Mobile Device Usage in Interactive, Co-located Presentations

Iris M. Schaffer

MASTERARBEIT

eingereicht am
Fachhochschul-Masterstudiengang

Interactive Media

in Hagenberg

im Juni 2016

Declaration

I hereby declare and confirm that this thesis is entirely the result of my own original work. Where other sources of information have been used, they have been indicated as such and properly acknowledged. I further declare that this or similar work has not been submitted for credit elsewhere.

Hagenberg, June 27, 2016

Iris M. Schaffer

Contents

Vorwort

Dies ist **Version 2015/09/19** der LaTeX-Dokumentenvorlage für verschiedene Abschlussarbeiten an der Fakultät für Informatik, Kommunikation und Medien der FH Oberösterreich in Hagenberg, die mittlerweile auch an anderen Hochschulen im In- und Ausland gerne verwendet wird.

Das Dokument entstand ursprünglich auf Anfragen von Studierenden, nachdem im Studienjahr 2000/01 erstmals ein offizieller LaTeX-Grundkurs im Studiengang Medientechnik und -design an der FH Hagenberg angeboten wurde. Eigentlich war die Idee, die bereits bestehende *Word*-Vorlage für Diplomarbeiten "'einfach"' in LaTeX zu übersetzen und dazu eventuell einige spezielle Ergänzungen einzubauen. Das erwies sich rasch als wenig zielführend, da LaTeX, vor allem was den Umgang mit Literatur und Grafiken anbelangt, doch eine wesentlich andere Arbeitsweise verlangt. Das Ergebnis ist – von Grund auf neu geschrieben und wesentlich umfangreicher als das vorherige Dokument – letztendlich eine Anleitung für das Schreiben mit LaTeX, ergänzt mit einigen speziellen (mittlerweile entfernten) Hinweisen für *Word*-Benutzer. Technische Details zur aktuellen Version finden sich in Anhang **??**.

Während dieses Dokument anfangs ausschließlich für die Erstellung von Diplomarbeiten gedacht war, sind nunmehr auch *Masterarbeiten*, *Bachelorarbeiten* und *Praktikumsberichte* abgedeckt, wobei die Unterschiede bewusst gering gehalten wurden.

Bei der Zusammenstellung dieser Vorlage wurde versucht, mit der Basisfunktionalität von LaTeX das Auslangen zu finden und – soweit möglich – auf zusätzliche Pakete zu verzichten. Das ist nur zum Teil gelungen; tatsächlich ist eine Reihe von ergänzenden "'Paketen"' notwendig, wobei jedoch nur auf gängige Erweiterungen zurückgegriffen wurde. Selbstverständlich gibt es darüber hinaus eine Vielzahl weiterer Pakete, die für weitere Verbesserungen und Finessen nützlich sein können. Damit kann sich aber jeder selbst beschäftigen, sobald das notwendige Selbstvertrauen und genügend Zeit zum Experimentieren vorhanden sind. Eine Vielzahl von Details und Tricks sind zwar in diesem Dokument nicht explizit angeführt, können aber im zugehörigen Quelltext jederzeit ausgeforscht werden.

Zahlreiche KollegInnen haben durch sorgfältiges Korrekturlesen und kon-

struktive Verbesserungsvorschläge wertvolle Unterstützung geliefert. Speziell bedanken möchte ich mich bei Heinz Dobler für die konsequente Verbesserung meines "‘Computer Slangs"’, bei Elisabeth Mitterbauer für das bewährte orthographische Auge und bei Wolfgang Hochleitner für die Tests unter Mac OS.

Die Verwendung dieser Vorlage ist jedermann freigestellt und an keinerlei Erwähnung gebunden. Allerdings – wer sie als Grundlage seiner eigenen Arbeit verwenden möchte, sollte nicht einfach ("‘ung’schaut"’) darauf los werken, sondern zumindest die wichtigsten Teile des Dokuments *lesen* und nach Möglichkeit auch beherzigen. Die Erfahrung zeigt, dass dies die Qualität der Ergebnisse deutlich zu steigern vermag.

Der Quelltext zu diesem Dokument sowie das zugehörige LaTeX-Paket sind in der jeweils aktuellen Version online verfügbar unter

https://sourceforge.net/projects/hgbthesis/.

Trotz großer Mühe enthält dieses Dokument zweifellos Fehler und Unzulänglichkeiten – Kommentare, Verbesserungsvorschläge und passende Ergänzungen sind daher stets willkommen, am einfachsten per E-Mail direkt an mich:

Dr. Wilhelm Burger, Department für Digitale Medien,
Fachhochschule Oberösterreich, Campus Hagenberg (Österreich)
wilhelm.burger@fh-hagenberg.at

Übrigens, hier im Vorwort (das bei Diplom- und Masterarbeiten üblich, bei Bachelorarbeiten aber entbehrlich ist) kann kurz auf die Entstehung des Dokuments eingegangen werden. Hier ist auch der Platz für allfällige Danksagungen (z. B. an den Betreuer, den Begutachter, die Familie, den Hund, ...), Widmungen und philosophische Anmerkungen. Das sollte allerdings auch nicht übertrieben werden und sich auf einen Umfang von maximal zwei Seiten beschränken.

Kurzfassung

An dieser Stelle steht eine Zusammenfassung der Arbeit, Umfang max. 1 Seite. Im Unterschied zu anderen Kapiteln ist die Kurzfassung (und das Abstract) üblicherweise nicht in Abschnitte und Unterabschnitte gegliedert. Auch Fußnoten sind hier falsch am Platz.

Kurzfassungen werden übrigens häufig – zusammen mit Autor und Titel der Arbeit – in Literaturdatenbanken aufgenommen. Es ist daher darauf zu achten, dass die Information in der Kurzfassung für sich *allein* (d. h. ohne weitere Teile der Arbeit) zusammenhängend und abgeschlossen ist. Insbesondere werden an dieser Stelle (wie u. a. auch im *Titel* der Arbeit und im *Abstract*) normalerweise *keine Literaturverweise* verwendet! Falls unbedingt solche benötigt werden – etwa weil die Arbeit eine Weiterentwicklung einer bestimmten, früheren Arbeit darstellt –, dann sind *vollständige* Quellenangaben in der Kurzfassung selbst notwendig, z. B. [Zobel J.: *Writing for Computer Science – The Art of Effective Communication.* Springer-Verlag, Singapur, 1997].

Weiters sollte daran gedacht werden, dass bei der Aufnahme in Datenbanken Sonderzeichen oder etwa Aufzählungen mit "'Knödellisten"' in der Regel verloren gehen. Dasselbe gilt natürlich auch für das *Abstract*.

Inhaltlich sollte die Kurzfassung *keine* Auflistung der einzelnen Kapitel sein (dafür ist das Einleitungskapitel vorgesehen), sondern dem Leser einen kompakten, inhaltlichen Überblick über die gesamte Arbeit verschaffen. Der hier verwendete Aufbau ist daher zwangsläufig anders als der in der Einleitung.

Abstract

This should be a 1-page (maximum) summary of your work in English.

Im englischen Abstract sollte inhaltlich das Gleiche stehen wie in der deutschen Kurzfassung. Versuchen Sie daher, die Kurzfassung präzise umzusetzen, ohne aber dabei Wort für Wort zu übersetzen. Beachten Sie bei der Übersetzung, dass gewisse Redewendungen aus dem Deutschen im Englischen kein Pendant haben oder völlig anders formuliert werden müssen und dass die Satzstellung im Englischen sich (bekanntlich) vom Deutschen stark unterscheidet (mehr dazu in Abschn. **??**). Es empfiehlt sich übrigens – auch bei höchstem Vertrauen in die persönlichen Englischkenntnisse – eine kundige Person für das "‘proof reading"’ zu engagieren.

Die richtige Übersetzung für "‘Diplomarbeit"’ ist übrigens schlicht *thesis*, allenfalls "‘diploma thesis"’ oder "‘Master's thesis"’, auf keinen Fall aber "‘diploma work"’ oder gar "‘dissertation"’. Für "‘Bachelorarbeit"’ ist wohl "‘Bachelor thesis"’ die passende Übersetzung.

Übrigens sollte für diesen Abschnitt die *Spracheinstellung* in LaTeX von Deutsch auf Englisch umgeschaltet werden, um die richtige Form der Silbentrennung zu erhalten, die richtigen Anführungszeichen müssen allerdings selbst gesetzt werden (s. dazu die Abschnitte **??** und **??**).

Chapter 1

Implementation

1.1 Project Scope

As the aim of the present work is to explore ways of incorporating mobile devices into presentation workflows, the goal of the project was to use an easily extensible presentation library to then build the mechanisms discussed in the previous chapter **??**. As the focus was placed on the interaction possibilities between speaker and audience, the creation of the presentation for the speaker or the management of slides and presentations were out of the project scope. Therefore the server used for connecting different users to the presentation was kept as simple as possible, allowing any potential other developer to work with their own servers and technology stacks.

In total, a system with several ways of interacting with the presentation from mobile or desktop devices was created, putting emphasise on mobile-optimised views and navigation possibilities. This system includes synchronisation of navigation state and state changes between viewers and speaker, the possibility to add sub-slides during the presentation for the audience, a speaker-view showing the next slides and controls, real-time voting (both created on-the-fly and prepared beforehands) and the possibility to create different paths through the presentation. In the following the technologies used in the project will be analysed and described to then go into details on the implementation, problems and solutions of the mentioned components.

1.2 Technologies

The project generally tries to follow best-practices in web development and utilises modern CSS3 and JavaScript features and frameworks. The software is written in ECMAScript2015, makes use of the *node package manager*[1](short *npm*) for managing dependencies and *Babel* to transpile to

[1] https://www.npmjs.com/

ECMAScript5. Additonally to relying on CSS3 features, this project also uses *Sass*[2] as a CSS pre-processor. Media-queries allow for mobile-friendly views.

On the front end, which this project focuses on, the JavaScript library *React* is the framework of choice, additionally applying the *reactive programming* paradigm using *RxJS* to allow for a simpler interface for event-driven operations. The communication between client and server is handled by *socket.io*[3]. This section tries to introduce the reader to the main technologies used to establish a base on which the following technical implementation details can be understood.

1.2.1 ECMAScript2015 and Babel[4]

JavaScript undoubtly is an integral part of front end web development and since the emergence of server-side JavaScript with Node.js[5] and its package manager npm has developed into a programming language widely used by web developers ([**gpm-meta-transcompiler**]). Both PYPL[6] and TIOBE[7] programming language indices rank JavaScript among the top 10 programming languages (PYPL at 5, TIOBE at 7 at the time of writing) ([**gpm-meta-transcompiler**]). Stack Overflow's 2015 Developer Survey even places JavaScript as the number 1, most-used programming language with 54.4% and JavaScript, Node.js and AngularJS[8] all three rank amongst the top 5 languages developers expressed an interest in developing with ([**stackoverflow-developer-survey**]).

However, like any front end technology, JavaScript suffers from slow end user adoption, as a multitude of browser versions exist for different devices and operating systems and many people still do not auto-update their browsers. Another factor is the time it takes for browser-vendors to implement new ECMAScript standards (the standard behind JavaScript) and roll out said updates. This is exactly what is happening with the new ECMAScript standard, ECMA-262, commonly known as ECMAScript 2015 or ES6: Although the General Assembly has adopted the new standard in June 2015 ([**ecma2015**]), *Kangax' ECMAScript compatibility tables*[9] still show a fairly low level of adoption, especially among mobile browsers. ES6 makes JavaScript easier and more efficient to write by providing new semantics for default values, arrow-functions, template-literals, the spread operator or ob-

[2] http://sass-lang.com/
[3] http://socket.io/
[4] https://babeljs.io/
[5] https://nodejs.org/en/
[6] http://pypl.github.io/PYPL.html
[7] http://www.tiobe.com/tiobe_index
[8] https://angularjs.org/
[9] https://kangax.github.io/compat-table/es6/

ject destructuring ([**es6**]). It also makes JavaScript easier to understand and safer to develop, with the introduction of block-scoped variables (`let` and `const`) and finally offers native support of modules and promises ([**es6**]). As these features are all included in the new ECMAScript standard, it is safe to assume browser-vendors will implement them in the near future. Until then, developers who want to already make use of them, can *transpile* ECMAScript 2015 code into ECMAScript 5, which is exactly what Babel does. With over 650000 downloads in March 2015 (according to npm) and companies like Facebook, Netflix, Mozilla, Yahoo or PayPal using this transpiler ([**babel-users**]), Babel is the de facto standard solution to transpile to ECMAScript 5 and was also chosen for this project.

1.2.2 Reactive Programming

Another problem with JavaScript, although integral part of the reason for its high popularity, is its asynchronous nature. Especially when working with highly interactive parts, the prime example being user interfaces, sequential programming quickly gets too inflexible to handle complex, event-driven applications ([**reactive-programming-survey**]). But also on the server, the possibility to concurrently serve a multitude of different clients, is crucial. In these cases JavaScript offers *event listeners* – functions called once a certain event happens. However, these event listeners or *asynchronous callback* ([**reactive-programming-survey**]), oftentimes executes more asynchronous code and in turn has to wait for another event, and another one, and another one... which can result in something known and dreaded by most any JavaScript developer: *Callback Hell* (see programm 2.1).

Different approaches have been employed to lower the hurdle of writing asynchronous code, one of them being *promises*: A promise is a value, yet to be computed ([**reactive-vs-promises**]). A promise can be a) pending (if it has not been assigned a value yet), b) resolved (if it has been assigned a value) or c) rejected (if an error occurred). In ECMAScript 2015 promises these objects can then be queued using the `then` keyword, to execute asynchronous code in a certain sequence (see programm 2.2).

However, promises can still create nested callbacks, especially when chaining promises that rely on other promisesŕesolution ([**reactive-vs-promises**]). This is where *reactive programming* comes in: The reactive programming paradigm works with streams of events, in which every event is handled as a new value and all other parts depending on this value are re-computed upon arrival of such a new value. To demonstrate this I would like to use Bainomugisha et al.ś illustrative example of a simple addition [**reactive-programming-survey**]:

```
1 var v1 = 1
2 var v2 = 2
3 var v3 = var1 + var 2
```

Program 1.1: *Callback Hell* – Nested callbacks in JavaScript. Simplified method taken from a previous project, which authenticates a user, creates a new google calendar for them and then saves the user to oneś own database, to then redirect them. `{...}` is used to shorten the code, error-handling was also omitted in the example for simplicity.

```
router.get('/callback', function(req, res, next) {
  var code = req.query.code;
  var name = JSON.parse(req.query.state);

  // get token from oauth library
  oauth2Client.getToken(code, function (err, tokens) {
    // load configuration
    Configuration.findOne({}, function (err, configuration) {
      var calendar = google.calendar('v3');
      // save to google calendar
      calendar.calendars.insert({...}, function (err, cal) {
        var member = new Member({...});
        // save member to own database
        member.save(function(err, member) {
          return res.redirect(getRedirectionUrl(name) + '&success=true')
        });
      });
    });
  });
});
```

Program 1.2: *Promises* – Simple example of chaining ECMAScript 2015 promises with `then` and `catch`.

```
var promise = new Promise(function(resolve, reject) {
  asyncCall(function(error, data) {
    if (error) {
      reject(error); // reject the pending promise
    } else {
      resolve(data); // resolve the pending promise
    }
  })
})

promise
  .then(function(data) {
    // this is executed after asyncCall returns
    // other asyncronous calls can be placed here
  })
  .catch(function(error) {
    // this is executed if an error occurs somwehere along the way
  })
```

Program 1.3: *RxJS* – simplified example of the touch controls used to swipe to the next or previous slide. An Observable is created from the browser's `touchmove` event and is then transformed with `map` and `filter`, to in the end call the `navigate()` method with the direction the user swiped into.

```
1  this.moveObservable = Observable.fromEvent(document, 'touchmove')
2    // data: event object with array of touches
3    .filter(this.touchStarted) // only procede if touchstart is set to true
4    .map(this.toXY) // transform initial event data to latest touch's xy position
5    // data: {x: xPosition, y: yPosition}
6    .map(this.toDirection) // transform xy position to direction literal
7    // data: right/left/up/down
8    .do(this.resetTouchStart) // set this.touchStarted to false
9    // data: right/left/up/down
10   .subscribe(this.navigate); // call this.navigate() with direction data
```

In sequentially executed code, `v3` will hold the value 3, no matter if or how `v1` or `v2` change. In reactive programming, however, `v3` will be re-computed as soon as either one of the values it depends on change ([**reactive-programming-survey**]). This way for a drag-and-drop feature, for example, the move of the mouse, continuously sending its location, could directly alter the position of an element in a page. JavaScript does not directly support reactive programming, but other, more functional languages, which can be transpiled to JavaScript, do. Another way of adding reactive programming concepts to JavaScript is using a library, such as *Bacon.js*[10] or the one chosen for this project, *ReactiveX*[11]. ReactiveX provides libraries for a multitude of different programming languages, C, C++, Java and of course JavaScript among them. The latter, called *The Reactive Extensions for JavaScript* or short *RxJs*, allows for the simple creation of event streams (*Observables*) from browser events or promises directly and uses the same method names JavaScript developers are familiar with from array-methods, most notably and well-known `map`, to apply a method to every element in the incoming stream and `filter`, to only let a subset of events pass. These methods can be chained to sequentially alter a value (see programm 2.3).

Additionally to Observables, RxJs also knows *Subjects*, which combine both a source of events and a consumer of such. Subjects are Observables, but also Observers at the same time and can be used to broadcast values to several consumers ([**rxjs-docu**]).

[10] https://baconjs.github.io/
[11] http://reactivex.io/

1.2.3 React[12]

As this project concentrates on the front end, a mature JavaScript framework was searched for. After previous experience with the big and complex, but slow AngularJS, because of promising performance benchmarks ([**react-benchmarks**]) and simply to explore new JavaScript libraries, I decided to give React a try. Since Facebook started developing React in 2013, it has challenged existing approaches and set new standards in front end web development ([**introduction-to-react**]). Instead of creating an entire MVC framework for the front end, React really concentrates on the view by offering a way of creating independent, lightweight view components. This gives React the huge advantage of beating other front end frameworks by far in performance benchmarks ([**react-benchmarks**]). Moreover, *React Native*[13] would make it possible to port the application to different mobile operation systems fairly easily. The arguably most important method these re-usable, lightweight components implement is the `render` method, defining what HTML or JSX[14] should be rendered by the browser:

```
1 export default class HelloWorld extends Component {
2   render() {
3     return (<h1>Hello World!</h1>);
4   }
5 }
```

The created Component can then be rendered into the virtual React DOM, JSX makes it possible to simply use the name of the component to create it:

```
1 ReactDOM.render(
2   <HelloWorld />,
3   document.getElementsByTagName('body')[0]
4 );
```

This would simply put an `h1` element with the text `Hello World!` into the `body` element of the HTML page. Additionally to the `render` method, components also have a *state* and *properties*(`props`), through which they can communicate with other components and maintain their inner state. `props` are passed in to the component during creation, in JSX this can be achieved by simply passing them in as XML attributes:

```
1   <HelloWorld greeting="Hi" />
```

This could in turn be used in the `HelloWorld` component's `render` method:

```
1   render() {
2     return (<h1>{this.props.greeting} World!</h1>);
3   }
```

[12] https://facebook.github.io/react/index.html
[13] https://facebook.github.io/react-native/
[14] https://facebook.github.io/jsx/

The children of a component are also available through [this.props.children]. The `state` variable, on the other hand, is responsible for handling internal updates e.g. through user interactions ([**react-docu**]). To alter the state, the method `setState` can be used, which will cause an update and re-rendering of the component. So if in the above example, the word "World" should be changed to something else by the user, a text field with an event handler can be added inside the component:

```
1 // set default state and props...
2
3 componentWillMount() {
4   // create observable from change event on input
5   this.observable = Observable.fromEvent('change', this.refs.input)
6     .pluck('target', 'value') // extract input text
7     .subscribe(this.update) // call update with value
8 }
9
10 componentWillUnmount() { this.observable.unsubscribe() }
11
12 update(text) { this.setState({name: text}) }
13
14 render = () => (
15   <div>
16     <h1>{this.props.greeting} {this.state.name}!</h1>
17     <input value={this.state.name} ref="input" />
18   </div>
19 )
```

Now, whenever a user changes the text in the `input` field, the RxJS Observable will receive a new value (a change event). The `value` of the field is extracted on line 6 and used to then update the state in the `update` method, which is implicitly called with the value passed through the Observable chain.

The two methods `componentWillMount`, `componentWillUnmount` as well as `componentDidMount`, `shouldComponentUpdate`, `componentWillUpdate`, `componentDidUpdate` and `componentWillReceiveProps` are *lifecycle methods*, called whenever the component is created, updated or destroyed.

As an end note on React, and a transition to the core presentation library, I want to add that React components can be nested, which is at the core of the project developed for the present thesis. To make it as easy as possible for other developers to use the created libraries, a presentation is built just as a usual HTML page, using JSX to reference the custom React components (see program 2.4).

Program 1.4: Nested React components. In this example, a 1-slide-long presentation is created.

```
<UnveilApp>
  <Slide name="start">
    <h1>Unveil</h1>
    <h2>a meta presentation</h2>
    <Notes>Greet people and welcome everyone</Notes>
  </Slide>
</UnveilApp>
```

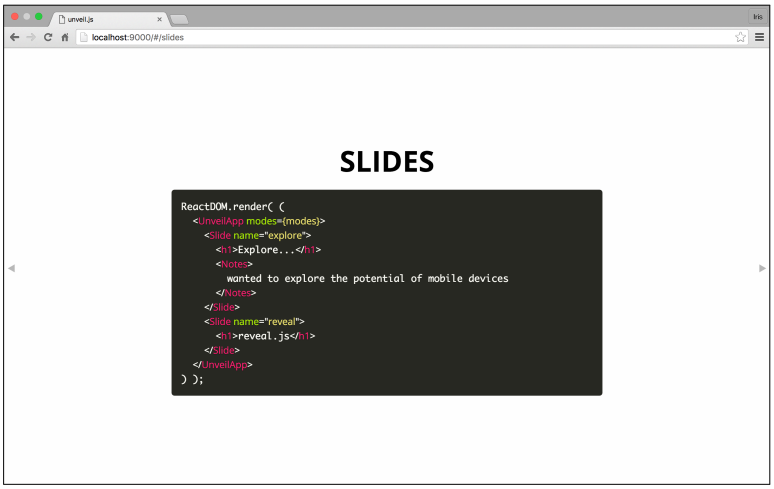


Figure 1.1: Screenshot of an example slide with unveil.js.

1.2.4 unveil.js[15]

As a presentation layer, this project uses the open-source JavaScript library *unveil.js*, which was developed by Leandro Ostera and myself in the beginning of the project and which I extended and adapted to my needs in an own fork[16] during the project. This fork will be covered in section 2.4 of this chapter, until then, a short overview of the different parts of unveil.js is given and key concepts of the library are discussed. Screenshots of what unveil.js looks like can be found in figure 2.1.

Generally, unveil.js operates on a 2-dimensional slide-space: Every slide can have a next and previous slide in x, as well as in y-direction. To generate the y axis, slides can be nested in other slides. These slides have an optional unique name as well as an index in the slide-tree, which they are identified by. As shown in program 2.4, slides are created inside the component

[15] https://github.com/ostera/unveil.js
[16] https://github.com/irisSchaffer/unveil.js

`UnveilApp`, the core of unveil.js. This component configures and sets up the entire application based on optional configuration passed in as properties. There are a few concepts unveil.js is built around to allow for extensibility and configurability, namely *presenters*, *controls* and *modes*:

- **Presenters** define the way slides are rendered, e.g. show notes, upcoming slides or hide them.
- **Controls** control a part of the application, e.g. navigating from one slide to the next using the arrow keys on the keyboard.
- **Modes** are what allows a speaker to have a different presenter and controls from an audience member. Each mode defines its own presenter and set of controls, the mode is determined by the url query parameter `mode`.

This allows anybody using unveil.js to define new modes, presenters and controls and thereby extend the base library as they wish. A few of these are already defined, namely a default `Presenter`, `UIControls` to navigate using buttons, `KeyControls` to navigate using the keyboard and `TouchControls` to navigate with swipe-gestures on touch screens. In section 2.8, modes for the audience (*default*), the speaker (*speaker*) and for use on the projection device (*projector*) will be introduced.

For these controls and the entire presentation to be navigatable, `UnveilApp` is responsible for the creation of two very important classes: `Router` and `Navigator`. These can be defined outside and passed into `UnveilApp` as properties, allowing users to customise their navigation logic. The `Router` is the class handling everything connected to the current url. It receives the slide-tree and can compute the indices of a slide by its name and vice versa. Whenever the browser history changes, the router finds the corresponding slide-indices, computes an array of possible directions to go into from this slide and propagates the event to `UnveilApp`, which can then re-render the application. `Navigator`, in turn, receives these directions and is responsible for the mapping of directions (*left*, *right*, *up*, *down*) to slide-indices. Controls know the navigator and can push new directions to the navigator subject, thus starting the navigation process described in detail in figure 2.2.

1.3 Project Structure

As the puropose of this project was not only to experiment with different ways of interacting with presentations using mobile devices, but also to create something worthwhile and contribute back to the vibrant open-source community, the project is enitrely open-source and separated into different repositories, which are all available on GitHub. These can be installed using npm, therefore allowing developers to rely only on the parts they really need.

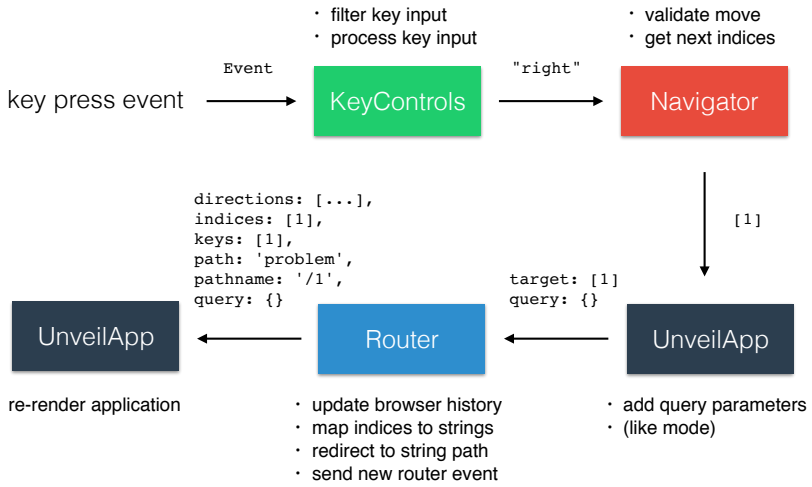


Figure 1.2: Navigation pipeline from user's key press to re-render of the presentation. The monospaced text next to the arrows symbolises the data transmitted. `KeyControls` listen for key events and process them, to then send a navigation request to go *right* to the `Navigator`. This component then maps the direction to the next slide's indices (1). `UnveilApp` then adds other information necessary for the `Router`, which then is responsible for updating the browser history, mapping the indices back to a human readable url and sending out a new router event. In the end `UnveilApp` receives this event and re-renders the presentation.

Extended unveil.js: As discussed in section 2.2.4, the project is based on the library unveil.js. During the development of the project, certain parts of the base library did not offer the flexibility needed for easy extensibility and so several parts were adapted and new presentation logic was added. This happened in a fork of the original library, which will be examined in section 2.4.

Network Synchronisation Layer: The first library of direct importance for the interaction between speaker and audience through personal devices is *unveil-network-sync*[17]. This rather small library relies on unveil.js and is responsible for connecting the client and the server through web sockets and enables the synchronisation of the current slide displayed between speaker, audience and projector. The implementation of the features will be discussed in detail in section 2.5.

Interactive Extension: As the name already suggests, this library is at the core of the present thesis: It includes a dedicated presenter for the

[17] https://github.com/irisSchaffer/unveil-network-sync

speaker, implements the insertion of additional slides and subslides and by that allows the audience to share content with the presentation. The voting mechanism, as well as the creation of new votings on-the-fly, also live within this library. The repository, called *unveil-interactive*[18] relies on unveil-network-sync for the socket-interaction. The interactive extension will be covered in section 2.6 of this chapter.

Server and Example Presentation: The last repository connected to this thesis is *unveil-client-server*[19], which includes a simple server as well as a real-world example of a presentation, which was used in the intermediate thesis project presentation as part of the Interactive Media course IM690, on the 2nd of February, 2016. In this chapter, a whole section was dedicated to the server (2.7), as well as to the example application (2.8), to separate concerns a bit more clearly and be able to conclude with a demonstration of how all parts discussed earlier play together in a final presentation.

1.4 Extended unveil.js

The biggest adaptions and additions were necessary in the main component `UnveilApp`. A state subject was added to allow all components in the presentation to interact with the application. This subject receives an event with type and data and depending on this type starts a certain state change. Two of these state events are the `state/navigation:enable` and `state/navigation:disable` events, which set a state-variable `navigatable` to true or false. This variable is used in the controls to determine navigatability, therefore making it possible to keep the audience locked to a slide, e.g. during votings. To make it possible for the audience to add subslides, as well as to dynamically add votings on-the-fly, another event is `state/slide:add`. It includes what slide to add (content), how (subslide or main slide) and where (under or after which slide). On occurence of this event, the slide-tree has to be re-built, the router and navigator re-started and the whole presentation re-rendered, which the library also had to be prepared for.

Another adaption in `UnveilApp` is the introduction of the `context` object: Additionally to state and properties, there is a third way of communicating between components in React, called *context*. Instead of having to pass properties from one nested component to the other, every child component can access the context of its parents. The navigator, needed in the controls, was formerly passed from `UnveilApp` to the controls through several layers. Using context, `UnveilApp` now defines a number of different variables which are available through context, including current slide and router state, navigator, mode and the state subject discussed in the last paragraph.

[18] https://github.com/irisSchaffer/unveil-interactive
[19] https://github.com/irisSchaffer/unveil-client-server

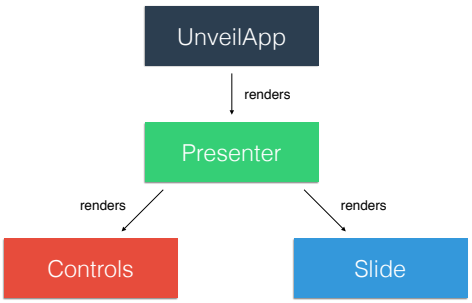


Figure 1.3: Overview over the render-flow of the application in the extended version of unveil.js. `UnveilApp` renders the presenter, which then takes care of rendering the (current) slide and all controls.

This makes it easy for new controls and presenters to access the data they need without other layers knowing about them or having to define them. This adaption was partly due to a change in the render hierarchy: Formerly, `UnveilApp` itself rendered the presenter (which rendered the current slide) and the controls. However, the presenter needs to be able to also control the rendering of controls (see figure 2.3), adding another layer between the rendering of controls and `UnveilApp`.

Another part that was added to the unveil.js base library is the `Notes` component. It allows adding speaker notes to each slide (see program 2.4). These, however, are not rendered by the slide, but by the presenter, as will be shown in section 2.6. One more important new feature is the possibility to configure the next slide in a certain direction (left/right/up/down) and therefore allow for jumping into different branches of the presentation, thus making a presentation even more interactive. The following code, for example

```
1  <Slide name="start" left={[0]}>
2    ...
3  </Slide>
```

means a navigation *left* (left arrow key pressed, swipe left etc.) will not go to the previous slide defined in the slide-tree, but rather jump to the first slide (of index 0).

1.5 Network Synchronisation Layer

As mentioned before, the network synchronisation layer is responsible for the communcation between server and client using web sockets. These are created with socket.io, a library which also provides fallbacks for browsers that do not support web sockets yet. However, this library also has a few drawbacks, especially when it comes to corporate networks. As Rob Britton

Program 1.5: Shortened version of `NavigationReceiver`. First the inherited context properties are set up, then an observable waiting for `state:change` events from the socket is created. If the incoming request is not the currently displayed slide, the navigator will be pushed a new value.

```
1  // imports...
2
3  export default class NavigationReceiver extends React.Component {
4    static contextTypes = {
5      navigator:   React.PropTypes.object.isRequired,
6      routerState: React.PropTypes.object.isRequired
7    }
8
9    componentDidMount () {
10     this.observable = Observable.fromEvent(SocketIO, 'state:change')
11       .filter((e) => !this.context.routerState.indices.equals(e))
12       .subscribe(this.props.navigator.next)
13   }
14
15   // ...
16 }
```

describes in [**socketio-problems**], socket.io seems to have problems getting through firewalls and can be blocked by some anti virus software. Because mobile browser support is essential for this project, I decided to still use this library.

The setup of the socket is simple: one helper function, called `createSocket` is called in the main entry point of the application to configure which server to connect to:

```
1 import { createSocket } from 'unveil-network-sync'
2 createSocket('46.101.166.172:9000')
```

This function creates the socket and returns it as a singleton, so every component uses the same connection. To make importing even easier, there is another helper, called `SocketIO` which dan be importet directly and internally calls `createSocket` to retrieve the singleton:

```
1 import { SocketIO } from 'unveil-network-sync'
```

These sockets can then be used to listen to events or to emit them (see program 2.5) Like the state subject events, the socket.io events used in this library follow the naming convention of scoping the object targeted in by the event separated by slashes, followed by a colon and the name of the action, e.g. `state:change` or `state/slide/voting:start`.

The second responisbilty of this library is synchronising the navigation state of the presentation between speaker and audience. To do this, two controls, `NavigationSender` and `NavigationReceiver` were implemented. As the names already say, the sender broadcasts the state update, while

the receiver is waiting for state updates and starts the navigation process. The latter is used in all modes (default, speaker and projector), whereas the sender is only added to the speaker mode. To make sure the sender does not end up in an infinite loop of sending and receiving its own state changes, the last received state is stored and only navigation events going to a different slide are processed further.

This mechanism, though relatively simple, already enables the audience to follow the presentation, the speaker to use his/her phone as a remote control and any number of projectors to be controlled by the speaker.

1.6 Interactive Extension

The interactive library includes several parts which will be discussed here: a speaker presenter (section 2.6.1) as well as different components connected to sharing media (section 2.6.2) and voting (section 2.6.3). Additionally controls handling the `state:initial` event were implemented to redirect new listeners to the current slide in the presentation.

1.6.1 Speaker Presenter

The speaker presenter, like the normal presenter, is responsible for rendering controls and slides. In speaker mode, where this presenter is used, notes as well as the upcoming slides (right and down) are also shown (see figure 2.4). This means the presenter has to find these next slides, using the router's information about the navigatable directions, and render them in a designated area. To make the presenter view usable on mobile devices, special attention was paid to mobile stylesheets (see figure 2.5 (b)). This ensures that everything is big enough to be readable and all buttons are clickable.

1.6.2 Media

Another responsibility of the interactive extension is the possibility for audience members to share content with the presentation. For this to work, three different controls were created: `MediaSender`, `MediaReceiver` and `MediaAcceptor`. The sender is used in the default mode so users can share their content (see figure 2.6 (a)), the acceptor is enabled in speaker mode, to accept or reject incoming media (see figure 2.6 (b)) and the receiver in the end handles the creation of a new slide if the content was accepted and is therefore necessary in all modes. As incoming content requests could disrupt the presentation flow and distract the speaker, an option to mute the requests was built into the application. If the *do not disturb* mode is turned on, slides will silently be added as subslides, without causing the acceptor modal to open. This way the audience' additions can be re-visited after the end of the presentation. Generally, this feature can be used to either post a

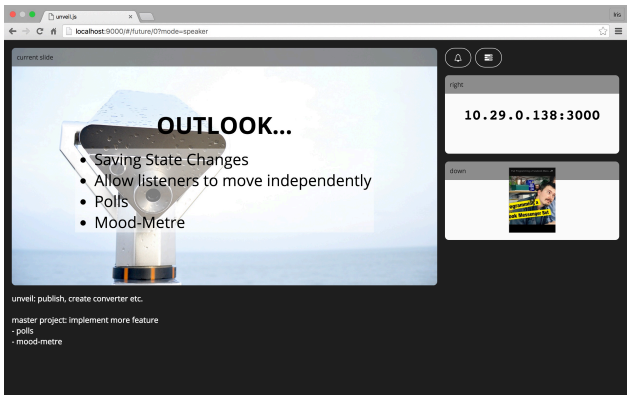
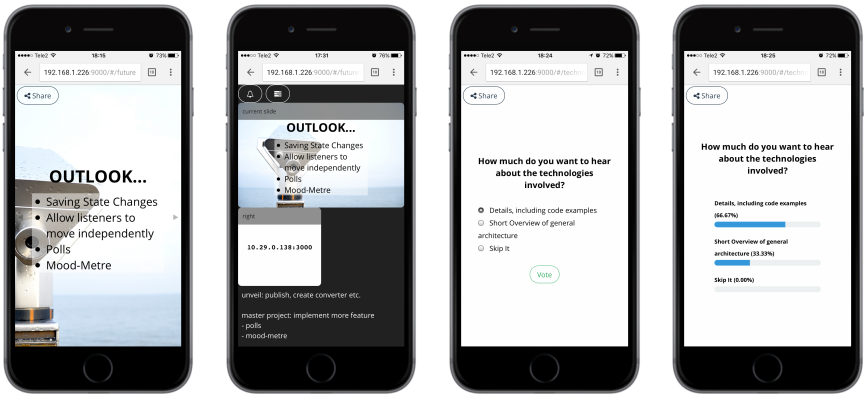


Figure 1.4: Screenshot of the speaker presenter with the current main slide, the upcoming slide to the right, available actions (muting and adding votings) and speaker notes.

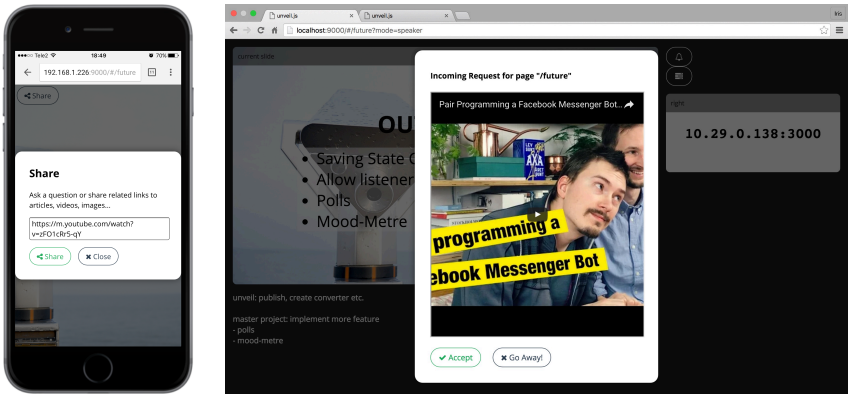


(a) (b) (c) (d)

Figure 1.5: Mobile view of a presentation slide in (a) default mode and (b) speaker mode as well as (c) voting view before voting and (d) voting view after voting. Mind the share button in the left upper corner in (a) as well as the buttons to mute content requests and create votings in (b).

link to an interesting picture, website or even youtube video, or also as text input, for example to add a comment or a question regarding a certain slide. This works through the introduction of the presentation components `Media` and `IFrame`, which, depending on the shared content, render an image-tag, blockquote or IFrame. The differentiation of these is carried out with regular expressions, as the following to check if the content is the link to an image:

```
1 isImg (str) {
2   let imgRegex = new RegExp(/\.(jpe?g|png|gif|bmp)$/i)
3   return imgRegex.test(str)
4 }
```



(a) (b)

Figure 1.6: Screenshots of sharing content. (a) media sender on mobile phone shares youtube link (b) speaker receives content request.

As far as the implementation of the controls is concerned, these ones are the first ones discussed in the present thesis makeing use of the `render()` method. It is used to output the *share* button in the left upper corner in default mode (see figure 2.5 (a)) as well as the share modal opened when clicking on said button (see figure 2.6 (a)). The same happens in the `MediaAcceptor`, which uses the `render()` method to display the *mute* button shown in figure 2.4 as well as the modal for accepting media (see figure 2.6 (b)).

These are also the first controls using state: in the sender a click on the share button sets the state variable `sharingMode` to `true` and through that enables the rendering of the modal. In the acceptor an array of `requests` is filled as new content is shared and emptied again, as the speaker accepts or denies them.

On event level, the socket events `state/slide/add:accept` and `state/slide:add` are included in the process of sharing new content, in the end an unveil state event of type `state/slide:add` lets `UnveilApp` create the new slide and add it to the slide tree. The whole flow is outlined in details in figure 2.7.

1.6.3 Voting

The last group of components connected to the interactive extension covered in this chapter allow speakers to create votings (both during in the preparation of the presentation and on-the-fly as shown in figure 2.8) and members of the audience to vote (see figure 2.5 (c) and (d)).

The main presentational component involved in the voting process is `Voting`, which keeps track of the current voting scores and has a `Question`

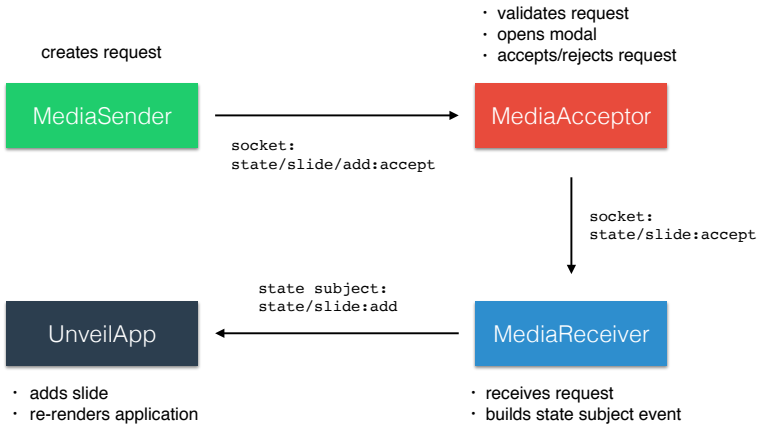


Figure 1.7: Flow of adding content, monospaced text symbolises type and name of events. First the `MediaSender` of the default mode sends a request, which the speaker mode's `MediaAcceptor` listens to. If the request is accepted by the speaker or requests are muted, another socket event is broadcast, which the `MediaReceiver` waits for. This component is enabled in all modes and emits the state subject event to add a new slide, which `UnveilApp` reacts to.

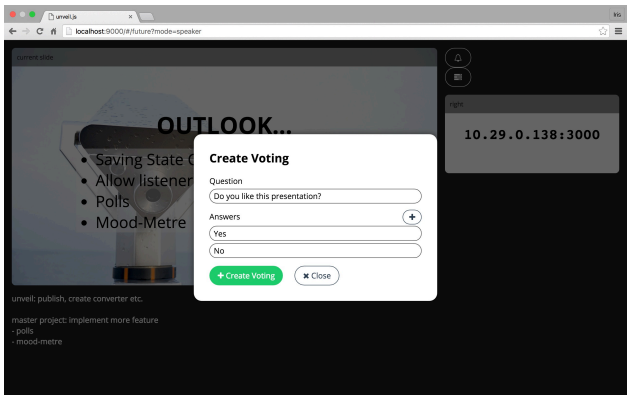


Figure 1.8: Screenshot of the desktop speaker mode, creating a new voting, on-the-fly, during a presentation.

and a number of `Answer` components as children (see program 2.6). `Voting` also remembers if an audience member has already voted and if so, displays `Result` components. In speaker and presenter mode only these results are shown.

The audience can start voting as soon as the speaker navigates to the slide with the voting, until then the vote button is disabled. Once the voting has started, the possibility for all audience members to navigate to a different

Program 1.6: Example code for preparing a slide with voting.

```
1 <Voting name="like">
2   <Question>Do you like these slides?</Question>
3   <Answer value="yes">Yes</Answer>
4   <Answer value="no">No</Answer>
5 </Voting>
```

slide is disabled. This happens in the `VotingNavigatableSetter`, which is only installed for default mode. The voting start event, as well as the voting end event are broadcast by the `VotingController`, which checks the speaker's current slide for the occurence of a `Voting`.

Once the voting has started, internally, the `Voting` component remembers which answer the user has clicked in the `answer` state variable. Once the submit button is pressed, a `state/slide/voting:answer` event is fired and broadcast to all clients, which update their internal voting results. Because the results of the voting should be available throughout the whole presentation and not be reset when leaving the slide, `Voting` also handles the communication with local storage, to store current results.

1.7 Server

1.8 Example Application

Chapter 2

Implementation

2.1 Project Scope

As the aim of the present work is to explore ways of incorporating mobile devices into presentation workflows, the goal of the project was to use an easily extensible presentation library to then build the mechanisms discussed in the previous chapter **??**. As the focus was placed on the interaction possibilities between speaker and audience, the creation of the presentation for the speaker or the management of slides and presentations were out of the project scope. Therefore the server used for connecting different users to the presentation was kept as simple as possible, allowing any potential other developer to work with their own servers and technology stacks.

In total, a system with several ways of interacting with the presentation from mobile or desktop devices was created, putting emphasise on mobile-optimised views and navigation possibilities. This system includes synchronisation of navigation state and state changes between viewers and speaker, the possibility to add sub-slides during the presentation for the audience, a speaker-view showing the next slides and controls, real-time voting (both created on-the-fly and prepared beforehands) and the possibility to create different paths through the presentation. In the following the technologies used in the project will be analysed and described to then go into details on the implementation, problems and solutions of the mentioned components.

2.2 Technologies

The project generally tries to follow best-practices in web development and utilises modern CSS3 and JavaScript features and frameworks. The software is written in ECMAScript2015, makes use of the *node package manager*[1](short *npm*) for managing dependencies and *Babel* to transpile to

[1] https://www.npmjs.com/

ECMAScript5. Additonally to relying on CSS3 features, this project also uses *Sass*[2] as a CSS pre-processor. Media-queries allow for mobile-friendly views.

On the front end, which this project focuses on, the JavaScript library *React* is the framework of choice, additionally applying the *reactive programming* paradigm using *RxJS* to allow for a simpler interface for event-driven operations. The communication between client and server is handled by *socket.io*[3]. This section tries to introduce the reader to the main technologies used to establish a base on which the following technical implementation details can be understood.

2.2.1 ECMAScript2015 and Babel[4]

JavaScript undoubtly is an integral part of front end web development and since the emergence of server-side JavaScript with Node.js[5] and its package manager npm has developed into a programming language widely used by web developers ([**gpm-meta-transcompiler**]). Both PYPL[6] and TIOBE[7] programming language indices rank JavaScript among the top 10 programming languages (PYPL at 5, TIOBE at 7 at the time of writing) ([**gpm-meta-transcompiler**]). Stack Overflow's 2015 Developer Survey even places JavaScript as the number 1, most-used programming language with 54.4% and JavaScript, Node.js and AngularJS[8] all three rank amongst the top 5 languages developers expressed an interest in developing with ([**stackoverflow-developer-survey**]).

However, like any front end technology, JavaScript suffers from slow end user adoption, as a multitude of browser versions exist for different devices and operating systems and many people still do not auto-update their browsers. Another factor is the time it takes for browser-vendors to implement new ECMAScript standards (the standard behind JavaScript) and roll out said updates. This is exactly what is happening with the new ECMAScript standard, ECMA-262, commonly known as ECMAScript 2015 or ES6: Although the General Assembly has adopted the new standard in June 2015 ([**ecma2015**]), *Kangax' ECMAScript compatibility tables*[9] still show a fairly low level of adoption, especially among mobile browsers. ES6 makes JavaScript easier and more efficient to write by providing new semantics for default values, arrow-functions, template-literals, the spread operator or ob-

[2] http://sass-lang.com/
[3] http://socket.io/
[4] https://babeljs.io/
[5] https://nodejs.org/en/
[6] http://pypl.github.io/PYPL.html
[7] http://www.tiobe.com/tiobe_index
[8] https://angularjs.org/
[9] https://kangax.github.io/compat-table/es6/

ject destructuring ([**es6**]). It also makes JavaScript easier to understand and safer to develop, with the introduction of block-scoped variables (`let` and `const`) and finally offers native support of modules and promises ([**es6**]). As these features are all included in the new ECMAScript standard, it is safe to assume browser-vendors will implement them in the near future. Until then, developers who want to already make use of them, can *transpile* ECMAScript 2015 code into ECMAScript 5, which is exactly what Babel does. With over 650000 downloads in March 2015 (according to npm) and companies like Facebook, Netflix, Mozilla, Yahoo or PayPal using this transpiler ([**babel-users**]), Babel is the de facto standard solution to transpile to ECMAScript 5 and was also chosen for this project.

2.2.2 Reactive Programming

Another problem with JavaScript, although integral part of the reason for its high popularity, is its asynchronous nature. Especially when working with highly interactive parts, the prime example being user interfaces, sequential programming quickly gets too inflexible to handle complex, event-driven applications ([**reactive-programming-survey**]). But also on the server, the possibility to concurrently serve a multitude of different clients, is crucial. In these cases JavaScript offers *event listeners* – functions called once a certain event happens. However, these event listeners or *asynchronous callback* ([**reactive-programming-survey**]), oftentimes executes more asynchronous code and in turn has to wait for another event, and another one, and another one... which can result in something known and dreaded by most any JavaScript developer: *Callback Hell* (see programm 2.1).

Different approaches have been employed to lower the hurdle of writing asynchronous code, one of them being *promises*: A promise is a value, yet to be computed ([**reactive-vs-promises**]). A promise can be a) pending (if it has not been assigned a value yet), b) resolved (if it has been assigned a value) or c) rejected (if an error occurred). In ECMAScript 2015 promises these objects can then be queued using the `then` keyword, to execute asynchronous code in a certain sequence (see programm 2.2).

However, promises can still create nested callbacks, especially when chaining promises that rely on other promisesŕesolution ([**reactive-vs-promises**]). This is where *reactive programming* comes in: The reactive programming paradigm works with streams of events, in which every event is handled as a new value and all other parts depending on this value are re-computed upon arrival of such a new value. To demonstrate this I would like to use Bainomugisha et al.ś illustrative example of a simple addition [**reactive-programming-survey**]:

```
1 var v1 = 1
2 var v2 = 2
3 var v3 = var1 + var 2
```

Program 2.1: *Callback Hell* – Nested callbacks in JavaScript. Simplified method taken from a previous project, which authenticates a user, creates a new google calendar for them and then saves the user to oneś own database, to then redirect them. `{...}` is used to shorten the code, error-handling was also omitted in the example for simplicity.

```
router.get('/callback', function(req, res, next) {
  var code = req.query.code;
  var name = JSON.parse(req.query.state);

  // get token from oauth library
  oauth2Client.getToken(code, function (err, tokens) {
    // load configuration
    Configuration.findOne({}, function (err, configuration) {
      var calendar = google.calendar('v3');
      // save to google calendar
      calendar.calendars.insert({...}, function (err, cal) {
        var member = new Member({...});
        // save member to own database
        member.save(function(err, member) {
          return res.redirect(getRedirectionUrl(name) + '&success=true')
        });
      });
    });
  });
});
```

Program 2.2: *Promises* – Simple example of chaining ECMAScript 2015 promises with `then` and `catch`.

```
var promise = new Promise(function(resolve, reject) {
  asyncCall(function(error, data) {
    if (error) {
      reject(error); // reject the pending promise
    } else {
      resolve(data); // resolve the pending promise
    }
  })
})

promise
  .then(function(data) {
    // this is executed after asyncCall returns
    // other asyncronous calls can be placed here
  })
  .catch(function(error) {
    // this is executed if an error occurs somwehere along the way
  })
```

Program 2.3: *RxJS* – simplified example of the touch controls used to swipe to the next or previous slide. An Observable is created from the browser's `touchmove` event and is then transformed with `map` and `filter`, to in the end call the `navigate()` method with the direction the user swiped into.

```
1  this.moveObservable = Observable.fromEvent(document, 'touchmove')
2    // data: event object with array of touches
3    .filter(this.touchStarted) // only procede if touchstart is set to true
4    .map(this.toXY) // transform initial event data to latest touch's xy position
5    // data: {x: xPosition, y: yPosition}
6    .map(this.toDirection) // transform xy position to direction literal
7    // data: right/left/up/down
8    .do(this.resetTouchStart) // set this.touchStarted to false
9    // data: right/left/up/down
10   .subscribe(this.navigate); // call this.navigate() with direction data
```

In sequentially executed code, `v3` will hold the value 3, no matter if or how `v1` or `v2` change. In reactive programming, however, `v3` will be re-computed as soon as either one of the values it depends on change ([**reactive-programming-survey**]). This way for a drag-and-drop feature, for example, the move of the mouse, continuously sending its location, could directly alter the position of an element in a page. JavaScript does not directly support reactive programming, but other, more functional languages, which can be transpiled to JavaScript, do. Another way of adding reactive programming concepts to JavaScript is using a library, such as *Bacon.js*[10] or the one chosen for this project, *ReactiveX*[11]. ReactiveX provides libraries for a multitude of different programming languages, C, C++, Java and of course JavaScript among them. The latter, called *The Reactive Extensions for JavaScript* or short *RxJs*, allows for the simple creation of event streams (*Observables*) from browser events or promises directly and uses the same method names JavaScript developers are familiar with from array-methods, most notably and well-known `map`, to apply a method to every element in the incoming stream and `filter`, to only let a subset of events pass. These methods can be chained to sequentially alter a value (see programm 2.3).

Additionally to Observables, RxJs also knows *Subjects*, which combine both a source of events and a consumer of such. Subjects are Observables, but also Observers at the same time and can be used to broadcast values to several consumers ([**rxjs-docu**]).

[10] https://baconjs.github.io/
[11] http://reactivex.io/

2.2.3 React[12]

As this project concentrates on the front end, a mature JavaScript framework was searched for. After previous experience with the big and complex, but slow AngularJS, because of promising performance benchmarks ([**react-benchmarks**]) and simply to explore new JavaScript libraries, I decided to give React a try. Since Facebook started developing React in 2013, it has challenged existing approaches and set new standards in front end web development ([**introduction-to-react**]). Instead of creating an entire MVC framework for the front end, React really concentrates on the view by offering a way of creating independent, lightweight view components. This gives React the huge advantage of beating other front end frameworks by far in performance benchmarks ([**react-benchmarks**]). Moreover, *React Native*[13] would make it possible to port the application to different mobile operation systems fairly easily. The arguably most important method these re-usable, lightweight components implement is the `render` method, defining what HTML or JSX[14] should be rendered by the browser:

```
1 export default class HelloWorld extends Component {
2   render() {
3     return (<h1>Hello World!</h1>);
4   }
5 }
```

The created Component can then be rendered into the virtual React DOM, JSX makes it possible to simply use the name of the component to create it:

```
1 ReactDOM.render(
2   <HelloWorld />,
3   document.getElementsByTagName('body')[0]
4 );
```

This would simply put an `h1` element with the text `Hello World!` into the `body` element of the HTML page. Additionally to the `render` method, components also have a *state* and *properties*(`props`), through which they can communicate with other components and maintain their inner state. `props` are passed in to the component during creation, in JSX this can be achieved by simply passing them in as XML attributes:

```
1   <HelloWorld greeting="Hi" />
```

This could in turn be used in the `HelloWorld` component's `render` method:

```
1   render() {
2     return (<h1>{this.props.greeting} World!</h1>);
3   }
```

[12] https://facebook.github.io/react/index.html
[13] https://facebook.github.io/react-native/
[14] https://facebook.github.io/jsx/

The children of a component are also available through [this.props.children]. The `state` variable, on the other hand, is responsible for handling internal updates e.g. through user interactions ([**react-docu**]). To alter the state, the method `setState` can be used, which will cause an update and re-rendering of the component. So if in the above example, the word "World" should be changed to something else by the user, a text field with an event handler can be added inside the component:

```
1  // set default state and props...
2
3  componentWillMount() {
4    // create observable from change event on input
5    this.observable = Observable.fromEvent('change', this.refs.input)
6      .pluck('target', 'value') // extract input text
7      .subscribe(this.update) // call update with value
8  }
9
10 componentWillUnmount() { this.observable.unsubscribe() }
11
12 update(text) { this.setState({name: text}) }
13
14 render = () => (
15   <div>
16     <h1>{this.props.greeting} {this.state.name}!</h1>
17     <input value={this.state.name} ref="input" />
18   </div>
19 )
```

Now, whenever a user changes the text in the `input` field, the RxJS Observable will receive a new value (a change event). The `value` of the field is extracted on line 6 and used to then update the state in the `update` method, which is implicitly called with the value passed through the Observable chain.

The two methods `componentWillMount`, `componentWillUnmount` as well as `componentDidMount`, `shouldComponentUpdate`, `componentWillUpdate`, `componentDidUpdate` and `componentWillReceiveProps` are *lifecycle methods*, called whenever the component is created, updated or destroyed.

As an end note on React, and a transition to the core presentation library, I want to add that React components can be nested, which is at the core of the project developed for the present thesis. To make it as easy as possible for other developers to use the created libraries, a presentation is built just as a usual HTML page, using JSX to reference the custom React components (see program 2.4).

Program 2.4: Nested React components. In this example, a 1-slide-long presentation is created.

```
<UnveilApp>
  <Slide name="start">
    <h1>Unveil</h1>
    <h2>a meta presentation</h2>
    <Notes>Greet people and welcome everyone</Notes>
  </Slide>
</UnveilApp>
```

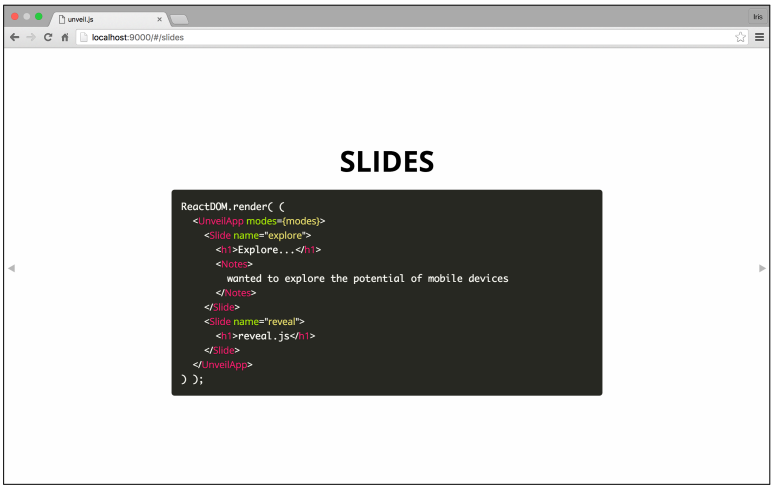


Figure 2.1: Screenshot of an example slide with unveil.js.

2.2.4 unveil.js[15]

As a presentation layer, this project uses the open-source JavaScript library *unveil.js*, which was developed by Leandro Ostera and myself in the beginning of the project and which I extended and adapted to my needs in an own fork[16] during the project. This fork will be covered in section 2.4 of this chapter, until then, a short overview of the different parts of unveil.js is given and key concepts of the library are discussed. Screenshots of what unveil.js looks like can be found in figure 2.1.

Generally, unveil.js operates on a 2-dimensional slide-space: Every slide can have a next and previous slide in x, as well as in y-direction. To generate the y axis, slides can be nested in other slides. These slides have an optional unique name as well as an index in the slide-tree, which they are identified by. As shown in program 2.4, slides are created inside the component

[15] https://github.com/ostera/unveil.js
[16] https://github.com/irisSchaffer/unveil.js

`UnveilApp`, the core of unveil.js. This component configures and sets up the entire application based on optional configuration passed in as properties. There are a few concepts unveil.js is built around to allow for extensibility and configurability, namely *presenters*, *controls* and *modes*:

- **Presenters** define the way slides are rendered, e.g. show notes, upcoming slides or hide them.
- **Controls** control a part of the application, e.g. navigating from one slide to the next using the arrow keys on the keyboard.
- **Modes** are what allows a speaker to have a different presenter and controls from an audience member. Each mode defines its own presenter and set of controls, the mode is determined by the url query parameter `mode`.

This allows anybody using unveil.js to define new modes, presenters and controls and thereby extend the base library as they wish. A few of these are already defined, namely a default `Presenter`, `UIControls` to navigate using buttons, `KeyControls` to navigate using the keyboard and `TouchControls` to navigate with swipe-gestures on touch screens. In section 2.8, modes for the audience (*default*), the speaker (*speaker*) and for use on the projection device (*projector*) will be introduced.

For these controls and the entire presentation to be navigatable, `UnveilApp` is responsible for the creation of two very important classes: `Router` and `Navigator`. These can be defined outside and passed into `UnveilApp` as properties, allowing users to customise their navigation logic. The `Router` is the class handling everything connected to the current url. It receives the slide-tree and can compute the indices of a slide by its name and vice versa. Whenever the browser history changes, the router finds the corresponding slide-indices, computes an array of possible directions to go into from this slide and propagates the event to `UnveilApp`, which can then re-render the application. `Navigator`, in turn, receives these directions and is responsible for the mapping of directions (*left*, *right*, *up*, *down*) to slide-indices. Controls know the navigator and can push new directions to the navigator subject, thus starting the navigation process described in detail in figure 2.2.

2.3 Project Structure

As the puropose of this project was not only to experiment with different ways of interacting with presentations using mobile devices, but also to create something worthwhile and contribute back to the vibrant open-source community, the project is enitrely open-source and separated into different repositories, which are all available on GitHub. These can be installed using npm, therefore allowing developers to rely only on the parts they really need.

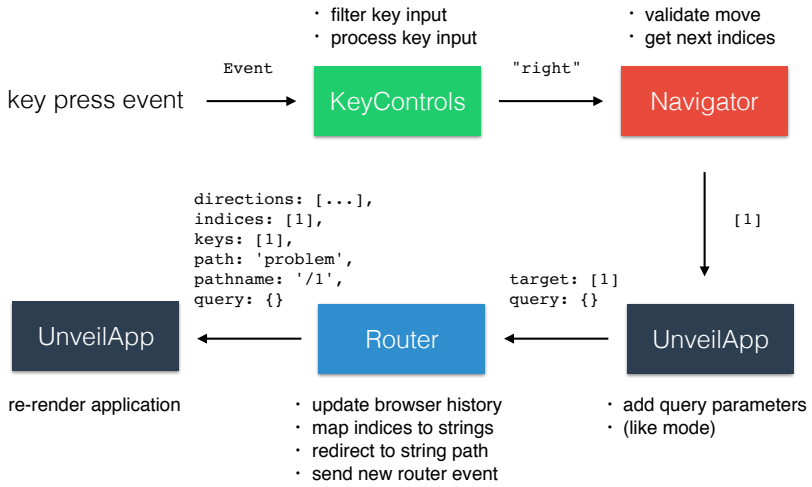


Figure 2.2: Navigation pipeline from user's key press to re-render of the presentation. The monospaced text next to the arrows symbolises the data transmitted. `KeyControls` listen for key events and process them, to then send a navigation request to go *right* to the `Navigator`. This component then maps the direction to the next slide's indices (1). `UnveilApp` then adds other information necessary for the `Router`, which then is responsible for updating the browser history, mapping the indices back to a human readable url and sending out a new router event. In the end `UnveilApp` receives this event and re-renders the presentation.

Extended unveil.js: As discussed in section 2.2.4, the project is based on the library unveil.js. During the development of the project, certain parts of the base library did not offer the flexibility needed for easy extensibility and so several parts were adapted and new presentation logic was added. This happened in a fork of the original library, which will be examined in section 2.4.

Network Synchronisation Layer: The first library of direct importance for the interaction between speaker and audience through personal devices is *unveil-network-sync*[17]. This rather small library relies on unveil.js and is responsible for connecting the client and the server through web sockets and enables the synchronisation of the current slide displayed between speaker, audience and projector. The implementation of the features will be discussed in detail in section 2.5.

Interactive Extension: As the name already suggests, this library is at the core of the present thesis: It includes a dedicated presenter for the

[17] https://github.com/irisSchaffer/unveil-network-sync

speaker, implements the insertion of additional slides and subslides and by that allows the audience to share content with the presentation. The voting mechanism, as well as the creation of new votings on-the-fly, also live within this library. The repository, called *unveil-interactive*[18] relies on unveil-network-sync for the socket-interaction. The interactive extension will be covered in section 2.6 of this chapter.

Server and Example Presentation: The last repository connected to this thesis is *unveil-client-server*[19], which includes a simple server as well as a real-world example of a presentation, which was used in the intermediate thesis project presentation as part of the Interactive Media course IM690, on the 2nd of February, 2016. In this chapter, a whole section was dedicated to the server (2.7), as well as to the example application (2.8), to separate concerns a bit more clearly and be able to conclude with a demonstration of how all parts discussed earlier play together in a final presentation.

2.4 Extended unveil.js

The biggest adaptions and additions were necessary in the main component `UnveilApp`. A state subject was added to allow all components in the presentation to interact with the application. This subject receives an event with type and data and depending on this type starts a certain state change. Two of these state events are the `state/navigation:enable` and `state/navigation:disable` events, which set a state-variable `navigatable` to true or false. This variable is used in the controls to determine navigatability, therefore making it possible to keep the audience locked to a slide, e.g. during votings. To make it possible for the audience to add subslides, as well as to dynamically add votings on-the-fly, another event is `state/slide:add`. It includes what slide to add (content), how (subslide or main slide) and where (under or after which slide). On occurence of this event, the slide-tree has to be re-built, the router and navigator re-started and the whole presentation re-rendered, which the library also had to be prepared for.

Another adaption in `UnveilApp` is the introduction of the `context` object: Additionally to state and properties, there is a third way of communicating between components in React, called *context*. Instead of having to pass properties from one nested component to the other, every child component can access the context of its parents. The navigator, needed in the controls, was formerly passed from `UnveilApp` to the controls through several layers. Using context, `UnveilApp` now defines a number of different variables which are available through context, including current slide and router state, navigator, mode and the state subject discussed in the last paragraph.

[18] https://github.com/irisSchaffer/unveil-interactive
[19] https://github.com/irisSchaffer/unveil-client-server

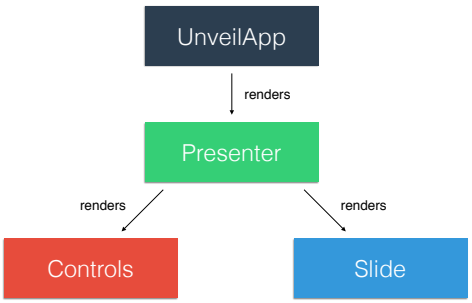


Figure 2.3: Overview over the render-flow of the application in the extended version of unveil.js. `UnveilApp` renders the presenter, which then takes care of rendering the (current) slide and all controls.

This makes it easy for new controls and presenters to access the data they need without other layers knowing about them or having to define them. This adaption was partly due to a change in the render hierarchy: Formerly, `UnveilApp` itself rendered the presenter (which rendered the current slide) and the controls. However, the presenter needs to be able to also control the rendering of controls (see figure 2.3), adding another layer between the rendering of controls and `UnveilApp`.

Another part that was added to the unveil.js base library is the `Notes` component. It allows adding speaker notes to each slide (see program 2.4). These, however, are not rendered by the slide, but by the presenter, as will be shown in section 2.6. One more important new feature is the possibility to configure the next slide in a certain direction (left/right/up/down) and therefore allow for jumping into different branches of the presentation, thus making a presentation even more interactive. The following code, for example

```
1  <Slide name="start" left={[0]}>
2    ...
3  </Slide>
```

means a navigation *left* (left arrow key pressed, swipe left etc.) will not go to the previous slide defined in the slide-tree, but rather jump to the first slide (of index 0).

2.5 Network Synchronisation Layer

As mentioned before, the network synchronisation layer is responsible for the communcation between server and client using web sockets. These are created with socket.io, a library which also provides fallbacks for browsers that do not support web sockets yet. However, this library also has a few drawbacks, especially when it comes to corporate networks. As Rob Britton

Program 2.5: Shortened version of `NavigationReceiver`. First the inherited context properties are set up, then an observable waiting for `state:change` events from the socket is created. If the incoming request is not the currently displayed slide, the navigator will be pushed a new value.

```
1  // imports...
2
3  export default class NavigationReceiver extends React.Component {
4    static contextTypes = {
5      navigator:   React.PropTypes.object.isRequired,
6      routerState: React.PropTypes.object.isRequired
7    }
8
9    componentDidMount () {
10     this.observable = Observable.fromEvent(SocketIO, 'state:change')
11       .filter((e) => !this.context.routerState.indices.equals(e))
12       .subscribe(this.props.navigator.next)
13   }
14
15   // ...
16 }
```

describes in [**socketio-problems**], socket.io seems to have problems getting through firewalls and can be blocked by some anti virus software. Because mobile browser support is essential for this project, I decided to still use this library.

The setup of the socket is simple: one helper function, called `createSocket` is called in the main entry point of the application to configure which server to connect to:

```
1 import { createSocket } from 'unveil-network-sync'
2 createSocket('46.101.166.172:9000')
```

This function creates the socket and returns it as a singleton, so every component uses the same connection. To make importing even easier, there is another helper, called `SocketIO` which dan be importet directly and internally calls `createSocket` to retrieve the singleton:

```
1 import { SocketIO } from 'unveil-network-sync'
```

These sockets can then be used to listen to events or to emit them (see program 2.5) Like the state subject events, the socket.io events used in this library follow the naming convention of scoping the object targeted in by the event separated by slashes, followed by a colon and the name of the action, e.g. `state:change` or `state/slide/voting:start`.

The second responisbilty of this library is synchronising the navigation state of the presentation between speaker and audience. To do this, two controls, `NavigationSender` and `NavigationReceiver` were implemented. As the names already say, the sender broadcasts the state update, while

the receiver is waiting for state updates and starts the navigation process. The latter is used in all modes (default, speaker and projector), whereas the sender is only added to the speaker mode. To make sure the sender does not end up in an infinite loop of sending and receiving its own state changes, the last received state is stored and only navigation events going to a different slide are processed further.

This mechanism, though relatively simple, already enables the audience to follow the presentation, the speaker to use his/her phone as a remote control and any number of projectors to be controlled by the speaker.

2.6 Interactive Extension

The interactive library includes several parts which will be discussed here: a speaker presenter (section 2.6.1) as well as different components connected to sharing media (section 2.6.2) and voting (section 2.6.3). Additionally controls handling the `state:initial` event were implemented to redirect new listeners to the current slide in the presentation.

2.6.1 Speaker Presenter

The speaker presenter, like the normal presenter, is responsible for rendering controls and slides. In speaker mode, where this presenter is used, notes as well as the upcoming slides (right and down) are also shown (see figure 2.4). This means the presenter has to find these next slides, using the router's information about the navigatable directions, and render them in a designated area. To make the presenter view usable on mobile devices, special attention was paid to mobile stylesheets (see figure 2.5 (b)). This ensures that everything is big enough to be readable and all buttons are clickable.

2.6.2 Media

Another responsibility of the interactive extension is the possibility for audience members to share content with the presentation. For this to work, three different controls were created: `MediaSender`, `MediaReceiver` and `MediaAcceptor`. The sender is used in the default mode so users can share their content (see figure 2.6 (a)), the acceptor is enabled in speaker mode, to accept or reject incoming media (see figure 2.6 (b)) and the receiver in the end handles the creation of a new slide if the content was accepted and is therefore necessary in all modes. As incoming content requests could disrupt the presentation flow and distract the speaker, an option to mute the requests was built into the application. If the *do not disturb* mode is turned on, slides will silently be added as subslides, without causing the acceptor modal to open. This way the audience' additions can be re-visited after the end of the presentation. Generally, this feature can be used to either post a

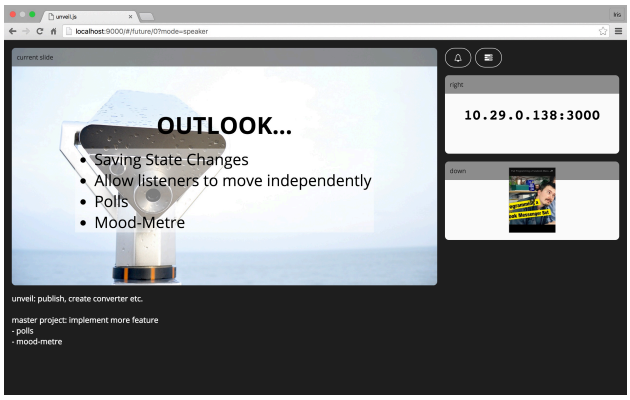
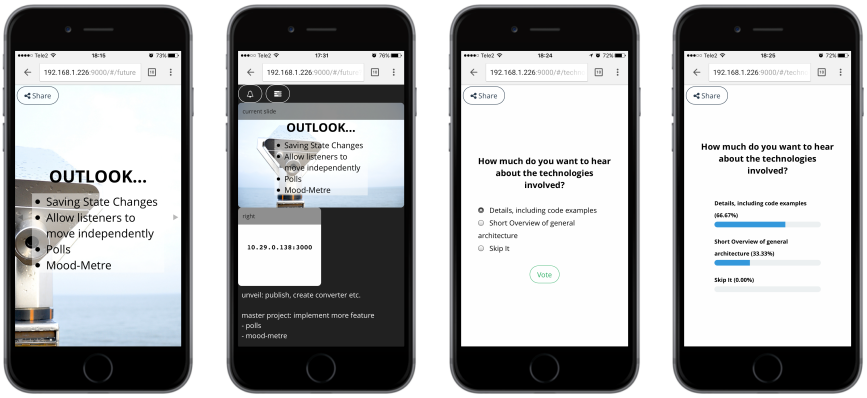


Figure 2.4: Screenshot of the speaker presenter with the current main slide, the upcoming slide to the right, available actions (muting and adding votings) and speaker notes.

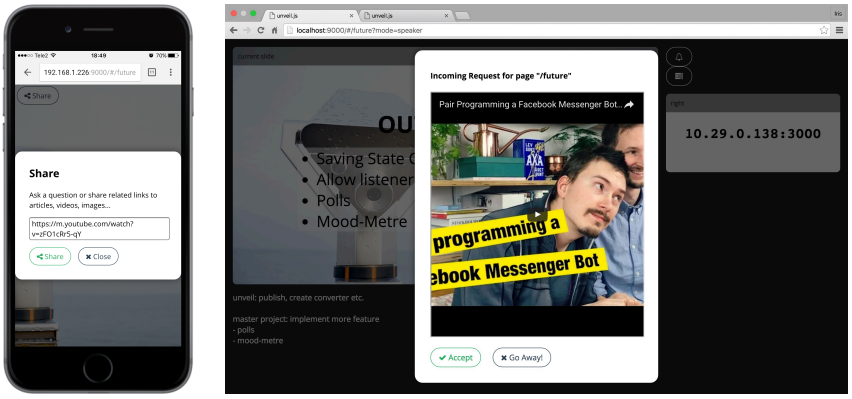


(a) (b) (c) (d)

Figure 2.5: Mobile view of a presentation slide in (a) default mode and (b) speaker mode as well as (c) voting view before voting and (d) voting view after voting. Mind the share button in the left upper corner in (a) as well as the buttons to mute content requests and create votings in (b).

link to an interesting picture, website or even youtube video, or also as text input, for example to add a comment or a question regarding a certain slide. This works through the introduction of the presentation components `Media` and `IFrame`, which, depending on the shared content, render an image-tag, blockquote or IFrame. The differentiation of these is carried out with regular expressions, as the following to check if the content is the link to an image:

```
1 isImg (str) {
2   let imgRegex = new RegExp(/\.(jpe?g|png|gif|bmp)$/i)
3   return imgRegex.test(str)
4 }
```



(a) (b)

Figure 2.6: Screenshots of sharing content. (a) media sender on mobile phone shares youtube link (b) speaker receives content request.

As far as the implementation of the controls is concerned, these ones are the first ones discussed in the present thesis makeing use of the `render()` method. It is used to output the *share* button in the left upper corner in default mode (see figure 2.5 (a)) as well as the share modal opened when clicking on said button (see figure 2.6 (a)). The same happens in the `MediaAcceptor`, which uses the `render()` method to display the *mute* button shown in figure 2.4 as well as the modal for accepting media (see figure 2.6 (b)).

These are also the first controls using state: in the sender a click on the share button sets the state variable `sharingMode` to `true` and through that enables the rendering of the modal. In the acceptor an array of `requests` is filled as new content is shared and emptied again, as the speaker accepts or denies them.

On event level, the socket events `state/slide/add:accept` and `state/slide:add` are included in the process of sharing new content, in the end an unveil state event of type `state/slide:add` lets `UnveilApp` create the new slide and add it to the slide tree. The whole flow is outlined in details in figure 2.7.

2.6.3 Voting

The last group of components connected to the interactive extension covered in this chapter allow speakers to create votings (both during in the preparation of the presentation and on-the-fly as shown in figure 2.8) and members of the audience to vote (see figure 2.5 (c) and (d)).

The main presentational component involved in the voting process is `Voting`, which keeps track of the current voting scores and has a `Question`

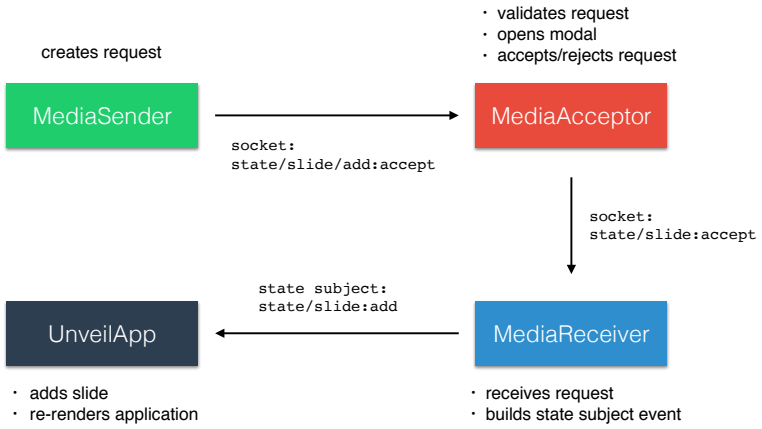


Figure 2.7: Flow of adding content, monospaced text symbolises type and name of events. First the `MediaSender` of the default mode sends a request, which the speaker mode's `MediaAcceptor` listens to. If the request is accepted by the speaker or requests are muted, another socket event is broadcast, which the `MediaReceiver` waits for. This component is enabled in all modes and emits the state subject event to add a new slide, which `UnveilApp` reacts to.

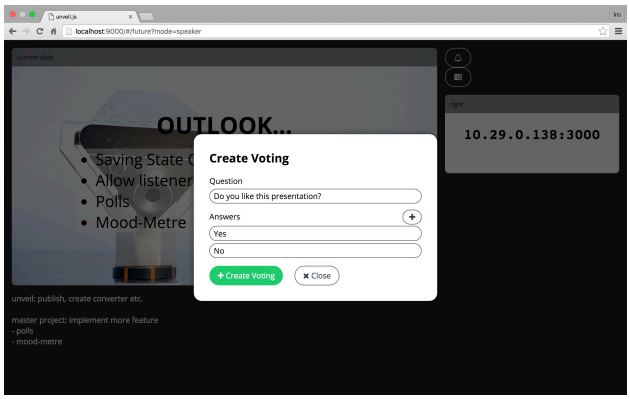


Figure 2.8: Screenshot of the desktop speaker mode, creating a new voting, on-the-fly, during a presentation.

and a number of `Answer` components as children (see program 2.6). `Voting` also remembers if an audience member has already voted and if so, displays `Result` components. In speaker and presenter mode only these results are shown.

The audience can start voting as soon as the speaker navigates to the slide with the voting, until then the vote button is disabled. Once the voting has started, the possibility for all audience members to navigate to a different

Program 2.6: Example code for preparing a slide with voting.

```
<Voting name="like">
  <Question>Do you like these slides?</Question>
  <Answer value="yes">Yes</Answer>
  <Answer value="no">No</Answer>
</Voting>
```

slide is disabled. This happens in the `VotingNavigatableSetter`, which is only installed for default mode. The voting start event, as well as the voting end event are broadcast by the `VotingController`, which checks the speaker's current slide for the occurence of a `Voting`.

Once the voting has started, internally, the `Voting` component remembers which answer the user has clicked in the `answer` state variable. Once the submit button is pressed, a `state/slide/voting:answer` event is fired and broadcast to all clients, which update their internal voting results. Because the results of the voting should be available throughout the whole presentation and not be reset when leaving the slide, `Voting` also handles the communication with local storage, to store current results.

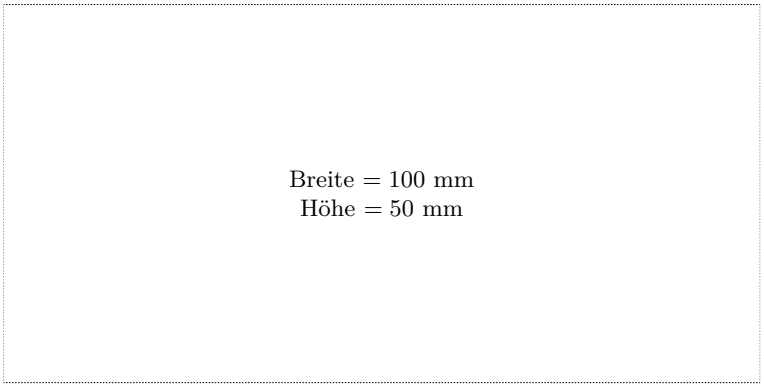
2.7 Server

2.8 Example Application

References

Messbox zur Druckkontrolle

— Druckgröße kontrollieren! —



— Diese Seite nach dem Druck entfernen! —