Live Visualization and Editing of User Behavior in iObserve

Bachelor Thesis

Daniel Banck

March 30, 2017

Kiel University
Department of Computer Science
Software Engineering Group

Advised by: Prof. Dr. Wilhelm Hasselbring
Dr. Reiner Jung

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Kiel, 27. April 2017

Abstract

The iObserve project collects user behavior information and derives from that information user behavior models. The visualization of these models supports operators to understand better how users are using an application, how performance bottlenecks can occur, and how the deployment of the system changes.

In this thesis, we present an approach and implementation for a behavior model visualization service. The service provides a live visualization of user behavior graphs for operators to inspect. Furthermore, operators can create and modify user behavior graphs to reflect potential changes in user behavior and take future marketing campaigns into account. In an evaluation, we compare different Javascript libraries for their technical fit for the visualization and editor service.

Contents

Chapter 1

Introduction

1.1 Motivation

Tracking user behavior in applications is helping to comprehend how users are interacting with an application. User behavior information can be used to visualize which pages users are visiting, how users are navigating through an application, and where users are leaving a website.

Based on this information performance bottlenecks can be found, the system evolved, and only the most frequented parts of an application scaled. For example, if we take future marketing campaigns into account, it is possible to make manual changes to the user behavior graph. With this, we have got a new model of potential future user behavior and can derive new system requirements in advance.

The iObserve project collects user behavior information and combines it into user behavior models. Visualization of these models can give operators a better understanding of how users are using an application. Manual changes of the visualization can be given back into iObserve for further actions. One goal of this thesis is to build an application for operators with appropriate technologies.

1.2 Goals

The goal of this thesis is to add live visualization of user behavior models to iObserve. That gives an operator better insights into an application. Furthermore, we enable operators to make dynamic changes to the user behavior model graph, which allows them to derive new system requirements from the modified graphs. For a better realization, we divide our main goal into multiple more specific goals.

1.2.1 G1: Evaluation of Technologies for Live Visualization of User Behavior

There are a lot of different technologies for building browser applications today. Our goal is to assess appropriate technologies and compare them using the "Goal Question Metric" approach. We evaluate different Javascript libraries which support a reactive programming pattern to support the incoming stream of live data which should be visualized.

1.2.2 G2: Implementation of User Behavior Visualization in iObserve

We use the selected technologies to develop the live user behavior model visualization. We build the application following a microservice [16, 21, 43] architecture. Each part is a small application. We define the requirements and design the application. The implementation is grouped into two sub-goals.

G2.1: We implement the Java service, which includes a service for operations on user behavior models, like creation, modification and deletion and a graph database for storing them. Furthermore, we implement the Javascript application running inside a browser with the visualization of the user behavior models.

G2.2: We extend the Javascript application with the possibility to create and edit the user behavior model graphs and store all modifications inside the graph database.

1.3 Document Structure

First, we describe the foundation and technologies one needs to understand for this thesis in Chapter 2. Furthermore, we list potential technologies which can be used but without evaluating those yet. In Chapter 3 we are explaining our overall approach and describe the steps we are taking for achieving the goals of this thesis in more detail. Chapter 4 describes the evaluation and selection of technologies we are using for the implementation. We present related work in Chapter 5, and summarize our work and point possible improvements of the approach and implementation in Chapter 6.

Chapter 2

Foundations and Technologies

In this chapter, we describe the technologies and underlying principles we use to realize the live visualization and editing of user behavior models. In Section 2.1 we explain the iObserve approach, followed by the concept of live visualization in applications in Section 2.2. Then we define two types of models, the Palladio Component Model in Section 2.3 and user behavior models in Section 2.4. In Section 2.6 we explain the concept of reactive programming. Next, we describe different technologies starting with WebSockets in Section 2.5 and Javascript, including two libraries, in Section 2.7. Finally, we describe tools for working with graphs, such as the graph library Cytoscape.js in Section 2.8 and the graph database Neo4j in Section 2.9.

2.1 iObserve

iObserve [22] supports operators to observe and detect anomalies of software systems. It also provides support for automatic and human in the loop adaptation.

Figure 2.1 shows a simplified version of the iObserve approach. iObserve facilitates a model-driven engineering approach. Based on design time models, depicted as application model, artifacts are generated to be executed during run-time. After deployment, iObserve monitors the application and the environment. This monitoring data can then be used to adapt the application model including architecture and user behavior. Based on this model, iObserve performs analysis to discover anomalies in the execution of the modeled software system.

2.2 Live Visualization

The iObserve project provides a runtime analysis of monitoring data for software systems. Based on the monitoring data, a set of models, including behavior models, are generated and continually updated to reflect the current state of the system. For a better understanding, these models need to be visualized.

Live visualization can be found in many applications dealing with user data. Users create new data while using an application and this data is supplied as a constant stream to the visualization. For example, AppDyamics [6] is using live visualization for infrastructure

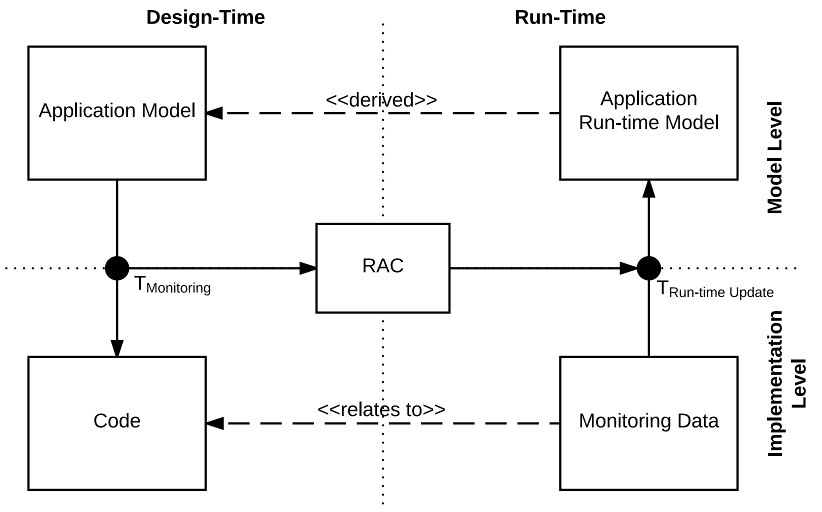


Figure 2.1. Simplified model of the iObserve approach

visibility and end-user monitoring. Another use case for live visualization is ExplorViz, which provides trace visualization of large software landscapes.

2.3 Palladio Component Model

The Palladio Component Model (PCM) [8] describes architectures, behavior models, and deployment and execution targets of software systems. It contains components, connectors, interfaces, individual service behavior models and more. With it, it is possible to make assumptions about software regarding performance and reliability during design time.

In the PCM components can offer and require interfaces, they can be assembled into a system by connecting provided and required interfaces. Allocations are the mapping of components to hardware. Hardware platforms are modeled as resource containers. A component can also offer services, which are described by a Service Effect Specification (SEFF). These specifications contain a range of actions, in which an action can be a call to a required interface or a demand to a specific resource.

2.4 User Behavior Models

A user behavior model contains data about a particular type of user of a specific application. It is created by combining data from users with similar behavior into one model.

One way to model user behavior is using Markov chains [3]. User interactions are represented as Markov states, and transition between those states are enhanced with probabilities. Another way is using graph-based models. These are usually used for fraud detection [9] or identity and access management [51]. The WESSBAS approach [49] describes a couple more possibilities.

2.5 WebSockets

WebSockets enable real-time communication between a server and a client. They are primarily used in web browsers and web servers, but any client or server application can use them. Its protocol is independent of HTTP and works over a single TCP connection. Contrary to HTTP this connection can be utilized in both ways. A server can push data to clients and clients can fetch data from a server without establishing a new connection for each request. Therefore, WebSockets are well suited for real-time applications like chats, collaboration tools, and live visualizations.

We use WebSockets to push changes of the user behavior model graph to multiple browsers used by operators. Therefore, the graph is always kept up to date without the need of refreshing the page.

2.6 Reactive Programming

Reactive programming [50] is programming with asynchronous data streams. Streams transport data objects, which can be anything like values, user input events, and complex data structures. For working with this data, trigger functions can be attached to a stream. These functions are triggered every time a new data object arrives and are called observers. Observers stop listening for new data objects when they are detached, or the stream is closed.

For working with streams, there are functions to merge, transform and filter them. For example in Figure 2.2, a stream containing random numbers and characters is filtered for only numbers, and then these numbers are multiplied by two. That results in a new stream containing only even numbers.

Our live visualization service must process incoming model changes and present the user with updates. These can be additions, modifications or deletions of models. While handling these changes, the user interface should be responsive and usable.

2.7 Javascript Libraries

Javascript is a scripting language which is intended to be used in browsers to validate forms and implement user interface functionality. It is the most common implementation of the ECMAScript standard. Today Javascript works in many more environments than browsers.

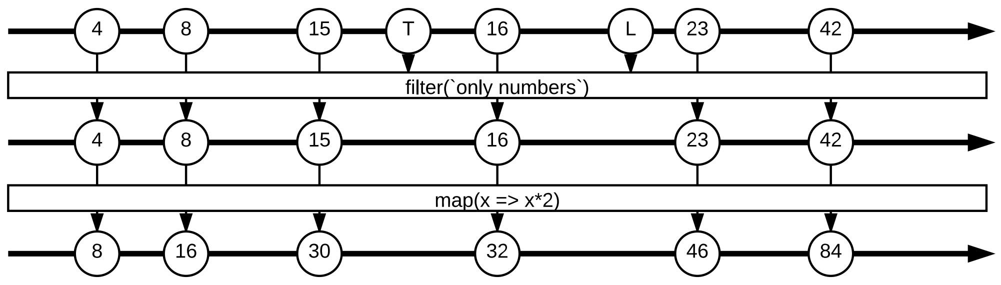


Figure 2.2. A stream of numbers and characters is filtered and transformed

With Node.js®, built on top of Chome's V8 Javascript engine, it is possible to write server and command line applications in Javascript. Furthermore, with the help of Electron or React Native, Javascript can be used to write desktop or native mobile applications.

ECMAScript 6 is a significant update to the language and adds many new features like *arrow functions*, *generators*, and *template literals*. *Arrow functions* are a shorter a function expression without an own binding of the current scope variable *this*. *Generators* are special types of functions that work as a factory for iterators. With them, it is possible to write a single function, which can maintain its state, to define an iterative algorithm. *Template literals* can contain expressions for interpolation and can be used as multi-line strings. We use all three features in our implementation.

Javascript allows building applications which do not need to reload the page on every update. All necessary code is retrieved with a single page load, and additional resources are loaded dynamically. There are frameworks like Angular, React, and Elm which support building a single page application. In this thesis, we investigate the two more modern libraries Elm and React which support reactive programming patterns (see Chapter 4).

2.7.1 Elm

Elm [17] is an open source programming language which compiles to Javascript. It uses a functional programming paradigm and addresses many weaknesses of Javascript, like confusing equality operators and implicit type conversion. All data structures in Elm are immutable by default; everything is typed, and the compiler can detect problems during compilation to avoid runtime exceptions.

Elm suggests a simple pattern for web application architectures, called the Elm Architecture. Each application can be separated into three parts. A model which holds the state of the application (model), a collection of functions which update the state (update) and a collection of functions which render the state as HTML (view).

Listing 2.1. A skeleton following the Elm Architecture

```
import Html exposing (..)

-- MODEL
type alias Model = { ... }

-- UPDATE
type Msg = Reset | ...

update : Msg -> Model -> Model
update msg model =
  case msg of
    Reset -> ...
    ...

-- VIEW
view : Model -> Html Msg
view model =
  ...
```

User input, like clicking a button or entering text, will call an update function, which updates the state and then is rendered back to a user with a view function. Instead of rendering HTML it is also possible to execute commands or create subscriptions. These methods can be used to make HTTP requests or listen for WebSocket messages.

2.7.2 React

React [35] is an open source Javascript library for creating user interfaces. It allows us to write declarative components, which then can be composed to make complex UIs. Components in React have a render method and can get a set of properties. Calling a component with the same properties always renders the same results. This feature makes the code more predictable and easier to test and debug.

Inside a render method, React mixes an XML-like syntax called JSX with Javascript. React allows to write components in plain Javascript, but with JSX, template code and logic can be stored together in one file.

Listing 2.2. A simple react component with JSX

```
class HelloMessage extends React.Component {
  render() {
    return <div>Hello {this.props.name}</div>;
  }
}
```

```

ReactDOM.render(<HelloMessage name="World" />, mountNode);
```

There are two ways to manage state with React. Each component can have a state, and it is possible to have a global state for the whole application. When applications are getting bigger, it is much easier to use the latter and follow an Elm Architecture like pattern. That ensures when data changes, only a single state has to be updated, and there is no need to keep track of which component contains what state. All components get notified of the state change and update their rendered content if required. *Redux* [41] is a library for keeping a single predictable state container for an application and inspired by the Elm Architecture. It can be used together with *React Redux* [39] to manage a global state and emit update actions for changing this state.

Since React is only a library for the view layer, it is missing a way to handle asynchronous tasks, like fetching data from another service. There are libraries like *Redux Thunk* or *Redux Saga*, which provide this functionality. *Redux Thunk* uses promises for executing tasks and waiting for results, while *Redux Saga* uses a more complex approach with generators. A *Saga* is a generator function which makes asynchronous calls and yields the results.

With *React Native* [36] is it possible to build native mobile applications using the same principles and knowledge of React. A *React Native* application is not running inside a web view. Instead, it renders real native components for the layout and runs a small Javascript engine for the business logic. For example, a *React Native* `ScrollView` component is rendered as `UIScrollView` on iOS and `ScrollView` on Android. The communication between the native Java or ObjectiveC code and the Javascript code is done by bridges provided by *React Native*.

This approach allows developers to reuse the business logic of a React web application for a native mobile application and only requires them to rewrite the view layer. When a developer is writing the view layer for a mobile application in *React Native*, the same design principles as React apply, and they do not need any specialized knowledge of mobile application development.

2.8 Graph Library Cytoscape.js

Cytoscape.js [15] is an open source graph library for graph analysis and visualization written in Javascript. Without further configuration, Cytoscape.js creates an interactive graph from a list of nodes and a list of edges. A user can interact with the graph using a range of gestures, like pinch-to-zoom or panning.

Cytoscape requires a layout and a stylesheet to create a graph visualization. It has a couple of predefined layouts, named "circle" and "breadthfirst." Every aspect of the graph is customizable with a stylesheet, including node size and color, edge thickness and whether to draw arrows on edges. Furthermore, Cytoscape offers an API for subscribing to events, like selecting or moving a node. Advanced use cases like graph traversal are supported too.

We use Cytoscape to render the user behavior model graph with a custom stylesheet. We make use of the different layouts, by allowing an operator to switch the layout of the graph at any time.

2.9 Graph Database Neo4j

Neo4j [32] is a graph database, written in Java, which supports a friendly query language and atomic, consistent, isolated and durable transactions. Graph databases are used to store highly connected information. A graph contains nodes and connections between those nodes, called edges. In a Neo4j database, nodes and edges can have properties, storing arbitrary information.

Neo4j is supplemented with a language to query graphs, called *Cypher Query Language*. Cypher is a declarative, SQL-inspired language and allows to describe patterns in graphs using an ASCII-art syntax.

Listing 2.3. Selecting a product called "Chocolade" with Cypher

```
MATCH (p:Product {productName:"Chocolade"})
RETURN p.productName, p.unitPrice;
```

Graph databases offer a range of algorithms to run on graphs [4], like finding the shortest path between two nodes and traversal to find all neighbors of a node. Furthermore, graph databases can be used for fraud detection [9] and real-time recommendation engines [31].

We use Neo4j for storing and querying our application data. The user behavior models are in graph-like structures so storing them in a graph database is a much better fit than a traditional relational database.

Chapter 3

Live Visualization of User Behavior Models

The goal of this thesis is to enable users to easily explore user behavior models in a graph and allow making changes to it later on. In Section 3.1, we discuss the requirements of the behavior visualization and editor services. In Section 3.2, we explain the architecture of our approach, followed by the design of our custom REST interfaces in Section 3.3. Section 3.4 describes our data model. In Section 3.5 we explain our implementation and which decisions we made, followed by an overview with screenshots of the resulting application in Section 3.6.

In this Chapter, we are using the following terms for describing the different parts of user behavior models. A *UserBehaviorGraph* is a graph with information about a part of a particular system. It contains many *Pages* which are entities a user can visit. The path a user can take from one *Page* to another *Page* is called *Visit*.

3.1 Visualization and Editor Requirements

The requirements for the visualization and editor services are motivated by the goals of this thesis and industry standards for web applications. Below, we list the requirements (R) for the visualization and editor services.

R1 The services should work in all browsers. Operators need a browser to access the application, so it should work in all of them, like Firefox 51, Chrome 56, Safari 10, Edge 14. If an operator is using an old or unsupported browser, we show a notice with an appropriate text for a browser update. There is no support for mobile devices since the application displays much information and large graphs are not suitable for small screens.

R2 The application displays user behavior models as an interactive graph. An operator can interact with the graph and customize the layout using a range of predefined layouts or move the nodes around to create one on its own. Nodes and edges can be inspected, and a sidebar displays additional information.

R3 When changes in a user behavior model occur, the graph must change accordingly in a certain time frame. A change may be an addition and a deletion of an entity. The update of the graph should not take longer than one second.

R4 The application must support user behavior model information which closely resembles the data structures from the Palladio Component Model (PCM). Incoming PCMs are accepted and stored in the database. The browser application can display these as a graph.

R5 An operator must be able to create new user behavior models and edit existing ones in an editor. Such editor allows an operator to create new graphs, and add, delete and modify nodes and edges along with their properties.

R6 When an operator opens an existing graph in an editor, a copy of this graph could be created. With a copy, the existing graph is still updatable with new user behavior model information.

3.2 Architecture

In this section, we describe the architecture of our visualization and editing services containing six components `Visualization`, `Editor`, `Data Collection and Update Component`, `Visualization Update Push Component`, `Graph Database` and `Mock Service`. We are not performing data collection since the user behavior models are provided by the iObserve analysis service. In this section, we describe the components of the architecture in more detail. We write all component names in a `monospace` font.

3.2.1 Visualization Component

An operator uses the `Visualization` to view user behavior models. It fetches graphs, nodes, and edges from the `Data Collection and Update Component` via secure HTTP. When an operator is inspecting a single graph, the `Visualization` establishes a WebSocket connection to the `Visualization Update Push Component`. When updates are pushed over this connection, the visualization is updated.

The `Visualization` communicates with the `Data Collection and Update Component` via a custom REST interface. Requests and responses are encoded in JSON. Messages received from the `Visualization Update Push Component` are encoded in JSON, too.

3.2.2 Editor Component

With the help of the `Editor`, an operator can create new graphs. Each insertion of a node and an edge is send to the `Data Collection and Update Component` via secure HTTP to get a unique id. After the insertion, the graph inside the `Editor` is updated. Modifications and

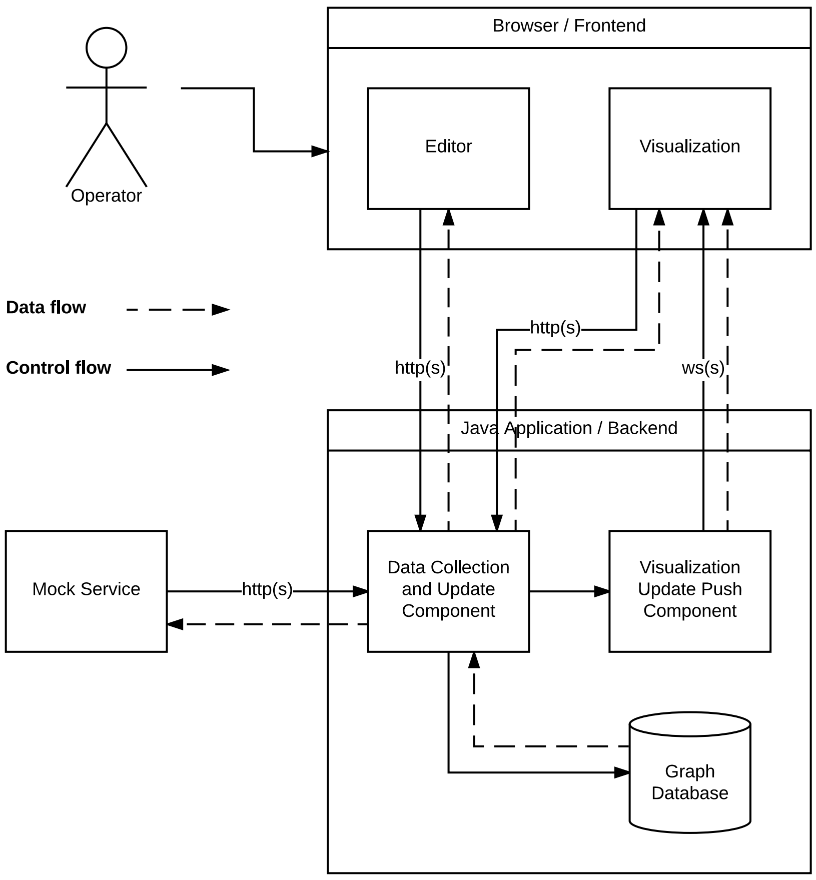


Figure 3.1. The architecture of the editing and visualization service

deletions work the same way, just without the generation of a new unique id. There is no WebSocket connection required for the `Editor` since the graphs are not updated with data coming from the iObserve service.

The `Editor` communicates with the `Data Collection and Update Component` via a custom REST interface. Requests and responses are encoded in JSON.

3.2.3 Data Collection and Update Component

The `Data Collection and Update Component` is the central data processing component, which receives data from the iObserve analysis service and `Editor`, and supplies the `Visualization` with data. It offers a REST interface for incoming requests. Furthermore, the `Mock Service` is using the `Data Collection and Update Component` for supplying sample data. After incoming write requests are processed, they are persisted in the `Graph Database` and handled to the `Visualization Update Push Component`. For incoming read requests, the data is read from the `Graph Database` and encoded in JSON.

The `Data Collection and Update Component` offers a custom REST interface. It communicates with the `Graph Database` using the standard Java driver. Communication with the `Visualization Update Push Component` is realized with Java calls.

3.2.4 Visualization Update Push Component

The `Visualization Update Push Component` allows browsers to subscribe to updates for specific graphs. It has an in-memory map of graphs and WebSocket connections for pushing changes, after each incoming write request, to all subscribers of the affected graph.

The `Visualization Update Push Component` transmits the change requests via WebSockets to the clients. The data is represented using JSON. It offers a Java method for broadcasting messages for a specific graph to all connected clients.

3.2.5 Graph Database Component

The `Graph Database` permanently stores all the application data. We store each graph with its nodes and edges separately of other graphs with their respecting nodes and edges.

The `Graph Database` supports a special network protocol for high-performance access, called *Bolt*, and a supplemental REST interface. The Java driver which we are using in the `Data Collection and Update Service` utilizes *Bolt*.

3.2.6 Mock Service Component

The `Mock Service` is a placeholder to simulate real incoming data. It supplies the `Graph Database` with sample data and helps with testing and demonstrating the application.

The `Mock Service` communicates with the `Data Collection and Update Component` via a custom REST interface. Requests and responses are in encoded JSON.

3.3 Component Interface Design

In this section, we describe the custom REST interfaces for adding, modifying and deleting user behavior model data in the application. We do not explain the generic interfaces because they have a dedicated documentation in an external project, like the Neo4j driver, or are documented in the code. Each interface offers a range of URLs to send data to, called endpoints. The endpoints are for create, read, updated, and delete operations on *UserBehaviorGraphs*, *Pages*, and *Visits*. Next, we explain all endpoints in detail.

All endpoints carry the prefix *v1* which indicates the version of the API. A versioned API is easier to upgrade and offers backward compatibility. An updated version of the API gets published with a new prefix, for example, *v2*. That allows developers to use the old API, which is still accessible using the *v1* prefix, while new implementations can use the new API. This approach enables a gentle transition from one API version to another, without disrupting any systems.

3.3.1 UserBehaviorGraphs

UserBehaviorGraphs are the highest level of abstraction and represent a user behavior model graph. We store a *UserBehaviorGraph* as a specific type of node in the graph database. A *UserBehaviorGraph* has the following attributes.

- *id*: A unique identifier for the graph
- *name*: The common name of the graph
- *description*: A short informative text about the graph
- *custom*: Indicator if the graph was created by an operator with the editor
- *createdAt*: Creation date of the graph

***GET** /v1/graphs* A request to this endpoint returns a list of all graphs with a status code of *200 - OK* [24]. The list contains information about each graph, like name, description, and creation date. The `Visualization` fetches data from this endpoint to display a list of all graphs to an operator for further selection.

***POST** /v1/graphs* A request to this endpoint creates a new graph and returns the information of the graph with a status code of *201 - Created* [24]. The request must contain a couple of required fields like the name of the graph. The `Mock Service` is using the endpoint to create new graphs for demonstration purpose. Also the `Editor` is using this endpoint to create new graphs. Later on, this endpoint is used to supply real user behavior models.

***GET** /v1/graphs/{gid}* A request to this endpoint returns detailed information about this graph with a status code of *200 - OK* and must contain a graph id. If we cannot find a graph with this id, we return a *404 - Not Found* [24] error. The `Visualization` and the `Editor` are fetching data from this endpoint to get more detailed information about a graph.

***PUT** /v1/graphs/{gid}* A request to this endpoint updates the information of this graph and must contain a graph id. It returns the complete updated graph with a status code of *200 - OK*. If we cannot find a graph with this id, we return a *404 - Not Found* error. The `Editor` and the `Mock Service` can use this endpoint to make updates to graphs.

DELETE */v1/graphs/{gid}* A request to this endpoint deletes the graph and everything belonging to it and must contain a graph id. It returns with an empty body and a status code of *200 - OK*. If we cannot find a graph with this id, we return a *404 - Not Found* error. The `Editor` and the `Mock Service` are using this endpoint to delete unused or old graphs.

3.3.2 Pages

Pages belong to a *UserBehaviorGraph*. In the graph database, we save *Pages* as a specific type of node and connect them to a *UserBehaviorGraph* with an edge. A *Page* has the following attributes.

▻ *id*: A unique identifier for the page

▻ *name*: The common name of the page

▻ *createdAt*: Creation date of the page

▻ *extra*: A collection (key, value) of additional information

A page cannot exist without a *UserBehaviorGraph*. Therefore all endpoints are prefixed with */v1/graphs/{gid}*. If we cannot find a *UserBehaviorGraph* with the specified id, all endpoints are returning a *404 - Not Found* error.

GET */v1/graphs/{gid}/pages* A request to this endpoint returns a list of all pages of the *UserBehaviorGraph* with a status code of *200 - OK*. The list contains information about each page like name and creation date.

POST */v1/graphs/{gid}/pages* A request to this endpoint adds a new page to the given *UserBehaviorGraph*. If all required fields are valid, the request returns the newly created page with a status code of *201 - Created*. The `Editor` is using this endpoint for adding pages to an existing *UserBehaviorGraph*.

GET */v1/graphs/{gid}/pages/{pid}* A request to this endpoint returns detailed information about this page with a status code of *200 - OK* and must contain a page id. If there is no page with this id, we return a *404 - Not Found* error. The `Visualization` and the `Editor` are using this endpoint for getting more information on the page an operator is inspecting.

PUT */v1/graphs/{gid}/pages/{pid}* A request to this endpoint updates the information of this page and must contain a page id. It returns the complete updated page with a status code of *200 - OK*. If there is no page with this id, we return a *404 - Not Found* error. The `Editor` is using this endpoint for updating the information of single pages.

DELETE */v1/graphs/{gid}/pages/{pid}* A request to this endpoint deletes the given page, including its visits and must contain a page id. It returns an empty body and a status code of *200 - OK*. If there is no page with this id, we return a *404 - Not Found* error. The `Editor` is using this endpoint for removing pages from a *UserBehaviorGraph*.

3.3.3 Visits

A *Visit* represents a movement of a user from one *Page* to another. We store a *Visit* as a directed edge between two pages (nodes), with the following attributes attached to it.

▻ *id*: A unique identifier for the visit

▻ *action*: A name for the transition between the pages

▻ *start*: The start page

▻ *end*: The destination page

▻ *count*: Information about how many users are taking this path

▻ *extra*: A collection (key, value) of additional information

The following endpoints for visits all require a *UserBehaviorGraph*, therefore all endpoints are prefixed with */v1/graphs/{gid}*. If we cannot find a *UserBehaviorGraph* with the specified id, all endpoints are returning a *404 - Not Found* error.

GET */v1/graphs/{gid}/visits* A request to this endpoint returns a list of all visits between the *UserBehaviorGraph* pages with a status code of *200 - OK*. The list contains information about each visit like starting and ending page, and user flows.

POST */v1/graphs/{gid}/visits* A request to this endpoint adds a new visit to the given *UserBehaviorGraph*. If all required fields are valid, the request returns the newly created visit with a status code of *201 - Created*. The `Editor` is using this endpoint for adding visits to existing pages.

GET */v1/graphs/{gid}/visits/{vid}* A request to this endpoint returns detailed information about this visit with a status code of *200 - OK* and must contain a visit id. If there is no visit with this id, we return a *404 - Not Found* error. The `Visualization` and the `Editor` are using this endpoint for getting more information on the visit an operator is inspecting, including more information on user flows.

PUT */v1/graphs/{gid}/visits/{vid}* A request to this endpoint updates the information of this visit and must contain a visit id. It returns the entire updated visit with a status code of *200 - OK*. If there is no visit with this id, we return a *404 - Not Found* error. The `Editor` is using this endpoint for updating information of single visits.

***DELETE** /v1/graphs/{gid}/visits/{vid}* A request to this endpoint deletes the given visit and must contain a visit id. It returns an empty body and a status code of *200 - OK*. If there is no visit with this id, we return a *404 - Not Found* error. The `Editor` is using this endpoint for removing visits from a *UserBehaviorGraph*.

Furthermore, a complete external documentation of the API build with API Blueprint [5] is available online [26]. API Blueprint is a high-level API description language for web APIs and can be used in conjunction with a couple of tools. These tools enable us to generate an HTML version of the documentation, create unit tests for a couple of languages, and build simple mock servers.

3.4 Data Model of the Behavior Visualization

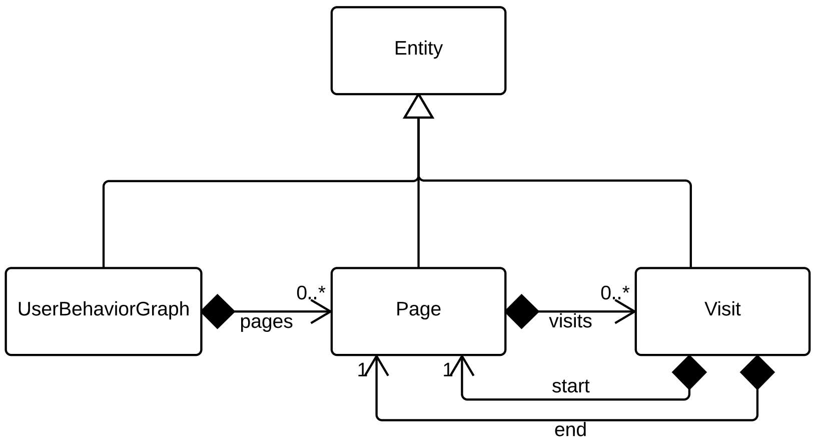


Figure 3.2. The behavior visualization and editing service data model in UML

In this section, we describe the data model of the behavior visualization and editing service. The types of the model correspond to the properties of the related entities of the custom REST interface (see Section 3.3). The main classes for representing our data are *UserBehaviorGraph*, *Page*, and *Visit*. All are extending the base class *Entity* for shared properties.

We save the model as following in a graph database. *UserBehaviorGraph* and *Page* are nodes with labels according to their type. The relation between them is saved as an edge with the label `CONTAINS`. A *Visit* is saved as a directed edge between two *Pages* with the label `VISIT`. The relations *start* and *end* are derived from the direction of the `VISIT` edge. The properties of *UserBehaviorGraph*, *Page*, and *Visit* are saved as attributes to their representation in the graph database. An *Entity* has no representation in the graph database. Next, we describe all classes in more detail.

3.4.1 Entity

Entity is the parent class for the *UserBehaviorGraph*, *Page*, and *Visit* classes. It is an abstract class and defines shared methods and attributes, like checking for equality.

▻ *id*: A unique identifier for the entity

▻ *createdAt*: Creation date of the entity

3.4.2 UserBehaviorGraph

UserBehaviorGraph is the name for a whole graph, which represents the user behavior model graph we are visualizing. A *UserBehaviorGraph* can have zero or more *Pages* as children. We save it as a node with the type `UBG` in the graph database. Each *UserBehaviorGraph* with its *Pages* and *Visits* is a graph in the database. Internally a *UserBehaviorGraph* has the following class properties.

▻ *name - string*: The name of the graph

▻ *description - string*: A short informative text about the graph

▻ *custom - boolean*: Indicator if the graph was created manually by an operator

▻ *pages - Set<Page>*: A set of *Pages* belonging to the graph

3.4.3 Page

A *Page* is a section of a *UserBehaviorGraph* users can visit. It can be included in zero or more visits as start or end page. We save it as a node with the type `PAGE` in the graph database. Internally a *Page* has the following class properties.

▻ *name - string*: The name of the *Page*

▻ *visits - Set<Visit>*: A set of visits the *Page* is included in

▻ *ubg - UserBehaviorGraph*: The *UserBehaviorGraph* the *Page* belongs to

▻ *extra - HashMap<String, String>*: A collection of additional information

3.4.4 Visit

A *Visit* is the transition of a user from one *Page* to another. It always has one start and one end *Page*. We save it as an edge with the type `VISIT` in the graph database. Internally a *Visit* has the following class properties.

- ▻ *action - string*: A name for the transition between the pages
- ▻ *start - Page*: The start *Page*
- ▻ *end - End*: The destination *Page*
- ▻ *count - int*: Information about how many users are taking this path
- ▻ *extra - HashMap<String, String>*: A collection of additional information

3.5 Design and Implementation Decisions

In this section, we discuss decisions we made while building the application. We split it into two parts, first building the operator facing frontend part, then the backend part.

3.5.1 Building the Browser Part

The frontend uses the React library [35] discussed in Chapter 4. React supports different approaches when starting the development of a new application.

Create everything from scratch When starting the development of a modern Javascript application, there are a lot of dependencies which need to be configured first, like Babel [7] and Webpack [52]. Babel is required for compiling next generation Javascript (ES2015) into the older ES5 version which can run in every browser. Webpack provides a development server with auto-reloading the browser window when files are changing and multiple pipelines for building the code for development or production with minification. The configuration of these tools and other dependencies requires a lot of time.

Use an application code generator *Create React App* [14] is being developed by Facebook and the official way to get started with React. It hides all the required configuration for tools like Babel and Webpack. When using *Create React App*, a developer can get straight to write React components, but still, needs to add any other React libraries theirself.

Use a boilerplate *React Boilerplate* [38] provides a pre-configured React application with many additional libraries. It includes libraries for routing, internationalization, asynchronous data fetching, web workers for offline support, and much more. The opinionated directory layout enables a developer to group application components by concern and should maintain clarity even if the application grows. *React Boilerplate* does not hide any of the build configurations and includes code generators for scaffolding new application parts. Using a boilerplate allows a developer to save much time when starting larger applications which require all of the already integrated libraries.

For our application requirements, *React Boilerplate* is too large and has many features we do not need. Building everything from scratch takes too much time and offers no advantage over *Create React App*. That is why we are using *Create React App* and adding the few required other React libraries manually.

Adding Routing to our Javascript Application

With Javascript, it is possible to navigate between pages on the client without sending a request to a server and doing a full page reload each time. This feature leverages the browser history API [23]. The project *React Router* [37] is wrapping this API, making it available through declarative React components. It offers helper components for navigating between sites, dynamic route matching, and lazy code loading. We integrate it into our application by first defining our routes, then accessing route parameters in our components when required.

Listing 3.1. The route configuration of the application

```
<Route component={App}>
  <Route path="/" component={Dashboard} />
  <Route path="/applications/:id" component={Application} />
  <Route path="/editor" component={EditorWrapper}>
    <IndexRoute component={EditorList} />
    <Route path="/editor/new" component={EditorNew} />
    <Route path="/editor/:id" component={Editor} />
  </Route>
  <Route path="*" component={NotFoundPage} />
</Route>
```

In Listing 3.1, we define our routes. A *Route* component expects an argument component which specifies the component to be rendered and an optional argument path for specifying on which URL the component should be rendered.

Each component which is rendered with *React Router* has additional properties with information about the current location. In our application, we use this information to access parameter data passed during requests. In Listing 3.2, we infer the application id from the passed parameters and start loading additional information about this application for display purposes.

Listing 3.2. Accessing route parameters

```
export class ApplicationView extends React.Component {

  componentDidMount() {
    this.props.loadApplication(parseInt(this.props.params.id, 10));
  }

  ...
}
```

Keeping the URL in sync with the content enables us to use deep linking in our single page application. That is a much better user experience instead of having just a single URL to access the application and not being able to share individual pages.

Handling State and Fetching Data

Most of our React components do not require an internal state. Therefore, we avoid side effects and write components which will produce the same output if they get the same properties as input. However, we still need a way to fetch data from external services and store this data while our application is running inside the browser.

With *Redux* [41] it is possible to keep a single predictable state container for an application. The state container is described as plain object and modifications are done with so-called *Reducers*. A *Reducer* accepts the whole state and an action as parameters and returns a modified version of the state. *Reducers* must be pure functions and always return a new state object instead of modifying the existing one. Making the state an immutable map with *immutable.js* [25] is a good way to ensure this.

We can fire *Actions* in different parts of the application to trigger state changes. An *Action* always contains a type that indicates which *Reducer* should handle it. Furthermore, it can carry a payload for getting new data into the state.

Redux Saga [42] allows us to have something like separate threads in Javascript for handling side effects. Internally it is using an ES6 feature called Generators. The *Sagas* (see Section 2.7.2) wait for an action, which can be fired at any point in the application, execute their tasks and then dispatch another *Action* with the new data. Inside a *Saga*, we are making secure HTTP calls to our custom REST interface. Furthermore, we are watching incoming WebSockets messages with a *Saga*, transforming them into actions, our reducers can understand, and then dispatching those.

Type Checking the Javascript Application

Javascript is a weakly and dynamically typed programming language. That makes it difficult to find common bugs, like silent type conversions or null dereferences, during development and may lead to errors during runtime. *Flow* [1] is a static type checker for Javascript and mitigates type related problems. It uses static typing and type inference and outputs helpful information about type mismatches during development.

Listing 3.3. Valid Javascript with silent type conversions

```
console.log({} + {}) // NaN
console.log({} + []) // 0
console.log([] + []) // ''
console.log({} + 2) // [object Object]2
```

Listing 3.3 shows some valid Javascript additions with unpredictable results. When checking this code for invalid types with *Flow*, we get an error for each of the lines as in Listing 3.4

Listing 3.4. Errors when checking the code with *Flow*

```
1: console.log({} + {}) // NaN
               ^^ object literal. This type cannot be added to
1: console.log({} + {}) // NaN
               ^^^^^^^ string
4: console.log({} + 2) // [object Object]2
               ^^ object literal. This type cannot be added to
4: console.log({} + 2) // [object Object]2
                    ^ number
```

Furthermore, we can use *Flow* to create type aliases, union types and much more. That allows us to describe all our models with type aliases and gradually typing the whole application.

Styling the Application

We are using Bootstrap [11] to create a simple layout with a header and a content area. Furthermore, Bootstrap provides many pre-styled components like buttons and forms. Bootstrap comes with a grid system, which we are using for splitting the application view into a graph area and a sidebar. We use the same sidebar layout for the editor. In the editor, the sidebar displays two forms for adding pages and visits to the application. The forms are styled with standard bootstrap input fields and buttons.

For an easier integration of Bootstrap with React, we are using reactstrap [40]. Reactstrap offers all Bootstrap elements as ready to use React components. Instead of manually including the stylesheet files and setting the CSS classes, we can import the specific component from reactstrap and use it like any other React component.

In some parts of our application, we cannot use predefined Bootstrap elements and need to style elements manually. For this, we are using *styled components* [46]. *Styled components* uses the ES6 template string feature (see Section 2.7) and allows us to apply CSS properties to existing components or HTML elements as demonstrated in Listing 3.5. With *styled components* it is also possible to change the style of a component based on the passed properties or do animations.

Listing 3.5. A styled div element for wrapping the graph

```
import styled from "styled-components";

const Wrapper = styled.div‘
  height: 85vh;
  width: 100%;
‘;
```

Adding Live Updates with WebSockets

We use a WebSocket connection for pushing updates of an application to all clients. A client needs to open such connection when an operator is inspecting an application. Each major browser ships with support for WebSockets today without requiring extra libraries.

In our application, a *Saga* (see Section 2.7.2) handles all tasks regarding the WebSocket connection. The *Saga* waits for a `SUBSCRIBE` action with an id and then establishes a connection to the `Visualization Update Push Component`. The connection is terminated when the server disconnects or a `UNSUBSCRIBE` action is received. While the connection is open an *eventChannel* sends all incoming messages to another *Saga*. This *Saga* evaluates the messages and dispatches actions accordingly. This way we can handle addition and deletion events for pages and visits and easily extend the code to handle other events, like modifications, later on. We log an error when an unknown message arrives.

3.5.2 Building the Java Part

In this part, we describe building the backend application with Java. When starting the development of a new Java application, we have to choose a build tool first. A build tool manages the dependencies and builds the application. New dependencies can be added to the build tools settings file and are downloaded and added to our application on each build. Furthermore, a build tool can be extended with plugins to have other commands, for example launching a development server with hot code replacement.

We are using Gradle [19] as the build tool, since the syntax for its configuration files is easy to learn, it is in active development, and it is widely supported like in IDEs and continuous deployment services. With Jetty [29] as our web server, we can make use of the Gretty [20] Gradle plugin. With this plugin we have hot code replacement while developing the application, so we do not have to restart the server after each change.

Adding the Neo4j Graph Database

We are using a graph database called Neo4j [32] for storing our application data. Neo4j can be used in different modes. We can embed the database inside our application or connect our application to an external database. We are connecting to an external database since this reduces the size of the application and allows maintenance and updates of the database without touching the application.

Neo4j offers a native Java driver which connects to the database using the Bolt (see Section 2.9) binary protocol. On top of that, we are using the Neo4j Object Graph Mapping (OGM) library [33], which allows us to map our domain objects, *UserBehaviorGraph*, *Page*, and *Visit* to Neo4j. Each domain object has its class with different properties and annotations how the data should be represented.

Adding Jersey for a RESTful Web Service

Jersey [28] is a JAX-RS reference implementation and allows us to expose data in different media types. We use Jersey to build a RESTful web service for our three domain objects *UserBehaviorGraph*, *Page*, and *Visit*. The web service implements the component interface design we describe in Section 3.3. We create a resource class for each model containing annotated methods for each endpoint. Listing 3.6 shows an example of a resource class.

Listing 3.6. A resource class for the Page model with one method

```
@Path("v1")
@Produces(MediaType.APPLICATION_JSON)
public class PageResource {

    @GET
    @Path("/graphs/{graphId}/pages")
    public Iterable<Page> getAll(@PathParam("graphId") final Long graphId) {
        return this.pageService.findAll(graphId);
    }
}
```

Each resource returns the data as JavaScript Object Graph (JSOG) [30] using Jackson [27]. JSOG strips duplicate information from JSON and replaces them with references to the first occurrence of the information. That reduces the size of the produced JSON and avoids stack overflow errors while serializing due to cycles in the graph. With Jackson, we just have to annotate the properties in the domain objects we want to include in our response. That makes adding new fields to a response easy and fast.

Pushing changes with WebSockets

For keeping track of incoming WebSocket connections, we create a handler, which adds a new connection to a *HashSet* for each *UserBehaviorGraph* id. When a connection is closed, we remove it from the *HashSet*.

Our handler offers a method to broadcast a message to all connections of a specific *UserBehaviorGraph* id. We are using this method in all Jersey resources after evaluating the request. For example, when we get a request for the creation of a page, we save the page to the graph database, send the response to the requester and then push the new page to all WebSocket connections.

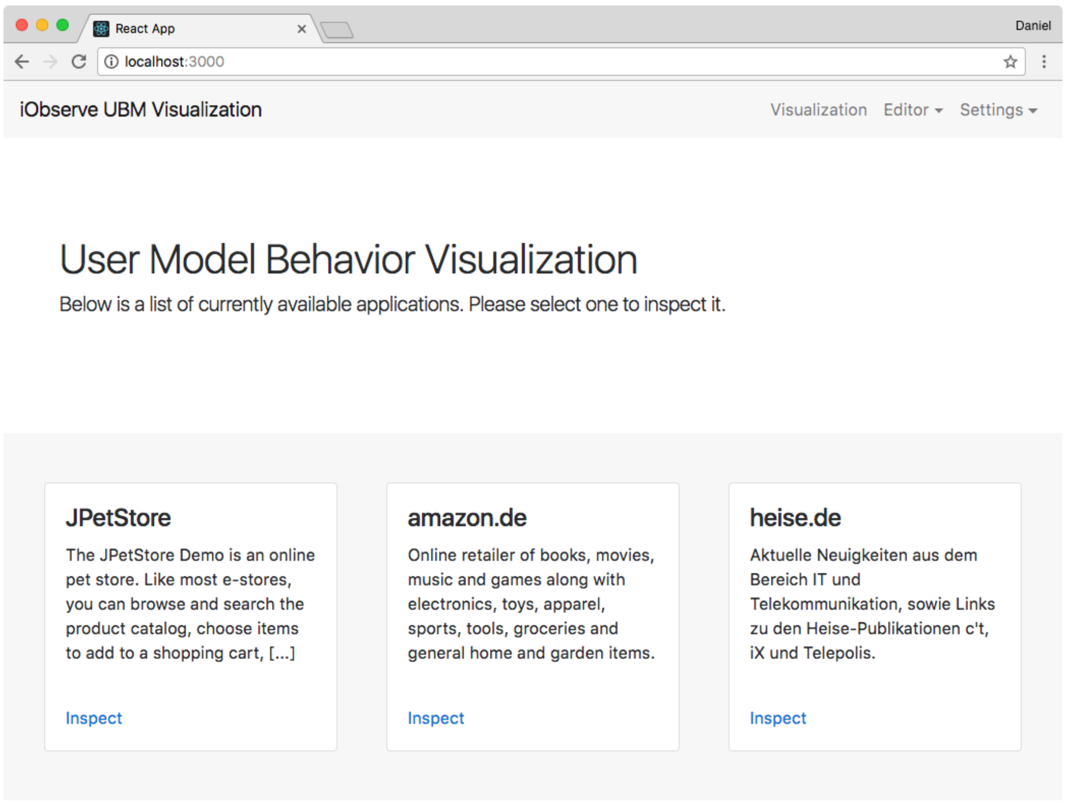
3.6 Overview of the Application

In this section, we give a short overview of the resulting application with the help of two screenshots. The first screenshot, depicted in Figure 3.3, shows the start page of the user behavior model visualization. It displays a short introduction text and shows a list of *UserBehaviorGraphs* with a title and a description for further selection. An operator can click on each of the *Inspect* links to get to the detailed view of this *UserBehaviorGraph*. On the top right is a menu with one link and two dropdowns. The link *Visualization* links to the current page (start). The first dropdown *Editor* contains two links, one for getting to an overview of all custom created *UserBehaviorGraphs* and another for creating a new *UserBehaviorGraph*. The second dropdown *settings* enables an operator to select different layouts for the graphs.

The editor is shown in Figure 3.4. In the sidebar on the left, an operator can add new pages and visits. The page form has just one field *name*. In the *Visit Editor* an operator can select a start and an end page, and set further properties like the performed action and how many users are following this path. Whenever a form is submitted the visualization is updated accordingly.

When an operator is inspecting a *UserBehaviorGraph*, the format of the page is similar. Instead of the forms we display additional information about the selected page or visit in the sidebar. The information is shown in a table and contains all attributes of the selected entity. In both views the layout of the graph can be changed anytime by using the *Settings* dropdown on the top right. Furthermore, while inspecting a *UserBehaviorGraph*, the graph is rerendered when an update arrives via the WebSocket connection.

The graph displays each page as a node and each visit as an edge. The names of the pages are rendered on top of each node. If a visit has an action defined, we display the action on top of each edge as a label. The thickness of each edge is defined by how many users are taking this path; the more users, the thicker the edge.



User Model Behavior Visualization

Below is a list of currently available applications. Please select one to inspect it.

JPetStore

The JPetStore Demo is an online pet store. Like most e-stores, you can browse and search the product catalog, choose items to add to a shopping cart, [...]

Inspect

amazon.de

Online retailer of books, movies, music and games along with electronics, toys, apparel, sports, tools, groceries and general home and garden items.

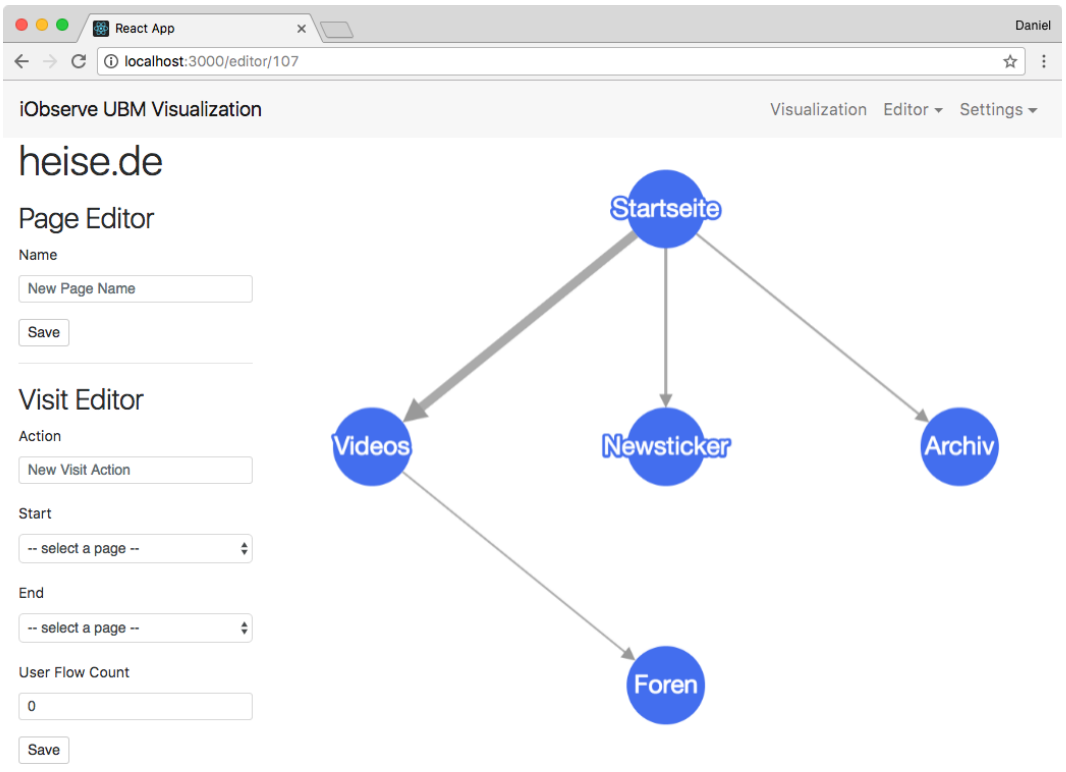
Inspect

heise.de

Aktuelle Neuigkeiten aus dem Bereich IT und Telekommunikation, sowie Links zu den Heise-Publikationen c't, iX und Telepolis.

Inspect

Figure 3.3. The start screen of the application with a list of all *UserBehaviorGraphs*



heise.de

Page Editor

Name

New Page Name

Save

Visit Editor

Action

New Visit Action

Start

-- select a page --

End

-- select a page --

User Flow Count

0

Save

Figure 3.4. The editor with a visualization of the current *UserBehaviorGraph* which is being edited

Chapter 4

Evaluation

We had to evaluate different Javascript libraries for building user interfaces before building the user behavior model visualization application. In this chapter, we explain the evaluation progress and our findings in detail.

4.1 Javascript Libraries for Building User Interfaces

We compare and evaluate the two Javascript libraries Elm and React using an adapted GQM [48] approach. We define our primary goal as: "Evaluate Javascript libraries for building user interfaces from the point of view of a software developer." Furthermore, we define three research questions (RQ) based on our primary goal.

4.1.1 Research Questions

RQ1: How well is the library maintained?

A well-maintained library is important for the future of a project. It ensures that we get bugfixes and new features for the whole time the project is being developed and future developers do not have to rebuild the project shortly using another upcoming library. To measure, if a library is well-maintained and answer the first research question, we have created the following metrics.

Contributors on Github This counts the individual people who have contributed to the project on Github. A contribution can be code, like features or bugfixes, improvements or additions to the documentation or modifications of anything else inside the git repository.

Project age The age of a project can be an indicator of its maturity. A young project is more likely to contain bugs and is not as proven as an older one.

Downloads on NPM NPM is the node package manager and the central platform inside the Javascript ecosystem. It provides statistics on how many times a package is downloaded each day, week and month.

What kind of release scheme is used? A defined release scheme indicates a planned development schedule and a well-maintained project. We are using the following values for possible release cycles: "none," "half year cycle/periodic cycle," "continuous delivery (with master branch)," "continuous delivery (with specific release branches)."

When was the last stable release? In the Javascript ecosystem, many projects are created and abandoned every day. In combination with the age of a project, the last stable release date indicates if a project is still maintained.

Does it follow a versioning scheme? The presence of a versioning scheme shows that developers have recognized that versioning is important and are enforcing it on this library.

How high is the test coverage? A high test coverage indicates that the library is well tested and bugs are more likely discovered in early stages of development.

How stable is the API? Breaking changes to the API are forcing every user of the library to update their code. Those changes should be as rarely as possible and if the occur communicated as clear as possible. We are using the following criteria's for assessing the API stability: "Stable (new additions do not affect the existing API)," "Unstable, major releases," "Very unstable, minor releases."

RQ2: To what extend does the library foster low complexity component data binding?

Using a library with a small API and low complexity enables us to develop the project at a fast pace and makes it easier for future developers to continue with the project. With React or Elm, we can create many small components for each part of the application and compose them together. With this component model, we need an easy way to get the right data to each component while keeping a good render performance to display changes to the user behavior model graph in an instant. We created the following metrics to make sure the library satisfies these requirements.

Does it support push updates? Push updates are essential for live visualization. The browser application should not poll the backend each second for new updates. Instead, the backend pushes each new update directly to the connected browsers.

What is the render performance? Rendering large and fast-changing graphs is challenging and requires a good rendering performance. For comparison, we are using a benchmark of a TodoMVC test application[10].

RQ3: To what extend does the library support interoperability with other libraries?

We are evaluating libraries for building user interfaces, but when building the complete application, we are facing other challenges too, like displaying a graph or connecting to a

WebSocket server. That is why we must ensure that the user interface library is modular and compatible with other libraries. The following metrics ensure compatibility with the most important libraries for our application.

Is it compatible with Cytoscape.js? We are using Cytoscape.js to render the user behavior model graph. Interoperability with the UI library is required.

Is there a wrapper for styling via Bootstrap? Bootstrap is a CSS library for styling UI elements. A wrapper of bootstrap elements for React or Elm makes building a UI with such elements much easier.

Is there a library for handling WebSockets? Using a library, instead of implementing every detail manually, makes working with WebSockets much easier. The library should handle issues like reconnecting on connection loss and offer an easy to use API.

4.1.2 Measurements

Next, we collect all values and answer all metrics for Elm and React to all research question in a tabular format. Table 4.1 contains all measurements for research question one, table 4.2 all measurements for research question two, and table 4.3 all measurements for research question three. Data as per end of March 2017.

Table 4.1. RQ1: How well is the library maintained?

Metric	Elm	React
Contributors on Github	86	956
Project age	2012	March 2013
Downloads on NPM in the last month	35,421	3,382,322
What kind of release scheme is used?	none	none
When was the last stable release?	January 23, 2017	January 6, 2017
Does it follow a versioning scheme?	Semantic Versioning [44]	Semantic Versioning [44]
How high is the test coverage?	hard to say, due code splitting in many small repositories	82% [13]
How stable is the API?	Unstable, major releases	Unstable, major releases

Table 4.2. RQ2: To what extend does the library foster low complexity component data binding?

Metric	Elm	React
Does it support push updates?	Yes	Yes
What is the render performance?	2244ms	3553ms

Table 4.3. RQ3: To what extend does the library support interoperability with other libraries?

Metric	Elm	React
Is it compatible with Cytoscape.js?	Might work via a Javascript bridge	Yes
Is there a wrapper for styling via Bootstrap?	Yes, elm-bootstrap-html [18]	Yes, reactstrap [40]
Is there a library for handling WebSockets?	Yes, websocket [53]	Yes, socket.io [45]

4.1.3 Evaluation of Results

After collecting all measurements and answering all research questions, we come to the following conclusion. In direct comparison React is the much bigger project with 100 times more downloads each month and ten times more contributors. The reason for this might be that Facebook backs React and heavily uses it for their applications, even though both projects are about the same age. Elm originated from a Master thesis and was developed by a single person in the beginning. Today different companies are using Elm in production environments and supporting the development.

Both libraries are well maintained and have recent releases. The test coverage of React is easy to measure and is 82 percent. We could not measure this for Elm since the Elm language consists of many small modules which are all published and tested separately. However, all modules we had a look at, had many tests. When comparing the render performance of both libraries, Elm performs better and is about 50 percent faster. For our application, the performance of both libraries is fast enough since we are not rendering large sets data, like tables with many rows. The rendering of the graph and the contained nodes and edges is handled by Cytoscape.js and independent of the libraries.

React has a much larger ecosystem and is compatible with all libraries we are using for our application. Elm lacks support for Cytoscape.js. Instead, we could use Elm with a bridge to run normal Javascript code along with Elm code, since Elm gets compiled to Javascript too. However, this approach requires us to write a wrapping layer between the Cytoscape.js library and Elm. Both libraries support styling with Bootstrap and push updates via WebSockets.

We decide on using React for our application because it supports Cytoscape.js without extra work and is easier to learn for other Javascript developers. They only need to learn to work with a new library for building applications and not a whole new language. That outweighs the performance benefits of Elm.

Chapter 5

Related Work

In this chapter, we discuss related work for modeling user behavior and live visualization. In Section 5.1 we discuss an approach for visualizing navigation patterns on a website, followed by a tool for visualization called *DTWRader* in Section 5.2. In Section 5.3 we analyze a patent for user behavior visualization, followed by a commercial tool in Section 5.4. Finally, we discuss a visualization based on Markov models in Section 5.5.

5.1 Web-site Navigation Patterns

The *Model-Based Clustering and Visualization of Navigation Patterns on a Web Site* approach of Igor Cadez et al. [12] presents a new method for visualizing navigation patterns on a website. Before visualizing the data, they partition the users into clusters using a model-based approach with first-order Markov models. Each cluster contains users with similar navigation patterns. After clustering, they are visualizing their data using a tool called *WebCANVAS*. This tool displays each cluster as own graphic containing rows with category blocks in different colors.

5.2 DTWRadar

Why We Search: Visualizing and Predicting User Behavior by Eytan Adar et al. [2] presents the results of their study of the behaviors of internet users on multiple systems. For analyzing the behaviors, they develop a model for the events and a tool for visualization called *DTWRadar*. With the model, they can compare reactions of users on different systems, like blogs or search engines. The tool displays time series of these reactions and makes it easy to spot patterns or differences.

5.3 Visualization of User Behavior in a Distributed Computer Network

The patent *System and method for logical view analysis and visualization of user behavior in a distributed computer network* [47] was filled in 2001 and describes the process of collecting raw data, refining and clustering it, apply scoring, and visualizing it in a networked computer environment. It uses the same graph-based approach for visualizing the data as we do in this thesis; the resulting graph has nodes and edges of different thickness. The patent expired in 2015.

5.4 Path analysis with SAS Visual Analytics

The commercial software *SAS Visual Analytics* [34] visualizes user behavior as paths. A path displays the choices a user makes on a website in a diagram from left to right. A path can contain the data of a single user or multiple users. Each time a path splits into two or more paths, an event happened, for example, navigating or making a purchase. Furthermore, paths can be segmented or filtered for a better understanding of different events.

5.5 Early Reliability Prediction

The *Technique for Early Reliability Prediction of Software Components Using Behaviour Models* approach of Awad Ali et al. [3] proposes a new reliability prediction technique for software components based on a state machine and a hidden Markov model. The behavior models of software components are drawn as a component probabilistic dependency graph (CPDG). The layout of the CPDG is similar to our graph, but instead of user behavior, it displays different states of a component and does not weight the edges.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

In this thesis, we built an application for visualizing and editing user behavior models as an addition to the iObserve project. We evaluated two Javascript libraries - Elm and React - regarding their fitness for our live visualization application. We chose React based on our evaluation (see Section 4.1.3), as it is a better fit for our application requirements.

After the technology evaluation, we built our application in the context of the iObserve microservice architecture. We built a Java application containing multiple components for handling data and storing data in a graph database (see Section 3.2). With building the browser application (see Section 3.5.1), we enable operators to view this data in a graph-based visualization and allow them to make edits and additions, and even create new graphs inside an editor. An operator can inspect each graph to get more detailed information about each entity. For keeping the visualized graphs up to date, we added a live update method using WebSockets.

The resulting application fulfills all our requirements defined in Section 3.1. We have published it as open source on Github in the iObserve research project. In the appendices (see Section A.1) we explain how to obtain the application and get it running in a local environment.

6.2 Future Work

There are a couple of things we were not able to implement in our application mainly due to time constraints. In future projects, the application could be extended with new features, improved stability, and improved performance.

The application must be integrated into the iObserve context. It is currently only processing sample data from the `Mock Service`. After integration, the application can display real user behavior models derived from real user behavior information.

The *Data Collection and Update Service* currently only validates the URL parameters of a request. It is necessary to validate the JSON body for missing fields and that all fields contain valid values. If a request contains references to other entities, like when adding a visit containing a start and an end page, we have to validate if those pages exist. In cases of validation errors, the operator should get a clear error message and a hint how to fix the error.

When an operator opens an existing user behavior model in the editor, all modifications are done to the existing model. That results in a conflict when new data for this model arrives from an external source. Instead of modifying the existing model, a full clone - including all nodes and edges - should be created. With that, we avoid all potential conflicts. In a future version, there could even be a way to merge both models back together.

We are currently updating each page and visit of an application on its own. A better way would be to batch these requests into a single one updating the whole application. That would reduce the request counts and traffic. Furthermore, it would reduce the re-renderings of the graph in the browser application. Presently, each update triggers a re-render of the graph. A batched update would only trigger one.

There is no authentication or authorization of operators currently. When an operator has access to the application, they can do anything they want. In a future version it might be necessary to create different operator roles. For example, there can be one for inspecting graphs, another one for editing them and one for creating new graphs. Each operator can have its set of roles which authorize what they can do.

Another addition could be other layouts for the graphs or entirely other forms of user behavior visualization. The whole application could stay as it is and only the Cytoscape.js component must be extended or replaced. We discussed potential other forms of visualization in Chapter 5.

Bibliography

[1] *A static type checker for JavaScript*. flowtype. URL: http://flowtype.org/ (visited on 03/02/2017) (cited on page 23).

[2] Eytan Adar et al. "Why we search: visualizing and predicting user behavior". In: *Proceedings of the 16th International Conference on World Wide Web*. WWW '07. New York, NY, USA: ACM, 2007, pages 161–170. URL: http://doi.acm.org/10.1145/1242572.1242595 (cited on page 35).

[3] Awad Ali et al. "Technique for early reliability prediction of software components using behaviour models". In: *PLOS ONE* 11.9 (Sept. 26, 2016), e0163346. URL: http://journals.plos.org/plosone/article?id=10.1371/journal.pone.0163346 (cited on pages 5, 36).

[4] R. Angles. "A comparison of current graph database models". In: *2012 IEEE 28th International Conference on Data Engineering Workshops*. 2012 IEEE 28th International Conference on Data Engineering Workshops. Apr. 2012, pages 171–177 (cited on page 9).

[5] *API blueprint*. A powerful high-level API description language for web APIs. URL: https://apiblueprint.org/ (visited on 02/08/2017) (cited on page 18).

[6] *Application performance monitoring & management*. AppDynamics. URL: https://www-origin.appdynamics.com/ (visited on 03/15/2017) (cited on page 3).

[7] *Babel*. The compiler for writing next generation JavaScript. URL: https://babeljs.io/ (visited on 03/02/2017) (cited on page 20).

[8] Steffen Becker, Heiko Koziolek, and Ralf Reussner. "The palladio component model for model-driven performance prediction". In: *J. Syst. Softw.* 82.1 (Jan. 2009), pages 3–22. URL: http://dx.doi.org/10.1016/j.jss.2008.03.066 (cited on page 4).

[9] Alex Beutel, Leman Akoglu, and Christos Faloutsos. "Graph-based user behavior modeling: from prediction to fraud detection". In: *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '15. New York, NY, USA: ACM, 2015, pages 2309–2310. URL: http://doi.acm.org/10.1145/2783258.2789985 (cited on pages 5, 9).

[10] *Blazing fast html round two*. URL: http://elm-lang.org/blog/blazing-fast-html-round-two (visited on 01/23/2017) (cited on page 30).

[11] *Bootstrap*. URL: https://v4-alpha.getbootstrap.com (visited on 03/02/2017) (cited on page 23).

[12] Igor Cadez et al. “Visualization of navigation patterns on a web site using model-based clustering”. In: *Proceedings of the Sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD ’00. New York, NY, USA: ACM, 2000, pages 280–284. URL: http://doi.acm.org/10.1145/347090.347151 (cited on page 35).

[13] *Coveralls react*. Test Coverage History & Statistics. URL: https://coveralls.io/github/facebook/react?branch=master (visited on 01/23/2017) (cited on page 31).

[14] *Create-react-app*. Create React apps with no build configuration. URL: https://github.com/facebookincubator/create-react-app (visited on 03/02/2017) (cited on page 20).

[15] *Cytoscape.js*. Graph theory / network library for analysis and visualisation. URL: http://js.cytoscape.org/ (visited on 02/25/2017) (cited on page 8).

[16] Eberhard Wolff. *Microservices*. dpunkt.verlag, 2015 (cited on page 2).

[17] *Elm*. A delightful language for reliable webapps. URL: http://elm-lang.org/ (visited on 12/03/2016) (cited on page 6).

[18] *Elm-bootstrap-html*. Html shorthand for working with Bootstrap in Elm. URL: http://package.elm-lang.org/packages/circuithub/elm-bootstrap-html (visited on 01/23/2017) (cited on page 32).

[19] *Gradle build tool*. Gradle helps teams build, automate and deliver better software, faster. URL: https://gradle.org/#close-notification (visited on 03/26/2017) (cited on page 24).

[20] *Gretty*. Advanced gradle plugin for running web-apps on jetty and tomcat. URL: https://github.com/akhikhl/gretty (visited on 03/26/2017) (cited on page 25).

[21] Wilhelm Hasselbring. “Microservices for scalability: keynote talk abstract”. In: *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering*. ICPE ’16. New York, NY, USA: ACM, 2016, pages 133–134 (cited on page 2).

[22] Robert Heinrich et al. *Run-time architecture models for dynamic adaptation and evolution of cloud applications*. Report. Kiel, Germany: Department of Computer Science, Apr. 20, 2015. URL: http://eprints.uni-kiel.de/28566/ (cited on page 3).

[23] *HTML 5*. A vocabulary and associated APIs for HTML and XHTML. URL: https://www.w3.org/TR/html5/browsers.html#history (visited on 03/29/2017) (cited on page 21).

[24] *HTTP 1.1: response*. Status Code and Reason Phrase. URL: https://www.w3.org/Protocols/rfc2616/rfc2616-sec6.html (visited on 03/30/2017) (cited on page 15).

[25] *Immutable.js*. Immutable collections for JavaScript. URL: https://facebook.github.io/immutable-js/ (visited on 03/22/2017) (cited on page 22).

[26] *iObserve UMB API documentation*. URL: https://github.com/research-iobserve/ubm-visualization/tree/master/docs/api (visited on 02/08/2017) (cited on page 18).

[27] *Jackson*. Jackson is a suite of data-processing tools for Java. URL: https://github.com/FasterXML/jackson (visited on 03/26/2017) (cited on page 25).

[28] *Jersey*. RESTful Web Services in Java. URL: https://jersey.java.net/ (visited on 03/26/2017) (cited on page 25).

[29] *Jetty*. Servlet Engine and Http Server. URL: https://eclipse.org/jetty/ (visited on 03/26/2017) (cited on page 25).

[30] *Jsog*. JSOG (JavaScript Object Graph) is a simple convention which allows arbitrary object graphs to be represented in JSON. URL: https://github.com/jsog/jsog (visited on 03/26/2017) (cited on page 25).

[31] Miller, Justin J. "Graph database applications and concepts with neo4j". In: *Proceedings of the Southern Association for Information Systems Conference, Atlanta, GA, USA*. Volume 2324. 2013, page 36 (cited on page 9).

[32] *Neo4j*. Graphs for Everyone. URL: https://github.com/neo4j/neo4j (visited on 12/03/2016) (cited on pages 9, 25).

[33] *Neo4j - OGM object graph mapper*. URL: https://neo4j.com/developer/neo4j-ogm/ (visited on 03/26/2017) (cited on page 25).

[34] *Path analysis with SAS visual analytics*. SAS Voices. URL: http://blogs.sas.com/content/sascom/2014/08/19/path-analysis-with-sas-visual-analytics/ (visited on 03/23/2017) (cited on page 36).

[35] *React*. A JavaScript library for building user interfaces. URL: https://facebook.github.io/react/ (visited on 12/03/2016) (cited on pages 7, 20).

[36] *React native*. A framework for building native apps using React. URL: https://facebook.github.io/react-native/index.html (visited on 03/14/2017) (cited on page 8).

[37] *React router*. Declarative Routing for React.js. URL: https://reacttraining.com/react-router/ (visited on 03/02/2017) (cited on page 21).

[38] *React.js boilerplate*. Quick setup for new performance orientated, offline–first React.js applications. URL: https://www.reactboilerplate.com/ (visited on 03/02/2017) (cited on page 21).

[39] *React-redux*. Official React bindings for Redux. URL: https://github.com/reactjs/react-redux (visited on 03/14/2017) (cited on page 8).

[40] *Reactstrap*. React Bootstrap 4 components. URL: https://reactstrap.github.io/ (visited on 01/23/2017) (cited on pages 23, 32).

[41] *Redux*. A predictable state container for JavaScript apps. URL: http://redux.js.org/ (visited on 12/03/2016) (cited on pages 8, 22).

[42] *Redux-saga*. a library that aims to make side effects in React/Redux applications easier and better. URL: https://redux-saga.github.io/redux-saga/ (visited on 03/14/2017) (cited on page 22).

[43] Sam Newman. *Microservices - konzeption und design*. 1st edition. mitp Verlags, 2015 (cited on page 2).

[44] *Semantic versioning*. Given a version number MAJOR.MINOR.PATCH. URL: http://semver.org/ (visited on 01/11/2017) (cited on page 31).

[45] *Socket.IO*. URL: http://socket.io/ (visited on 01/23/2017) (cited on page 32).

[46] *Styled-components*. Visual primitives for the component age. URL: http://styled-components.com (visited on 03/22/2017) (cited on page 24).

[47] "System and method for logical view analysis and visualization of user behavior in a distributed computer network". US7165105 B2. David Reiner et al. U.S. Classification 709/224, 709/217, 714/E11.18, 707/999.007, 707/999.01; International Classification G06Q30/02, H04L12/24, G06F11/34, G06F15/173, H04L29/08, G06F11/32; Cooperative Classification Y10S707/99937, H04L41/22, G06F11/32, G06F11/3495, G06F2201/875, G06Q30/02, H04L67/22, H04L67/20, H04L69/329; European Classification G06Q30/02, H04L41/22, G06F11/32, H04L29/08N21, H04L29/08N19, H04L29/08A7. Jan. 16, 2007. URL: http://www.google.com/patents/US7165105 (cited on page 35).

[48] Victor R. Basili, Gianluigi Caldiera, and H. Dieter Rombach. *The goal question metric approach*. 1994 (cited on page 29).

[49] Christian Vögele et al. "WESSBAS: extraction of probabilistic workload specifications for load testing and performance prediction—a model-driven approach for session-based application systems". In: *Software & Systems Modeling* (Oct. 20, 2016), pages 1–35. URL: https://link.springer.com/article/10.1007/s10270-016-0566-5 (cited on page 5).

[50] Zhanyong Wan and Paul Hudak. "Functional reactive programming from first principles". In: *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*. PLDI '00. New York, NY, USA: ACM, 2000, pages 242–252. URL: http://doi.acm.org/10.1145/349299.349331 (cited on page 5).

[51] Jim Webber and Ian Robinson. *Graph databases in the enterprise: identity & access management*. Neo4j Graph Database. Nov. 2, 2015. URL: https://neo4j.com/blog/enterprise-identity-access-management/ (visited on 03/28/2017) (cited on page 5).

[52] *Webpack module bundler*. A bundler for javascript and friends. URL: https://webpack.github.io/ (visited on 03/02/2017) (cited on page 20).

[53] *Websocket*. URL: http://package.elm-lang.org/packages/elm-lang/websocket (visited on 01/23/2017) (cited on page 32).

Appendix A

Appendices

A.1 Obtaining the application

The implementation of this work is hosted as open source project on Github: `https://github.com/research-iobserve/ubm-visualization`. This repository can be cloned with git or downloaded to get a local copy of the source.

A.1.1 Installing the Frontend Application

The frontend application has the following requirements:

- ▻ Node.js in version 6 (or higher)
- ▻ The package manager `yarn` in version 0.20 (or higher).

After obtaining the source and installing the requirements, the application dependencies have to be installed. This is possible by running `yarn install` inside the `code/frontend` directory.

When all dependencies have been installed, the application can be started with `yarn start`. This command will start a development server which is accessible at `http://localhost:3000`.

The readme (`code/frontend/README.md`) contains more information and commands.

A.1.2 Installing the Backend Application

The backend application has the following requirements:

- ▻ Docker in version 17 (or higher)
- ▻ Docker Compose in version 1.11 (or higher).

After obtaining the source and installing the requirements, the application can be started by running `docker-compose up` inside the `code/backend` directory. This will start two docker containers, one containing the application and one containing the graph database. The backend is accessible at `http://localhost:8080/ubm-backend` and some sample data can be imported by accessing `http://localhost:8080/ubm-backend/v1/populateDatabase`.

The readme (`code/backend/README.md`) contains more information and commands.