Yrkkö Äkkijyrkkä

Dynamic Web-Applications with Meteor.js

Helsinki Metropolia University of Applied Sciences

Bachelor of Engineering

Information Technology

Thesis

4 December 2015

Author(s) Title Number of Pages Date	Yrkkö Äkkijyrkkä Dynamic Web-Applications with Meteor.js 36 pages December 4, 2015
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Specialisation option	Software engineering
Instructor(s)	Simo Silander, Senior Lecturer

This thesis studies the viability of using Meteor.js in implementing a content management system by using the User Centred Design and Generic programming paradigm.

The goal of the project was to program generic visual templates and software modules to be used in the aggregation and displaying of user-created data.
The need for a content management system came from Aalto University Department of Architecture, which was in dire need for a more accessible way of interacting with their audience and the academic community.

During the development a great deal of care was placed on making the intranet side accessible to different user profiles, the common factor being the fact that the user base is stems from the architecture department or similar academic circles.

The author's knowledge and experience gained during studies and working as a freelance web-developer during the past 5 years were well applied in the project, as well as the know-how acquired at Conmio Oy building high concurrency web-applications that once served up to 100 million page hits per day.

Within this frame work, the study tries to objectively ascertain how production-ready Meteor actually is, how flexible it is when considering a particular software design and how well it performs when carrying out the design.

Keywords	Meteor, application, UCD, full-stack, JavaScript, generic, DRY, modular

Tekijä(t) Otsikko Sivumäärä Aika	Yrkkö Äkkijyrkkä Dynamic Web-Applications with Meteor.js 31 sivua 03.12.2015
Tutkinto	Insinööri AMK
Koulutusohjelma	Tietotekniikka
Suuntautumisvaihtoehto	Ohjelmistotekniikka
Ohjaaja(t)	Lehtori Simo Silander

Tämä lopputyö tutkii Meteor.js-sovelluskehyksen käytettävyyttä sisällönhallintajärjestelmän toteutukseen hyväksikäyttäen käyttäjäkeskeisen suunnittelun ja geneerisen ohjelmoinnin paradigmoja.

Projektin tavoitteena oli ohjelmoida geneerisiä HTML-malleja sekä ohjelmistomoduuleja, joita käytettäisiin sisällöntuotantoon sekä tiedon koostamiseen Aalto-yliopiston arkkitehtuurin laitoksen uudella internetsivustolla. Arkkitehtilaitos oli Aalto-yliopiston muodostamisen yhteydessä siirtänyt tiedotuksen uuden yliopiston keskitetylle tiedotusosastolle. Yhteistyö tiedotusosaston kanssa ei ollut toiminut, ja he tarvitsivat kipeästi oman tiedotuskanavan akateemiseen maailmaan.

Kehityksen aikana käytettiin paljon aikaa, että varmistettaisiin sisällönhallintajärjestelmän helppo käytettävyys ja lähestyttävyys erilaisille käyttäjille, joiden it-taitotaso ja käyttötarpeet ovat erilaisia, yhteisenä tekijänä kuitenkin arkkitehtilaitoksen yhteisö.

Projektissa pääsin käyttämään kaikkia tietoja ja taitojani, jotka olen hankkinut viimeisen viiden vuoden aikana toimiessani freelance-ohjelmoijana internetsivustojen parissa. Koin hyödylliseksi myös viimeisen parin vuoden aikana saamani kokemuksen Conmio Oy:n palveluksessa, missä tein mobiilisivustoja, joissa oli parhaimmillaan 100 miljoonaa kävijää päivässä. Tällä tausta-asetelmalla yritän opinnäytetyössä arvioida objektiivisesti, kuinka valmis Meteor on tuotantosovelluksiin, miten joustava se on ohjelmistoarkkitehtuurin suhteen ja kuinka hyvin se suorittaa tätä projektia varten suunniteltua arkkitehtuuria.

Kyseessä oli erittäin vaativa ja pitkä projekti, joka kesti maaliskuusta 2015 joulukuuhun 2015. Voin sanoa, että ainakin yksittäisten ohjelmistosuunnitteluun liittyvien ratkaisujen suhteen projekti oli vaativin, mitä olen koskaan tehnyt. Näin ollen, opin erittäin paljon prosessista ja kehityin valtavasti projektin aikana. Saavutin hyvin tavoitteet, joita asetin itselleni. Projekti paljasti myös Meteorin sovelluskehyksessä tiettyjä heikkouksia, jotka rajoittavat monimutkaisempien ohjelmistojen toteuttamista.

Avainsanat	Meteor, sovelluskehys, käyttäjäkeskeinen suunnittelu, DRY, WebSocket, JavaScript

Contents

List of Abbreviations

Javascript	A high level dynamic loosely typed programming language developed originally by Netscape Communications for creating rich content on the web, maintained now in co-operation by Mozilla foundation and Ecma international.
MongoDB	Document-oriented database management system. Documents are stored in BSON-format (binary JSON). Maintained and developed further by MongoDB Inc.
REST	Representational state transfer, de-facto standard interface of communication between applications in networks. Uses typically HTTP verbs for retrieving and storing data. Has been recently threatened with the increased use of sockets and other proprietary protocols.
Node.js	Application framework invented in 2009 by Ryan Dahl which uses JavaScript to implement an event-driven architecture and a non-blocking I/O API designed to optimize an application's throughput and scalability for real-time web applications.
MDG	Meteor Development Group is a privately funded organisation based in San Francisco but operating globally in the open-source community. It is the governing body directing Meteor development.
DDP	Distributed Data Protocol, is a simple protocol for fetching structured data from a server, and receiving live updates when that data changes.
WebSocket	is a protocol providing full-duplex communication channels over a single TCP connection. The WebSocket protocol was standardized by the IETF as RFC 6455 in 2011.
DOM	The Document Object Model is a platform- and language-neutral interface that will allow programs and scripts to dynamically access and update the content, structure and style of documents.

HTML	HyperText Markup Language, commonly referred to as HTML, is the standard markup language used to create web pages.
CSS	Cascading Style Sheets (CSS) is a style sheet language used for describing the presentation of a document written in a markup language.
DRY	In software engineering, don’t repeat yourself (DRY) is a principle of software development, aimed at reducing repetition of information of all kinds, especially useful in multi-tier architectures.
HTTP	The Hypertext Transfer Protocol (HTTP) is an application protocol for distributed, collaborative, hypermedia information systems.
TCP	The Transmission Control Protocol (TCP) is a core protocol of the Internet protocol suite. It originated in the initial network implementation in which it complemented the Internet Protocol (IP). Therefore, the entire suite is commonly referred to as TCP/IP.
NoSQL	A NoSQL (originally referring to "non SQL" or "non relational") database provides a mechanism for storage and retrieval of data that is modelled in means other than the tabular relations used in relational databases.
Angular	AngularJS (commonly referred to as "Angular" or "Angular.js") is an open-source web application framework mainly maintained by Google and by a community of individual developers and corporations to address many of the challenges encountered in developing single-page applications.
React	(sometimes styled React.js or ReactJS) is an open-source JavaScript library providing a view for data rendered as HTML.
dataURL	The data URI scheme is a uniform resource identifier (URI) scheme that provides a way to include data in-line in web pages as if they were external resources. It is a form of file literal or here document.

1 Introduction

Meteor is a modern full-stack web-application framework based on **Node.js** which is a back-end framework based on Google's V8 JavaScript engine. Meteor enables the programmer to hasten development speed by eliminating the use of REST interfaces from the server-client communication cycle. Instead of this de-facto standard communication architecture of web-applications, it is based on the technology by Meteor development group (henceforth MDG), called DDP that uses WebSocket-based communication between the client and server. In WebSocket-based communication database-connection between client (browser) must be kept alive during a DDP-session by sending small packets of data periodically. (MDG DDP-specification, 2015)

The approach is used to maintain *reactive programming* paradigm between a view in the browser and a database in the backend that enables Meteor applications respond to the changes in the collections, be it the collection in the server or the collection in the client used in the current context. Practically it means any changes to the attributes of the document are reflected on the view immediately. These updates reflect to all users of the application, so for example when someone creates new "post", it is pushed to all browsers that are listening for changes in the collection "posts". This is not always required or needed, but in scenarios where instant updates enhance the feeling of interactivity, it makes (with the additional help of directives called helpers in Meteor) manipulation of the HTML DOM much easier than with traditional document method calls. Developers can avoid tedious manipulation routines that are required on data addition and deletion. When the reactivity is not required, it can be turned off for rendering static content on the template context. Such use cases include the rendering of a single page, where it is not practical to change content concurrently as the user is reading it, and for example when building static elements of the layout, which are based on a dynamic data e.g. menus based on the aforementioned pages.

Through a maturing process, Meteor is now in version 1.2 that is stated by the MDG as *production-ready,* which means that it is considered to be stable enough for production of large-scale applications. Upon hearing that Aalto department of architecture were planning a major overhaul to their website and that it would be completed as a student project, the author offered his services to design the technical aspects of the project.

Throughout the process of outlining features required to be implemented on the site, it gradually became more evident that using Meteor would benefit the project most. Its fast development cycle and light implementation would help to attain the qualities necessary to satisfy the end-user (henceforth user) needs.

2 Case Aalto Department of Architecture

After the merger of the three major universities in 2010, the newly formed Aalto University has concentrated its maintenance of web site to a central it service. The department of architecture used to have its own web site until the merger, after which it became a subdomain of the Aalto University. However, having concentrated service has led to significant difficulties in the governing of the content of the website of the department. The content is being edited from the outside of the department by the central Aalto communication team. The consequence of this arrangement resulted in irrelevant or outdated information.

In spring 2015 the department of architecture launched this project in order to restore an important communication channel to the outside world. The main objectives of the new web site were to accurately communicate about happenings in the department, promote the department as a high standard educational entity and provide tools for internal management of the web site content. Particularly the communication about ongoing research projects and important events of the department was channelled only internally. In addition, the lack of key information about studies was exasperating to the staff of the department, as it was causing continuous inquires from the new students. Simplicity and unambiguity were the main focus areas in the design of the tools. The fluency of tools for editing was particularly important in order to maintain the high frequency of the updates.

3 Design Philosophy

When considering the user base and client, the design must reflect the values and image presented. Not talking only in terms of graphical design, but the omnipresent design of the product. It means that nothing in the application is by chance or without critical thinking applied first. Especially regarding the architecture department which has long history of being in the forefront of Nordic minimalism, it was crucial to reflect this idea on the technical implementation.

3.1 Generic

Generic design is applied thoroughly in the design and structure of the web-application by breaking down the content into domain models and using them to render context-aware templates that when possible, are agnostic about the data objects provided. In practice this means they will render only the values that can be found in the respective context data keys. By using identical keys on the models, one template can be used as ontological base to consolidate the render process.

The source code of the html is generated based on the ontological paradigm, which defines in this project terms such as “template”, “content” and “snippet”. All pages on the application comprise of these blocks, being built upon the instructions and context passed on by the base model, e.g. “template”.

An important feature of the application is the editor module which is generalized for file types and the database model being edited. Upon instantiation, it will initialize all necessary interface elements and render the content accordingly.

Using generalized content types, templates and helpers improve the programmer’s productivity and once the initial ontology is well defined, development of new features is a relatively unequivocal process.

3.1.1 Reusable

In accordance to the DRY programming paradigm, no single part of code should be repeated anywhere else in the codebase. Benefits from code reuse are fewer lines of code written, increased efficiency and scalability along with reduced complexity of the architecture. A statement can be also made that reduced codebase leads to fewer faults, as corroborated by Figure 1:

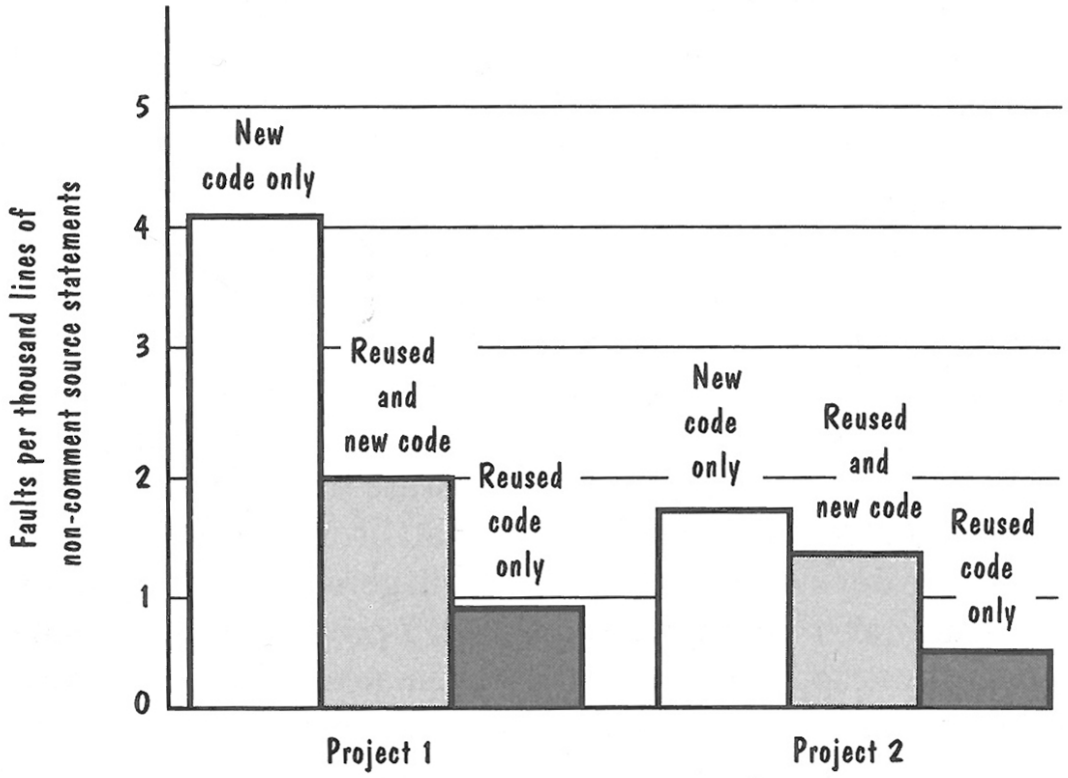

Figure 1. Case study results by Lim (1994) at Hewlett Packard

Arguments against code reuse include prolonged development time per component of software as the complexity of the function increases when it has to accommodate to different argument types, harder debugging, possibly compromising implementations of certain functionalities within the module and in the case of highly abstract modules decreased readability of the code for future developers.

3.1.2 Associative

One of the reasons for choosing Meteor as the development platform was that in order to have a flexible display of data, the application must use a NoSQL-type database. Conveniently Meteor uses MongoDB by default, hence providing a lot of freedom in designing the database structure.

To have an efficient aggregation and to retain flexibility, nesting the data inside the domain models as in composition would not be the right solution. By using the *association* pattern it would be possible to fetch initial data set much faster. And by implementing the *proxy* pattern for the related documents, the actual data is only fetched from the database just before they are loaded into the view; in other words, to the html template.

3.2 UCD

UCD, short for user-centred design is a framework of processes in software design that considers the users needs, abilities and limitations in all aspects of the product. ISO Ergonomics of human-system interaction (2015) defines UCD in a different appellation; Human centered design, in reality the two being the same concept:

> ISO 9241-210:2010 provides requirements and recommendations for human-centred design principles and activities throughout the life cycle of computer-based interactive systems. It is intended to be used by those managing design processes, and is concerned with ways in which both hardware and software components of interactive systems can enhance human–system interaction.

Adopting UCD establishes a frame of reference in which to operate on when considering design choices regarding user experience vs required features. All these requirements were refined by having several meetings and planning sessions with the personnel involved, which includes without limiting to: Professors, management, students and staff of the department.

To modularize content creation and following the unambiguous nature of the interface, the personnel using the intranet side of it would be divided into 3 intranet user groups:

- *Administrators*, who can do every action possible.
- *Editors* who are able to add and edit articles and student users.
- *Students*, who may upload their work for display on the site.

The professors would be the main users of the intranet side of the site and while quite competent using professional applications, the time investment in learning such application is considerable. What this means is that per UCD guidelines the application must conform to the user and not vice versa. Editors are meant to be engaged with the interface by having an immediate response, e.g. as soon as the edit is made, it will be published to the site and by having a very shallow learning curve in using the editor features. By forcing every edit to be substantial and not providing a separate publish function, all content must be polished and explicit from the beginning to the end.

Context-aware actions emphasize the immediacy of the interface, further improved by the performance gains demonstrated in Chapter 3.2.1 Context-awareness ensures user orientation, and was actualized by placing relevant actions always within, or in immediate proximity of the document that the action concerns.

The secondary main user group would be site visitors that are expected to find information as quickly as possible, providing rich but conspicuous navigation and an instant-search to find any page within seconds, by simply typing into the form first letters of the content desired and choosing from the results that appear directly under the search input. Additionally, filter and sorting are provided in pages which list larger sets of data, for narrowing the result set. In practise, it was equally important to maintain relatively flat structure of the sitemap to achieve usability in terms of navigation. By looking at Figure 2. the flat topology of the site becomes evident.

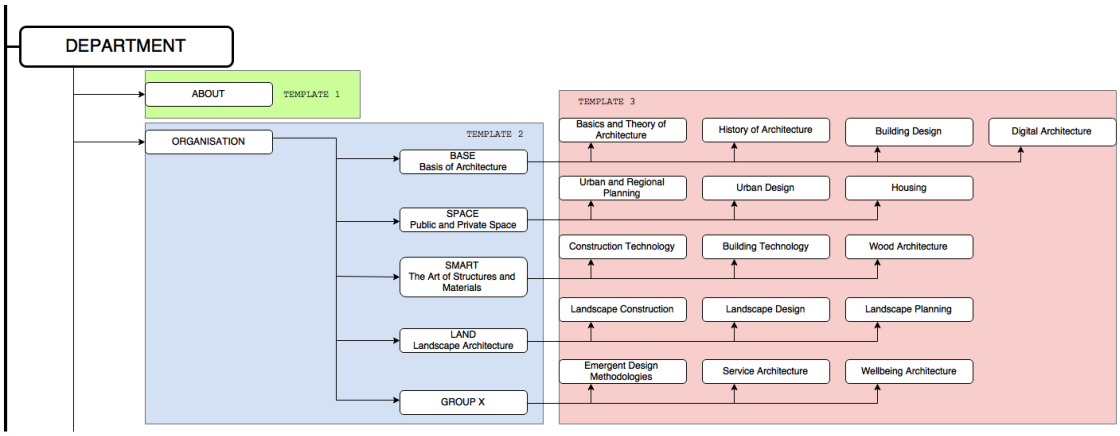

Figure 2. “Department” branch of the sitemap

Navigation topology is limited to 3 levels deep as illustrated in Figure 2. It is possible to navigate from any part of the site to another within 3 clicks. Each colour represents a template, hence the contained child objects are listed items and thereby the template represents a page for the user in practical terms.

User experience author Jakob Nielsen (2007) states that:

> The more engaged users are, the more features an application can sustain. But most users have low commitment -- especially to websites, which must focus on simplicity, rather than features.

Interpreted in the context of the site, one must strike the right balance between features, user experience and *simplicity* to keep the editors engaged and the content flowing. This directly affects the consumers of the content (i.e. visitors) whose engagement is directly associated to the relative informational value within the content.

3.2.1 Performance

For a great user experience, performance is always an important consideration. Nielsen (2010) points out that users engage more to a site when they can move about and enjoy the content rather than focus on waiting the results most of the time.

Nielsen (2010) also lists response-time limits that are based on human psychological traits that assists in defining what is a performant site:

- **0.1 seconds** gives the feeling of instantaneous response — that is, the outcome feels like it was caused by the user, not the computer. This level of responsiveness is essential to support the feeling of direct manipulation
- **1 second** keeps the user's flow of thought seamless. Users can sense a delay, and thus know the computer is generating the outcome, but they still feel in control of the overall experience and that they're moving freely rather than waiting on the computer. This degree of responsiveness is needed for good navigation.
- **10 seconds** keeps the user's attention. From 1–10 seconds, users definitely feel at the mercy of the computer and wish it was faster, but they can handle it. After 10 seconds, they start thinking about other things, making it harder to get their brains back on track once the computer finally does respond.

Acknowledging this, one can examine more closely an intrinsic part of Meteor; the DDP. Arun Gupta (2014) provides a number of arguments that promote WebSockets as a beneficial way of communicating between computers:

- **Bi-directional**: HTTP is a uni-directional protocol where a request is always initiated by client, server processes and returns a response, and then the client consumes it. WebSocket is a bi-directional protocol where there are no pre-defined message patterns such as request/response. Either client or server can send a message to the other party.
- **Full-duplex**: HTTP allows the request message to go from client to server and then server sends a response message to the client. At a given time, either client is talking to server or server is talking to client. WebSocket allows client and server to talk independent of each other.
- **Single TCP Connection**: Typically, a new TCP connection is initiated for a HTTP request and terminated after the response is received. A new TCP connection need to be established for another HTTP request/response. For WebSocket, the HTTP connection is upgraded using standard HTTP Upgrade mechanism and client and server communicate over that same TCP connection for the lifecycle of WebSocket connection.
- **Lean protocol**: HTTP is a chatty protocol, and as such there is a large overhead included with every request/response.

In the context of performance, the last point is most interesting. A regular HTTP header can be anywhere between ~200 bytes to over 2KB with the average being 700-800 bytes. (SPDY-whitepaper, 2015) now Mr. Gupta made some tests using 1000 bytes as a reference, and the results can be seen in Figure 3:

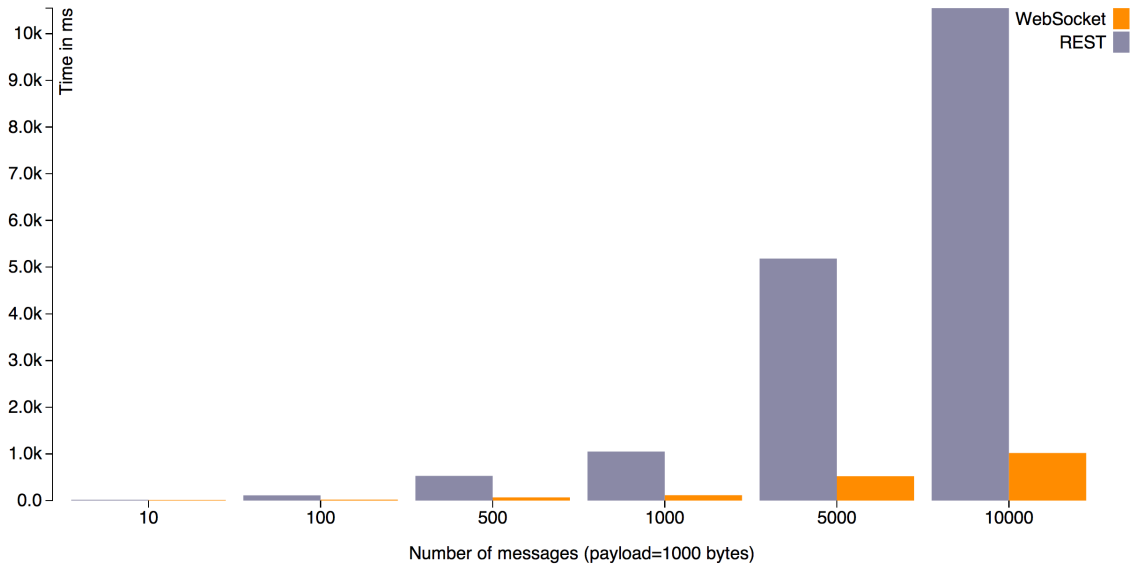

Figure 3. WebSocket vs. HTTP REST, equal payload Gupta (2014)

As can be seen in Figure 3, by saving bandwidth, response time and amount of data processed in each end, WebSocket messages can be dramatically more performant. Why is this great for the website? It means that Mr. Nielsen's guidelines can be followed and the site actually behaves more or less like a native application running only in the user's device.

There are still advantages in using HTTP such as caching, search engine optimization and security, all of which are rather poorly implemented in WebSocket-based communication.

3.2.2 Simplicity

Simplicity in this project is defined as economical use of visual elements and technical efficiency of data flows. Simplicity for aggregation of data means each page in the site is context-aware and will only display information relevant to the subject of the page. This is not limited to the current application as the relations can be re-associated later on in case there is need to combine the existing content in a new way. It enhances the *scalability* and is achieved by free association between the data models, enabled by using MongoDB, something that is not traditionally desired because of the clear threats projected upon integrity and validity of data. However, by using simple-schema it is possible to maintain the integrity of the data through double validation; primary on client-side and secondary on the server-side.

By simplifying the user interface and data context, it is possible to optimize the application by requesting only necessary data to the client data through the subscription model. Meteor will drop data from the client database accordingly after a necessary time period by unsubscribing publications that are not used anymore in the view context. Furthermore, by using indexes on the queried fields of the collection, it is possible to even further narrow down the amount of data needed to process by the database. The *publish-subscribe* paradigm is further discussed in Chapter 4.2.

3.2.3 Unobtrusive Interfaces

In the user interface it is desired not to display unnecessary JavaScript effects/animations or information not relevant to the user in order maintain a clutter-free visual layout.

Commitment on uncluttering the interface meant keeping in mind the technical competence of the user base when implementing required features. Analysing thoroughly the common use cases led to removing redundant buttons and directing user actions through the use of *modals*. It was also of highest priority to maintain a coherent activity by allocating only one action per button.

In the editor module the workflow is designed in such way as to always have clear direction for the next required action. When adding content, the relevant uploader will open a modal type container that cannot be closed without selecting an action to upload the file or discard it altogether.

For the editor the content creation cycle must be as straightforward as possible. All buttons and elements for the intranet side of the application are positioned in such a way that they will not interfere with the browsing experience. Most editor actions are reflected by direct, simple yet innocuous visual feedbacks about the actions being executed.

The user experience of an editor differs very little from the one of site visitor, as the application is aware of the user rights and abilities involved for each session. This enables showing only the actions relative to the current user, in essence if it cannot be seen, it cannot be done.

3.3 Responsive Web Design

The term responsive web design was first coined by Ethan Marcotte (2010) in his eponymous article: “responsive web design” in which he laid out the fundamental guidelines which have remained more or less the same: The web-application responds by disabling features or selectively showing, resizing or hiding elements from the view according viewport size of the device that requested a view and/or data from the application. Using relative sizing of elements helps to maintain consistent overall visual presentation of the interface, while re-positioning content enables efficient use of the viewing area. This ensures the best experience for the user as per intended by the designers of the application.

Responsive design can be implemented from scratch by writing all media queries as needed or by using existing frameworks. Some of the functionality these frameworks

offer in HTML mark-up, are CSS-classes and directives that activate upon meeting the required criteria provided by the device context.

This all means in some cases minor sacrifices for the overall design, in terms of performance, visual effects or functionality. However, these limitations are diminishing as the device-base of internet users becomes more unified and capable in terms of raw processing power and features supported.

In this project responsive design is implemented thoroughly in such a way that it is usable with full features available in phones, tablets and desktop. Some compromises were made on visual cues such as hovering on elements with a cursor as iOS devices tend to interpret these CSS-conditions very differently from other devices.

If there are multiple elements being displayed on the page, the elements will change size accordingly to the screen resolution to provide an optimal viewing experience for the user. Most commonly in array items this means showing one element on extra-small, 2-3 on small and 3-6 from medium screen resolution and up. Bootstrap frontend-library provides media queries that determine the pixel-sizes for these semantic size definitions, which is called the grid system. Bootstrap library is further elaborated in Chapter 4.1.1.

4 Technical Execution

After a rather lengthy survey and conception phase throughout spring 2015, the development undertook in the course of 3 months during autumn 2015. The development was accomplished using IntelliJ Webstorm as an integrated development environment and Git as the version control software with a repository in Bitbucket.

Initial work focused on the distribution of the planned content into the ontological model using static data placeholders. Some difficulties arose within this phase as it was hard to decide where to draw the line between in the taxonomy and which HTML-elements to include in each level.

Thereon, most work was allocated to defining the domain models within the schema. This proved to be very challenging for the reason that the models needed to be flexible enough to provide different aggregation compositions without making the editors have to determine too many variables.

A rather lengthy and hard module to implement, the editor took much longer than expected to function completely as intended. A considerable amount of development time was allocated to make it possible that only pdf and image files are accepted when uploading files by dropping into, or clicking the upload area.

When the structure and all content creation elements were completed, static placeholders from the templates were replaced with aggregated data. This also depended upon implementing new relational search methods that combine and filter database results.

4.1 Application Architecture

Meteor is very flexible in the way the developer can organize the files. It does have few rules to it that are very important to keep in mind when designing the application architecture:

> By default, any JavaScript files in your Meteor folder are bundled and sent to the client and the server. However, the names of the files and directories inside your project can affect their load order, where they are loaded, and some other characteristics. (Meteor documentation 2015)

The load order is very important, especially when considering libraries which are not loaded with the Meteor package manager, Atmosphere. More information of the matter can be found in the Meteor documents.

Meteor has an interesting approach when it comes to serving HTML files, as even they are transformed into the application JavaScript, which is sent to the client when the user requests a page. The files are analysed in the client and all rendering is computed with the user's device. Figure 4 further exhibits the details of the interesting architecture that transfers a monolithic core which contains only the logic how to fill it with content, which trickles from the server as browsing progresses.

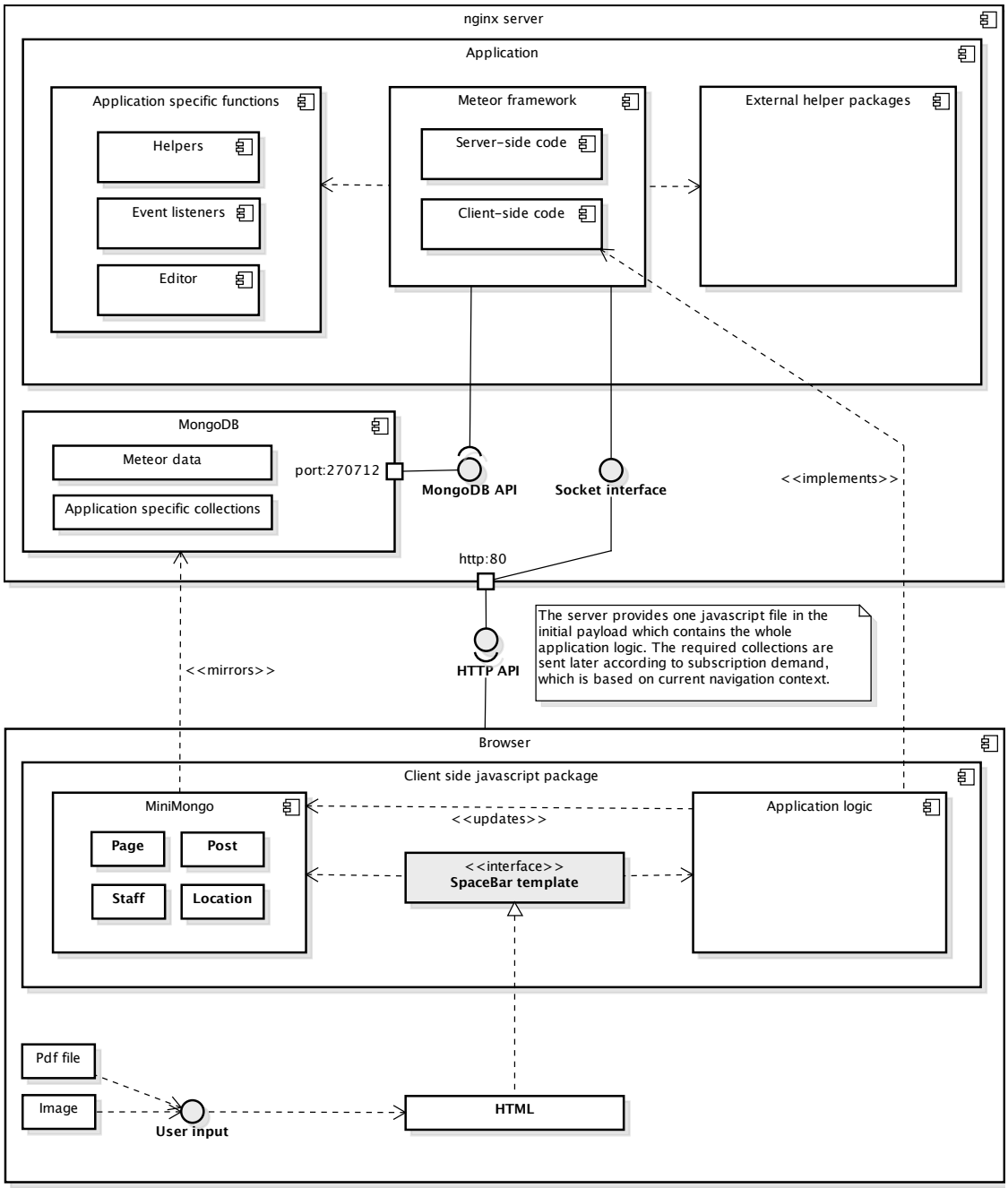

Figure 4. Architectural overview showing the interaction between modules

A developer would have mostly interest in the application specific functions and external helper packages as presented in Figure 4. They are the fundamental areas which make a Meteor application operate according to specification. Not forgetting the database that will reside application specific instance data etc. and also the defined collections. Both the application logic and collections are separated to client-side and server-side. Collection manipulation and any critical operations should only be executed on the server-side to be considered “safe code”.

These modules are represented on the client by single file that contains the logic and receives cacheable updates on the change of state within the application and collections. The client implements an instance copy that is a part or a *result set* of the whole database, which can be used to navigate and sort data in the view almost instantly.

4.1.1 Core Functions and Primary Libraries

Meteor offers two important tools for developer to build your own logic on top of the existing abstraction layer and the methods provided there:

Helpers are used to provide templates with variables or functions that are context-tied or global. They can return variables, arrays, objects, functions etc. In accordance to JavaScript language specification, a function extends object so in reality these are always defined as functions.

Events are HTML DOM events, such as “click” and “onchange” along with others, simplified to a cordial interface that is easy to use in the Meteor template context. Global events cannot be defined per se, nonetheless it is possible to associate events to the HTML <body> element, to which most elements belong under hierarchy-wise automatically.

Core functionality was extended with a multitude of libraries, however few of them deserve a mention in order to be more clear when referencing them later in the text:

- **Simple-schema,** to enable schemas on MongoDB, which by nature is dynamically-typed database system.
- **Collection2,** an extension on the Meteor collection object which enables more complex structuring and validation of documents.
- **Autoform,** generative programming-style forms that enables quick development of forms that self-validate based on the schema and are fully responsive.
- **Iron router,** library for analysing navigation requests and routing traffic appropriately.
- **User-roles,** extension library for the user model provided by Meteor.

These libraries were used mostly for front-end functionality or helper functions:

- **Bootstrap** was chosen for its ease-of-use and community support, meaning high availability of support libraries that can use bootstrap for their styling.
- **jQuery-cropper** library was used in the document editor for resizing and cropping images, **summernote-editor** for the text content WYSIWYG-editor and **sanitize-html** for sanitizing the user inputs.
- **Fullcalendar** provides the calendar generation based on the event-type posts residing in the database.

Accompanying libraries were abstraction of helper functions, convenience libraries or related to notifying user or adding visual elements. Using the Atmosphere package manager is a breeze and gives the developer really good set of tools for implementing ones design. The community support for the libraries is also pretty extensive, in the form of discussion forums and issue tracking in GitHub.

4.1.2 Database Schema

Although it is a common belief that NoSQL or document-based data stores such as MongoDB and CouchDB comprise of schema or type-free documents, that is not entirely true. In actuality, they are dynamically-typed as opposed to statically typed schemas as they are in SQL databases. Dynamic typing takes effect without developer action unless explicitly defined in the application schema.

In practice MongoDB's own internal schema is realized in the format of BSON-documents. The format incorporates the type information of each value of a field, which upon closer inspection are not so dissimilar from the variable types in static languages such as Java. (BSON types 2015)

The package simple-schema lets developers to apply this feature and define the domain objects in a static fashion. It enables to use validation and makes sure that the database stays organised and clean, with a clear understanding of the values being stored. This helps debugging problems where arising vs. in a dynamic schema where the composition of the stack that built and stored the object affects the type that gets stored in the database.

In Figure 5, a part of the “page” type domain object specification is introduced. The code illustrates how easy it is to set up static typing and really here lies the power of NoSQL, being free when required yet enabling constrains where needed.

```
...
content: {
    type: String,
    label: "Content",
    max: 10000,
    autoform: {
        afFieldInput: {
            type: 'summernote',
            class: 'editor', // optional
            settings: SummerNoteSettings
        }
    },
    autoValue: function () {
        if (Meteor.isServer && this.value) {
            return sanitizeHtml(this.value, {
                allowedTags: ['span', 'p', 'h2', 'br', 'ul', 'ol', 'li', 'a'],
                allowedAttributes: {
                    'span': ['style'],
                    'a': ['href', 'target']
                }
            }).replace(/\s/, " ");
        }
    }
},
template: {
    type: String,
    label: "Template",
    max: 50
},
parent: {
    type: String,
    label: "Parent page (optional)",
    autoform: {
        options: function () {
            return Pages.find(
                {
                    _id: {
                        $ne: Session.get("currentPage")
                    }
                }).map(function (c) {
                var selector = "";
                var findParent = function (page) {
                    var parent = Pages.findOne({_id: page.parent});
                    if (parent) {
                        selector += "- ";
                        if (parent.parent) {
                            findParent(parent);
                        }
                    }
                };
                findParent(c);
                selector += c.title;
                return {label: selector, value: c._id};
            });
        }
    },
    optional: true
}
...
```

Figure 5. Part of “page” domain object schema specification

One can see in the example illustrated in Figure 5. that the typing is static and there are some additional properties set, that are of some interest to us. As mentioned before, this schema is used for the validation process by comparing the fields of the document that is received from a form to the schema's fields. They must meet the requirements set by the schema as illustrated above. Failure to do so will result in the form being invalidated and the user prompted to fix the matter. Autoform fields are used to define options for the generative forms that are used throughout the project. In the example above, the options function sets values for selecting a parent page without being able to select the current page as parent. AutoValue() functions trigger when this type of object is being inserted in the collection. It is a very effective solution that on the other hand retains a lot of freedom regarding how the relations are defined. In this project the relations are simply String type variables that act as proxies for the actual documents.

4.1.3 Defining Templates

In consonance with the ontological model, it was important to designate semantic hierarchy to the taxonomy of templates. From this approach, three classes of templates emerged as mentioned earlier on 3.1: "template", "content" and "snippet". Three levels were designated as it was decided that this represented the layout structure most efficiently. The ontology model is represented in Figure 6 below.

Figure 6. Structure of a page by hierarchical elements.

By breaking down the layout, it was noticed that in most cases the re-usable elements concentrated among content originating from the domain model. Examples of this type of content include images, related and/or children documents or simply listing a larger set of base-level documents.

This way it is possible to simplify the mark-up writing process as demonstrated by the isolation acknowledged in Figure 7. Additional benefits are gained by modularizing the CSS-styles. For example, CSS-styles for snippet blocks are in their own file and common styles in a separate file.

```
<template name="snippetPostColumn">
    <h2>Related posts</h2>
    {{#each categoryPosts}}
        <a href="{{pathFor route='post.show' slug=slug}}">
            <p class="newspage-related-links" style="margin-bottom: 10px;">
                {{title}}{{{getDuration}}}
            </p>
        </a>
    {{/each}}
</template>
```

Figure 7. Snippet for showing category-related post items.

Each level contains it own mark-up, context-aware helpers and events. The context can be changed from default (which is the object given to template in the router) in the render call for this snippet by re-defining the context object. Isolation of the code gains the developer strong awareness of any potential problems in the mark-up and fast debugging in case there is any. It can, however, sometimes be cumbersome to navigate the snippets; in case they are all lumped in one large .html file. Meteor gives flexibility in this matter and it is possible to break them down to files in any way desired. It should be noted though that this can result in complicated file structure.

4.2 Reactive Programming

For those who are familiar with the *observer* pattern, it is not hard to imagine what reactive programming tries to accomplish. Similarities have been discussed in various forums but it can be summarized with the following:

In an *imperative* programming setting, $a := b + c$ would mean that a is being assigned the result of $b + c$ in the instant the expression is evaluated, and later, the values of b and c can be changed with no effect on the value of a. However, in reactive programming, the value of a would be automatically updated based on the new values, the opposite of functional programming.(Huber, 2013)

What this means in practise is that when a database collection has a new document, e.g. a “post”, the news feed template should know about it and display it as soon as it is possible. This is closely related to the observer pattern but it is not the same, the fundamental difference being that the observer-observable relationship is between objects. In reactive programming, the change notifying can extend to the variables included in the

observed object and recursively deeper. Meteor uses the *publish–subscribe* pattern to achieve reactivity for the database in the views.

Obviously this is not automatic (actually by default Meteor projects pre version 1.2 came with a package auto-publish that does it for the user), the relationship has to be established by defining publications and subscribing to them. Publications are methods on the server that expose a collection or part of it to the client and make sure that the changes are relegated when necessary. Publications return a database cursor that the subscription will operate thereon.

The manner in which the communication is handled, is the new protocol MDG developed: DDP. It uses multiple small packets of data to transfer large entities from the database. It uses WebSocket technology as the underlying technology. It is not unlike http-frames in its implementation data-wise as perceived by the example illustrated in Figure 8:

```
["{\"msg\":
    \"sub\",
    \"id\":\"pKG4qKjDQWp5wJxNe\",
    \"name\":\"newsFeedPosts\",
    \"params\":[]
}"]
```

Figure 8. A DDP-packet in which the client requests a subscription.

The fundamental difference between the two arises in the amount of data used. In the case of DDP, this is not the case, befitting the reactive paradigm. In the client Meteor uses jQuery to manipulate the DOM and insert the data acquired.

Without needing to refresh the browser page between page loads, which actually are not page loads in the traditional sense, it is possible to acquire near-instantaneous load times. It is only limited by the performance of the computer running the client. Certainly large data sets will take time to render into the view but most definitely not longer than first fetching the whole document from the server and then rendering the changed parts.

There are scenarios which will not benefit from this behaviour, such as “static” page content. In quotation because of course in this type of application it is not actually sensible to have any static business data. But static in the sense that it is not desirable to change it while the user is actually using it. This can be solved by the option *reactive: false* when defining the data context for the template.

However, the implementation of reactive programming in Meteor has one major drawback; it is limited to database collections and a special Session object that can nevertheless hold any standard type of JavaScript object inside of itself. It is possible to use a package called **reactive-var** that enables an object called *reactiveVar*, but it is in most aspects a replica of the Session object, primary difference being that unlike Session object, the reactiveVar can be tied to a template instance whereas the Session object exists in global scope.

Using the Session object is cumbersome compared to for example **Angular**, where each object within the controller scope is automatically reactive. The only way to access the Session object is through get() and set() methods, which really complicates even the most rudimentary tasks like incrementing numbers. This is exemplified in Figure 9:

```
//METEOR
if (!target.data("visible")) {
    target.data("visible", true);
    Session.set("page", Session.get("page") + 1);
}
//ANGULAR
if (!target.data("visible")) {
    target.data("visible", true);
    $scope.page++;
}
```

Figure 9. Implementing reactive increment in Meteor and Angular

While this might not appear issue at first, it really causes some features to be very tricky to implement and special care must be taken when using them, since they are global. The Session object is tied to the client so the global nature has some benefits too when used in the right context, examples include navigation states, filter states etc.

Another grief imposed by the reactive model in Meteor is that the template helper model is not extensible. One cannot create a generalized set of functions that then would be enhanced by extending the model on which they reside. In Angular and **React** it is a central part of the framework design, enabling complex frontend interactions through use of patterns and architectural superiority. A simple analogy would be saying that in Meteor one builds with only bricks that are basically just fine but one would not like building a house out of only them, compared to the aforementioned frameworks which give you bricks and mortar, that do make for a much better house.

4.3 Data Flows

The content in the site is based on data flows that are defined by two opposite concepts acting on same set of resources: Content creation and aggregation. As discussed earlier the primary source of the content are the editors, as illustrated in Figure 10.

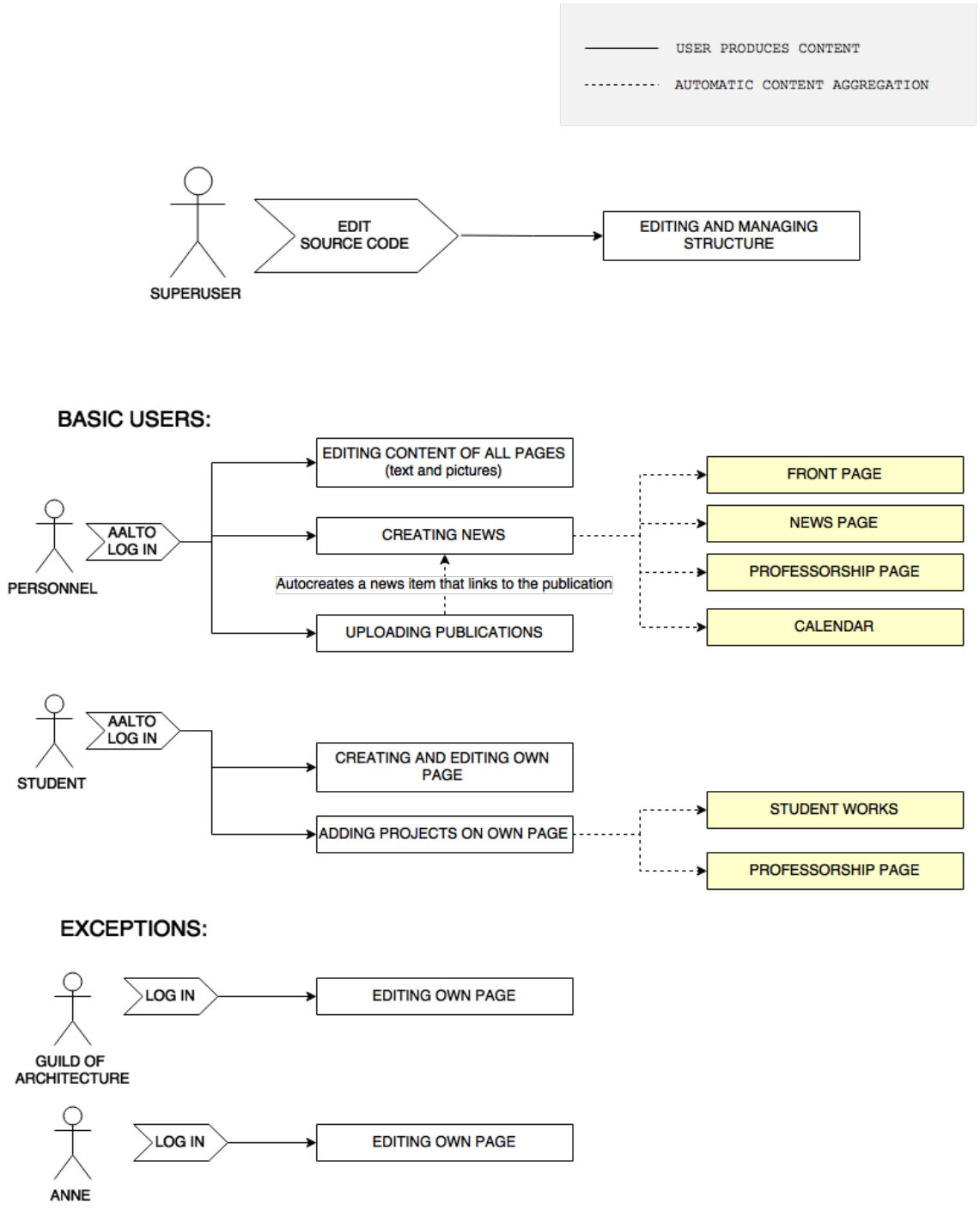

Figure 10. Content creation and aggregation model

Figure 10 introduces a concise overview on how the data flows through the application. The user roles and groups and their domains of influence from chapter 3.2. are realized here.

Aggregation combines documents according to their “parent” field per described in their schema. Aggregation helpers use recursive functions when necessary to find and sort the related content accordingly and pass it on to the templates.

4.3.1 Content Creation

The editor is a collection of JavaScript functions in the form of event listeners, helpers jQuery method calls and library instances. It is always context aware and will render elements according to the type of data being edited. Limiting user actions and responding to errors in the process is also a crucial part of its operation.

The editor module provides the users with a simple interface for content creation that simply renders the fields from the schema with empty values and enables the user to input the necessary information. After completing the form, the user will click save which will validate the provided document extracted from the form. If there are missing or invalid fields the user will be notified of the situation. Editing content is very similar, it is activated by navigating to a single “page” or “post” and clicking the edit button. It opens the document in the form with the values populated from database as demonstrated in Figure 11.

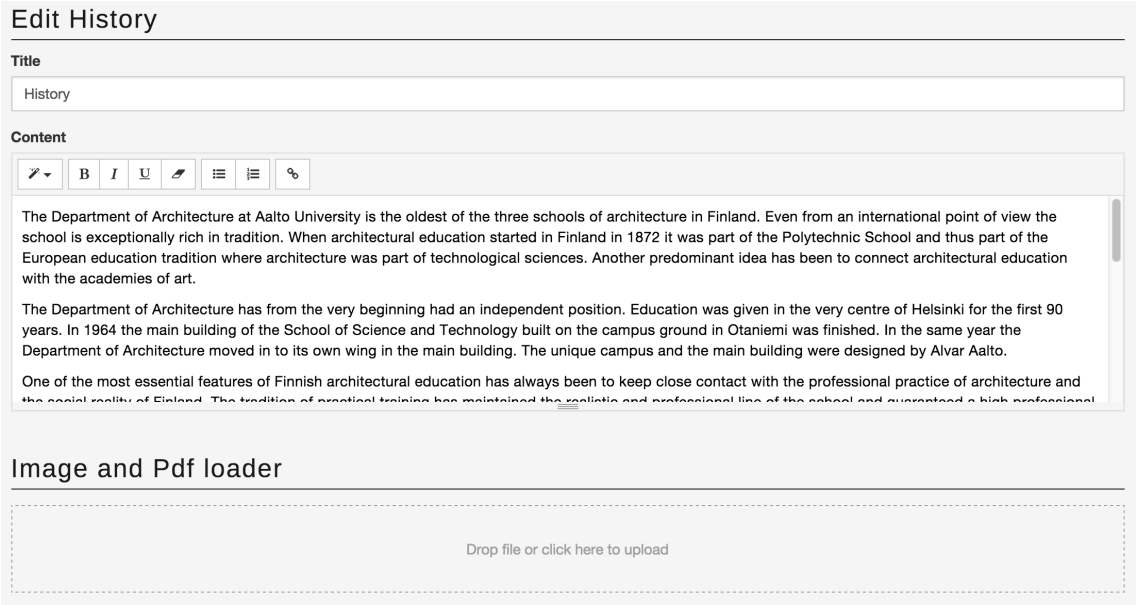

Figure 11. Editor module in action

The most important content in a website that relies heavily on visual communication is of course images and the editor provides an excellent tool for this. It is able to resize images of any size to more appropriate for web use and lets the user define a cropped portion of the original image to use in the templates. This is done to retain the visual consistency of the layout and minimize database payload.

Figure 12 demonstrates us how easy it is to add new images to the site. The user can simply drag the crop area around and resize it if one wishes to do so. The cropped area has a fixed ratio that is determined by the type of document the image will be belong to.

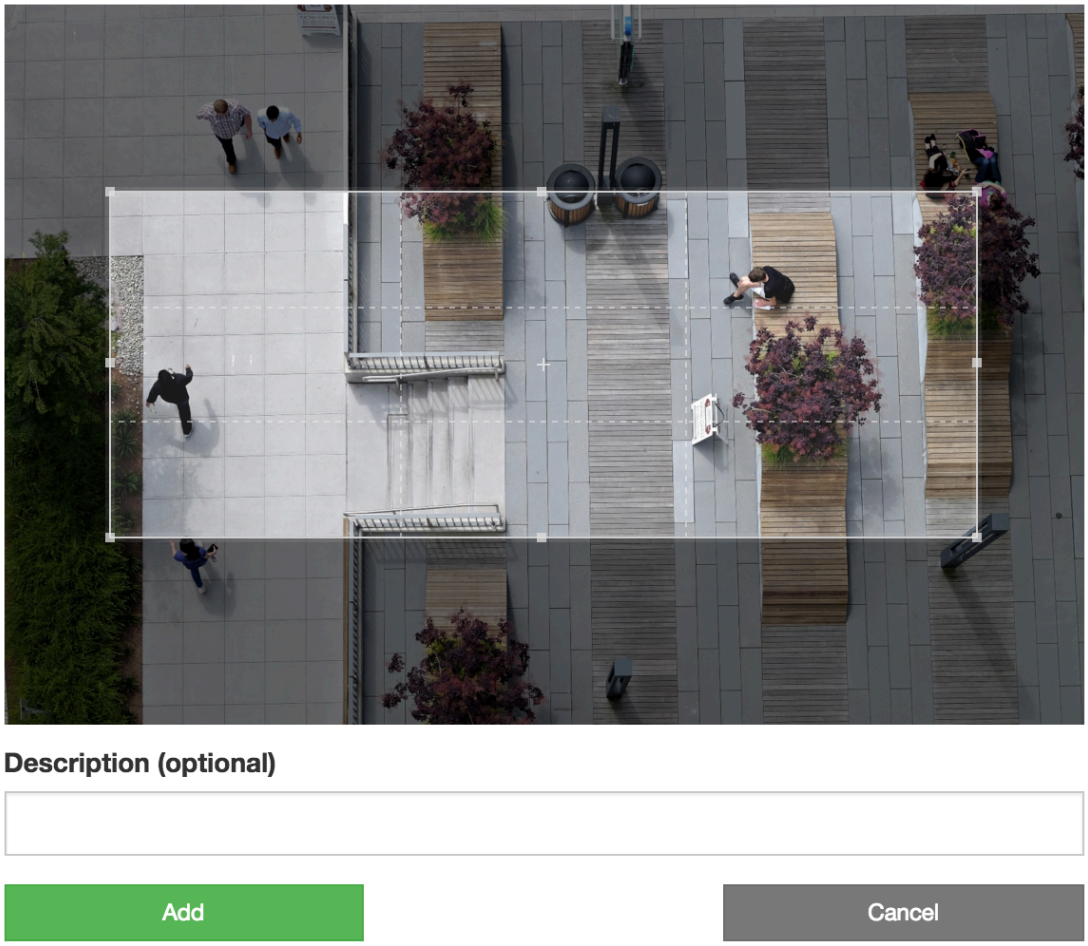

Figure 12. Using the editors crop modal

There was significant work involved in the optimization of the load order of the individual libraries to present a smooth experience in using the crop tool. Behaviour was very inconsistent between browsers and in the end it was compulsory to incorporate a 100ms delay between inserting the dataURL to the html <img>-tag and instantiating the crop library.

Sequence of the image uploading progresses as follows:

- The user clicks the upload area or drops a file into it.
- The file is analysed whether it is image- or pdf-type.
- If image, sent to the resizing library which returns a dataURL object.
- After resizing is done, a bootstrap modal is instantiated.
- Inside the modal the image object is given to the cropper library.
- The user sets a crop area and saves the image.
- The cropper returns another dataURL object which is sent to database in chunks.
- When the upload is finished, the image reference id string is saved as part of the document to be saved.
- Per reactive paradigm, once the image id is part of the document it can be fetched and displayed.

Uploading pdf-files happens in a similar manner, but of course it is not required to crop this type of file in any way so it can just be uploaded and include the reference to the current document.

It was challenging yet fulfilling to also design the sequence of the user interface feedback related to each step of the editor actions. It can also be said that there were really no compromises in terms of the functionality desired and what was accomplished.

4.3.2 Content Aggregation

Gathering the information starts in the iron-router module, which sets the context used to determinate the next course of action. The router consists of routes that individually contain directives for behaviour based on the request parameters. Once it is established which route matches the given URI, the logical operations are executed and initial data load begins by means of subscripting to a DDP stream/streams. While the data is loading, a waiting animation is displayed on the layout's main container. In Figure 13, it can be seen that the interface for configuring the router is very simple.

```
Router.route('/pages/:slug', {
    name: "page.show",
    waitOn: function () {
        var subs = [];
        if (this.params.slug === "facilities") {
            subs.push(Meteor.subscribe('locations'));
        }
        subs.push(Meteor.subscribe('pages'));
        subs.push(Meteor.subscribe('images'));
        subs.push(Meteor.subscribe('files'));
        return subs;
    },
    action: function () {
        var page = Pages.findOne({
            slug: this.params.slug
        }, {
            reactive: false
        });
        if (page) {
            Session.set("previousPage", "/pages/" + page.slug);
            this.render(page.template, {data: page});
        }
        else {
            this.render("template4");
        }
    }
});
```

Figure 13. A single “page” router

When all subscriptions are completed, the initial rendering begins. Now the template may request more subscriptions based on the context and sequential loading begins, showing content as it arrives reactively from the server. The primary means of acquiring this data is through the aforementioned helper functions. They populate the template and when all data is loaded and rendered, the process is complete.

```
Template.registerHelper("childPages", function (page) {

    return Pages.find(
        {
            parent: page._id
        },
        {
            sort: {
                index: 1
            }
        }, {
            reactive: false
        }
    ).fetch();
});
```

Figure 14. Helper function that returns child pages of a single “page”

The example in Figure 14. illustrates clearly how easy it is to combine data in the client. Now, because the router in Figure 13. defined subscription to “pages” without any limiting parameters, all “page” objects are already in the browsers working memory, meaning

this function returns the result near-instantly. For the user, it seems that there is no delay whatsoever (sub-100ms.).

These features make it extremely viable to implement filtering of content in the client, as in the case of “post” items. When the user selects any of the filtering options, the data is aggregated from the server and sorted in the client. In practise, this also happens very rapidly, thanks to the small overhead made possible by WebSocket.

4.4 Deployment

The virtual server was provided by Aalto systems management team in the form of Linux Red Hat RHEL 7. After a short research period, Passenger by Phusion Inc. was chosen as the deployment platform, by the virtue of a having very clear and concise guide into production deployment with Meteor.

It was not, however, a breeze to complete, as the virtual server provided had very tight security, forbidding for example MongoDB to write to the hard drive. These small difficulties were not tied to the Passenger software and once the directory ownership and CHMOD rights were correctly set, it was very easy to get the server running.

5 Summary

A very challenging experience, sometimes intensified by the quirks of the framework, the development took in all 4 months. In retrospect, if it were possible start over the author would substitute the Meteor Blaze library that provides the helper functions, with Angular. Many aspects of the frontend logic could be much more readable code and easier to extend in the future.

User feedback on the past 2 weeks of internal testing by the actual users have proved to be very positive. Comments have been made that the interface is really easy to fathom and as is the case usually with technical qualities perceived by non-technical persons, it is good if no complaints are heard. Regarding the actual benchmarks of the page, an instrumentation tool called Kadira was used and the metrics look very promising. The method response time has constantly been under 100ms, memory usage is under between 150-200 MBs and CPU usage stays usually under 5% with the highest peak seen at 18%.

The application has fulfilled the design goals quite beautifully. Meteor can be used for implementing a modern web-application, especially when speed is a consideration. It also enables native wrappers for compiling the project into iOS- or Android-application. There was no opportunity to test this feature yet, but it could be a complete solution for a start-up company or small institution looking for an effective framework in which to realise an idea from a prototype into production in months.

Responsive aspects were well fulfilled and the mobile experience is very good and does not feel any lesser compared to that of the desktop.

The fact that it was possible to implement (although simple by design) a fully working content management system from scratch in 4 months is a testament to Meteor's practicality as a web development tool.

Please check out http://architecture.aalto.fi after reading this!

References

MDG DDP-Specification, 2015. Available from: https://github.com/meteor/meteor/blob/devel/packages/ddp/DDP.md [2 December 2015]

Lim Wayne, 1994. “Effects of reuse on quality, productivity and economics.” IEEE software, p. 23-30 [2 December 2015]

ISO, Ergonomics of human-system interaction, 2015. Available from: http://www.iso.org/iso/catalogue_detail.htm?csnumber=52075 [2 December 2015]

Jakob Nielsen, 2007, “Feature Richness and User Engagement” Available from: http://www.nngroup.com/articles/feature-richness-and-user-engagement/ [2 December 2015]

Jakob Nielsen, 2010, “Website Response Times” Available from: http://www.nngroup.com/articles/website-response-times/ [2 December 2015]

SPDY-whitepaper Google, 2015. Available from: http://dev.chromium.org/spdy/spdy-whitepaper [3 December 2015]

Arun Gupta, 2014, “REST vs WebSocket Comparison and Benchmarks” Available from: http://blog.arungupta.me/rest-vs-websocket-comparison-benchmarks/ [2 December 2015]

Ethan Marcotte, 2010 “Responsive web design” Available from: http://alistapart.com/article/responsive-web-design [2 December 2015]

BSON types, 2015. Available from: https://docs.mongodb.org/manual/reference/bson-types/ [28 November 2015].

Huber Val, 2013. Available from: https://dzone.com/articles/reactive-programming-database [29 November 2015]

Meteor documentation, 2015 Available from: http://docs.meteor.com/#/full/ [2 December 2015]