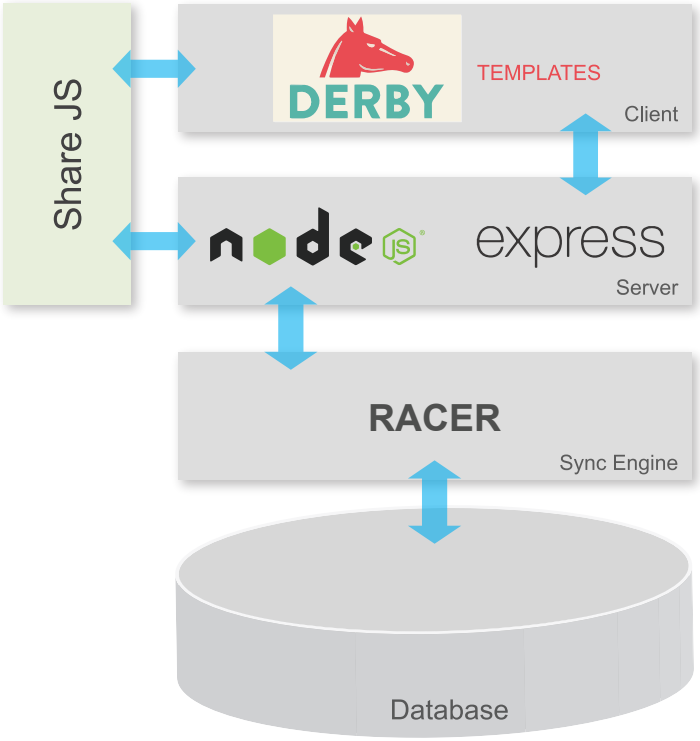

Figure 3.3: The technology stack of the DerbyJS [Derby, 2011] framework. [Drawn by Sanja Ibraimović Ljajić and used with kind permission.]

3.4.1 Data Synchronisation with Racer

Derby's secret is a powerful data synchronisation engine called Racer. Although it functions somewhat differently, Racer to DerbyJS is what ActiveRecord [Rails, 2016] is to Rails. Racer allows models to subscribe to changes on defined objects and queries, providing granular control of data propagation without having to define channels. This means that data is automatically synchronised between browsers, servers, and a database. To make the writing of collaborative, multi-user applications as simple as possible, Racer supports advanced features such as conflict detection and resolution.

Derby's mission is to make it effortless to develop applications that "load as fast as a search engine, are as interactive as a document editor, and work offline" [Derby, 2016a]. In that sense, Derby provides on top of Racer a powerful templating, data binding, and routing features. In fact, every feature of Racer and DerbyJS is written with server and client rendering in mind.

3.4.2 Real-time Collaboration with ShareJS

Powered by ShareJS's [ShareJS, 2011] operational transformation of JSON and text, effortless syncing of data across clients and servers, including automatic conflict resolution, is available out of the box. By default, every text input field is a collaborative text editor, and each piece of data in the model can be collaboratively edited in real-time or offline.

3.4.3 Client and Server Routing and Rendering

As in the previous frameworks, routing in DerbyJS is provided by Express. DerbyJS supports one or more single-page applications, as well as server applications, and the same routes can service them all.

DerbyJS packages up all of an application's templates, routes, and application code when rendering, so when a request comes, it is able to render a view within the same application client-side regardless of what URL is initially requested. If it cannot handle a URL, it will be passed through and requested from the server. If errors are thrown in the process of route handling, such requests will also be passed through to the server [Derby, 2016b].

Pages requested from the server, are generated there and served fully rendered. Once the HTML arrives at the client, subsequent request no longer need to make the round-trip to the server, but can be fully serviced by the client. For this to work, the rendering code would need to be duplicated on the server and on the client, but in DerbyJS it is not. DerbyJS is an isomorphic framework, which means that the same code can run both on the server and the client.

3.4.4 HTML Templates and View Bindings

DerbyJS uses its own templating syntax, based loosely on the semantic templating language known as Handlebars [Katz, 2010]. Templates are parsed as HTML and transformed into Template objects. `Template` objects can either directly create HTML for server rendering or DOM nodes for client rendering. When rendered on the server pages display immediately, before a single script is loaded.

In addition to HTML rendering, templates define live bindings between the view and the model. When model data changes, the view updates the properties, text, or HTML, as necessary to reflect the new data. Analogously, should users interact with the page, the model data is updated.

3.4.5 Getting Started with Derby

Similar to the previously described frameworks, a prerequisite to installing DerbyJS is having Node installed. In addition, Redis [Sanfilippo, 2009] and MongoDB [MongoDB, 2009] need to be set up. After that, DerbyJS is installed by running the command:

```
$ npm install -g derby
```

DerbyJS also comes with a project generator. Those who favour CoffeeScript [Ashkenas, 2009], can set up a bare-bones application by adding the option `--coffee`.

```
$ DerbyJS new --coffee derby-project
```

An application is started by navigating to its root directory and issuing the following command:

```
$ cd derby-project $ npm start
```

3.4.6 Conclusion

DerbyJS eliminates the tedious work of "wiring together a server, server templating engine, CSS compiler, script packager, minifier, client MVC framework, client JavaScript library, client templating and/or bindings engine, client history library, real-time transport, ORM, and database" [Derby, 2016a]. This does not mean, however, that DerbyJS does not play well with others. On the contrary, Derby follows the conventions and best practices of Node. Additionally, DerbyJS eliminates the complexity of keeping application state synchronised between models and views, clients and servers, multiple windows, multiple users, and models and databases. These two features make DerbyJS a good candidate for a beginner web developer to start their first full-stack JavaScript project. Community interest in the framework does not differ much that of the previously discussed frameworks, so getting support when needed might prove challenging.

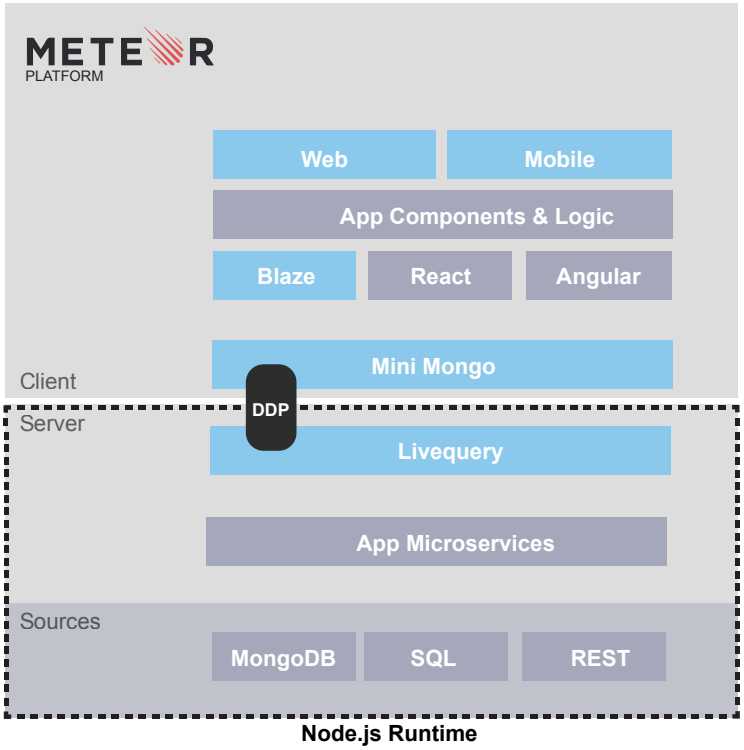

Figure 3.4: The technology stack of the Meteor [MDG, 2012] framework. [Adapted from the original by Dejaeger [2015]. Redrawn by Emil Ljajić and used with kind permission.]

3.5 Meteor

Meteor [MDG, 2012] is a platform built on top of Node, and licensed under an MIT licence [OSI, 1999], which enables the development of real-time web applications. Meteor sits between the database and the user interface (see Figure 3.4), and ensures that both are synchronised. Meteor has been around since 2012, but has only recently received widespread attention. Being built on Node, Meteor uses JavaScript both on the server and the client and, like Derby, Meteor is isomorphic. Hence, Meteor is a very straightforward and efficient platform that resolves many difficulties and pitfalls in the real-time web application development [Robinson et al., 2016].

In order to better understand the specific features Meteor has, it is necessary to step back and take another look at the JavaScript MVC family. Frameworks such as Angular [Google, 2010], Ember [Tilde, 2011], Backbone [Ashkenas, 2010] and others, all follow different methodologies, but share the characteristic of being pure client-frameworks, made to "live" in the browser. Even though it is still possible to develop complex data models within them, control over the state and whereabouts of data ends with the next server round-trip. This requires developers to implement and test comparable business logic on both ends independently, clearly violating the `Don't-Repeat-Yourself` (DRY) principle. Meteor solves this by taking an entirely different approach. The platform was designed from the beginning to support the consistent use of JavaScript code on both the server and client-side. This means that developers only have to implement the model, validation, and business logic once. Additionally, XHR-requests are completely hidden, thus achieving the effect of a direct connection between the application in the browser and the database.

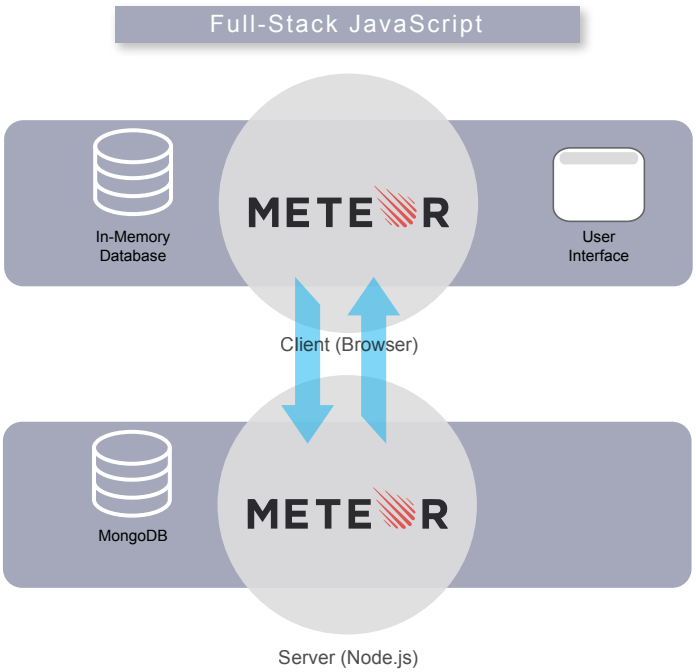

Figure 3.5: The "Database Everywhere" concept. Meteor keeps a subset of the database in the browser's memory, which makes for a far better application performance and user experience. [Adapted from the original by Greif [2014]. Redrawn by Sanja Ibraimović Ljajić and used with kind permission.]

3.5.1 Full Stack Reactivity – "Realtime by Design"

Meteor is a step forward when it comes to enhancing the communication between server and client, in that it offers `Publish/Subscribe` control mechanisms. Furthermore, through the WebSocket protocol [IETF, 2011], browsers are able to maintain bidirectional connections with the server: The server informs Meteor of an event, a change in the database, which results in the platform re-rendering the views (live update). `Publish/Subscribe` declarations also enable flexible, event-driven couplings of all involved components, according to the `Observer` design pattern. In other words, every part of an application – from the database up to the HTML-templates – can react to changes out of the box. This interplay is called "Full Stack Reactivity" by the Meteor team. Applications using Meteor are diverse, including chat platforms, admin dashboards, and stock exchange monitors.

3.5.2 Database Everywhere and Latency Compensation

One of Meteor's biggest innovations is what the Meteor team calls "Database Everywhere". Meteor takes the server-side database, or more usually a subset of it, and duplicates it on the client, in the browser's memory. So instead of doing the whole round-trip from the client to the server to the database, the data already resides in the client (see Figure 3.5), ready to be changed and displayed, which makes for much faster applications and a far better user experience.

When the client notices a change, it is first applied against the client-side `Minimongo` instance. The client then propagates the change down the stack, using Meteor's `Distributed Data Protocol` (DDP). If the server accepts it, all connected clients, including the one that made the change, are informed about it. In case the change is rejected, or a newer one has come in, the `Minimongo` instance on the client is

corrected, and any affected UI elements are updated as a result. The important thing here is that round-trip latency is avoided by not having the client wait until the response has come back from the server. The UI can immediately be updated, based on what the `Minimongo` instance holds, and should the change in any way be illegal, the client can correct it as soon as the word from the server arrives [Strack, 2015].

3.5.3 Getting Started with Meteor

Similar to other frameworks, Meteor supports OS X, Windows, and Linux, but unlike them, it comes bundled with Node [NF, 2009], npm [npm, 2010] and everything that is necessary to run it. The installation process merely required running the following command:

```
$ curl https://install.meteor.com/ | sh
```

or downloading and executing the official Meteor installer on Windows. If for some reason installing Meteor locally is not an option, it is possible to develop and run a Meteor application on directly in the browser using Nitrous.io [Nitrous, 2016; Coleman and Greif, 2015].

Once Meteor is installed, creating a project and running it locally requires just a few commands:

```
$ meteor create meteor-app
$ cd meteor-app
$ meteor npm install
$ meteor # Meteor server running on: http://localhost:3000/
```

Unlike other frameworks, Meteor does not rely on Express [NF, 2010] to provide routing functionality, but instead offers the possibility to choose one of many community-provided routers. In fact, a variety of different functional features [Percolate Studio, 2016] can be added to a Meteor project by running the `add` command. The following command will install the Iron Router [Mather, 2013]:

```
$ meteor add iron:router
```

To further facilitate the techniques and concepts of Meteor development, an example application is provided with a complete tutorial on how to build it from scratch. The application implements all the essential features of a collaborative task management app, in less than 600 lines of JavaScript [MDG, 2012]. It's called "Todos" and can be created by executing:

```
$ meteor create --example todos
```

3.5.4 Conclusion

Meteor is a significant step toward the simplification of development and operation of modern, scalable web applications. The platform deliberately targets the upcoming generations of data and event-driven real-time systems. Since Meteor is still a relatively young framework, there are a few open questions remaining, one of which is database support. Currently, Meteor only works with MongoDB [MongoDB, 2009] at the persistence level, but there are promises from the developer team to provide support for further databases.

Learning and starting to work with Meteor can be somewhat challenging at the beginning, given that Meteor is a full-featured framework, resembling giants such as .NET [Microsoft, 1985] and JavaEE [Oracle, 1999]. On the other hand, Meteor comes with extensive online documentation and tutorials tailored for several different frontend frameworks. Along with example applications and thriving community support, beginning a development project in Meteor should be relatively straightforward.

Chapter 4

The React Frontend JavaScript Framework

React [Facebook, 2013] is a JavaScript library for building user interfaces. It was published in 2013 under the BSD 3-Clause License [OSI, 1988] by Facebook, and has since created much interest in the JavaScript community. React is used by Facebook, Whatsapp, Yahoo, AirBnB, Atom editor, and many others [Facebook, 2016].

jQuery [Resig, 2006] dominated the JavaScript development landscape for many years. Even though the web has undergone constant development, the basic requirements of a web page remained the same for a long time. The expansion of mobile and web apps started to drive the development of frameworks which could meet these new requirements. One of the oldest and most popular such frameworks was Google's Angular [Google, 2010]. Angular recognised the root problem as being that HTML was not designed for dynamic views, and addressed it by enabling the developer to extend the HTML vocabulary for the needs of their application. Besides Angular, many other frontend frameworks appeared, offering solutions for bundling and structuring the important and most often used functions. Nonetheless, Angular remains very popular [Heller, 2016].

The React framework [Facebook, 2013] appeared recently, and has quickly become a strong alternative to Angular, because of some controversial decisions the React team made: implementing a virtual DOM, defining the UI in JavaScript rather than templates, and inventing a superset of JavaScript called JSX which supports the embedding of HTML-like tags directly in JavaScript code. These decisions, however, combine together to create an environment in which the developers write code which describes how the DOM is supposed to look "right now", rather than writing code to manipulate the DOM. Behind the scenes, React does the hard work of figuring out what manipulations are needed to make the DOM actually look that way [Shore, 2014].

4.1 The React Virtual DOM

The main problem with the DOM is it was never optimised for creating dynamic UIs. Unlike traditional websites, modern web applications are usually built into a single web page (Single-Page Application – SPA), and thus can have tens of thousands of nodes in the DOM tree. Interacting with such web applications efficiently is a huge problem.

The virtual DOM is an abstraction of the HTML DOM. It is lightweight and is detached from browser-specific implementation details. React is fast because it maintains a fast, in-memory representation of the DOM, and hence never talks to the DOM directly. Components' `render` methods return a "description" of the DOM, which React compares with the in-memory representation to compute the fastest way to update the browser, as illustrated in Figure 4.1.

React implements a full synthetic event system such that all event objects are guaranteed to conform to the UI Events Specification [W3C, 2015] despite browser differences. Furthermore, even some

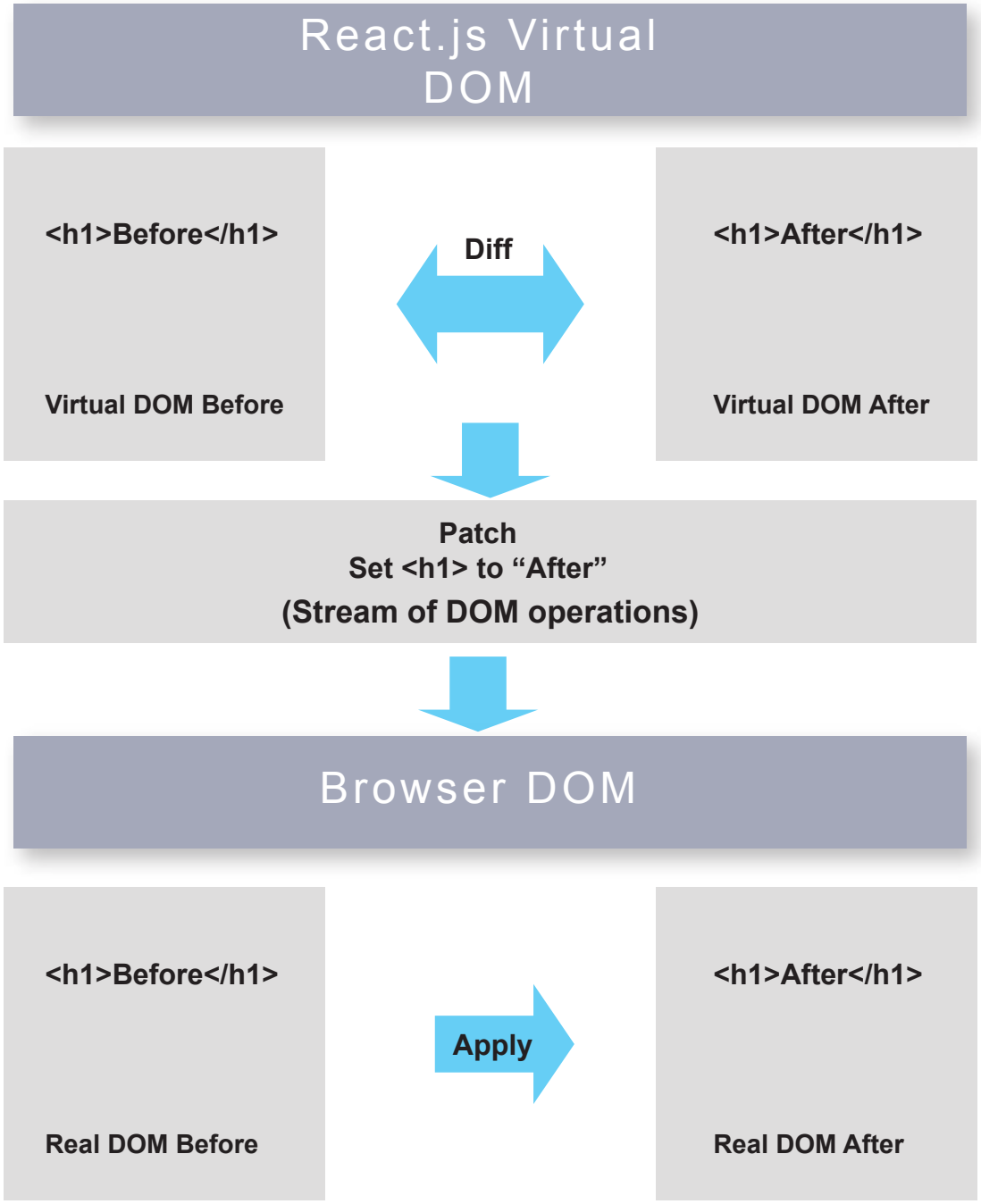

Figure 4.1: The virtual DOM comparison and update process. React compares the "new" with the "old" virtual DOM to pinpoint the changes. When located, only the affected elements in the real DOM are updated.
[Adapted from the original by Hollidge [2014]. Redrawn by Sanja Ibraimović Ljajić and used with kind permission.]

newer HTML5 elements become available in older browsers, even though they would ordinarily not be supported [Facebook, 2013].

4.2 React Components

The central and only building blocks in React are the components. Conceptually, components are to React, what directives are to Angular. Each React component must at least contain the `render` method, where the markup for the component's DOM representation is defined. This markup can either be a virtual representation of a native DOM component, or another custom-defined component [Facebook, 2013].

React components can receive external data through attributes, which are specified when the component is used, and are made accessible within the component through the `this.props` object. Components can also maintain internal state by initialising the `this.state` object. Whenever the `this.state` object changes, the component is re-rendered.

4.3 JSX

JSX is a thin syntactic abstraction of JavaScript function calls, which remarkably resembles the HTML syntax. JSX was for many the first stumbling block, when React first appeared. Why use an XML-like syntax, instead of a customary templating language? And even more importantly, why suddenly write markup in a JavaScript file, after many years' practice of HTML, CSS, and JavaScript separation?

For the authors of React, however, modularisation by technologies was not a criterion for a meaningful "separation of concerns" [Hunt, 2013]. Rather, components should be used as a basis for separation of different aspects of an application, and consequently, everything that belongs to a component belongs in the same file, regardless of whether it is HTML, JavaScript, or CSS. This makes JSX exactly the opposite of the "logicless templates" approach which templating languages like Handlebars [Katz, 2010] endorse.

Chapter 5

Sortit

`Sortit` is an online card sorting application, developed in JavaScript. Card sorting managers can register, create a card sorting project, and distribute its URL to potential sorters (study participants). A sorter enters the URL and an ID provided to them, and can then perform a sort, leave comments, pause the sorting, and continue at a later time. Completed sorts are made available for download and analysis.

This chapter will cover the requirements, design, and implementation details of the backend part of the `Sortit` application.

5.1 User Roles

`Sortit` has three types of user: admins, project managers, and sorters. `Sortit` has the following corresponding home pages:

- Application home page.
- Admin home page for each admin.
- Manager home page for each manager.
- Card sorting page for each sorter.

5.1.1 Admin Functionality

An administrator (or admin) of a particular `Sortit` installation handles global settings and user registration.

5.1.2 Project Manager Functionality

A project manager sets up and manages particular card sorting project. The functionality available to a project manager includes:

- Creating a card sorting project. A project manager needs to be able to create a new card sorting project. An interface allows the manager to specify the project's name and upload or enter the concepts for the cards. Each card typically has a title, subtitle, and an ID (typically a number). A URL for the sorting project is then created.
- Activating and deactivating a card sorting project. A card sorting manager needs to be able to mark a certain project as inactive, so that running further sorts for that project is disabled.

- Accessing the project's dashboard. A card sorting manager can gain an overview of a project's state by means of a project dashboard page displaying statistics such as the number of completed sorts, number of incomplete sorts, activation date, etc.
- Downloading the `Sortit` data for analysis and archiving.
- In a future phase of `Sortit` various analysis functions will be added

After successfully logging in, a manager is redirected to their home page, which provides an overview of their sort projects, and options for managing them. In case a manager is already logged in when accessing the application's root URL, they are automatically redirected to their home page.

5.1.3 Sorter Functionality

Each sorter receives a publicly accessible URL for a particular card sorting project. The sorter must identify themselves by entering a unique ID assigned to them. This ID is important for two reasons:

1. To be able to distinguish sorters during analysis.
2. In case the sorter wants to pause the sort, they are able to continue the sorting session at a later time using the ID.

A set of cards is provided for each sort, and the sorter is able to group these cards together by dragging and dropping. The sorter can name, rename, and delete the groups, and eventually submit the sort. It is also possible to leave a comment during a sort, which is important to help understand the reasoning behind certain sorting decisions.

Every action the sorter makes during sorting is automatically recorded to prevent data loss. The sorter does not need to explicitly save current progress. This also makes it possible to replay sorting sessions at a later time.

5.2 Software Architecture

`Sortit` relies on the Meteor [MDG, 2012] framework to provide the necessary functionality in the backend. At the frontend the React [Facebook, 2013] framework is used. Redux [Abramov, 2015] is used to manage the application state. Pince [MadEye, 2013] provides logging functionality throughout the frontend and the backend. The software architecture of the `Sortit` application is illustrated in Figure 5.1. The `Sortit` application stack is shown in Figure 5.2. The `Sortit` data model is shown as an entity-relationship diagram in Figure 5.3.

Meteor was chosen as the development framework for `Sortit` because it is a fairly mature, stable, and very robust framework, which comes pre-bundled with all the technologies needed for full-stack application development. In particular, it eliminates the need to set up a development environment, which can otherwise be tedious and time-consuming.

5.2.1 User Accounts

Meteor has several community-contributed packages providing out-of-the-box functionality which extend the core `useraccounts` module. One such package is the `accounts-password` [Meteor, 2016b] – a login service that enables secure password-based login. `useraccounts` and `accounts-password` provide the following functionality:

- User registration.

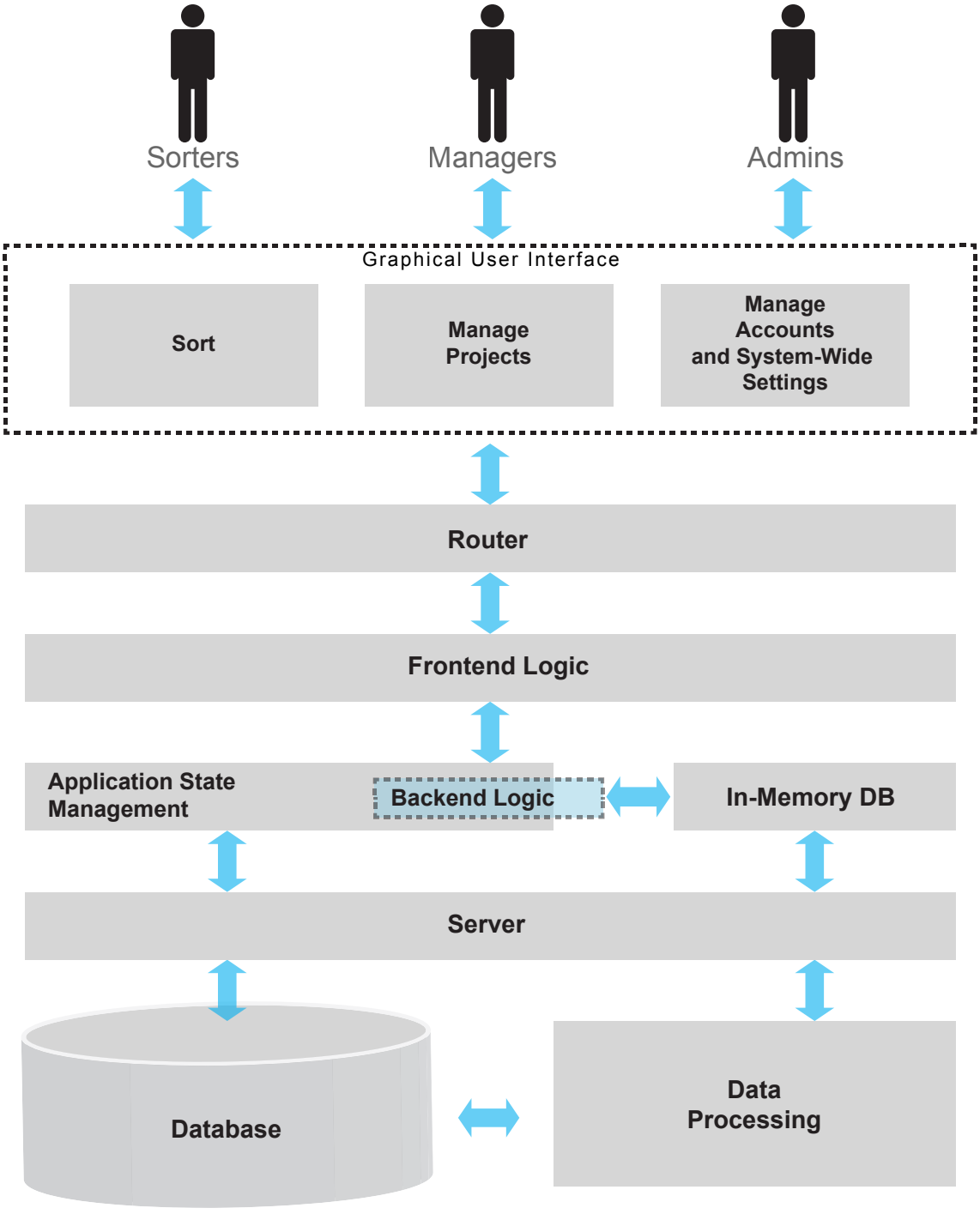

Figure 5.1: The software architecture of the `Sortit` application. [Drawn by Sanja Ibraimović Ljajić and used with kind permission.]

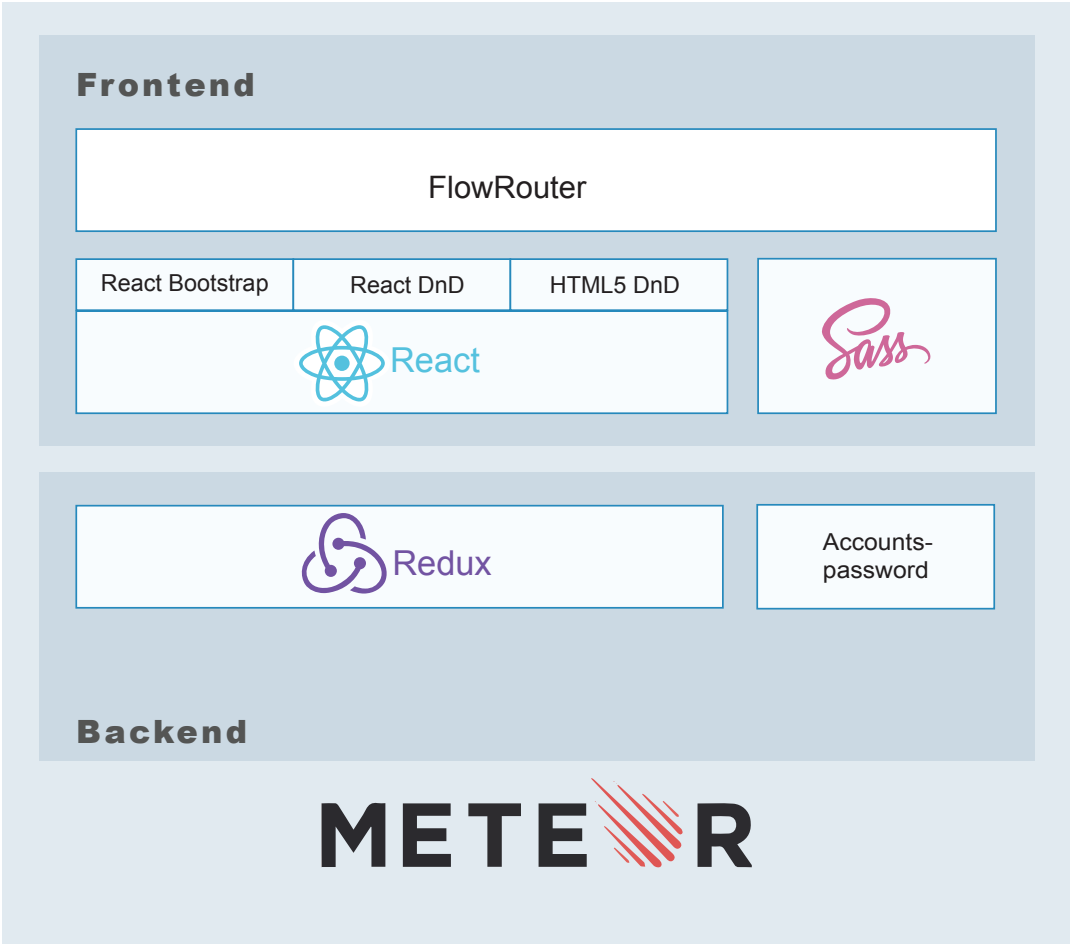

Figure 5.2: The `Sortit` application stack. [Drawn by Emil Ljajić and used with kind permission.]

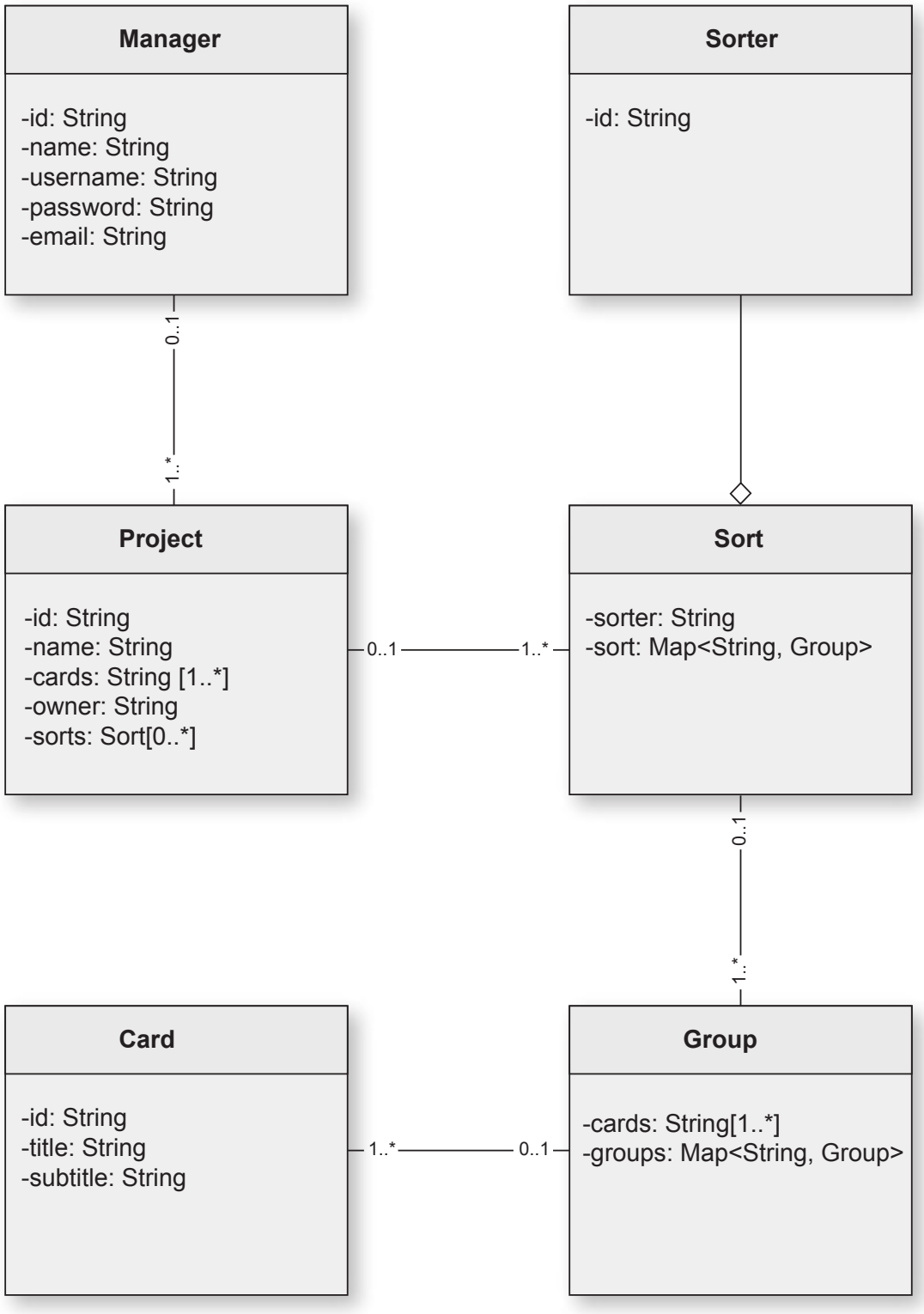

Figure 5.3: The `Sortit` entity-relationship model. [Drawn by Emil Ljajić and used with kind permission.]

```
{
    id: "string",
    name: "string",
    cards: [
        {
            title: "string",
            subtitle: "string",
            id: "string"
        }
    ],
    owner: "string",
    sorts: [
        {
            sorter: "string",
            sort: {
                <group-name>: {
                    cards: [],
                    groups: {
                        name: "string",
                        cards: [],
                        groups: {}
                    }
                }
            }
        }
    ]
}
```

Listing 5.1: The MongoDB document schema used for persisting a Sortit project.

- Secure user login.
- Session management.

`accounts-password` requires no additional setup, and is ready to use upon installation.

5.2.2 Data Persistence

Meteor uses MongoDB as its database of choice. This is well-suited to `Sortit`, since application state snapshots, which get created on every user action during sorting, are kept in a JSON-like structure, which is exactly what MongoDB documents look like. The MongoDB document schema used for persisting `Sortit` projects and sorts into the database is illustrated in Listing 5.1.

When created, each project is persisted under a unique ID, which is also embedded in the card sorting page URL. This ID is created using the `meteor-node-uuid` [Sewell, 2014] package.

5.2.3 Application State Management

Redux [Abramov, 2015] is used hold and manage application state. It serves as an additional layer between React on the client-side and Meteor to provide a kind of a facade to the frontend for exchanging application data with the server. Redux is “a predictable state container for JavaScript applications”. It is a very lightweight implementation of Flux [Facebook, 2014]. Redux works by defining three mandatory elements:

```
Actions.getProject = function () {
    var project = Projects.findOne();
    LOG.debug('Project from DB:', project);

    if (project && project.cards) {
        return {
            type: 'GET_PROJECT',
            cards: project.cards
        };
    } else {
        return {
            type: 'PROJECT_NOT_FOUND'
        };
    }
};
```

Listing 5.2: The Sortit GET_PROJECT action. When a sorter accesses the card sorting page, GET_PROJECT is executed.

- a `store`: a global state container which holds the current application state,
- `actions`: which define the application behaviour, and
- `reducers`: which generate a new copy of the application state, based on the action.

When an action is dispatched, the store passes the action to the corresponding reducer, which generates a new copy of the state based on the action [Konkle, 2015].

The Redux `store` is an immutable JavaScript object, created by invoking the `createStore` function. In addition to the three mandatory elements for managing the application state, `Sortit` also uses `middleware`, "created by composing functionality that wraps separate cross-cutting concerns which are not part of the main execution task" [Eagle, 2015]. In `Sortit`, `middleware` performs logging and helps with debugging the Redux code. Since `middleware` are also used, the store is not created directly but through the `applyMiddleware` function. The code is in the `store.jsx` file, which resides in the `redux` folder.

The actions which modify the application state are:

- `GET_PROJECTS`: modifies the state to include all projects created by the currently logged-in user. The definition of `GET_PROJECT` action is shown in Listing 5.2.
- `GET_PROJECT`: adds the requested project to the application state object.
- `PROJECT_NOT_FOUND`: in case the project is not found in the database, an empty cardset is added to the application state.
- `GET_USER`: adds the currently logged-in user to the application state object.
- `LOAD_SORT`: when a user wants to continue sorting, the `LOAD_SORT` action adds the unfinished sort to the application state object.

`Sortit` also uses several "pseudo" actions, which are not handled by reducers since they do not change the state. These actions have a debugging role to help determine which of the state-changing actions could not be executed, or to signal that an action (usually in the backend) was performed which does not modify the state. The pseudo actions are:

```
switch (action.type) {
    case 'GET_PROJECTS':
        state.set('projects', action.projects);
        return state;
    case 'GET_PROJECT':
        state.set('cards', action.cards);
        return state;
    case 'GET_USER':
        state.set('user', action.user);
        return state;
    case 'PROJECT_NOT_FOUND':
        state.set('cards', []);
        return state;
    case 'LOAD_SORT':
        state.set('snapshot', action.snapshot);
        return state;
    default:
        return state;
}
```

Listing 5.3: Sortit reducers. The Redux store supplies each action to the reducers, which then modify the state accordingly.

- `UNEXPECTED`: fires in place of `GET_PROJECTS` when a user is not logged in,
- `NO_ACTION_LOAD_SORT`: fires if `LOAD_SORT` cannot be performed (i.e. there is no sort saved for a user),
- `NO_ACTION_SAVE_SNAPSHOT`: notifies that an application state snapshot is being persisted.

All actions are defined in the `action_creators.jsx` file.

Reducers handle actions by returning the modified application state. Reducer definitions are located in the `reducers.jsx` file. Listing 5.3 shows the characteristic `switch` statement where actions are examined.

5.2.4 Routing

Routing maps URLs to corresponding application functions. The `Sortit` web application separates private routes, accessible only by logged-in users, from publicly accessible routes. This functionality is provided by `FlowRouter` [Kadira, 2014], a very simple router for Meteor, developed with performance in mind. It focuses only on routing for client-side applications and does not handle rendering.

Because of the JavaScript's event-based architecture, when the card sorting page is requested, the program execution does not wait until the sorting project is retrieved from the database, but immediately proceeds to, in this particular case, render the page. However, to properly render the page, data from the database is needed. In such a situation, a callback must be registered, which will be executed once the data arrives from the database and supply the data to the page for rendering. Meteor's `Tracker` automatically reruns templates and other computation whenever `Session` variables, database queries, and other data sources change.

Routing code resides in the `Router.jsx` file. Since routing is mainly a concern of the frontend part of the application, it will not be discussed further here.

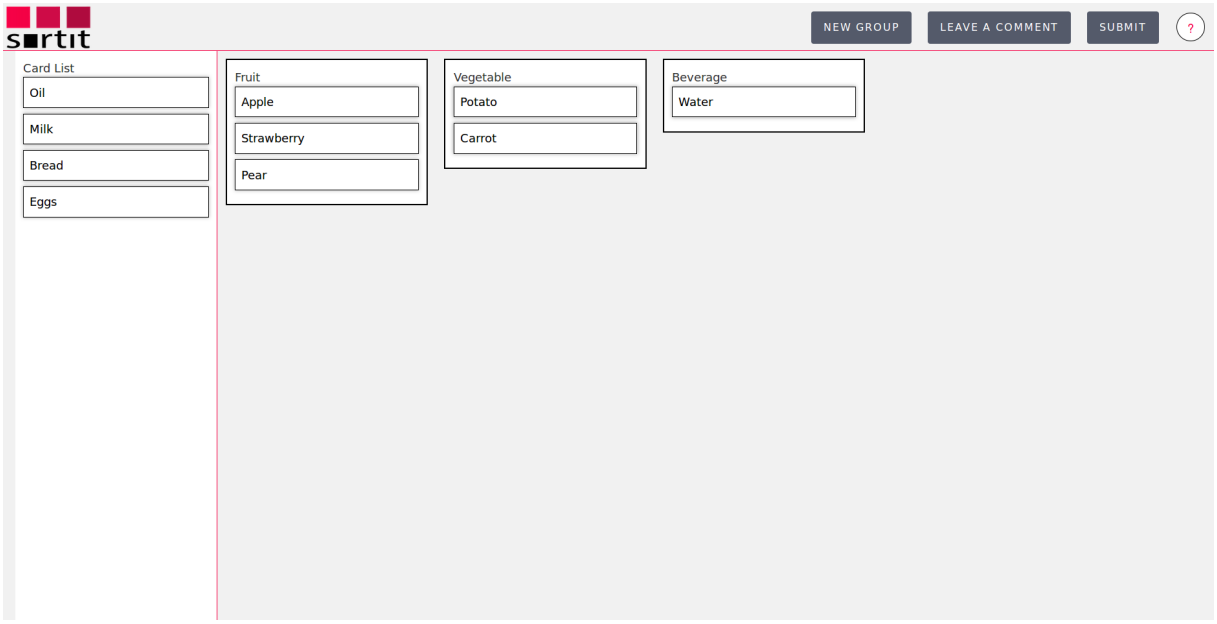

Figure 5.4: The user interface of the prototype frontend. [Screenshot taken by the author of the thesis.]

5.2.5 Prototype User Interface

User interface development is beyond the scope of this thesis, but in order to properly develop the backend, it was beneficial to build a frontend prototype. The frontend of `Sortit` is developed using React [Facebook, 2013]. React makes modularisation and separation of concerns exceptionally simple through its components. Furthermore, since React is a very popular technology, many supporting component libraries have been developed. One such library is React DnD [Abramov, 2014], which helps build complex drag-and-drop interfaces, and is used in the prototype implementation. The prototype relies on the HTML5 Drag and Drop API [W3C, 2011] to provide the event-based drag-and-drop mechanism. However, HTML5 Drag and Drop API does not come with support for touch events, which are necessary if the application is to work on touch devices. Fortunately, the third-party Touch Backend for React DnD [Yahoo, 2015] provides drag-and-drop functionality for touch devices, and to use it only requires replacing the `react-dnd-html5-backend` with the `react-dnd-touch-backend`. The user interface of the prototype frontend is shown in Figure 5.4.

5.3 Sortit Project Structure in Meteor

Following Meteor best practice, the `Sortit` project is structured with server and client code separation in mind. As illustrated in Figure 5.5, there are five directories residing in the project's root folder:

- `client`: All frontend code, including markup, styling, and React components.
- `lib`: Code which runs on both server and client. This is where internal data types are defined, Redux code resides, and Meteor methods are written.
- `node_modules`: Libraries defined in the `package.json` file are installed here by `meteor npm install` commands.
- `public`: Static files, such as images, that are served directly.
- `server`: Server related code, such as Meteor publish definitions. Publish statements specify which

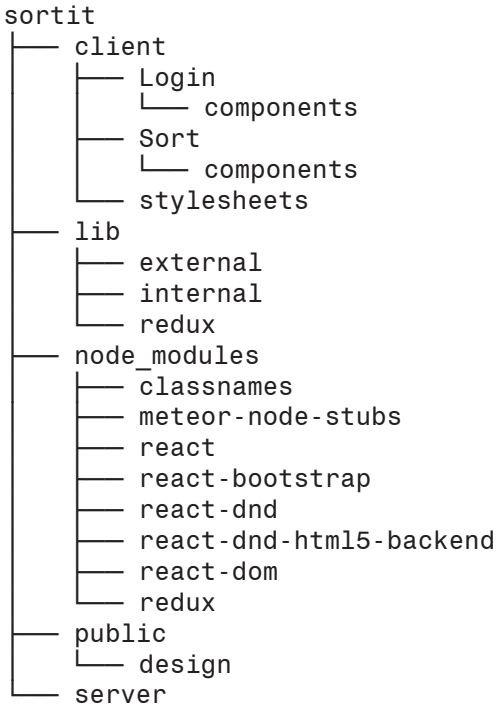

Figure 5.5: The `Sortit` project directory structure. [Drawn by Sanja Ibraimović Ljajić and used with kind permission.]

```
1 import {Meteor} from 'meteor/meteor';
2
3 Meteor.publish('project', function (projectId) {
4     return Projects.find({id: projectId});
5 });
```

Listing 5.4: The publish statement for a card sorting project. When a sorter accesses the card sorting page, the statement instructs Meteor to copy the part of the database containing the card sorting project to browser memory.

parts of the database are replicated on the client. The publish statement which is executed when a sorter accesses the card sorting page is shown in Listing 5.4.

5.4 Installing and Running Sortit

The `Sortit` source code is currently hosted on a private GitLab server. Assuming access is available to the repository, the following commands will download the sources, setup the development environment, and start `Sortit` on a local machine:

```
$ git clone git@git.iicm.edu:kandrews/sortit.git
$ cd sortit
$ meteor npm install
$ meteor
```

At this stage, the application can be further developed. In case Meteor is not installed, refer to Chapter 3.5 for installation instructions.

Once the application reaches the point where it is ready to be deployed, `meteor build` will create a deployment bundle which contains a plain Node.js application. This means that the application can be hosted on any server which has Node installed (Meteor installation is not required). Prior to building the application bundle, any npm dependencies must be installed. Also, if the bundle is not being created on the target machine, the server architecture needs to be specified during the build. The following commands will build the application bundle for deployment to an Ubuntu Linux server:

```
$ npm install --production
$ meteor build sortit --architecture os.linux.x86_64
```

This will generate a bundled application `.tar.gz` which can be extracted and run using Node. Since Meteor is no longer used, it is important to make sure the correct Node version is installed:

- Node `4.4.7` for Meteor `1.4.x`, or
- Node `0.10.43` for Meteor `1.3.x` and earlier.

The application can now be started by invoking `node`, and specifying:

- `ROOT_URL`: The base URL of Sortit.
- `MONGO_URL`: A Mongo connection URI.

The commands are:

```
$ cd sortit
$ (cd programs/server && npm install)
$ MONGO_URL=mongodb://localhost:27017/sortit ROOT_URL=http://sortit.com
    node main.js
```

Further information on deploying Meteor applications can be found on the Deployment and Monitoring page on Meteor's website [Meteor, 2016a].

In a production environment, `Sortit` runs independently of Meteor, so managing database backups would also need to be performed separately. The `mongodump` and `mongorestore` utilities work with BSON data, and are useful for creating backups of `Sortit`-sized deployments.

To backup the database, the following command is required:

```
$ mongodump --host sortit.com --port 3017 --username user --password
   pass --out /opt/backup/mongodump-YYYY-MM-DD
```

To restore a database dump, the following command is used:

```
$ mongorestore --host sortit.com --port 27017 --username user --
   password pass /opt/backup/mongodump-YYYY-MM-DD
```

More details on MongoDB backup methods can be found in the MongoDB documentation [MongoDB, 2016].

Chapter 6

Concluding Remarks

This thesis presented work to implement an online card sorting application. Chapter 2 introduced the topic of card sorting and examined already existing, web-based card sorting applications. Chapter 3 surveyed some of the more prominent full-stack JavaScript frameworks, from the aspect of developing a card sorting application in one of them. The React UI JavaScript development framework was introduced in Chapter 4. Chapter 5 discussed the requirements, design, and implementation of the `Sortit` web application itself, with a particular focus on the backend.

Bibliography

Abramov, Dan [2014]. *React DnD*. 2014. http://gaearon.github.io/react-dnd/ (cited on page 35).

Abramov, Dan [2015]. *Redux*. 2015. http://redux.js.org/ (cited on pages 28, 32).

Andrews, Keith [2012]. *Writing a Thesis: Guidelines for Writing a Master's Thesis in Computer Science*. Graz University of Technology, Austria. 22 Oct 2012. http://ftp.iicm.edu/pub/keith/thesis/ (cited on page ix).

Ashkenas, Jeremy [2009]. *CoffeeScript*. 2009. http://coffeescript.org/ (cited on page 19).

Ashkenas, Jeremy [2010]. *Backbone.js*. 2010. http://backbonejs.org/ (cited on page 20).

Balderdash [2012]. *Waterline ORM*. 2012. https://github.com/balderdashy/waterline (cited on page 16).

Bitovi [2014]. *Embedded JavaScript*. 30 Nov 2014. http://www.embeddedjs.com/ (cited on page 16).

Bretz, Adam [2014]. *Full-Stack JavaScript Development With MEAN*. 22 Dec 2014. http://sitepoint.com/full-stack-javascript-development-mean/ (cited on page 11).

Coleman, Tom and Sacha Greif [2015]. *Building Real-Time JavaScript Web Apps*. 2015. http://discovermeteor.com/ (cited on page 22).

Daig, Cody B. and Roie Cohen [2016]. *meanjs/mean: MEAN.JS - Full-Stack JavaScript Using MongoDB, Express, AngularJS, and Node.js*. 2016. https://github.com/meanjs/mean (cited on page 13).

Dejaeger, Glenn [2015]. *Comparing Angular, Aurelia and React: Is there a next-gen JS framework that rules them all?* 15 Dec 2015. http://www.ae.be/blog-en/comparing-angular-aurelia-react-js-framework/ (cited on page 20).

Derby [2011]. *Derby*. 2011. http://derbyjs.com/ (cited on pages 17–18).

Derby [2016a]. *Derby*. 2016. http://derbyjs.github.io/derby/ (cited on pages 18–19).

Derby [2016b]. *DerbyJS*. 2016. http://derbyjs.com/docs/derby-0.6 (cited on page 19).

Eagle, Mark [2015]. *Understanding Redux Middleware*. Medium. 26 Aug 2015. https://medium.com/@meagle/understanding-87566abcfb7a#.8tr8c05hz (cited on page 33).

Facebook [2013]. *React*. 2013. https://facebook.github.io/react/ (cited on pages 1, 23, 25, 28, 35).

Facebook [2014]. *Flux*. 2014. https://facebook.github.io/flux/ (cited on page 32).

Facebook [2016]. *Sites Using React*. GitHub. 2016. https://github.com/facebook/react/wiki/Sites-Using-React (cited on page 23).

Fomin II, Slava [2015]. *Sailing With Sails.js: An MVC-style Framework For Node.js*. Smashing Magazine. 24 Nov 2015. https://smashingmagazine.com/2015/11/sailing-sails-js-mvc-style-framework-node-js/ (cited on page 16).

Google [2010]. *AngularJS*. 2010. https://angularjs.org/ (cited on pages 12–13, 20, 23).

Google [2016]. *Dependency Injection*. 2016. https://docs.angularjs.org/guide/di (cited on page 13).

Greif, Sacha [2014]. *Introduction to Meteor - April Devshop SF*. Meteor Development Group Inc. 02 May 2014. https://youtu.be/q9pA2xApdY0 (cited on page 21).

Harris, Amber [2013]. *The Birth of Node: Where Did it Come From? Creator Ryan Dahl Shares the History*. siliconANGLE. 01 Apr 2013. http://siliconangle.com/blog/2013/04/01/the-birth-of-node-where-did-it-come-from-creator-ryan-dahl-shares-the-history/ (cited on page 11).

Haviv, Amos [2014]. *MEAN.JS*. 2014. http://meanjs.org/ (cited on pages 12–13).

Heller, Martin [2016]. *Beyond jQuery: An expert guide to JavaScript frameworks*. 02 Mar 2016. http://infoworld.com/article/3039817/application-development/beyond-jquery-an-expert-guide-to-choosing-the-right-javascript-framework.html (cited on page 23).

Hollidge, Steven [2014]. *React.js Concepts*. Insights of a Full Stack Developer. 30 Nov 2014. http://stevenhollidge.blogspot.co.at/2014/11/reactjs-concepts.html (cited on page 24).

Hunt, Pete [2013]. *React: Rethinking Best Practices - JSConf EU 2013*. JSConf. 30 Oct 2013. https://youtu.be/x7cQ3mrcKaY (cited on page 25).

IETF [2011]. *The WebSocket Protocol*. Dec 2011. https://tools.ietf.org/html/rfc6455 (cited on page 21).

Jenkov, Jakob [2015]. *Dependency Injection*. 05 Jul 2015. http://tutorials.jenkov.com/dependency-injection/index.html (cited on page 13).

Kadira [2014]. *FlowRouter*. GitHub. 2014. https://github.com/kadirahq/flow-router (cited on page 34).

Katz, Yehuda [2010]. *Handlebars*. 2010. http://handlebarsjs.com/ (cited on pages 19, 25).

Konkle, Brandon [2015]. *State Management with Redux*. 05 Sep 2015. http://konkle.us/state-management-with-redux/ (cited on page 33).

MadEye [2013]. *Pince*. 2013. https://github.com/mad-eye/pince (cited on page 28).

Mardan, Azat [2014]. *Pro Express.js*. Apress, 22 Dec 2014. ISBN 1484200381 (cited on page 13).

Mather, Chris [2013]. *Iron Router*. 2013. https://iron-meteor.github.io/iron-router/ (cited on page 22).

McNeil, Mike [2012]. *Sails*. 2012. http://sailsjs.org (cited on pages 15, 17).

MDG [2012]. *Meteor*. Meteor Development Group Inc. 2012. https://meteor.com/ (cited on pages 20, 22, 28).

Meteor [2016a]. *Deployment and Monitoring*. 2016. https://guide.meteor.com/deployment.html (cited on page 37).

Meteor [2016b]. *Users and Accounts*. 2016. https://guide.meteor.com/accounts.html (cited on page 28).

Microsoft [1985]. *.NET*. Microsoft. 1985. https://microsoft.com/net (cited on page 22).

MongoDB [2009]. *MongoDB*. 2009. https://docs.mongodb.org/manual/introduction/ (cited on pages 12, 19, 22).

MongoDB [2016]. *MongoDB Backup Methods*. 2016. https://docs.mongodb.com/manual/core/backups/ (cited on page 37).

NF [2009]. *Node.js*. Node.js Foundation. 2009. https://nodejs.org/en/about/ (cited on pages 11–12, 22).

NF [2010]. *Express*. Node.js Foundation. 2010. http://expressjs.com/ (cited on pages 12–13, 22).

Nitrous [2016]. *Nitrous.io*. 2016. https://nitrous.io/ (cited on page 22).

npm [2010]. *Node Package Manager*. npm. 2010. https://npmjs.com/ (cited on page 22).

Optimal [2016]. *OptimalSort*. Optimal Product. 2016. https://optimalworkshop.com/optimalsort (cited on page 8).

Oracle [1999]. *Java Platform, Enterprise Edition*. Oracle. 1999. http://oracle.com/technetwork/java/javaee/overview/index.html (cited on page 22).

OSI [1988]. *The BSD 3-Clause License*. Open Source Initiative. 1988. https://opensource.org/licenses/BSD-3-Clause (cited on page 23).

OSI [1999]. *The MIT License*. Open Source Initiative. 1999. https://opensource.org/licenses/MIT (cited on pages 12, 15, 17, 20).

Percolate Studio [2016]. *Atmosphere*. 2016. https://atmospherejs.com/ (cited on page 22).

Rails [2016]. *Active Record Basics*. Rails Guides. 2016. http://guides.rubyonrails.org/active_record_basics.html (cited on page 18).

Resig, John [2006]. *jQuery*. 2006. https://jquery.com/ (cited on page 23).

Robinson, Josh, Aron Gray, and David Titarenco [2016]. *Intruducing Meteor*. Apress, 02 Jan 2016. ISBN 1430268360 (cited on page 20).

Sanfilippo, Salvatore [2009]. *Redis*. 2009. http://redis.io/ (cited on page 19).

SAP [2012]. *SAP Hybris*. Hybris AG. 2012. https://hybris.com/ (cited on page 16).

SCS [2016]. *Simple Card Sort*. 2016. http://simplecardsort.com/ (cited on page 5).

Sewell, Jason [2014]. *Meteor Node UUID Library*. 2014. https://github.com/jaywon/meteor-node-uuid (cited on page 32).

ShareJS [2011]. *ShareJS*. 2011. https://github.com/share/ShareJS (cited on page 18).

Shore, James [2014]. *An Unconventional Review of React*. Let's Code JavaScript. 22 Sep 2014. http://letscodejavascript.com/v3/blog/2014/09/react_review (cited on page 23).

Silva, João [2014]. *End-to-End JavaScript with the MEAN Stack*. 22 Dec 2014. https://github.com/joaopsilva/joaopsilva.github.io/tree/master/talks/End-to-End-JavaScript-with-the-MEAN-Stack/ (cited on page 12).

Spencer, Donna [2009]. *Card Sorting: Designing Usable Categories*. Rosenfeld, Apr 2009. ISBN 1933820071 (cited on pages 3–4).

Strack, Isaac [2015]. *Getting Started with Meteor.js JavaScript Framework*. 2nd Edition. Packt Publishing, 30 May 2015. ISBN 1785285548 (cited on page 22).

Tilde [2011]. *Ember*. 2011. http://emberjs.com/ (cited on page 20).

usabiliTEST [2016]. *Online Card Sorting Tool with Build-In Data Analytics*. 2016. http://usabilitest.com/CardSorting (cited on page 7).

W3C [2011]. *Drag and Drop – HTML5*. 2011. https://www.w3.org/TR/2011/WD-html5-20110525/dnd.html (cited on page 35).

W3C [2015]. *UI Events*. 2015. https://www.w3.org/TR/uievents/ (cited on page 23).

Wikipedia [2016]. *Create, Read, Update and Delete*. Wikipedia. 26 Jul 2016. https://en.wikipedia.org/wiki/Create,_read,_update_and_delete (cited on page 14).

Yahoo [2015]. *Touch Backend for react-dnd*. GitHub. 2015. https://github.com/yahoo/react-dnd-touch-backend (cited on page 35).