Bachelor Thesis

Development of a web-based application for visually collecting thoughts

by

Milan Bargiel

First reviewer: Prof. Dr.-Ing. Luigi Lo Iacono (Technische Hochschule Köln)

Second reviewer: Prof. Dr.-Ing. Arnulph Fuhrmann (Technische Hochschule Köln)

November 2016

Institut für
Medien- und Phototechnik

Fakultät für Informations-,
Medien- und Elektrotechnik

Technology
Arts Sciences
TH Köln

Bachelor Thesis

Title: Development of a web-based application for visually collecting thoughts

Reviewers:

- Prof. Dr.-Ing. Luigi Lo Iacono (TH Köln)
- Prof. Dr.-Ing. Arnulph Fuhrmann (TH Köln)

Abstract: This thesis describes the development of *Kosmo*, a web-based application for collaborative archiving and interactive exploration of notes. Textual thoughts float through a virtual space, driven by a physical simulation and invite an exploration of content. Using modern technologies an application was developed that instantly reacts to user interactions and allows for collaboration in real-time.

Keywords: Web application, Real-time collaboration, Data visualization, Meteor, D3.js

Date: November 7, 2016

Bachelorarbeit

Titel: Entwicklung einer webbasierten Anwendung zur visuellen Sammlung von Gedanken

Gutachter:

- Prof. Dr.-Ing. Luigi Lo Iacono (TH Köln)
- Prof. Dr.-Ing. Arnulph Fuhrmann (TH Köln)

Zusammenfassung: Diese Arbeit beschreibt die Entwicklung von *Kosmo*, einer web-basierten Anwendung zur gemeinschaftlichen Archivierung und interaktiven Erkundung von Notizen. Verschriftliche Gedanken werden auf Basis einer physikalischen Simulation durch den virtuellen Raum bewegt und können so durch die Benutzer erkundet werden. Durch den Einsatz von modernen Technologien wurde eine Anwendung entwickelt die ohne Verzögerungen auf Benutzereingaben reagiert und des weiteren Kollaboration in Echtzeit ermöglicht.

Stichwörter: Webanwendung, Echtzeit Kollaboration, Datenvisualisierung, Meteor, D3.js

Datum: 07.11.2016

Contents

1 Introduction

1.1 Background

> *“Very often, gleams of light come in a few minutes’ sleeplessness, in a second perhaps; you must fix them. To entrust them to the relaxed brain is like writing on water; there is every chance that on the morrow there will be no slightest trace left of any happening.”* (A. Sertillanges 1863 – 1948)

Thoughts are fragile; they may appear and vanish in a matter of seconds. By taking notes one ensures that sudden moments of inspiration are not forgotten. Once a thought is brought on paper, on screen, it becomes a concrete representation that is part of the visually perceivable space. Its author may come back at a later point in time to revive an idea that has been inspirational to him.

With the rise of information technology and the proliferation of smart phones, note taking nowadays is mostly done on electronic devices. Popular note-taking application such as Evernote aim at enabling the user to rapidly insert and retrieve notes from the system. They present notes in form of a list, a representation that allows for a quick scan of information. While this might be an appropriate solution for a number of cases, it may fall short in explorative tasks.

The purpose of this thesis is the development of a web-based application that provides a virtual space for users to create and collaboratively explore note archives. The envisaged application distributes notes spatially within an interactive visualization, inviting for an exploration of content. Through the integration of modern technologies, online collaboration in real-time shall be made possible.

1.2 Structure

The second chapter deals with the concept of the application. It elaborates on how the exploration of a thought archive may lead to inspiration and further points out the basic functioning of the system. It then presents the prototype of the user interface. The chapter ends with a list of technological demands that have evolved from the concept.

The third chapter details theories and concepts that are necessary to understand the technological implementation of the application. It explains how a web framework eases the building of complex applications and furthermore presents the tools that are used to program the application. By describing their underlying concepts, how these tools appropriately meet the technological demands defined in chapter two is clarified.

The fourth chapter discusses the system’s design and its actual implementation. An overview of the application’s architecture is given and its modular design principles are highlighted. Then, the configuration of the system is detailed that enables real-time collaboration on note archives. The chapter further explains how reactive note data is inserted into the application’s visualization. The chapter ends with a description of the application’s mobile version.

In the fifth chapter it is evaluated to what extent the technological demands were achieved. In addition, a brief reflection on the application’s ability for meaningful online collaboration is given. The report ends with an outlook on future work.

2 Concept of the application

Kosmo is a reservoir for thoughts that groups of users supply. The user interface portrays a planetary system in which each planet represents a collection of thoughts. Floating circles are used to visualize thoughts and are driven by a physical simulation. They reside on the surface of a planet, representing the “thought world” of a user. Hovering the mouse over a floating circle reveals its content.

The application aims at providing a pool for archiving and exploring notes. The program places notes within the visualization, without hierarchies and without a constructed order – every thought holds equal importance. One may not know what substance future notes will have when opening up a thought collection. The application allows for the insertion of ideas without having to think about how to integrate them into the context of other ideas. This shall facilitate the insertion of spontaneous and associative ideas.

Notes have no fixed position inside of the application’s user interface, instead floating around, changing their positions in space. According to Luhmann (1982), the incorporation of irritation into a system may provoke new ideas. While exploring a thought archive, a user might stumble upon an idea in the environment of other ideas, although these ideas were not originally related, i.e. a thought archive may lead to inspiration. Through the integration of interactive elements, the application aims to engage the user. A user might not have a specific question in mind while browsing a thought archive. By chance, he might rediscover a thought that is of value to him.

Kosmo incorporates the grouping of interrelated ideas through tags. The purpose of tags is to group notes and to serve as an index for entering the archive. Following the conventions of prevalent social media applications, a tag is initiated with a hash key and can be placed anywhere inside of a note text. Building on a syntax that is familiar to most users ought to scale down the learning curve.

Each collection of thoughts is accessible by a unique URL. As thought archives may contain private data, *Kosmo* makes a distinction between public and private note collections. A private collection is only visible to its owner. To collaborate on ideas, users can make their collection public and share its link with friends and colleagues. As no authentication is needed to explore a public thought collection, visitors may be invited to explore the contents of an archive.

When developing a concept, it is important to build upon existing ideas. The application allows the state of a visualization to be saved inside of the URL. By doing so users may share links with each other that lead to a preselected thought, and therefore have a common point of reference.

2.1 Prototype of the user interface

Figure 1 shows an individual thought collection henceforth referred to as the planet page. The prototype of the user interface was made with *Sketch*, a tool for digital design. The planet page represents the heart of the application. It is the graphical user interface that enables users to collaborate on collections of notes.

The user interface of the planet page divides into three major components: the *navigation*, the *visualization* and the *content view*.

At its center is the interactive visualization of all archived notes represented as floating circles. The circles levitate inside of the boundaries defined by the planet. When a user hovers the mouse over a circle, this action reveals the circle's content inside of the content view component located to the right of the visualization. The content view component portrays a note's text as well as its author and the creation date.

The navigation component to the left of the visualization either displays a list either of all tags or of all users who contributed to the collection. The list consists of selectable elements, which when chosen, highlight the corresponding notes that are visualized as circles.

To insert a note into the system a user has to click on the button at the top right corner of the planet page. The form to insert notes reveals itself at the same position as the content view component. This area serves for displaying notes as well as inserting them.

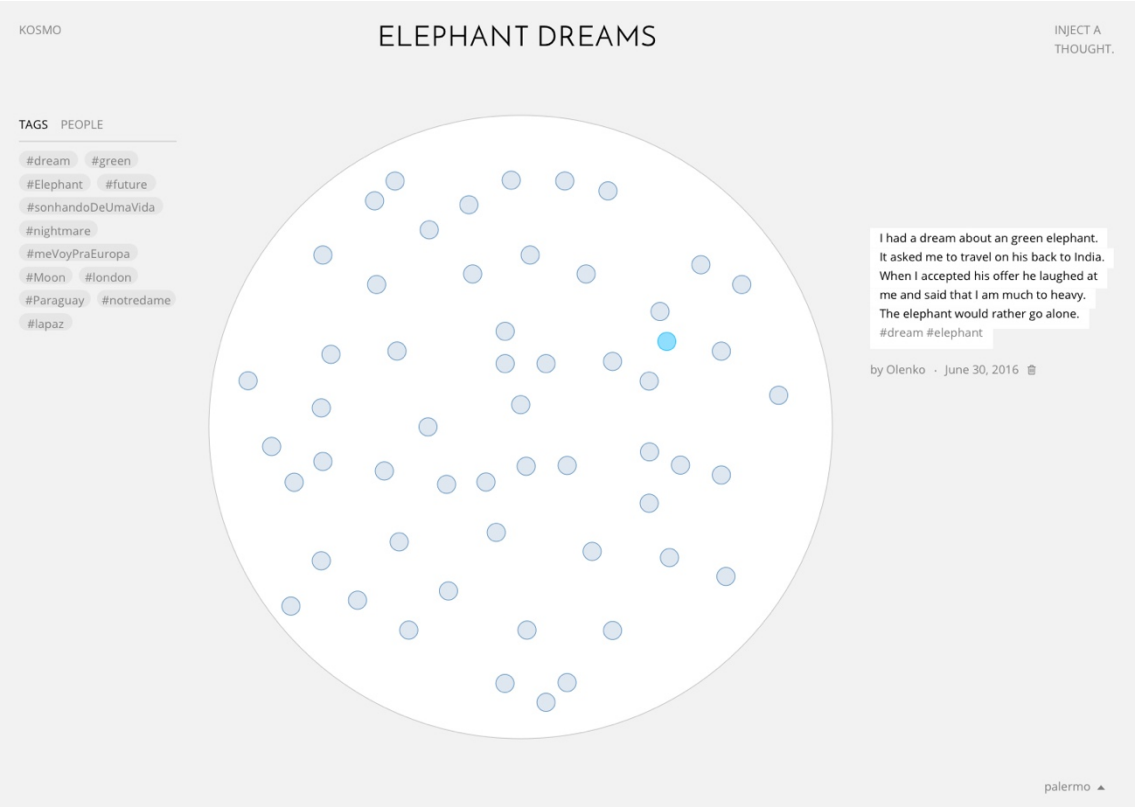

Figure 1: A screen from the prototype of the user interface portraying the main view of *Kosmo*. The prototype was designed with *Sketch*, a tool for digital design.

2.2 Technological demands

The application as defined in the prototype has certain technological demands as presented in the following:

1. Collaboration on ideas is fundamental to the application. To enable users to respond to ideas of others, it is necessary that newly inserted notes appear on the screen of all connected users as soon as they are published.
2. Even though a web-based approach, which in the past and is still currently viewed as static, was chosen, the highly interactive nature of the application requires a dynamic user experience that feels like a desktop application. Users shall experience direct feedback on their interaction with the visualization and the application itself.
3. Circles inside of the visualization need to be animated. To simulate levitation, their position on the screen needs to change over time. Furthermore, a note needs to be connected to its graphical representation to ensure that the right content is revealed upon the selection of a circle.
4. Moments of inspiration are not predictable. Since the user needs to be able to insert a thought into a collection from any given location, the user interface of *Kosmo* adjusts to mobile devices. Due to the narrow screen-size of a smartphone, the mobile version of *Kosmo* is limited to the insertion of notes.

3 Technologies

To implement the prototype defined in chapter two a set of technologies was chosen. The foreseen application comprises two logical parts: the interactive visualization representing archived notes as floating circles inside of a virtual space, and the web application itself, hosting the visualization and enabling online collaboration between users.

To create a web application, developers may build upon a web framework, which is an interlocked collection of programs that serve as a skeleton for the development of web applications. A framework eases the building of complex applications by providing core functionalities, such as data management and template systems. Instead of reinventing solutions, developers can focus on implementations that are unique to their application. *Kosmo* is built on top of *Meteor*, a web framework that supports real-time web applications.

3.1 The JavaScript framework Meteor

Meteor is an open source framework for developing modern web and mobile applications. It comprises a set of interlocked technologies bundled for the creation of real-time web applications (Introduction | Meteor Guide, 2016). The Meteor framework runs on top of *Node.js*, a server-side environment that enables running JavaScript code on the server. It is built on *Chrome's V8 JavaScript engine*, responsible for compiling JavaScript into machine code (Node.js, 2016).

The interaction with a web-based Meteor application is fluid. Users receive immediate feedback on their interactions with the application and with other users. Once the application has been loaded into the user's browser, no page reloads are necessary to access different views of the user interface. As users input new information, it is immediately published to all connected clients, without users having to refresh the page or the system having to execute periodical checks. These qualities are referred to as real-time and provide the basis for *Kosmo's* envisioned capabilities of online collaboration.

Meteor is built from the ground up to support reactive application that exhibit real-time behavior (Introduction | Meteor Guide, 2016). To comprehend Meteor's unique operating principles, one must possess an understanding of the architecture of a server-side web application. Traditionally, web applications operate on the server, whereas a Meteor application resides on the server as well as the client.

3.1.1 A server-side web application

A web page is divided into a client and server side. When a user opens up a browser and accesses a web page, the browser acts as a client sending a request for content to a server. The server is a computer with web server software that responds to client requests.

A server-side web application (e.g. built with *Ruby on Rails*) only lives on the server. On a request for content from a browser, the server-side application selects corresponding data from a database, generates HTML markup out of it and serves it to the client (Mills & Willee, 2016). The browser in return renders the HTML markup and displays the web page on the

screen. The browser therefore has a rather passive role. The creation of the web page occurs on the server side, and the browser is responsible for its rendering.

The communication between client and server follows a request-response pattern (Mills & Willee, 2016). Every time a user clicks on a link within the same web page, the server creates a new page and sends it to the browser. This process leads to a page reload, whereby the user sees a blank screen until the new page is loaded. When the underlying data of a website changes, a user needs to reload the page to see a representation of the modified data.

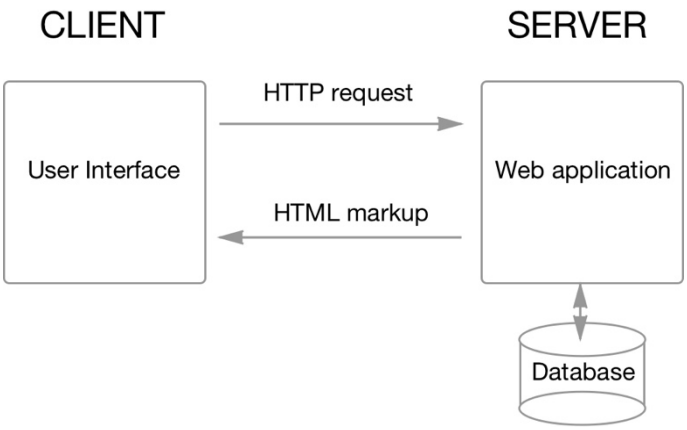

Figure 2: The architecture of a server-side web application.

3.1.2 The anatomy of a Meteor application

In contrast to a traditional web application, a Meteor application lives both on the client and on the server side. Upon an initial request to a web server that hosts a Meteor application, the client-side component is established inside of the user's browser. It contains all templates needed to build the entire user interface, as well as JavaScript application logic. Unlike in a traditional web application, HTML markup is generated in the browser. Therefore, no roundtrip to the server is required to update the user interface (Stubailo, 2015).

Data on the wire

Once the client possesses the initial state of the application, the client receives only the actual raw data (Greif, 2014). The client-side component of Meteor embeds that data into HTML code and draws the page on the screen. Not having to transfer complete HTML files to the browser saves bandwidth and therefore makes the application faster.

Whenever the underlying data changes, the user interface updates automatically. This paradigm is called reactivity, as the user interface reacts to changes in the database (Greif & Coleman, 2015). Meteor facilitates reactivity because it remains an open connection between client and server. Upon a change in underlying data, the Meteor server-side component pushes relevant data to clients. The connection is based on the *Distributed Data Protocol*, which is an implementation of *WebSocket* (Greif & Coleman, 2015).

One language

In Meteor, both client and server-side code is written solely in JavaScript. Having the same programming language at the client- and server-side enables new ways of designing an application. It allows for sharing code between environments, a concept known as *isomorphism* (Brehm, 2013).

Database everywhere

Meteor integrates *MongoDB*, a document-oriented database program, to store persistent data on the server-side. In addition, it has a local database program on the client called *Minimongo*, holding a subset of the server-side data (Collections and Schemas | Meteor Guide, 2016). The developer defines which subset of data is pushed from the trusted server-side database to the local database. The data of these two databases is kept synchronized by the application (Greif & Coleman, 2015).

Having an additional local database that lives inside of the user's browser serves two purposes. The local database is the unique source of data to the client-side templates. Therefore, data duplication is avoided, and it is ensured that all user interface elements are consistent (Stubailo, 2015). Furthermore, a local database allows for simulating user interactions that aim to change data on the client-side, a concept called *Optimistic UI* (Stubailo, 2015).

Optimistic UI

When a user inserts data into a Meteor application, for example by submitting a form, the result of the operation is immediately portrayed by the user interface. In a traditional web application, the user would have to wait until the insertion is evaluated by the server-side application before he receives visual feedback on his interaction. In contrast, a Meteor application behaves "optimistic" by showing the result of an operation on data before it has been evaluated on the server. First, the changes are written to the local database that serves as the source of data to fill the user interface's templates. The same operation is then performed on the server. When the modification is legitimate, the data is written to the server-side database as well (Stubailo, 2015). Otherwise the user interface is adapted to portray the true state of data as defined by the server-side database.

Optimistic UI is realized by defining methods that trigger the modification of data once for both the client and the server (Stubailo, 2015). User interactions that aim to change data initiate the execution of an isomorphic method in both environments.

Embrace the ecosystem

In programming, the reuse of code is of great importance. Programs are built upon existing ones to save time and resources. Rather than reinvent solutions, developers can focus on writing code that is specific to their application.

Meteor itself is a collection of interlocked reusable pieces of code, referred to as *packages*. They supply the fundamental operations of the framework. Furthermore, it is possible to integrate additional packages from third party developers into one's application. For example, a developer might decide to install a package that supplies form validation, instead of implementing the functionality himself.

Meteor is not a monolithic platform that seeks to provide its own solutions for every conceivable feature. Rather, its authors view it as a part of a broader ecosystem of JavaScript developers sharing code. This conception takes form in the principle *embrace the ecosystem*, specified in the official Meteor guide (Introduction | Meteor Guide, 2016).

To install, share and distribute code, Meteor employs package management. Meteor supports two package managers: *Atmosphere*, Meteor's own package system, and *npm*, the Node.js equivalent. To add a package from Atmosphere, a simple command inside of the operating system's terminal window is sufficient.

```
Meteor add fourseven:scss
```

Listing 1: Adding a package with Meteor's package manager

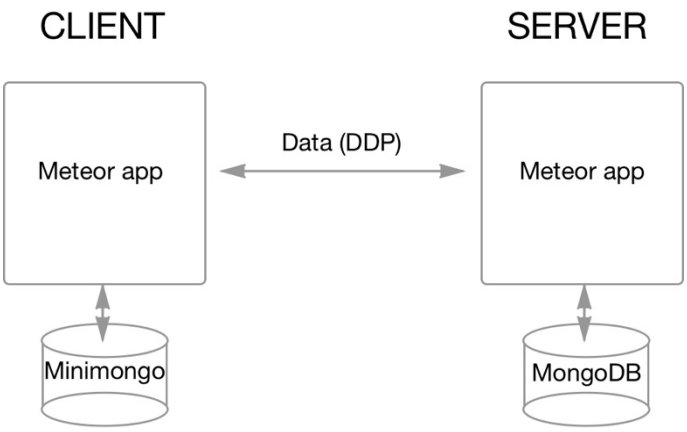

Figure 3: The anatomy of a Meteor application.

3.2 The JavaScript library D3

To represent archived notes as floating circles, a connection between note data and graphical elements must be established. The JavaScript library *D3* ("data-driven documents") is used for implementing the described functionality. Furthermore, it provides the means for an interactive animation of graphical elements based on a physical simulation.

D3 is a library for creating dynamic data visualizations for the web. The library enables the binding of arbitrary data to graphical elements on a web page (Bostock, 2016a). When the underlying data changes, the visualization transforms. For the creation and styling of graphics, it uses web standards such as HTML, SVG and CSS (Bostock, 2016a).

Native representations instead of abstractions

For the creation of a web page, different technologies are employed. Amongst others, HTML is used for page content, SVG for creating two dimensional vector graphics and CSS for defining the visual appearance of a page. The Document Object Model (DOM), a shared representation of a web page, enables the cooperation of these technologies (Bostock, Ogievetsky, & Heer, 2011).

When a web page is loaded, the browser builds up a Document Object Model for the page, and then uses this model to draw the page on the screen (Haverbeke, 2014). It is a representation of the page in which all document elements become nodes. D3 can access, change, create and delete these nodes. When nodes are modified, the page on the screen is updated to reflect these changes (Haverbeke, 2014).

D3 does not encapsulate the Document Object Model with abstractions. Instead the library enables direct manipulation and inspection of it (Bostock, Ogievetsky, & Heer, 2011). The library can create any type of element the browser supports, and set any attribute or style property (Bostock, 2012a). This process enables the use of all features supported by browser vendors as well as the use of external stylesheets created with CSS. Because of native representation, all elements can be debugged in the browsers built-in element inspector (Bostock, 2016a).

Binding data to graphical elements

D3 uses a generalized approach to bind data to DOM elements. First, a relationship between data and a selection of elements is declared. The relationship is then implemented by defining what happens to elements on a change of underlying data (Bostock, 2012b). The library is capable of creating and destroying elements.

In D3, a selection is an array of DOM elements pulled from the document. CSS selectors are used to identify elements for selection. Operators act on selections, modifying content (Bostock, Ogievetsky, & Heer, 2011).

```
d3.selectAll('p').style('color', 'white');
```

Listing 2: A D3 operator modifying a selection

The `data` operator binds an array of data to a selection of elements. The data array may hold arbitrary values, such as numbers, strings or objects. A data join results in three new sub-selections representing the possible states: *enter*, *update* and *exit* (Bostock, 2012b). On these sub-selections, one can define what happens to DOM elements on a data change.

```
// Join new data with old elements
const circles = svg.selectAll('circle')
  .data(data)
  // Update old elements
  .attr('r', d => d.radius);

// Add missing elements
circles.enter()
  .append('circle')
  .attr('r', d => d.radius);

// Remove surplus elements
circles.exit().remove();
```

Listing 3: Handling the *enter*, *update* and *exit* sub-selections that result from a data join

When a data join is performed, existing elements with corresponding data represent the *update* selection. The *enter* selection represents elements to be added. It contains placeholders for each data value that has no corresponding DOM element. The *exit* selection contains unbound DOM elements, representing elements to be removed (Bostock, 2012b).

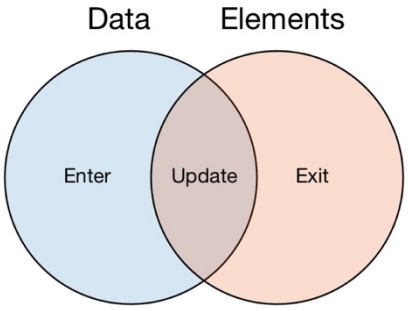

Figure 4: A visualization of the *enter*, *update* and *exit* sub-selection that results from a D3 data join.

The force simulation

The *D3 force layout* is an implementation of a force-directed graph algorithm (Bostock, 2016b). In a force-directed graph, physical laws are applied to distribute the nodes of a graph in an esthetic way. The algorithm assigns repulsive forces to nodes and spring forces to the edges of a graph, which leads to edges of more or less equal length as well as the best use of available space. In a D3 force layout, a pseudo-gravity force keeps nodes centered in the visible area (Bostock, 2016b). By applying repulsive forces to nodes, one can guarantee that nodes do not overlap.

In a force layout, elements are animated. The force layout calculates the position of elements on the screen. A cooling parameter defines the decay of element's velocity and when the animation stops. Upon the addition of new elements to the simulation, or when a user drags-and-drops an existing element, the animation is reheated (Bostock, 2016b).

Documentation

One major advantage of the D3 library is its documentation. Besides various online tutorials and the official wiki, Michael Bostock, the creator of D3, describes core concepts of the library on his personal blog. Furthermore, a large number of examples exist online that demonstrate the manifold possible applications of D3 (https://github.com/d3/d3/wiki/Gallery). Often, these examples provide source code to follow along.

3.3 Sass and Susy

3.3.1 Sass

The stylesheet language *Sass* (Syntactically Awesome Stylesheets) is used to define the layout and the visual appearance of *Kosmo*. Sass augments conventional CSS with features like variables and nested rules (Sass Basics, 2016). These additional functionalities help to organize and maintain large and complex stylesheets.

Sass allows for the splitting of a stylesheet into multiple files. This enables the design of modular stylesheets, in which each file belongs to certain components of a web page.

Variables can be used to define a value once and to reuse it throughout all stylesheets. A designer might use a certain shade of blue to highlight interactivity on a web page. By assigning the corresponding color value to a variable, he can then reuse it for each interactive element on the page. He therefore changes the value of a variable in one place, rather than on all elements individually.

In a stylesheet, rules define the visual appearance of content. A component of a web page, such as a form, might consist of elements like form fields and a button. The form is parent to its fields and the button. Nested rules visually represent parent-child relationships as seen in Listing 4:

```
$background-color: #f2f2f2;
$highlight-color: #dc143e;

.form {
  background-color: $background-color;

  .form-field {
    width: 100%;
  }

  .button {
    color: $highlight-color;
  }
}
```

Listing 4: Example of Sass syntax

Sass is a pre-processor for the creation of CSS (Sass Basics, 2016). Sass stylesheets are compiled into CSS. The Sass code from above would produce the following CSS:

```
.form {
  background-color: #f2f2f2;
}

.form .form-field {
  width: 100%;
}

.form .form-button {
  color: #dc143;
}
```

Listing 5: Simple CSS code

3.3.2 Susy

Susy is a grid framework based on Sass that allows for the defining of custom grids (Susy, 2016). In the field of web design, grid systems are used to create balanced and consistent screen layouts that respond to target device's screen-size. A horizontal grid is an auxiliary construction that divides the space of a screen into columns of equal width separated by gaps. Elements are placed upon columns to define their relative width and position on the screen.

As a web application is retrieved from various electronic devices, such as notebooks, tablets or smartphones, a grid system helps to establish harmonic proportions of content. Furthermore, it allows for the adaption of the screen layout to the viewport size. Instead of pre-defining a certain grid system, Susy facilitates the construction of a custom one.

4 System design and implementation

4.1 Architecture

The *Kosmo* application consists of two main components: a web application built with Meteor that enables real-time collaboration on note collections, and an interactive visualization of input data built with D3.

Both of these components are independent from each other. The web application serves as a frame to host the visualization and provides the visualization component with data. The visualization, in return, presents the received data as floating circles.

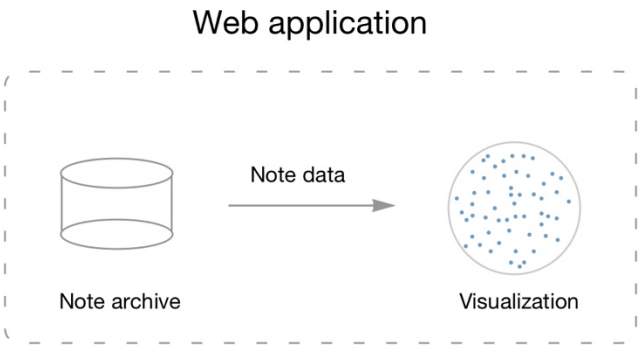

Figure 5: A visualization of the web application that serves as a frame to the D3 visualization.

4.1.1 Technological overview

Figure 6 provides an overview of the technologies that were used to implement *Kosmo*. The web application is constructed with Meteor. A Meteor application resides both on the client as well as on the server-side, providing *Kosmo's* core functionalities, such as storing note data inside of a database and pushing them to connected clients when data changes. The JavaScript library D3 operates inside of the user's browser and is used to implement the visualization component that is responsible for binding input data to graphical elements. Susy also runs inside of the user's browser. Stylesheets that define the layout's adaption to viewport sizes are based on Susy's calculations. D3 and Susy were integrated into the web application by making use of Meteor's package manager.

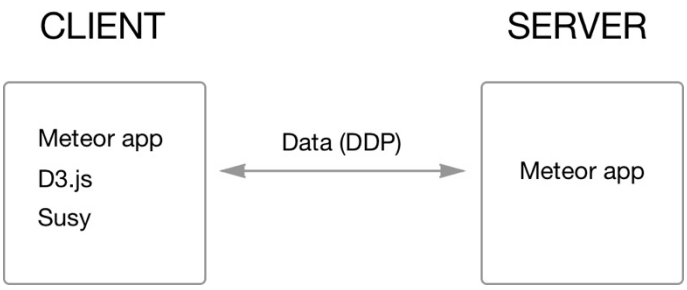

Figure 6: Technological overview of client- and server-side tools.

4.1.2 Special folders

Since the Meteor application lives on the client and the server, it has to be determined which code runs in which environment. In a Meteor application, one uses special folders to organize code (Application structure | Meteor Guide, 2016). All code that is located inside of a directory named `/client` is only available to the client-side of the application. On the other hand, code that is situated inside of a directory named `/server` is only available to the server-side. Code outside of these two special folders is accessible from both environments.

Meteor supports *ES2015*, at the time of writing (October 22, 2016) the latest version of the JavaScript programming language. ES2015 introduces a module system to JavaScript. A module is a cluster of code that provides a certain functionality inside of an individual file. It can be imported from other parts of the program. In Meteor, files inside of a directory named `/imports` are not loaded by default. As recommended by the official Meteor Guide, all of *Kosmo's* application code is placed inside the `/imports` directory. There are two files outside of the `/imports` directory that define entry points for the client and the server: the `/client/main.js` file imports all client-side code and the `/server/main.js` file imports all server-side code. By importing files explicitly, a developer can make sure that only essential files are loaded. This reduces transfer weight and allows the developer to control the file load order.

Inside of the `/imports` directory, data logic is separated from template related code. All code that is concerned with the insertion, modification and publication of data is placed inside of the `/api` directory. Code that defines templates resides in the `/ui` directory. The directory `/startup` is designated for code that configures the system on an initial loading. This separation of concerns inside of *Kosmo's* directories leads to a better reuse of code. Multiple templates may subscribe to the same publication. Changing a publication can be completed in one location and affects all templates that subscribe to that publication.

Defining code outside of the special folders `/client` and `/server` leads to isomorphic code, that is available on the client and the server. *Kosmo* makes use of isomorphic code to define its data logic.

4.1.3 Modularity

Kosmo's code is designed in a modular way to provide structure and to enable reusability. In the following the concept of *Kosmo's* templates, visualization components and stylesheets is described.

Smart and reusable components

The templates of the user interface are designed after a modular principle, which is described in the official Meteor Guide (User Interfaces | Meteor Guide, 2016).

In a Meteor application, templates have a major responsibility, as they are used to subscribe to data from the server-side database and to embed that data into HTML. Furthermore, they contain JavaScript logic to listen and to react to user interactions. Therefore, templates are

called *components* in Meteor to account for their special characteristics (User Interfaces | Meteor Guide, 2016).

The Meteor Guide distinguishes between “reusable” and “smart” components. A reusable component renders solely because of transfer parameters and its internal state. It has no access to global data sources. A smart component, in contrast, is responsible for passing data to reusable components.

Kosmo’s smart components reside in the `/ui/pages` directory. They represent the main views of the application that are connected to their own URL. *Kosmo’s* smart components subscribe to server-side data and distribute that data to reusable components. Furthermore, they define what happens on user interactions. Reusable components are to be found in the `/ui/components` directory. They define the composition and the logic of the user interface’s graphical elements like forms and navigations.

Visualization components

Kosmo defines modules for the creation of interactive visualizations of input data that reside in the `/ui/d3` directory. These modules are hosted and supplied with data by *Kosmo’s* smart components.

Two types of visualizations exist in *Kosmo*: the planet and the universe visualization. The planet visualization takes the data of notes as an input and creates floating circles inside of a container element. The universe visualization in return visualizes complete thought archives as planets in a planetary system that can be selected to enter an archive.

Kosmo’s visualizations are encapsulated within JavaScript functions that can be instantiated with the `new` keyword. To make a visualization accessible to the smart component that hosts it, a set of methods is defined that can be used to add and remove elements from the visualization. The planet visualization is explained in detail in section 4.3.1.

Modular stylesheets

The stylesheets of *Kosmo*, responsible for defining the visual appearance of the user interface, are created according to a simple naming convention for classes in HTML and CSS referred to as the *Block*, *Element*, *Modifier* methodology (BEM) (Rendle, 2015).

Kosmo’s graphical components divide into blocks and elements that are manipulated by modifiers. A block is a top-level abstraction of a graphical component, such as a button, that might consist of sub elements (Rendle, 2015). The same type of button may appear in different forms and colors on a web page. To define alterations of a prototype modifiers are employed.

BEM’s naming convention mirrors the modular nature of *Kosmo’s* graphical components. By employing the BEM methodology, code reuse is enforced and the maintenance of stylesheets is improved. All of *Kosmo’s* stylesheets are to be found in the `/client/stylesheets` directory.

Directory	Content
`client/`	Entry point that imports all client-side code, Stylesheets
`imports/`	
`api/`	Data logic: Collections, Methods and Publications
`startup/`	Code that configures the system on startup
`ui/`	User interface related code
`components/`	Reusable components
`d3/`	Visualization components
`pages/`	Smart components
`server/`	Entry point that imports all server-side code

Table 1: A shortened version of *Kosmo's* folder structure that shows essential folders.

4.2 Enabling real-time collaboration

To enable real-time collaboration a web application was build that is based on the Meteor framework. A real-time web application reacts to changes in the server-side database by pushing the modified data set to all connected clients. In addition, Meteor completes the real-time user experience by responding immediately to user interactions on data, a concept referred to as Optimistic UI (section 3.1.2). To configure the system, the setup in the following section was constructed.

4.2.1 Conceptual data model

The *Kosmo* application allows data sets to be saved in a database. To describe the relationships between those data sets, an entity-relationship diagram was created. Entity-relationship diagrams were coined by Peter Pin-Shan Chen in 1976 as a tool for database design (Chen, 1976).

For *Kosmo,* three types of entities are defined: *Projects*, *notes* and *users*. A project represents an individual thought collection and may contain multiple notes. It belongs to one user, whereas a user might have multiple projects. A note also belongs to one user. In contrast, a user might be the author of multiple notes.

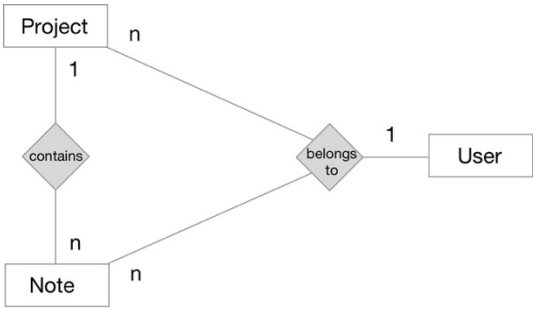

Figure 7: An entity-relationship diagram of *Kosmo*.

4.2.2 Collections

To implement the conceptual data model, Meteor's mechanism for storing persistent data inside of *collections* is used. A Meteor collection is an abstraction of the MongoDB API and enables communication with the server-side Mongo database and the client-side Minimongo (Collections and Schemas | Meteor Guide, 2016).

A collection consists of individual items called documents. *Kosmo* declares the following collections:

- The documents inside of the `Projects` collection represent individual projects. Amongst others, they contain information about the name and the author of a specific project. Furthermore, they define whether a project is public or private.
- The `Notes` collection holds documents embodying notes. A note contains the unique identification number of the project to which it belongs. Therefore, the relation between a project and its notes can be traced by the application. Moreover, it consists of textual content and information on its author. To allow for selecting notes based on tagging, a note contains an array of keywords.
- The `Users` collection contains documents on users. Among others, a user is defined by a username and a password.

In Meteor, collections are by default schema-less (Collections and Schemas | Meteor Guide, 2016). This means that no structure is enforced on individual documents of a collection. A document of a specific collection may have arbitrary fields. This can lead to confusion, because a developer might not know the fields of a collection's documents. To define common characteristics for collections, *Kosmo* integrates a package called `aldeed:simple-schema`. The package allows for the attachment of schemas to collections. A schema determines which fields belong to a certain collection and which values they are permitted to have. For example, the schema attached to the `Notes` collection ensures that all documents within that collection contain a field that gives information on its parent project. When inserting a document into the `Notes` collection, the document will be automatically validated against the schema. If it differs from the defined blueprint, the insertion will not be allowed. Listing 6 presents an abbreviated version of the `Notes` collection and its schema.

```
const Notes = new Mongo.Collection('notes');
Notes.schema = new SimpleSchema({
  projectId: {
    type: String,
    regEx: SimpleSchema.RegEx.Id,
  },
  author: {
    type: String,
  },
  text: {
    type: String,
    max: 512,
  },
});
```

Listing 6: Abbreviated version of the validation schema defined for the Notes collection

4.2.3 Publications and subscriptions

To push server-side data to client-side collections, the server defines publications. The client-side smart components subscribe to those publications, and embed the reactive data into HTML code. As soon as data changes in the server-side collection, the new data set is pushed to the client-side collection, and the client updates the representation of that data (Greif & Coleman, 2015).

A Meteor publication allows for configuring which subset of data is published to clients (Publications and Data Loading | Meteor Guide, 2016). In *Kosmo* the publication `notes.inProject` not only publishes data of all notes that belong to a particular project, but also the data of the specified project itself. The client-side smart component `planet.js`, managing *Kosmo's* planet view, subscribes to that data and draws the visualization of notes on the screen. To publish related documents the package `reywood:publish-composite` was integrated into *Kosmo*.

Notes and projects may contain sensible data. Therefore, it is important to verify whether the current user is allowed to view the content of the requested documents. Validation is done in server-side publications. The publication `notes.inProject` only publishes data when the project either belongs to the current user or is public.

```
if (!project.belongsTo(this.userId) && project.isPrivate()) {
  /* Abort publication */
  return this.ready();
}
```

Listing 7: Verifying if the current user is allowed to subscribe to a publication that pushes note and project data to the client.

4.2.4 Methods

Methods operate on collections, and allow for inserting, modifying, and removing documents. When declared isomorphic, methods run on both the client- and the sever-side component of a Meteor application. The execution of a methods is triggered by a user interaction that aims at modifying data. When a user inserts a thought into the system by entering text into the `Note_create` form, a method for inserting a document into the `Notes` collection is executed. On the client-side, the method inserts a note into the local collection. The user thus experiences direct feedback, i.e., the note is immediately displayed inside the visualization. However, it is only a simulation of the insertion, because the representation of that note is based on local data, where the server-side collection is the trusted source of data (Methods | Meteor Guide, 2016). At the same time, the same method is executed on the server. If the result of the server-side insertion differs from the simulation, the simulation is adapted (Stubailo, 2015).

Before inserting a note into a `Notes` collection, the method verifies whether the current user has the right to insert a note into a specific project. If he is not logged in, or the project does not exist, the insertion is aborted.

4.2.5 Routing

The prototype of *Kosmo* foresees that users can share planets. To do so the owner of a planet needs to change its privacy setting to public, and send the link of the planet to people with whom he wants to collaborate with. This feature implies that a routing mechanism needs to be included in the application. By default, Meteor does not ship with a routing functionality. To integrate routing into *Kosmo,* a package called `kadira:flow-router` is incorporated. The package allows for mapping URLs to templates. Listing 8 portrays the abbreviated route for the planet page of *Kosmo*.

```
FlowRouter.route('/:username/:projectSlug', {
  name: 'planet',
  action() {
    BlazeLayout.render('App_body', { navigation: 'Nav_main', main:
'Planet_page' });
  },
});
```

Listing 8: The definition of a route with the flow-router package.

If a user enters an URL with a path similar to `/felix/first_project` into his browser, a route named planet will be invoked. The planet route triggers the rendering of the templates that form the planet view: template `App_body` as the main layout, template `Nav_main` for navigation elements and the smart template `Planet_page` to hold the interactive visualization. The route parameters `username` and `projectSlug` are used to identify the corresponding data set which fills the visualization. The parameter `username` refers to the username of the project owner, and the parameter `projectSlug` represents the project name, converted to characters that are valid inside of URLs. The combination of `username` and `projectSlug` is unique, due to the fact that the system manages usernames, preventing two or more users from having the same username.

4.3 Visualizing reactive data

4.3.1 The planet visualization component

The module `d3/planet.js` consists of the function `Planet` that creates an interactive visualization of input data inside of a specified container element. The visualization consists of a main circle that holds smaller circles, referred to as nodes, which reference data. These nodes are part of a physical simulation that assigns repulsive charge forces to them. A pseudo-gravity force pulls nodes to the center of the main circle. The physical simulation is a modification of the D3 force layout as described in section 3.2. The visualization’s graphical elements (the main circle and the nodes) are of the format Scalable Vector Graphics (SVG).

The function acts as a class; it is a blueprint for visualizations of the type planet. In JavaScript a function is an object, therefore the `Planet` function can be instantiated with the `new` keyword. It takes a CSS selector as a transfer parameter to indicate in which container element the visualization will be initialized.

```
const planet = new Planet('.visualization');
```

Listing 9: Creating a visualization of the type planet inside of a specified container element.

On instantiation the `Planet` function creates an SVG container element to hold the visualization. Within this container, the function inserts an SVG circle element to hold the visualization's nodes. The D3 force layout is then instantiated and its gravity, friction and charge parameters are configured. The size of the visualization is based on the width of the container element specified by the CSS selector. The visualization scales to changes in size of the browser window due to its `resize` function.

To operate on an instance, the `Planet` function provides for methods. It is possible to add or remove nodes from the visualization by calling a method on the instance of the `Planet` function. The `addNode` method for example takes data as a transfer parameter and creates a graphical element that is bound to that data.

```
planet.addNode({ text: 'I had a dream about an green elephant' });
```

Listing 9: Adding a graphical element to the planet visualization by passing data to the addNode method.

In addition, the `Planet` function offers methods to highlight nodes. In *Kosmo* nodes are highlighted to flag them as selected. A selected node is of a striking color. The visualization's nodes can be highlighted by calling the methods `selectNode`, to highlight a single node, and `selectNodes` to highlight a group of nodes. To restore all nodes' colors to their initial value, the method `clearSelection` is called.

The `Planet` function is truly modular, meaning that it is independent of the Meteor web application that surrounds it. It would function in the same way when embedded into another context. Its methods provide an interface for applications to connect to.

4.3.2 The planet page

The planet page is the core of the application. It offers a graphical user interface to collaborate on note collections and to explore contents as described in chapter 2.

The user interface of the planet page is based on the `Planet_page` smart component that subscribes to the reactive data of a `Notes` collection and provides its data to the three building blocks that form the planet page:

- The `filter` component lists tags and collaborators that belong to a project. When a user selects a tag or a collaborator, corresponding notes from the visualization are highlighted.
- The `Planet` function establishes an interactive visualization of notes represented as floating circles driven by a physical simulation.
- When a user selects a circle from the visualization, this action reveals the textual content inside of the `Note_contentView` component.

To react to data changes and to user interactions, it defines *event listeners*. A listener triggers the execution of code on a given event. Two types of listeners are relevant for this template: those that listen to a change in data and those that listen to user interactions.

To fill the components of the planet page with content, the `Planet_page` template subscribes to the dynamic data of the current project and the notes that belong to it. The subscription is specified by passing the URL parameters `username` and `projectSlug` to the `notes.inProject` publication. These parameters represent a unique combination and therefore allow for an unambiguous allocation of a project and its notes.

```
const author = FlowRouter.getParam('username');
const slug = FlowRouter.getParam('projectSlug');
this.subscribe('notes.inProject', { author, slug }, () => {..};
```

Listing 10: Subscribing to a project and its notes based on the username and projectSlug parameters.

When all of the initial data has been sent to the client, the `Planet` function is instantiated. To fill the visualization with circles, a listener is established, that adds a circle for each document of the `Notes` collection. The listener observes the `Notes` collection. When a user adds or removes a document from the collection, the listener reacts by creating or destroying circles within the visualization.

```
const planet = new Planet('.visualization');

Notes.find().observe({
  added(newDocument) {
    planet.addNode(newDocument);
  },
  removed(oldDocument) {
    planet.removeNode(oldDocument._id);
  },
});
```

Listing 11: A listener observes the Notes collection and instructs the visualization component to create and to destroy circles.

Revealing the content of notes

The `Planet_page` component defines a variable called `noteId` that indicates whether a circle of the visualization is selected. A helper function listens for changes on the `noteId` variable. When the variable contains a valid id from a document of the `Notes` collection, the helper function is triggered, and the document's data is passed to the `Note_contentView` component for rendering.

There are two ways to assign a value to the `noteId` variable and trigger the revelation of a note's content:

- When a user hovers the mouse pointer over a circle of the visualization, the `mouseover` event listener retrieves the `_id` value from the circle element and assigns it to the template variable.
- The prototype of *Kosmo* foresees that it shall be possible to share links that lead to a visualization in which a note is already preselected. On a click event the `_id` value from a circle element is appended to the URL as a query parameter. A listener is set up to listen for changes in the URL. If the URL contains a valid note id, the listener calls a method that flags the corresponding circle as selected and sets the template's `noteId` variable.

Highlighting circles from the visualization by selecting a tag

The `filter` component lists all tags and all users that belong to a project. It receives its data from the parent `Planet_page` smart component.

When a user clicks on an element from the list its text is pushed into the URL as a query parameter. A listener recognizes that the URL contains a filter element and searches through the note archive for notes that contain the specified element. An array of note ids is passed to the visualization's `selectNodes` method, and all corresponding circles within the visualization are highlighted.

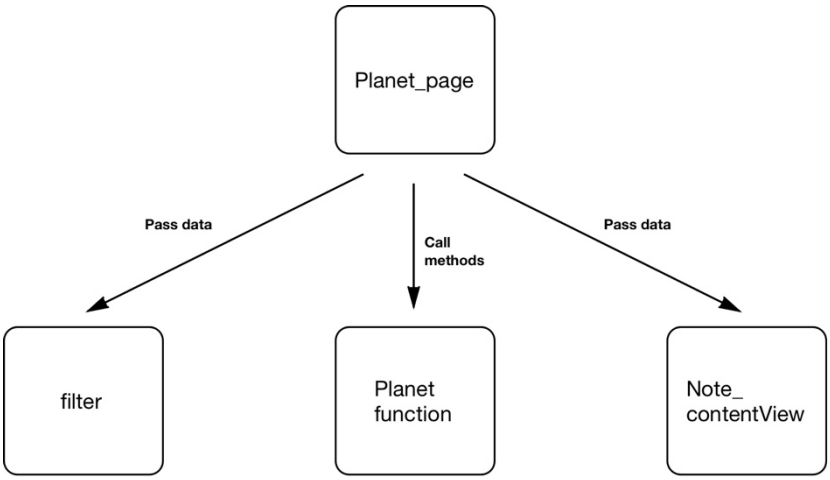

Figure 8: The Planet_page smart component distributes data to the components of the planet page.

4.4 Mobile version

According to a survey conducted by *ARD* and *ZDF*, the mayor public service broadcaster in Germany, 66 percent of the German population uses a smartphone to access the internet (Frees & Koch, 2016). Web applications and web pages are by default accessible to every device that has a browser software, such as smartphones, tablets or notebooks. These devices differ in screen-size and a web page needs to be optimized to present well-organized information across devices.

Two common approaches for doing so are presented in the following.

- *Responsive design:* The layout of the web page responds to the screen-size of the device being used to view that site (Marcotte, 2010). A responsive web page arranges and presents the document's elements differently when the size of the viewport changes. It is important to underscore that in a responsive web page, the HTML document, responsible for structuring contents, stays the same. Stylesheets are used to determine the viewport size and to adapt the presentation of contents (Marcotte, 2010).
- *Mobile web page:* A mobile web page exists besides a desktop web page. Depending on the viewport size of the target device either the mobile web page or the desktop web page is send to the client.

Kosmo uses a hybrid approach of the two presented solutions. Its main target devices are desktop computers, notebooks and tablets, which offer the necessary screen size to present the interactive visualization of notes and to allow for a meaningful exploration of the content. Across these devices, the web application behaves responsively. The grid framework Susy is used to divide the screen into columns that scale to viewport sizes. The planet page of *Kosmo* comprises 12 columns (a column being the basic unit of width). The three main components of the planet page - the filter component, the visualization and the content view - span a specific number of columns. Upon a change in viewport size, the components are scaled accordingly.

Nevertheless, the prototype of *Kosmo* also envisages a mobile view of the application that enables the insertion of thoughts into archives from wherever the user is located. The mobile view differs from the desktop view of the user interface in one crucial point: elements of the mobile view are static, whereas elements of the desktop view are animated within D3's force layout. These different forms of visualizing elements require individual application logic. Therefore, the decision to implement two separate sets of smart components for the mobile view and for the desktop view was made. When a user enters the URL of *Kosmo* inside of a browser, the application determines whether the target device is of the size of a smartphone. When the evaluation is positive, the mobile view smart components are sent to the browser and when it is negative, the desktop version is returned. However, reusable components are shared between the mobile and the desktop version of *Kosmo*.

```
const mobile = () => $(window).width() < 800;
Session.set('mobile', mobile());
FlowRouter.route('/', {
  name: 'universe',
  action() {
    if (Session.get('mobile')) {
      BlazeLayout.render('App_body', { navigation: 'Nav_mobile',
main: 'Universe_mobile_page' });
    } else {
      BlazeLayout.render('App_body', { navigation: 'Nav_main',
main:'Universe_page' });
    }
  },
});
```

Listing 12: The view port size of the target device is retrieved and corresponding smart components are send to the client.

5 Discussion

The result of this thesis is *Kosmo,* a web-based application for visually collecting thoughts. The application can be visited at http://kosmo-92397.onmodulus.net/.

The motivation behind the conceptualization and development of the application was to create a reservoir of thoughts in which a user could insert spontaneous thoughts and ideas, with the potential of leading to inspiration. The layout of these thoughts is meant to enable users to interactively explore their own ideas. Conventional note taking applications did not satisfy these claims due to their presentation of notes in the form of a list. A prototype was therefore designed in which notes are represented as floating circles inside of a virtual space. To allow for creative brain storming sessions on note collection, it was deduced that the envisaged system would need to be based on a real-time web application, one that pushes server-side data to all connected clients on a data change.

After the prototype was built, a set of technologies was chosen to implement the envisaged application. The JavaScript framework Meteor was used to implement *Kosmo's* real-time capabilities and to provide for the application's core functionalities such as storing notes inside of a database and creating users. The Meteor web application serves as a frame for the interactive visualization, which was implemented with the JavaScript library D3. A module was realized to take the application's note data as an input and represent that data as animated graphical elements inside a specified container.

5.1 Evaluation

The technological demands described in chapter two envisioned a system that immediately responds to user interactions and that portrays inserted notes to all clients as soon as they are inserted. These qualities, referred to as real-time, could be implemented by making use of the Meteor framework. This approach has proved to be efficient and reduced the complexity of the system. Real-time web applications are usually implemented through a combination of different libraries and frameworks. To get a real-time application up and running, a developer needs to configure all the individual components of the system to work together properly. Meteor eased the implementation of *Kosmo's* envisioned real-time capabilities by providing a well-orchestrated stack of technologies. In addition, Meteor's packages ecosystem, Atmosphere, provided for useful packages that were then integrated into *Kosmo*. Functionalities such as routing, creating users and client-side form validation are based on Atmosphere packages. In addition, the D3 library, the Sass preprocessor, and the Susy framework were integrated into *Kosmo* as a package with Meteor's package manager. Not having to include those packages manually and managing their dependencies further eased the development process.

The D3 library was used to connect note data to graphical elements and to simulate levitation. The declarative way of binding data to graphical elements made it easy to add and remove elements from the visualization. Instead of instructing D3 to create circles, a relationship between data and elements was declared. In addition, it is advantageous that D3 uses web standards for the creation of elements. Therefore, external stylesheets could be used

to define the look of the visualization component's graphical elements. However, defining the composition of a graphical element is quite complex. Defining templates, which serve as a blueprint for graphical elements, is not possible in D3. Instead, a complex element which consists of various nested sub elements, such as a menu, must be defined by executing D3's `append` operator multiple times. Code that is created in such a way is quite convoluted and can not be reused.

The cooperation of D3 and Meteor primarily lead to confusion due to the fact that D3 does not rely on Meteor's rendering engine. Therefore, D3 related code could not simply be included into Meteor's templates. The problem was solved by creating the visualization component, which is a function that creates a D3 visualization on instantiation. It provides for a set of methods to add and remove elements from the visualization. A set of listener was established to observe *Kosmo's* collections. When data is altered, the listener instructs the visualization component to adapt its representation.

Another technological demand for the application was that it can be retrieved from mobile devices. This functionality was implemented by creating two separate sets of smart components and by making the layout of the user interface responsive to the target device's screen-size. Even though the technological demand of enabling users to insert notes from a mobile device was achieved, the mobile version of *Kosmo* remains quite limited.

5.2 Future work

The project resulted in a web application that visualizes notes supplied by groups of users within a physical simulation. However, the application's features are not yet complete at the time of writing and more development is necessary to make *Kosmo* fully functional.

To test *Kosmo's* usability, a small experiment was conducted. A group of people were asked to insert a dream they recently had into a thought collection. Further, they were instructed to give feedback on how they managed to orient themselves within the user interface, and to state whether the application seemed useful to them. Even though the settings of the test do not allow for general statements, its results were an inspiration for future work. The test collection can be visited at http://kosmo-92397.onmodulus.net/milan/Traumfaenger.

At its current state, the user interface of *Kosmo* lacks instructions explaining how to use the application. Users of the test group reported that they did not know how to share a note collection and how to insert tags into the system. Therefore, it would make sense to include a help function into the application that guides new users through the functionalities of the application.

Kosmo's conceptual objective of allowing for a meaningful online collaboration was only partially achieved. Even though the application is based on an architecture that makes online collaboration in real-time possible, *Kosmo* lacks the function of responding to thoughts that were inserted into the system by others. Therefore, communication is difficult and ideas can not relate to others. A possible solution would be to enable users to drag-and-drop a note on top of another note to establish a connection. This connection could be visualized as an edge of a force-directed-graph.

Furthermore, the mobile version of *Kosmo* needs to be enhanced. At its current state, it only serves for entering notes into the system. For further development, it would be necessary to also present thought collections on mobile devices and to allow for their exploration. Therefore, new ways of presenting notes on a small screen-size need to be researched.

Kosmo introduces new ways of collecting thoughts within an application. Due to its simplicity, possible applications are manifold.

Figures

References

Application structure | *Meteor Guide*. (2016). Retrieved 22 October, 2016, from https://guide.meteor.com/structure.html

Bostock, M. (2016a). *D3.js - Data-Driven Documents*. Retrieved September 15, 2016, from https://d3js.org/

Bostock, M. (2012a). *d3.js - For Protovis Users*. Retrieved September 18, 2016, from http://mbostock.github.io/d3/tutorial/protovis.html

Bostock, M. (2016b). *Force Layout*. Retrieved September 16, 2016, from https://github.com/d3/d3-3.x-api-reference/blob/master/Force-Layout.md

Bostock, M. (2012b). *Thinking with Joins*. Retrieved September 18, 2016, from https://bost.ocks.org/mike/join/

Bostock, M., Ogievetsky, V., & Heer, J. (2011). D³: Data-Driven Documents. *IEEE transactions on visualization and computer graphics*, (pp. 2301-2309).

Brehm, S. (2013, November 11). *Isomorphic JavaScript: The Future of Web Apps - Airbnb Engineering*. Retrieved September 05, 2016, from http://nerds.airbnb.com/isomorphic-javascript-future-web-apps/

Chen, P. (1976). The Entity-Relationship Model - Toward a Unified View of Data. *ACM Transactions on Database Systems*, (pp. 9-36).

Collections and Schemas | *Meteor Guide*. (2016). Retrieved September 03, 2016, from https://guide.meteor.com/collections.html

Frees, B., & Koch, W. (2016). *Dynamische Entwicklung bei mobiler Internetnutzung sowie Audios und Videos.* Retrieved October 04, 2016, from http://www.ard-zdf-onlinestudie.de/fileadmin/Onlinestudie_2016/0916_Koch_Frees.pdf

Greif, S. (2014, January 02). *Understanding Meteor Publications & Subscriptions*. Retrieved August 29, 2016, from https://www.discovermeteor.com/blog/understanding-meteor-publications-and-subscriptions/

Greif, S., & Coleman, T. (2015, September 30). *Episode 01: What is Meteor?* Retrieved August 29, 2016, from https://www.discovermeteor.com/blog/episode-01-what-is-meteor/

Haverbeke, M. (2014). *Eloquent JavaScript: A Modern Introduction to Programming.* San Fransisco: No Starch Press.

Introduction | *Meteor Guide*. (2016). Retrieved August 27, 2016, from https://guide.meteor.com/

Luhmann, N. (1982). Kommunikation mit Zettelkästen. In *Öffentliche Meinung und sozialer Wandel* (pp. 222-228). Opladen: VS Verlag für Sozialwissenschaften.

Marcotte, E. (2010, May 25). *Responsive Web Design*. Retrieved October 02, 2016, from http://alistapart.com/article/responsive-web-design

Methods | *Meteor Guide.* (2016). Retrieved October 24, 2016, from https://guide.meteor.com/methods.html

Mills, C., & Willee, H. (2016, September 11). *Client-Server Overview*. Retrieved October 01, 2016, from https://developer.mozilla.org/en-US/docs/Learn/Server-side/First_steps/Client-Server_overview

Node.js. (2016). Retrieved August 27, 2016, from https://nodejs.org/en/

Publications and Data Loading | *Meteor Guide*. (2016). Retrieved October 26, 2016, from https://guide.meteor.com/data-loading.html

Rendle, R. (2015, April 02). *BEM 101*. Retrieved October 24, 2016, from https://css-tricks.com/bem-101/

Sass Basics. (2016). Retrieved October 02, 2016, from http://sass-lang.com/guide

Stubailo, S. (2015, May 27). *Optimistic UI with Meteor*. Retrieved September 03, 2016, from http://info.meteor.com/blog/optimistic-ui-with-meteor-latency-compensation

Susy. (2016). Retrieved October 02, 2016, from http://susy.oddbird.net/

User Interfaces | *Meteor Guide*. (2016). Retrieved October 23, 2016, from https://guide.meteor.com/ui-ux.html

Eidesstattliche Erklärung

Ich versichere hiermit, die vorgelegte Arbeit in dem gemeldeten Zeitraum ohne fremde Hilfe verfasst und mich keiner anderen als der angegebenen Hilfsmittel und Quellen bedient zu haben.

Köln, den TT. Monat JJJJ

Unterschrift

(Vorname, Nachname)

TH Köln
Gustav-Heinemann-Ufer 54
50968 Köln
www.th-koeln.de

Technology
Arts Sciences
TH Köln