Hasso Plattner Institut

Digital Engineering · Universität Potsdam

Bachelor's Thesis

# Using Embedded React to enable complex yet performant web applications

**Mithilfe von Embedded React komplexe und dennoch performante Webanwendungen möglich machen**

by

**Nico Knoll**

Potsdam, June 2017

**Supervisor**

Prof. Dr. Christoph Meinel,
Jan Renz

**Internet-Technologies and Systems Group**

## Disclaimer

I certify that the material contained in this dissertation is my own work and does not contain significant portions of unreferenced or unacknowledged material. I also warrant that the above statement applies to the implementation of the project and all associated documentation.

Hiermit versichere ich, dass diese Arbeit selbständig verfasst wurde und dass keine anderen Quellen und Hilfsmittel als die angegebenen benutzt wurden. Diese Aussage trifft auch für alle Implementierungen und Dokumentationen im Rahmen dieses Projektes zu.

Potsdam, June 28, 2017

(Nico Knoll)

## Kurzfassung

Durch das steigende Bedürfnis nach Software zur Ausführung vielschichtiger Aufgaben wird diese zunehmend komplexer. In den letzten Jahren wurden viele Frameworks und Bibliotheken vorgestellt, die die Implementierung solcher Software vereinfachen sollen. Diese wird jedoch häufig an den falschen Stellen verwendet und machen dadurch die Anwendungen weniger leistungsstark und zuverlässig. In dieser Arbeit werden die viel benutzen Frameworks / Bibliotheken Express und React bezüglich ihrer Komplexität und Performance evaluiert. Am Ende der Arbeit wird "Embedded React" als ein Weg präsentiert, der die Vorteile beider Technologien vereint.

**Schlüsselwörte**: Express, React, Javascript, Komplexität, Performance, Embedded React

## Abstract

With a growing need of software that solves complex tasks, elaborateness and size of software keeps increasing. Many libraries and frameworks have been produced in recent years to ease the implementation but often they are used in the wrong places which makes applications less perfomant and less reliable. In this paper the two commonly used libraries/frameworks Express and React get evaluated based on complexity and performance. In conclusion "Embedded React" is presented as a way to combine the advantages of both technologies.

**Keywords**: Express, React, Javascript, complexity, performance, embedded React

# Contents

# 1. Introduction

## 1.1. Motivation

With more and more real life tasks being brought to the digital world the complexity and size of applicable software keeps increasing. In the same time more and more Libraries and Frameworks are created to cope with this. Those Libraries and Frameworks are originally often specialized on solving just one task but being abused as a one-fits-all solution. Therefore simple tasks often become overly complex or complex tasks end up in far too complicated and inperformant solutions.

Alan Kay, an American computer scientist and Turing Award winner, said: "Simple things should be simple; complex things should be possible." (Kay, 1982) Aiming to materialize these words in the context of programming a client application for the "Schul-Cloud" (see 1.2) different Libraries and Frameworks were evaluated in this paper. In that process two clients were produced with the same scope of features but using two completely different technologies: Express and React.

To decide which one to use, the two clients were assessed by performance (4.2) and complexity (4.1). This paper describes the evaluation process and it's results (4), a solution to enable complex tasks with Express (5) and alternative methods to fasten up React (2).

## 1.2. Project

This paper is written based on research that was performed during the "Schul-Cloud" ("school cloud") project. Intention of the project was to build a prototype solution of a cloud platform for all schools in Germany as schools have been left out by digitalization in most places so far (Meinel et al., 2017).

The "Schul-Cloud" follows a modular approach: Each of its features is developed as a microservice that can be called through the API Server (3.5). The API Server runs completely separated from the client application which allowed me to test the different technologies without adjusting the API.

The bachelor project "Schul-Cloud" is executed in collaboration with the Federal Ministry of Education and Research (BMBF) and the school excellence network "MINT-EC". Bachelor projects are meant to prepare students for real-life challenges in constructive software companies.

# 2. Related Work

## 2.1. Server Side Rendering (SSR)

As Malek Hakim describes in his paper *"Speed index and critical path rendering performance for isomorphic single page applications"* (Hakim, 2016) one approach to cope with the long time before the content being displayed when using React - which is after the browser loaded and evaluated the bundle [1] - is server side rendering: The React components of the requested page are loaded and rendered already on the server and delivered as static HTML. The JavaScript bundle is loaded in the background and as soon as it is initialized replaces the static DOM with React's virtual one.

To use this technique the React application has to be isomorphic - all functions have to work on the server side as well as on the client side. This is especially complicated in the case of routing as all virtual routes that are used in the React application have to work on the server as well so that the correct static HTML can be rendered. There are some React routing modules that try to achieve this and allow to define isomorphic routes like Flow Router [2] does for the Meteor Framework [3]. This feature is still experimental though. [4]

## 2.2. Automatic code splitting

Rasmus Eneman researched different ways to improve the load time of "Single Page Applications" (SPAs) (Eneman, 2016) for his bachelor thesis. Besides SSR (2.1) - which got discussed before - he looked into automatic code splitting with webpack (3.3). With webpack it is possible to split the bundle into smaller context-sensitive bundle chunks that get only loaded if needed. This works by looking at the import statements in the components and render a chunk e.g. for each component.

[1] A JavaScript file containing all of the applications code.
[2] `https://github.com/kadirahq/flow-router/`, accessed on 10 June 17
[3] `https://www.meteor.com/`, accessed on 10 June 17
[4] `https://guide.meteor.com/routing.html#server-side`, accessed on 10 June 17

Using this approach the page can load faster as less code is needed to be loaded initially. A problem it still has is that if modules are required at multiple places they get bundled into multiple files and produce unnecessary code repetitions when loaded. This can be improved by e.g. using a commons file that includes the most common modules. Figuring out which ones to include here and setting up the code splitting is still a heavy task and requires a lot of time and expertise.

# 3. Technologies

## 3.1. Node.js

Node.js is built on Chrome's V8 JavaScript engine and allows developer to run JavaScript code server side. It follows a modular approach and offers developers *"the largest ecosystem of open source libraries in the world."* [5] through their package ecosystem "npm" [6]. This makes it easy to create huge web applications in a short time.

## 3.2. Babel

Babel [7] is a JavaScript compiler/transpiler. It can be used to compile JavaScript dialects such as Flow or JSX into JavaScript code. It can also be used to convert ES6 code into ES5 code to make it run on browsers that don't support ES6 so far.

## 3.3. Webpack

Webpack [8] is a JavaScript module bundler. Its main task is to bundle JavaScript applications into bundle files that include all module dependencies and can run in the browser. It can also be used to load and move assets such as images and stylesheets.

## 3.4. Gulp

Gulp [9] is a task runner built on Node.js. It mainly gets used as a build system in front-end development. Following a modular approach functionalities such as minifying or transpiling can be added to Gulp using Node.js modules.

[5] `https://nodejs.org/en/about/`, accessed on 8 April 2017
[6] `https://www.npmjs.com/`, accessed on 8 April 2017
[7] `https://babeljs.io/`, accessed on 8 April 2017
[8] `https://webpack.js.org/`, accessed on 8 April 2017
[9] `http://gulpjs.com/`, accessed on 8 April 2017

## 3.5. API Server

In the following API Server references the Schul-Cloud Server [10]. The Schul-Cloud Server is based on FeathersJS [11] and does implement a REST-like interface through various microservices. It is connected to a mongo database and stores models and relations. It also works as proxy to external microservices (the full list can be found at `https://github.com/schulcloud/`).

Communication with the API Server can be done using JSON requests. It offers an abstraction of logic so client applications can be compared in the following evaluation (4).

## 3.6. Express

Express [12] is a minimal Node.js web application framework. It runs server side and offers routing and support for several template engines among other features. It is easily extendable through npm modules (see 3.1).

## 3.7. React

React [13] is a JavaScript library for building user interfaces. It runs client side and allows developers to easily develop state-driven applications. An application can contain several components that can be used multiple times in multiple places.

[10] `https://github.com/schulcloud/schulcloud-server`, accessed on 9 April 2017
[11] `https://feathersjs.com/`, accessed on 9 April 2017
[12] `http://expressjs.com/`, accessed on 9 April 2017
[13] `https://facebook.github.io/react/`, accessed on 9 April 2017

# 4. Evaluation of ExpressJS and React as Clients

As mentioned before (see 1.1) in the process of testing Libraries and Frameworks two clients were produced with the same scope of features. One using Express, the other using React.

There are several differences between those two technologies. The main disparity is that Express runs server side and delivers only the results to the client, React however runs completely client side. Both technologies have advantages and disadvantages in executing tasks.

In this chapter a closer look on those differences is taken and the proper use case of both technologies evaluated.

## 4.1. Complexity

Complexity is hard to define. In this context the complexity of the client describes how easy it is to understand it's code, it's dependencies and how the client works. This is an important factor as the Schul-Cloud is an open source project and therefore has to be easy to be setup and be understand to get new developers involved.

To obtain unbiased opinions on that topic a questionnaire was evolved interrogating other developers about how easy it is for them to get started with React and Express, to create a new application from scratch with those technologies, what they like about them, where they see the limitations and what they would use them for. The questionnaire's results are integrated into the following evaluation.

### 4.1.1. Setup

For setting up an Express application only a Node.js server is required. Express can be installed as a module using npm:

```
npm install express
```

Code 1: Setting up Express

A minimal sample Express application looks like this:

```
const express = require('express');
const app = express();

app.get('/', function (req, res) {
  res.send('Hello World!');
});

app.listen(3000, function () {
  console.log('Example app listening on port 3000!');
});
```

Code 2: Taken from `https://expressjs.com/` (4 June 2017)

Setting up a React application has some more requirements. As it has to run client side it is recommended [14] to use a module bundler like webpack. The output of the module bundler is a JavaScript file that has to be delivered - in addition to an HTML output that includes the JavaScript file and offers an element that can contain the react virtual DOM e.g. a `<div>` tag with the ID `root` - by a server. Therefore a server is required as well. A sample React application using a module bundler looks like this:

[14] `https://facebook.github.io/react/docs/installation.html#hello-world-with-es6-and-jsx`, accessed on 4 June 2017

```
import React from 'react';
import ReactDOM from 'react-dom';

ReactDOM.render(
  <h1>Hello, world!</h1>,
  document.getElementById('root')
);
```

Code 3: Taken from `https://facebook.github.io/` (4 June 2017)

It is easy to recognize that the Node.js modules `react` and `react-dom` are required as well.

React can also be used without a module bundler (see section 5).

### 4.1.2. Request Handling / Routing

As Express runs server side routing is done server side as well.

Routing with Express begins as soon as a new request is received. The request gets routed through all routes/middleware that have been setup. Each route can modify the response until either one route calls `res.send()` to deliver a response to the user or all routes are processed.

```
app.get('/hello-world/', function (req, res) {
  res.send('hello world');
});
```

Code 4: Routing with Express

React doesn't offer native routing. There are different packages that can be used. For comparability to Express react-router was used, which is the most popular react routing solution [15].

[15]It has more then 22.000 stars on GitHub: `https://github.com/ReactTraining/react-router`

```
<Route path="/hello-world/" component={HelloWorldView} />
```

Code 5: Routing with React-Router

As a React application gets loaded once and does virtual routing, different routes for e.g. POST and GET requests cannot be defined.

### 4.1.3. Loading and Delivering Data

There are several technologies that can be used to connect to a server to receive data such as HTTP requests and websockets. In the case of the API Server only HTTP requests are allowed by offering a REST-like interface to the microservices.

Using Express the user's browser sends a request to load a different page or resource by e.g. clicking a link, browsing another URL or via AJAX to the Express application. The Express application needs to load the requested data from the API Server and therefore sends another HTTP request to the API Server. It has to wait for the API Server's response before it is able to send the result to the user and serve the request.

Express supports different view engines such as Pug (formerly known as "Jade") or Handlebars. Those are used to render the received data into valid HTML which gets delivered to the client as the result of the request

It is not possible for Express to change parts of the page after it got delivered to the user. This is because Express works on a per-request basis (see 4.1.2). The only way to change the page's HTML code natively from Express is by reloading the current page or loading another page.

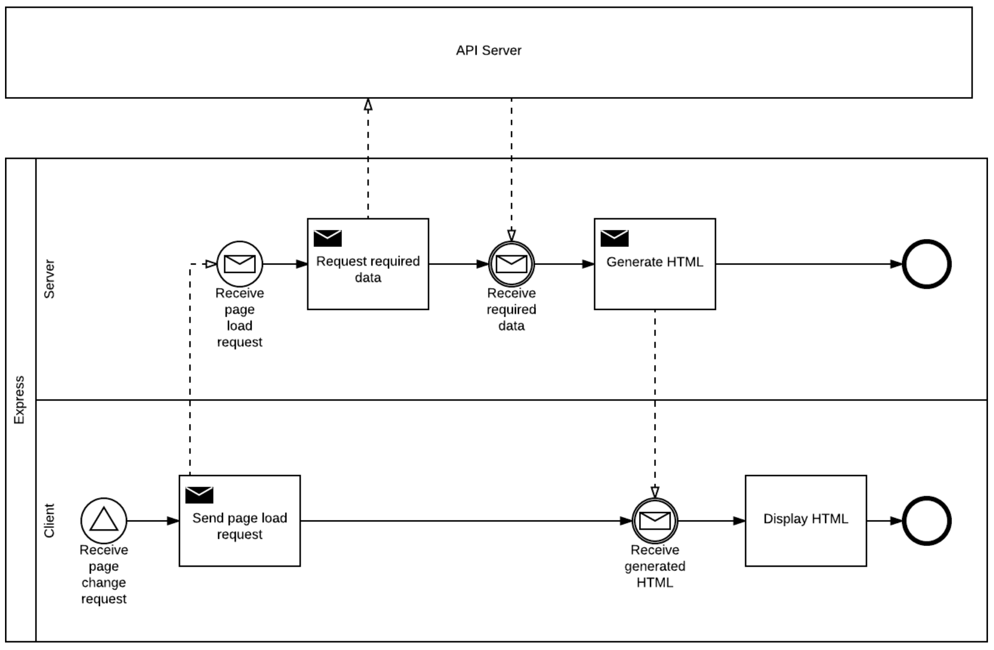


Figure 1: Loading data from the API Server with Express

For React there are different ways of loading data or handling incoming data: Using a composer (e.g. `react-komposer`) it is possible to defer displaying the data until all of it is loaded. This mimics the same behaviour server-side frameworks such as Express or Sails.js have.

This behaviour makes sense if the React component deeply relies on all data being loaded before it can be displayed or if the developer doesn't want the page to have blocks of content popping up after the page got displayed.

The other option is to display the data as soon as it comes in. This is possible because of the fact that React is running client side and can change it's virtual DOM as often as it wants. It is not limited to answer to a request by the client to display new data.

This is a behaviour that is known from interactive components such as chats or live tickers where it makes sense to show the most recent information as soon as possible without forcing the user to reload the page.

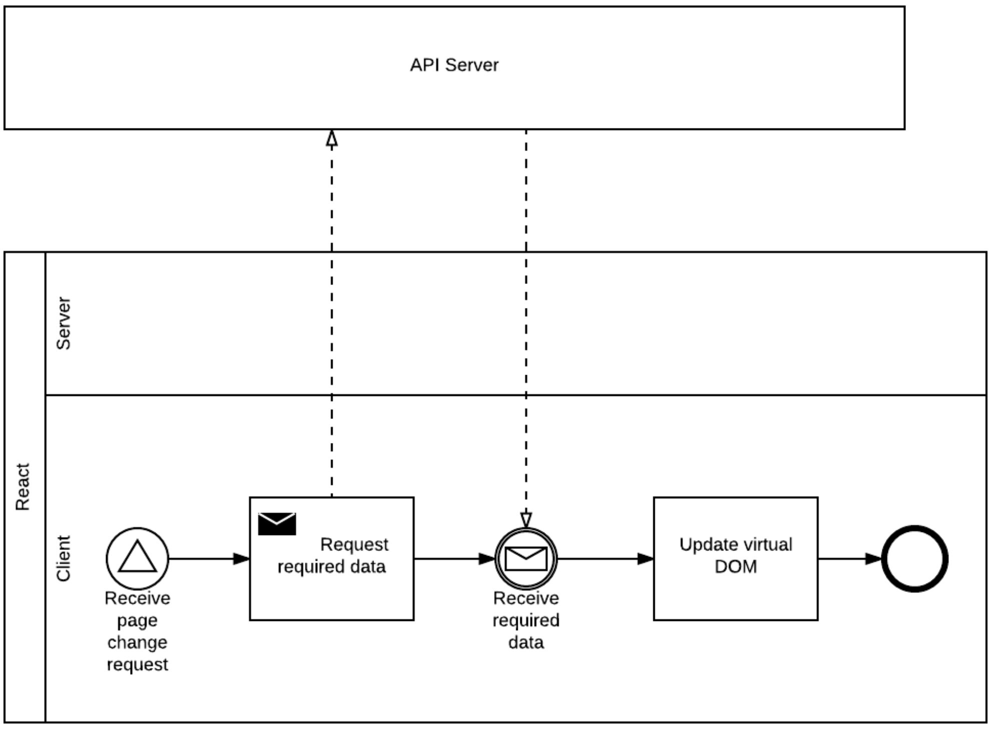


Figure 2: Loading data from the API Server with React

### 4.1.4. Error Handling

Error handling has to be a key aspect in writing any software. Errors shouldn't crash the application but also must not be ignored as the following functions that depend on the correct result may would work on wrong data afterwards.

There are different kinds of errors that may happen within running an web application. In the following a closer look at non-breaking errors such as HTTP-Errors and breaking errors such as "JavaScript Runtime Errors" is taken.

A good example for a HTTP-Errors is the "Error 404: Not found." which is called when a user tries to open a path that is not available.

In Express non-breaking errors get handled by an error middleware. Regular middleware functions take three objects as arguments: the request object, the response object and a callback function usually called `next()`. In the case of the error handling middleware there is an additional error object argument. That object gets set when a middleware calls `next(err)` with an error object as only argument. A sample error handling middleware looks like this:

```
app.use(function (err, req, res, next) {
  if (res.headersSent) {
    return next(err);
  }

  const status = err.status || err.statusCode;
  res.status(status);
  res.render('error', { error: err });
});
```

Code 6: Express error handling middleware

For triggering a 404 error a wildcard route is added at the end of the router configuration. As the request object gets passed through all routes and middlewares in the order they were registered in the router, the wildcard route gets the request only at the end if no other route serves it before by sending the response to the client. In the wildcard route a new error object is created and send to the error handler by calling `next()` with it:

```
app.use(function (req, res, next) {
  const err = new Error('Not Found');
  err.status = 404;
  next(err);
});
```

Code 7: Triggering a 404 error in Express

Breaking errors such as "Uncaught TypeError: Cannot read property '...' of undefined" which often appear if an API Server sends an unexpected response and the following routines try to work on it can crash the node process the Express server runs on. This can be prevented by adding data validation before trying to access the received data.

In addition by using a demon such as `nodeman` or `forever` the Node.js server gets restarted automatically and is able to handle new responses again even after the process crashed. This is important as there is only one application that gets accessed by all clients.

React doesn't offer any default implementation of an error handler for non-breaking errors.

And while React doesn't have a native router as mentioned before `react-router` allows handling 404 errors almost the same way Express does. At the end of the router configuration it is possible to add a wildcard route that displays e.g. an error page:

```
<Router history={browserHistory}>
  <Route path="/" component={App}>
    {/* Your other routes here */}
    <Route path="*" component={NotFound} />
  </Route>
</Router>
```

Code 8: "Page not found" error handling with `react-router`

For React running client side the application crashes on an breaking error and cannot be restarted like the server side application but the user has to reinitialize the application by e.g. reloading the request page.

There are different ideas and modules that try to prevent crashing of the react application in case of breaking errors. But they all share the same approach: The React render function gets wrapped into a `try-catch` construct that prevents any runtime error from being thrown and therefore crashing the application. In the catch part the error then can be handled appropriately.

### 4.1.5. Application Architecture

In the case of Express the traditional split of models, views and controllers that is suggested by Express [16] was performed whereas models aren't stored in the client but directly on the API server. `controllers` mainly includes the routing logic.

Additionally the following directories and files exist: `test` (which contains frontend tests), `static` (which contains static assets such as images and stylesheets), `helpers` (which contains various helpers that get included into multiple controllers) and `api.js` (which contains the API server adapter).

Therefore the directory tree looks like this:

```
/
├── api.js
├── app.js
├── bin
│   └── www
├── controllers
├── gulpfile.js
├── helpers
├── node_modules
├── package.json
├── static
│   ├── fonts
│   ├── images
│   ├── scripts
│   ├── styles
│   └── vendor
├── test
└── views
```

Figure 3: Directory tree of the Express client

For React the "Mantra application architecture" [17] - that currently is a "Working Draft" written by Kadira Inc. - was applied.

[16] `http://expressjs.com/en/starter/generator.html`, accessed on 7 June 2017
[17] `https://kadirahq.github.io/mantra/`, accessed on 7 June 2017

The main goals of Mantra are to make React applications maintainable and future proof. This is done by forcing the developer to split the entire application into modules. A module usually contains `actions` (which contain business logic such as validations and state management), `components` (which only display data, only manage their own state and can be used in multiple containers) and `containers` (which form the integration layer and are responsible for loading data from the API Server, composing components and provide them with the loaded data).

A regular mantra directory tree looks like this:

```
/
├─ src
│  ├─ app.js
│  ├─ configs
│  ├─ modules
│  │  ├─ courses
│  │  │  ├─ actions
│  │  │  ├─ components
│  │  │  ├─ containers
│  │  │  ├─ index.js
│  │  │  ├─ routes.jsx
│  │  │  └─ styles
│  │  └─ ...
│  └─ static
│     ├─ fonts
│     └─ images
├─ README.md
├─ package.json
├─ postcss.config.js
├─ server.js
└─ webpack.config.js
```

Figure 4: Directory tree of the React client

## 4.2. Performance

Loading times are a major aspect in the evaluation of the clients as user satisfaction as well as business success - e.g. in the case of Amazon (Eaton, 2012) - is directly connected to it. Research by Fiona Nah has shown that the average user is willing to wait two seconds at most before leaving the page. (Nah, 2003)

There are several factors that influence how long a website needs to load including the speed of the users internet connection or the number and size of assets (JavaScript files, stylesheets, images, ...). While the first factor can't be improved programmatically, the number of requests the browser has to make, the size of each requested file and the cachability of those files can be optimized.

### 4.2.1. Method Description

Based on the research by Rasmus Eneman (Eneman, 2016) the indicators used to evaluate the perceived performance of the application are the "Time To First Paint", "Time To Fully Loaded" and navigation time.

The "Time To First Paint" describes the time that the browser needs before first elements can be shown. Eneman used the "Time To First Meaningful Paint" instead but the results have shown that for the evaluated applications those two times are approximately the same with a possible deviation of a few milliseconds.

The experiment was executed using Google Chrome (Version 58.0.3029.110) on an Apple MacBook Pro (Early 2015, 16 GB RAM, 2,7 GHz Intel Core i5) running macOS 10.12.3. For both of the tested applications the experiments were run once with caching enabled and once with caching disabled.

The measurements were taken from Chrome's implementation of the "Navigation Timing API" [18]. The code used to receive the results and repeat each test 100 times is based on sample code by Sara Gonçalves [19] and can be found in the Appendix A as well as the raw measurements.

[18] `https://w3c.github.io/navigation-timing/`, accessed on 12 June 2017

[19] `https://medium.com/outsystems-engineering/measuring-web-app-runtime-performance-dfd8a6931418`, 10 June 2017

### 4.2.2. Interpreting the results

Using Express most of the work is done on the server and only the rendered HTML gets delivered in the response. The browser doesn't have to wait until all assets have been loaded before displaying the HTML. This behaviour can result in a "FOUC" [20] as CSS files may not be loaded fast enough. As soon as they are available the browser repaints the view with the styling rules defined in the CSS code. Therefore the Time To First Paint is really small.

Using a naive approach to run React client side all of it dependencies get bundled into one bundle file. As it includes all dependencies the bundle file usually gets really big. The browser cannot paint any frame until loading, parsing and evaluating is done which can take the browser long on initial load. Caching the bundled file improves this drastically as shown in Figure 6.

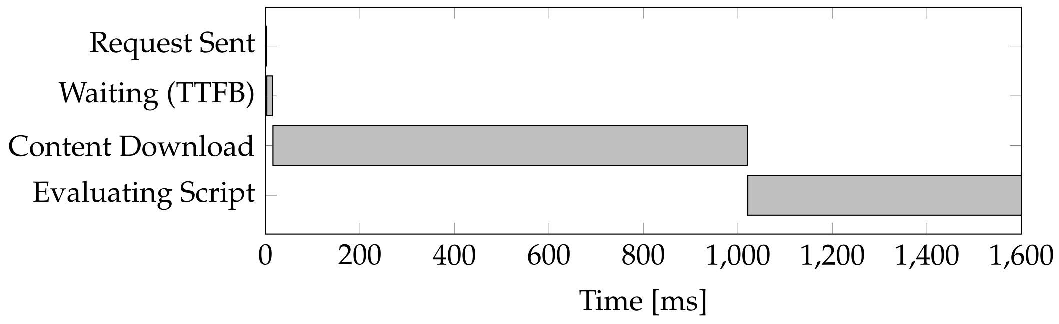


Figure 5: Example of parsing a bundled file from Schul-Cloud React Client, 5mb

In addition React can only start to load variable data from the API Server as soon as the entire loading and evaluating phase of the bundled application is done and by using `react-komposer` to prevent the components from being shown before their data is loaded this can result into an even longer Time To First Paint.

[20]"Flash of Unstyled Content"

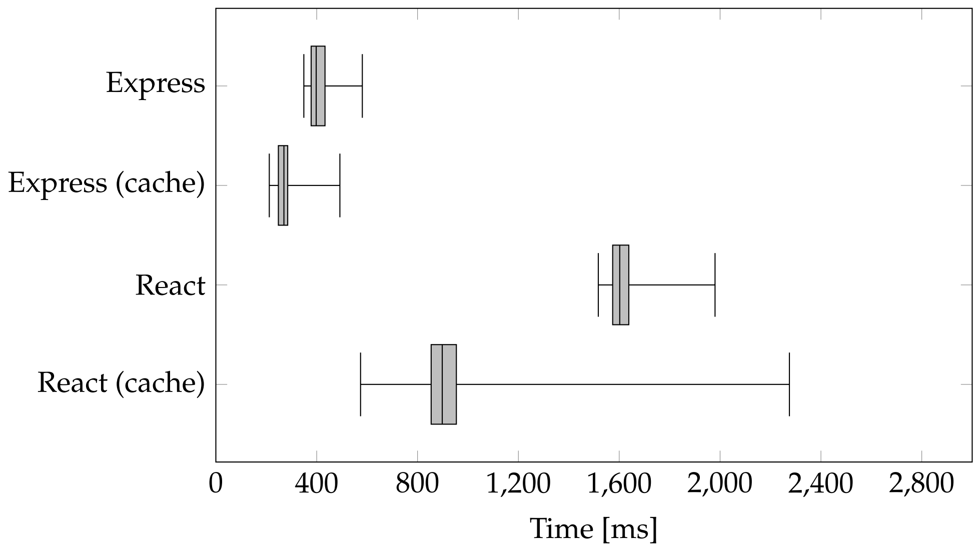


Figure 6: Time To First Paint

As with Express the browser can paint the received data as soon as it gets it the Time To First Paint is always smaller than the time to fully loaded.

With React it is the opposite. The browser has to first load the entire bundle and evaluate it before the first paint can happen. Therefore the Time To First Paint is always bigger than the Time To Fully Loaded.

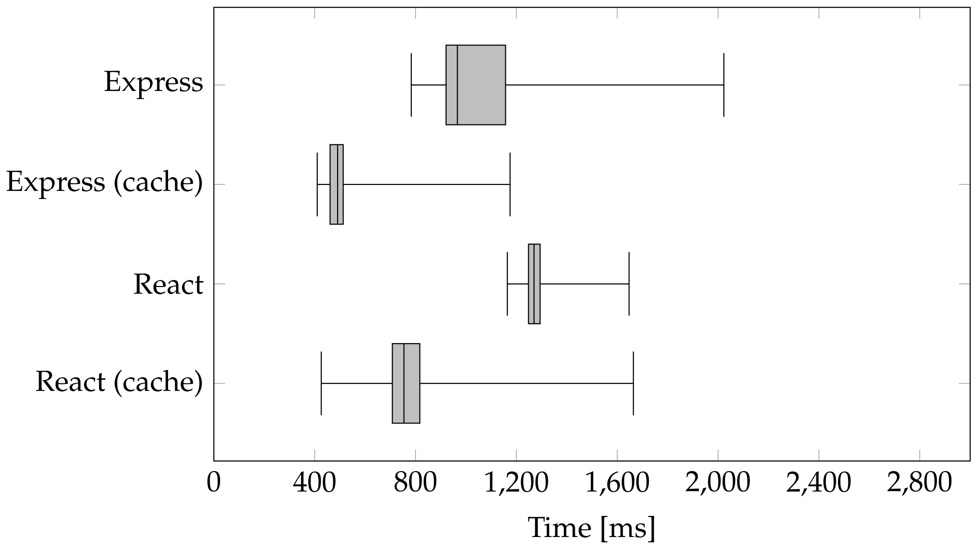


Figure 7: Time To Fully Loaded

While React only loads the one bundled JavaScript file, Express sends multiple smaller JavaScript files to the browser. An advantage of loading multiple smaller files is that each file can get cached by the browser individually. Changes in the code only force the changed files to be reloaded and cached again.

**Navigating on the site**

With Express clicking a link in the client sends a new request to the Express server which results in a new server response and a reload of the page. As most of the assets are cached already only newly referenced files get loaded from the server and the performance is the same as shown in the figures above under "Express (cache)". The advantage of React here is that once the application is loaded the entire website is loaded. It won't send additional request to the server to load a new view which makes page changes instant. As `react-komposer` is used to delay showing new components until the entire data is loaded from the API Server the navigation speed only depends on the speed of the API Server and the user's internet connection.

## 4.3. Evaluation Results

### 4.3.1. Express

Regarding loading and processing times the main advantage Express has over React is that it runs server side. Therefore calculations and loading times don't depend on the client's device but on the server's infrastructure. The client only receives the result of the server's calculations instead of the entire application which enables smaller loading times as the application is already running on the server and doesn't have to be loaded in the browser.

Unbundled resources allow the client to use efficient caching techniques which improves the Time To Fully Loaded as well. Common tasks like routing are easy and intuitively doable.

That all users share the same application on the server can be a disadvantage. The server has to generate the entire output and deliver it each time to each client. That the server can only response to requests from the client and the client needs to load a new page to make a request makes it hard to use it for interactive tools or applications that need to load and display a lot of additional or changing data.

### 4.3.2. React

While loading a react application requires the loading and evaluating of the entire application in the client which results in a long Time To Fully Loaded and even longer Time To First Paint it has some advantages over Express. Especially if it comes to interactive application that need to load data dynamically or react on data changes without reloading the page.

As React is state-driver and works on a virtual DOM it allows websites to change the DOM easily from the inside by changing the state instead of changing the real DOM from the outside with Libraries such as jQuery or plain JavaScript.

As mentioned above React applications require the client to be able to run JavaScript and to initialize the entire application before the DOM can be displayed. This can be problematic for e.g. older devices or crawlers that only take the real DOM which is almost empty in the case of React applications. As React is only a JavaScript Library for building user interfaces it doesn't come with native Routing or other features you get from Frameworks like Express.

# 5. Embedded React

## 5.1. Concept

In its core "Embedded React" acts and gets used like an external widget e.g. the <video> or <iframe> tag or an flash object:

```html
<div id="embedded-widget"></div>
```

Code 9: Embedded React sample widget

But as it is a native HTML tag it comes with complete browser support. To populate the Embedded React widget the standalone `react` and `react-dom` libraries are needed. Those can be used without any further dependencies. [21]

The page including the React libraries and the `<script>` tag gets rendered on the server and delivered to the client. The client renders the received DOM and paints the page. While parsing the received HTML it adds the initialization of the widget to the initialization queue. The initialization process replaces the widget in the DOM with an virtual DOM that's state is managed by React.

As a result the user sees the static page contents first really fast and after initialization the embedded React component instead of waiting for the browser to parse the entire bundled JavaScript file. Those bundled files can be really big depending on how many libraries are used. As shown above loading this one big file and parsing it does take longer than loading chunks and initializing the virtual DOM afterwards.

[21] `https://facebook.github.io/react/docs/installation.html`, accessed on 28 May 2017

## 5.2. Implementation / Usage

### 5.2.1. Convenience

It is a common misconception that JSX has to be used to write React components and that you have to use ES6 class syntax. At heart a react component gets constructed by a call to `React.createClass()` which is a function offered by the React library. [22] The virtual DOM elements for this class get created by `React.createElement()` which is offered by the React library as well.

As ES6 is currently not completely supported by major browsers it is recommended to use plain ES5 for client side code. [23]

```
var Greeting = React.createClass({
  render: function() {
    return React.createElement('div', null, 'Hello ' + this.props.
    toWhat);
  }
});

ReactDOM.render(
  React.createElement(Greeting, {toWhat: 'World'}, null),
  document.getElementById('embedded-widget')
);
```

Code 10: Defining React components in plain JavaScript

But after all it is possible to use the more convenient JSX and ES6 syntax by using e.g. Babel with an ES6 to ES5 and JSX to ES5 transpiler:

[22] `https://facebook.github.io/react/docs/react-without-es6.html`, accessed on 28 May 2017

[23] `http://caniuse.com/#search=es6`, accessed on 28 May 2017

```
babel({
    presets: [["es2015", { modules: false }]],
    plugins: ["transform-react-jsx"]
});
```

Code 11: Babel setup for Embedded React

Important for the babel configuration is to use `modules: false` as dependencies shouldn't be bundled into the generated ES5 code.

With transpiling the code above can be written in a convenient and more familiar way:

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

ReactDOM.render(
  <Greeting toWhat="World" />,
  document.getElementById('embedded-widget')
);
```

Code 12: Defining React components with ES6 and JSX

### 5.2.2. Provide Data

HTML5 introduced the data attribute [24] for providing additional data to an HTML object. As React needs to be initialized into a real DOM element, this attribute can be used to hold the data that should be provided directly to the React component.

One feature of the data attribute specification comes in really handy: By pars-

[24] `https://www.w3.org/TR/2011/WD-html5-20110525/elements.html#embedding-custom-non-visible-data-with-the-data-attributes`, accessed on 7 June 2017

ing the data attribute it's key becomes camelCase so it can be used directly in the JavaScript code (e.g. `<... data-our-provided-data="">` becomes accessible through `element.dataset.ourProvidedData`) and the value automatically gets parsed as well if it contains JSON.

So all that has to be done to provide data is the following: Regarding the HTML part the following code has to be added:

```html
<div id="root" data-our-provided-data="SOMETHING"></div>
```

Code 13: Using the HTML5 data tag to provide data to React component

The React component can now be initialized like this:

```javascript
const root = document.getElementById('root');

ReactDOM.render(
  <OurComponent data={root.dataset.ourProvidedData} />,
  root
);
```

Code 14: Initializing the React component with provided data

In `OurComponent` the data will be accessible through its properties [25].

## 5.3. Evaluation

As we use Express as main application we get all the benefits it provides such as routing and request handling. The Time To First Paint is the same as shown in Figure 6. The Time To Fully Loaded without caching takes a longer then those in Figure 7 as additional JavaScript files have to be loaded and evaluated. The whole application but the embedded React component is already usable as soon as the HTML gets displayed.

[25]Properties can be accessed through `this.props.data`.

The disadvantage Express had in reacting on changing and reloading data is solved as the Embedded React component can behave independently from the other parts of the application. In addition as the required files (`react.js`, `react-dom.js` and the Embedded React component's code) are not bundled they can be cached independently and don't take as long to load and evaluate as shown with the one bundled file in Figure 5.

# 6. Conclusion

As shown in the evaluation both techniques - Express and React - do have a right to exist if used correctly.

Express should be used for running server side applications. It runs fast and reliable on the server and only delivers what the browser needs to display the page with a minimal overhead. The questionnaire has shown that most developers score Express applications very easy to setup and work with and do like especially the easy way to setup routes or use Express as API as well.

React allows to build complex user interfaces and applications with lots of interaction. By offering complex functionalities developers annotate it is not that easy to setup a new React application as one has two setup bundling as well as a server that delivers the bundled script to the client.

Performancewise Express beats React as it comes with a faster Time To First Paint as well as a speedier Time To Fully Loaded.

Embedded React concedes to take the best of both techniques and combine it: It makes simple things simple and complex things possible.

## 6.1. Future Research

At the current state React components work like widgets. They get initialized after the page has already been rendered. So the user faces a blank section for a short time.

A way to improve could be the use of Server Side Rendering for components. Hereby the component would get rendered on the server, included in the generated HTML of the response and be delivered in the initial page load. After the JavaScript code is loaded and evaluated it would get replaced by the real component with virtual DOM as described in the "Server Side Rendering" section of the "Related Work" chapter.

Another open question is how to properly structure dependencies. As no module bundler is used at the moment node modules cannot be used directly from npm but have to be available as standalone libraries. Some npm modules offer UMD ("Universal Module Definition") builds which are already including all dependencies the module has so that they are capable of working everywhere.

An interesting research topic would be trying to use module bundling without repeatedly bundling the same libraries like React or React-DOM over and over again.

# Glossary

### AJAX

"Asynchronous JavaScript and XML" defines a way to load data using JavaScript after the website got fully loaded and displayed by sending new requests to a server in the background.

### API

An "application programming interface" defines an interface that e.g. (web) applications offer that can be used by other (web) applications to communicate. One standarized API pattern is the "RESTful API".

### Assets

Assets of a website are e.g. images, music, videos, stylesheets or JavaScript files that are requested after the HTML got parsed.

### Compiling

"Compiling is the general term for taking source code written in one language and transforming into another" [26].

### DOM

The "Document Object Model" represents the parsed XML/HTML as a tree structure. Nodes inside this tree can be modified, deleted or created. The painted webpage shown by the browser is based on this tree.

### ES2015 / ES6 / ECMAScript 6

The latest JavaScript specification.

[26] via `https://www.stevefenton.co.uk/2012/11/compiling-vs-transpiling/`, accessed on 12 June 2017

### HTTP

The “Hypertext Transfer Protocol” is a stateless-protocol defines how data is transfered between a client and a server. Usually data transfer starts with a request from the client and ends with a response from the server.

### Isomorphic

The code of an isomorphic web application can be run server side as well as client side. This is important for techniques such as Server Side Rendering.

### JSX

JSX is an XML/HTML-like syntax used by React for defining components. It gets compiled into JavaScript.

### Microservices

Following a microservice architecture the functionalities of an application get encapsulated in smaller loosely coupled services. This benefits the maintainability and reusability of those functionalities.

### REST / RESTful APIs

The “Representational State Transfer” defines an interface for applications by defining request methods such as GET, POST, PATCH, DELETE and defines how web application should react on those.

### Transpiling

"Transpiling is a specific term for taking source code written in one language and transforming into another language that has a similar level of abstraction" [27].

[27] via `https://www.stevefenton.co.uk/2012/11/compiling-vs-transpiling/`, accessed on 12 June 2017

### Websockets

Websockets offer a full-duplex connection between e.g. a client and a server. In contrast to the HTTP request a websocket connection stays opened so all involved parties can ongoing send and receive data.

## References

Eaton, Kit (2012).
*How One Second Could Cost Amazon $1.6 Billion In Sales.*
URL: `http://www.fastcompany.com/1825005/how%5C-one%5C-secondcould%5C-cost%5C-amazon-16-billion-sales` (accessed on 07 June 2017).

Eneman, Rasmus (2016).
*Improving load time of SPAs - An evaluation of three performance techniques.*
URL: `http://www.diva-portal.org/smash/get/diva2:945665/FULLTEXT01.pdf` (accessed on 07 June 2017).

Hakim, Malek (2016).
*Speed index and critical path rendering performance for isomorphic single page applications.*
URL: `http://cs.winona.edu/CSConference/2016conference.pdf` (accessed on 07 June 2017).

Kay, Alan (1982).
"Simple things should be simple; complex things should be possible."
In: *Byte Magazine Volume 07 Number 04 - Human Factors Engineering*, p. 274.

Meinel, Christoph et al. (2017).
*Die Cloud für Schulen in Deutschland.*
Konzept und Pilotierung der Schul-Cloud.
ISBN: 978-3-86956-397-8.

Nah, Fiona Fui-Hoon (2003).
*A Study on Tolerable Waiting Time: How Long Are Web Users Willing to Wait?*
URL: `https://pdfs.semanticscholar.org/e09f/f31852c87e19bf921a0e38565a901da61f5c.pdf` (accessed on 01 June 2017).

# Appendices

## A. Performance Measuring Code

```
window.onload = function () {
  var now = performance.now();
  var sampleSize = 100;
  var refreshAfter = 5000;

  if((JSON.parse(localStorage.getItem(’perf’)) || []).length < sampleSize) {

    window.setTimeout(function() {
      // Get results from Navigation Timing API
      // https://developer.mozilla.org/en-US/docs/Web/API/Navigation_timing_API
      var timing = performance.timing;
      var requestEntries = performance.getEntries();
      var perfData = {
        JSFilesCount: requestEntries.filter(function(element){return element.name.indexOf(’.js’)> -1;}).length,
        CSSFilesCount: requestEntries.filter(function(element){return element.name.indexOf(’.css’)> -1;}).length,
        FilesCount: requestEntries.length + 1,
        TimeToFirstByte: timing.responseStart - timing.navigationStart,
        TimeToDOMContentLoad: timing.domContentLoadedEventEnd - timing.navigationStart,
        TimeToFirstPaint: Math.round((window.chrome.loadTimes().firstPaintTime * 1000) - (window.chrome.loadTimes().startLoadTime * 1000)),
        TimeToLoad: timing.loadEventEnd - timing.navigationStart,
        TimeToFinish: Math.ceil(now),
        IsFirstLoad: requestEntries.filter(function(element){return element.duration == 0;}).length == 0 ? 1 : 0,
        LoadDuration: timing.loadEventEnd - timing.loadEventStart
      }

      // Add results to localstorage so we can export them at the end
    all together
      storedPerf = JSON.parse(localStorage.getItem('perf')) || [];
      storedPerf.push(perfData)
      localStorage.setItem('perf', JSON.stringify(storedPerf))

      // reload page for next run
      // first argument describes if cache should be disabled or not
      location.reload(false);
    }, refreshAfter);
  } else {
    alert('Done.')
  }
};
```

## B. Performance Measuring Results

In the following tables only the relevant results from the performance measuring code in Appendix A are shown.

Table 1: Raw results for Express without cache

| JS Files Count | Time To First Byte | Time To First Paint | Time To Fully Loaded |
|---|---|---|---|
| 4 | 89 | 428 | 928 |
| 4 | 89 | 476 | 1024 |
| 4 | 86 | 461 | 971 |
| 4 | 80 | 425 | 1040 |
| 4 | 126 | 541 | 985 |
| 4 | 81 | 558 | 933 |
| 4 | 95 | 480 | 890 |
| 4 | 103 | 472 | 1015 |
| 4 | 82 | 445 | 964 |
| 4 | 84 | 425 | 986 |
| 4 | 91 | 472 | 982 |
| 4 | 81 | 453 | 1202 |
| 4 | 77 | 468 | 947 |
| 4 | 83 | 462 | 1345 |
| 4 | 76 | 462 | 1427 |
| 4 | 100 | 545 | 1003 |
| 4 | 77 | 461 | 1893 |
| 4 | 100 | 459 | 996 |
| 4 | 90 | 484 | 923 |
| 4 | 77 | 434 | 947 |
| 4 | 76 | 459 | 943 |
| 4 | 104 | 481 | 996 |
| 4 | 80 | 501 | 1037 |
| 4 | 80 | 445 | 996 |
| 4 | 113 | 466 | 984 |
| 4 | 75 | 374 | 1109 |
| 4 | 70 | 373 | 949 |

Table 1: Raw results for Express without cache

| JS Files Count | Time To First Byte | Time To First Paint | Time To Fully Loaded |
|---|---|---|---|
| 4 | 90 | 386 | 925 |
| 4 | 93 | 386 | 1506 |
| 4 | 72 | 398 | 1403 |
| 4 | 71 | 376 | 946 |
| 4 | 70 | 397 | 816 |
| 4 | 64 | 370 | 1312 |
| 4 | 70 | 383 | 1315 |
| 4 | 71 | 409 | 935 |
| 4 | 70 | 352 | 947 |
| 4 | 63 | 391 | 933 |
| 4 | 70 | 362 | 2023 |
| 4 | 88 | 417 | 943 |
| 4 | 75 | 365 | 913 |
| 4 | 66 | 377 | 800 |
| 4 | 89 | 411 | 933 |
| 4 | 71 | 369 | 922 |
| 4 | 75 | 380 | 1016 |
| 4 | 101 | 392 | 936 |
| 4 | 74 | 427 | 966 |
| 4 | 73 | 364 | 931 |
| 4 | 91 | 432 | 1224 |
| 4 | 77 | 425 | 923 |
| 4 | 103 | 407 | 946 |
| 4 | 82 | 380 | 815 |
| 4 | 82 | 417 | 925 |
| 4 | 107 | 441 | 1013 |
| 4 | 94 | 431 | 923 |
| 4 | 78 | 412 | 920 |
| 4 | 78 | 386 | 902 |
| 4 | 76 | 384 | 875 |
| 4 | 77 | 386 | 1549 |

Table 1: Raw results for Express without cache

| JS Files Count | Time To First Byte | Time To First Paint | Time To Fully Loaded |
|---|---|---|---|
| 4 | 71 | 392 | 1157 |
| 4 | 70 | 394 | 1218 |
| 4 | 70 | 379 | 1223 |
| 4 | 88 | 420 | 1161 |
| 4 | 79 | 405 | 1144 |
| 4 | 72 | 406 | 1245 |
| 4 | 86 | 423 | 1191 |
| 4 | 72 | 370 | 1154 |
| 4 | 72 | 377 | 1209 |
| 4 | 95 | 426 | 1186 |
| 4 | 68 | 377 | 1170 |
| 4 | 70 | 378 | 1248 |
| 4 | 68 | 412 | 1171 |
| 4 | 68 | 380 | 1183 |
| 4 | 78 | 399 | 1178 |
| 4 | 62 | 381 | 1166 |
| 4 | 70 | 387 | 1176 |
| 4 | 72 | 382 | 1148 |
| 4 | 66 | 360 | 1274 |
| 4 | 67 | 392 | 887 |
| 4 | 68 | 382 | 884 |
| 4 | 70 | 368 | 790 |
| 4 | 69 | 427 | 901 |
| 4 | 61 | 369 | 910 |
| 4 | 92 | 378 | 926 |
| 4 | 69 | 355 | 790 |
| 4 | 79 | 412 | 898 |
| 4 | 95 | 417 | 923 |
| 4 | 69 | 392 | 919 |
| 4 | 83 | 470 | 997 |
| 4 | 87 | 433 | 976 |

Table 1: Raw results for Express without cache

| JS Files Count | Time To First Byte | Time To First Paint | Time To Fully Loaded |
|---|---|---|---|
| 4 | 72 | 349 | 920 |
| 4 | 73 | 395 | 1121 |
| 4 | 68 | 452 | 828 |
| 4 | 72 | 350 | 973 |
| 4 | 75 | 393 | 902 |
| 4 | 79 | 581 | 1032 |
| 4 | 75 | 360 | 783 |
| 4 | 65 | 355 | 921 |
| 4 | 68 | 364 | 793 |
| 4 | 66 | 393 | 894 |
| 4 | 72 | 403 | 888 |

Table 2: Raw results for Express with cache

| JS Files Count | Time To First Byte | Time To First Paint | Time To Fully Loaded |
|---|---|---|---|
| 4 | 181 | 492 | 932 |
| 4 | 117 | 388 | 818 |
| 4 | 109 | 343 | 550 |
| 4 | 72 | 298 | 504 |
| 4 | 71 | 289 | 496 |
| 4 | 69 | 255 | 499 |
| 4 | 81 | 311 | 564 |
| 4 | 64 | 242 | 456 |
| 4 | 79 | 285 | 474 |
| 4 | 71 | 249 | 454 |
| 4 | 66 | 275 | 458 |
| 4 | 64 | 285 | 525 |
| 4 | 69 | 273 | 521 |
| 4 | 58 | 274 | 447 |
| 4 | 64 | 276 | 454 |
| 4 | 57 | 245 | 426 |

Table 2: Raw results for Express with cache

| JS Files Count | Time To First Byte | Time To First Paint | Time To Fully Loaded |
|---|---|---|---|
| 4 | 62 | 262 | 485 |
| 4 | 68 | 247 | 450 |
| 4 | 63 | 254 | 454 |
| 4 | 85 | 214 | 517 |
| 4 | 63 | 288 | 477 |
| 4 | 63 | 274 | 508 |
| 4 | 105 | 321 | 537 |
| 4 | 87 | 295 | 513 |
| 4 | 60 | 273 | 502 |
| 4 | 62 | 267 | 454 |
| 4 | 66 | 253 | 475 |
| 4 | 60 | 285 | 950 |
| 4 | 62 | 246 | 492 |
| 4 | 64 | 280 | 499 |
| 4 | 64 | 282 | 494 |
| 4 | 62 | 246 | 499 |
| 4 | 62 | 237 | 492 |
| 4 | 60 | 267 | 455 |
| 4 | 58 | 230 | 424 |
| 4 | 58 | 249 | 473 |
| 4 | 65 | 273 | 502 |
| 4 | 53 | 234 | 483 |
| 4 | 60 | 251 | 491 |
| 4 | 62 | 268 | 449 |
| 4 | 56 | 255 | 463 |
| 4 | 70 | 280 | 508 |
| 4 | 73 | 275 | 450 |
| 4 | 67 | 248 | 461 |
| 4 | 91 | 212 | 759 |
| 4 | 99 | 310 | 539 |
| 4 | 61 | 266 | 449 |

Table 2: Raw results for Express with cache

| JS Files Count | Time To First Byte | Time To First Paint | Time To Fully Loaded |
|---|---|---|---|
| 4 | 99 | 312 | 511 |
| 4 | 130 | 337 | 542 |
| 4 | 124 | 342 | 557 |
| 4 | 90 | 314 | 521 |
| 4 | 68 | 278 | 496 |
| 4 | 85 | 341 | 564 |
| 4 | 63 | 254 | 480 |
| 4 | 58 | 272 | 445 |
| 4 | 68 | 303 | 578 |
| 4 | 59 | 262 | 511 |
| 4 | 58 | 290 | 542 |
| 4 | 59 | 263 | 443 |
| 4 | 61 | 234 | 471 |
| 4 | 63 | 246 | 473 |
| 4 | 64 | 278 | 485 |
| 4 | 57 | 243 | 467 |
| 4 | 60 | 276 | 490 |
| 4 | 81 | 254 | 541 |
| 4 | 63 | 280 | 883 |
| 4 | 63 | 280 | 484 |
| 4 | 57 | 256 | 492 |
| 4 | 59 | 247 | 493 |
| 4 | 62 | 280 | 520 |
| 4 | 59 | 283 | 462 |
| 4 | 59 | 235 | 458 |
| 4 | 87 | 299 | 488 |
| 4 | 60 | 259 | 469 |
| 4 | 62 | 235 | 468 |
| 4 | 94 | 292 | 501 |
| 4 | 95 | 337 | 562 |
| 4 | 70 | 270 | 485 |

Table 2: Raw results for Express with cache

| JS Files Count | Time To First Byte | Time To First Paint | Time To Fully Loaded |
|---|---|---|---|
| 4 | 59 | 241 | 494 |
| 4 | 67 | 289 | 1115 |
| 4 | 73 | 280 | 456 |
| 4 | 65 | 261 | 465 |
| 4 | 60 | 278 | 461 |
| 4 | 62 | 290 | 511 |
| 4 | 57 | 234 | 508 |
| 4 | 63 | 288 | 508 |
| 4 | 56 | 229 | 517 |
| 4 | 56 | 275 | 722 |
| 4 | 60 | 283 | 443 |
| 4 | 61 | 262 | 481 |
| 4 | 68 | 242 | 1175 |
| 4 | 64 | 245 | 492 |
| 4 | 58 | 265 | 448 |
| 4 | 62 | 233 | 410 |
| 4 | 58 | 274 | 489 |
| 4 | 56 | 239 | 470 |
| 4 | 83 | 266 | 498 |
| 4 | 61 | 270 | 481 |
| 4 | 61 | 251 | 460 |
| 4 | 88 | 307 | 521 |

Table 3: Raw results for React without cache

| JS Files Count | Time To First Byte | Time To First Paint | Time To Fully Loaded |
|---|---|---|---|
| 1 | 37 | 1946 | 1451 |
| 1 | 33 | 1577 | 1252 |
| 1 | 32 | 1571 | 1259 |
| 1 | 33 | 1686 | 1354 |
| 1 | 32 | 1628 | 1276 |

Table 3: Raw results for React without cache

| JS Files Count | Time To First Byte | Time To First Paint | Time To Fully Loaded |
|---|---|---|---|
| 1 | 31 | 1602 | 1261 |
| 1 | 37 | 1650 | 1266 |
| 1 | 25 | 1552 | 1248 |
| 1 | 36 | 1556 | 1259 |
| 1 | 32 | 1586 | 1268 |
| 1 | 34 | 1635 | 1319 |
| 1 | 36 | 1619 | 1265 |
| 1 | 39 | 1631 | 1316 |
| 1 | 30 | 1537 | 1252 |
| 1 | 33 | 1608 | 1290 |
| 1 | 33 | 1522 | 1245 |
| 1 | 30 | 1610 | 1277 |
| 1 | 38 | 1980 | 1647 |
| 1 | 33 | 1556 | 1232 |
| 1 | 36 | 1638 | 1308 |
| 1 | 29 | 1702 | 1350 |
| 1 | 23 | 1588 | 1226 |
| 1 | 30 | 1618 | 1269 |
| 1 | 26 | 1519 | 1164 |
| 1 | 27 | 1619 | 1287 |
| 1 | 30 | 1561 | 1234 |
| 1 | 33 | 1640 | 1273 |
| 1 | 32 | 1651 | 1321 |
| 1 | 33 | 1657 | 1289 |
| 1 | 34 | 1519 | 1201 |
| 1 | 38 | 1638 | 1324 |
| 1 | 32 | 1587 | 1274 |
| 1 | 36 | 1597 | 1266 |
| 1 | 32 | 1522 | 1200 |
| 1 | 35 | 1517 | 1249 |
| 1 | 34 | 1641 | 1281 |

Table 3: Raw results for React without cache

| JS Files Count | Time To First Byte | Time To First Paint | Time To Fully Loaded |
|---|---|---|---|
| 1 | 30 | 1690 | 1331 |
| 1 | 29 | 1596 | 1246 |
| 1 | 30 | 1544 | 1237 |
| 1 | 36 | 1661 | 1305 |
| 1 | 28 | 1577 | 1251 |
| 1 | 35 | 1642 | 1304 |
| 1 | 33 | 1563 | 1240 |
| 1 | 33 | 1651 | 1283 |
| 1 | 34 | 1586 | 1275 |
| 1 | 35 | 1589 | 1265 |
| 1 | 28 | 1602 | 1279 |
| 1 | 34 | 1593 | 1237 |
| 1 | 31 | 1566 | 1221 |
| 1 | 31 | 1668 | 1313 |
| 1 | 35 | 1611 | 1273 |
| 1 | 35 | 1564 | 1229 |
| 1 | 35 | 1546 | 1239 |
| 1 | 27 | 1610 | 1286 |
| 1 | 39 | 1553 | 1219 |
| 1 | 34 | 1582 | 1270 |
| 1 | 37 | 1588 | 1234 |
| 1 | 32 | 1597 | 1294 |
| 1 | 36 | 1574 | 1247 |
| 1 | 29 | 1656 | 1303 |
| 1 | 34 | 1707 | 1341 |
| 1 | 33 | 1588 | 1279 |
| 1 | 36 | 1618 | 1280 |
| 1 | 25 | 1609 | 1284 |
| 1 | 35 | 1640 | 1311 |
| 1 | 24 | 1608 | 1221 |
| 1 | 34 | 1611 | 1259 |

Table 3: Raw results for React without cache

| JS Files Count | Time To First Byte | Time To First Paint | Time To Fully Loaded |
|---|---|---|---|
| 1 | 33 | 1634 | 1287 |
| 1 | 33 | 1614 | 1303 |
| 1 | 32 | 1562 | 1234 |
| 1 | 28 | 1693 | 1329 |
| 1 | 35 | 1640 | 1306 |
| 1 | 27 | 1641 | 1277 |
| 1 | 27 | 1654 | 1340 |
| 1 | 31 | 1641 | 1303 |
| 1 | 35 | 1590 | 1266 |
| 1 | 34 | 1600 | 1284 |
| 1 | 34 | 1569 | 1246 |
| 1 | 30 | 1555 | 1252 |
| 1 | 29 | 1554 | 1255 |
| 1 | 36 | 1537 | 1219 |
| 1 | 37 | 1618 | 1305 |
| 1 | 34 | 1609 | 1286 |
| 1 | 37 | 1608 | 1264 |
| 1 | 30 | 1591 | 1274 |
| 1 | 31 | 1598 | 1295 |
| 1 | 37 | 1602 | 1250 |
| 1 | 33 | 1595 | 1263 |
| 1 | 26 | 1661 | 1292 |
| 1 | 23 | 1596 | 1260 |
| 1 | 24 | 1604 | 1232 |
| 1 | 35 | 1595 | 1268 |
| 1 | 38 | 1743 | 1377 |
| 1 | 23 | 1584 | 1260 |
| 1 | 27 | 1622 | 1271 |
| 1 | 29 | 1565 | 1251 |
| 1 | 27 | 1615 | 1280 |
| 1 | 38 | 1665 | 1312 |

Table 3: Raw results for React without cache

| JS Files Count | Time To First Byte | Time To First Paint | Time To Fully Loaded |
|---|---|---|---|
| 1 | 30 | 1575 | 1226 |
| 1 | 23 | 1621 | 1327 |

Table 4: Raw results for React with cache

| JS Files Count | Time To First Byte | Time To First Paint | Time To Fully Loaded |
|---|---|---|---|
| 1 | 21 | 2275 | 1664 |
| 1 | 23 | 1034 | 833 |
| 1 | 23 | 854 | 713 |
| 1 | 23 | 990 | 881 |
| 1 | 26 | 878 | 728 |
| 1 | 29 | 941 | 831 |
| 1 | 24 | 869 | 709 |
| 1 | 15 | 934 | 817 |
| 1 | 21 | 862 | 712 |
| 1 | 19 | 952 | 836 |
| 1 | 19 | 835 | 702 |
| 1 | 22 | 935 | 833 |
| 1 | 20 | 851 | 704 |
| 1 | 25 | 943 | 805 |
| 1 | 26 | 856 | 698 |
| 1 | 21 | 924 | 776 |
| 1 | 23 | 852 | 722 |
| 1 | 22 | 929 | 793 |
| 1 | 18 | 854 | 704 |
| 1 | 25 | 917 | 768 |
| 1 | 25 | 860 | 711 |
| 1 | 22 | 939 | 794 |
| 1 | 24 | 859 | 710 |
| 1 | 23 | 910 | 770 |
| 1 | 23 | 893 | 736 |

Table 4: Raw results for React with cache

| JS Files Count | Time To First Byte | Time To First Paint | Time To Fully Loaded |
|---|---|---|---|
| 1 | 23 | 1163 | 931 |
| 1 | 15 | 861 | 709 |
| 1 | 22 | 955 | 847 |
| 1 | 28 | 880 | 711 |
| 1 | 17 | 953 | 800 |
| 1 | 15 | 847 | 703 |
| 1 | 23 | 961 | 820 |
| 1 | 20 | 854 | 697 |
| 1 | 24 | 962 | 803 |
| 1 | 22 | 855 | 710 |
| 1 | 21 | 993 | 847 |
| 1 | 26 | 873 | 718 |
| 1 | 19 | 1013 | 830 |
| 1 | 22 | 862 | 714 |
| 1 | 21 | 956 | 845 |
| 1 | 22 | 853 | 700 |
| 1 | 19 | 942 | 813 |
| 1 | 19 | 875 | 737 |
| 1 | 20 | 946 | 796 |
| 1 | 25 | 853 | 704 |
| 1 | 20 | 947 | 798 |
| 1 | 20 | 853 | 701 |
| 1 | 16 | 938 | 801 |
| 1 | 21 | 866 | 725 |
| 1 | 21 | 967 | 791 |
| 1 | 24 | 869 | 711 |
| 1 | 21 | 932 | 793 |
| 1 | 22 | 865 | 726 |
| 1 | 18 | 972 | 855 |
| 1 | 19 | 845 | 693 |
| 1 | 24 | 990 | 868 |

Table 4: Raw results for React with cache

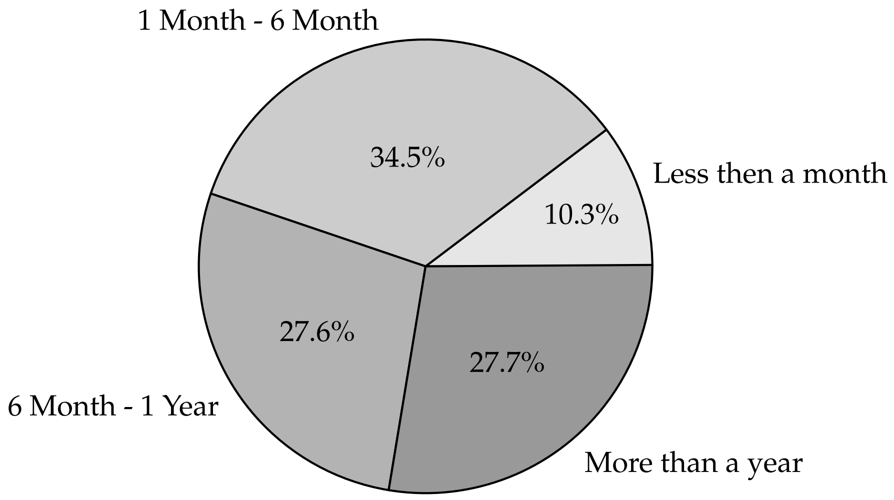
| JS Files Count | Time To First Byte | Time To First Paint | Time To Fully Loaded |
|---|---|---|---|
| 1 | 20 | 891 | 754 |
| 1 | 24 | 1019 | 837 |
| 1 | 21 | 848 | 695 |
| 1 | 19 | 978 | 863 |
| 1 | 19 | 835 | 685 |
| 1 | 20 | 969 | 868 |
| 1 | 19 | 865 | 711 |
| 1 | 28 | 996 | 877 |
| 1 | 16 | 837 | 701 |
| 1 | 22 | 945 | 787 |
| 1 | 19 | 831 | 697 |
| 1 | 20 | 996 | 813 |
| 1 | 24 | 854 | 708 |
| 1 | 20 | 956 | 867 |
| 1 | 18 | 574 | 426 |
| 1 | 23 | 834 | 708 |
| 1 | 22 | 946 | 797 |
| 1 | 23 | 869 | 735 |
| 1 | 24 | 1131 | 989 |
| 1 | 19 | 858 | 720 |
| 1 | 22 | 954 | 797 |
| 1 | 24 | 959 | 801 |
| 1 | 27 | 983 | 867 |
| 1 | 22 | 831 | 698 |
| 1 | 22 | 943 | 796 |
| 1 | 16 | 857 | 719 |
| 1 | 25 | 958 | 820 |
| 1 | 21 | 842 | 696 |
| 1 | 22 | 953 | 786 |
| 1 | 22 | 898 | 745 |
| 1 | 20 | 940 | 785 |

Table 4: Raw results for React with cache

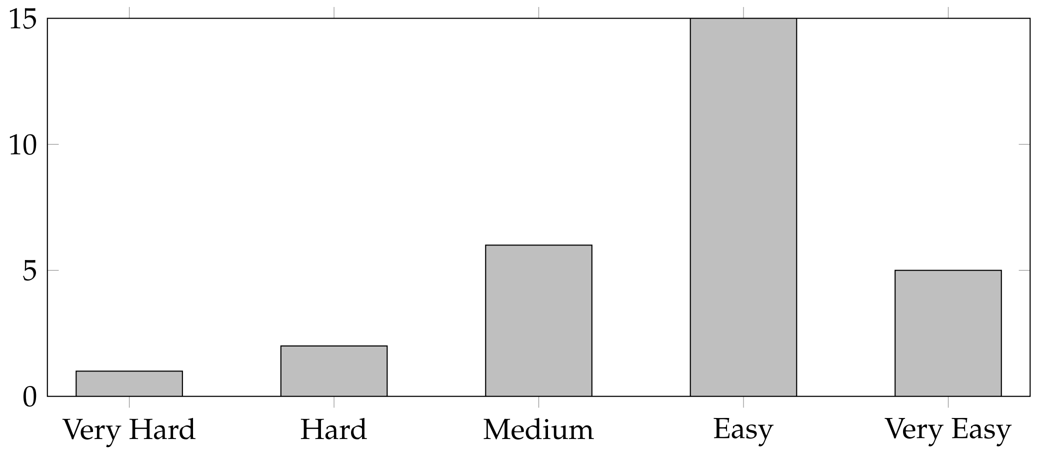
| JS Files Count | Time To First Byte | Time To First Paint | Time To Fully Loaded |
|---|---|---|---|
| 1 | 18 | 875 | 741 |
| 1 | 25 | 941 | 786 |
| 1 | 23 | 849 | 704 |
| 1 | 23 | 955 | 846 |
| 1 | 21 | 852 | 702 |
| 1 | 20 | 924 | 778 |
| 1 | 23 | 831 | 697 |
| 1 | 26 | 952 | 848 |
| 1 | 24 | 849 | 695 |
| 1 | 30 | 996 | 854 |
| 1 | 19 | 873 | 717 |
| 1 | 25 | 964 | 856 |
| 1 | 19 | 834 | 701 |

## C. Questionnaire Results

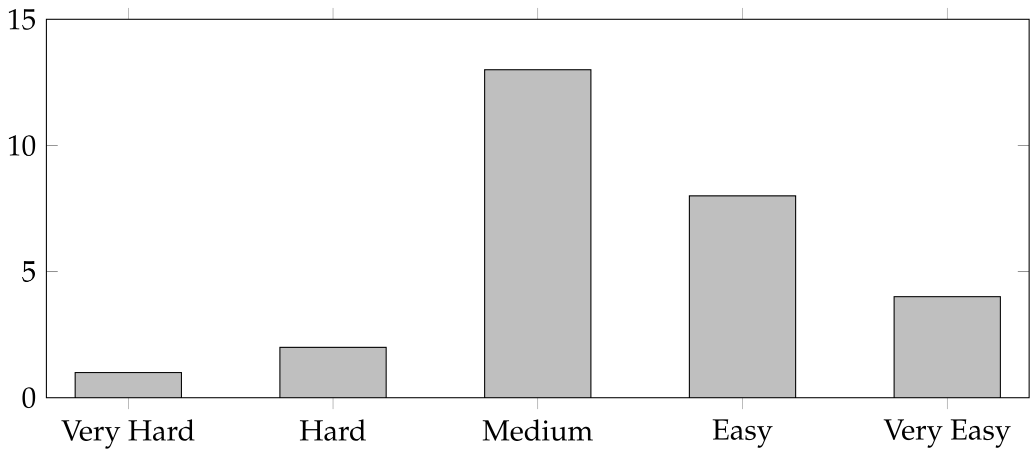
### How long have you used React?



### How easy is it to get started with React?



### How easy is it to setup a new React application?



### When would you use React?

- always when developing web applications
- Visualizing lots of data or building quick prototypes of CRUD applications etc.
- any frontend app with logic
- For bigger projects
- If I have a problem that can easily be solved by using stateful components, or if I don't have enough time to learn a better suited framework, since I know react pretty well by now and can develop pretty fast using it.
- Complex Userinterfaces/single Page applications
- Almost any web app or mobile app (using react native)
- In web programming
- on a one-page app
- Maybe on very large websites/SPAs with very complex markup and/or when mostly REST based.
- For interactive front ends
- Apps with at least minimal complexity.
- Whenever I wanna start a more complex web applications
- Whenever I want to build a webapp
- It's my go-to framework

- complex web applications
- Getting going quickly
- any not completely trivial web frontend
- When building a one-page site

## When would you NOT use React?

- only when not possible
- Complex applications that require computationally or graphically intensive computations or renderings
- static webpages
- for small one page websites
- If the solution is so simple that react would just be overkill. I don't need react for a static imprint page, for example.
- Small Websites
- scraping, html/ xml static generation (when not intended to do server rendering, e.g. for a script or so)
- For 3D web applications
- on not one-paged apps
- On small to mid size web sites
- For very small projects
- Dead-drop simple apps.
- Single page applications with only css and simple javascript
- When working with react-native and building an extensive app which will sooner or later need ios or android specific components.
- When inheriting an Angular2 codebase
- static websites, (very) simple web apps
- Possibly things with longer maintenance phase, since React does not enforce an architecture e.g. the way angular does it. But maybe I'd use it regardless.
- trivial web UI
- Building a site with many different pages or when having to integrate many other frameworks / libraries

### What are limitations of React?

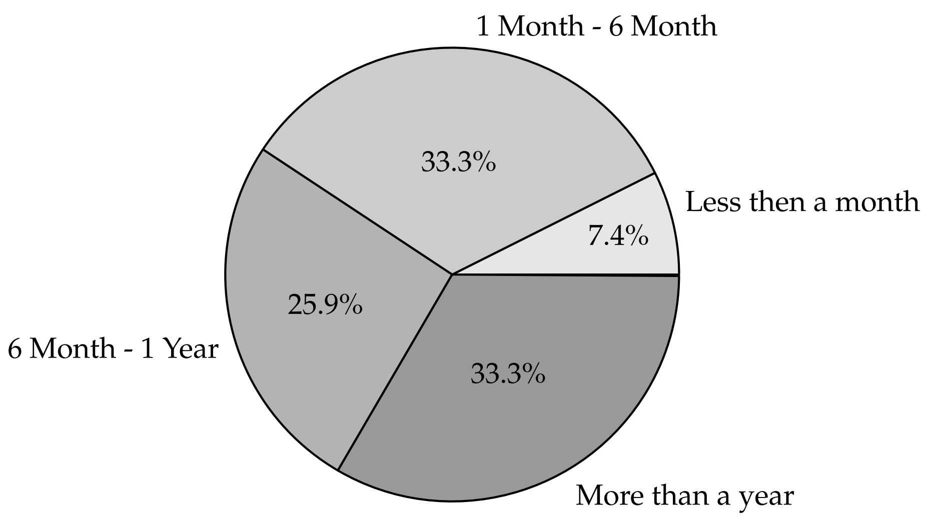
- Mostly inline styles, sometimes very slow, hard to combine with some other frameworks (like e.g. D3 etc.)
- Animations are shit
- Lot's and lot's of extensions and other tools (e.g. redux and webpack, to name only a few) that quickly increase the complexity.
- From a business perspective: The legal clause that gives facebook ownership of the product if it collides with facebooks business.
- Performance and SEO
- managing communications in big hierarchies of DOM (only context possible). you have to extend react with redux or similar flow management in these cases. integrating react with frameworks or non-react projects requires deep thinking.
- Client-side only. Needs additional libs to make sense, which in turn leads to code that resembles native browser functionality (request/response) with some XHR/Fetch/Routing stuff you otherwise do not have to care so much about in the application. I would prefer using ServiceWorkers and consequently use Caching mechanisms. Mostly latency is the problem, not bandwidth. In my opinion, React's "update only what has changed" in markup solves a problem that does not exist on small/mid web sites. Loading data is imminent in web based applications. When using caching for static assets then the bandwidth overhead and additional rendering time should not be the problem.
- You have to first read some of the many good tutorials on it. So much boilerplate.
- As everything in the javascript ecosystem things change rapidly
- -
- Haven't really come across anything yet
- interoperability with web components, complicated setup (as for any frontend app): module bundling, testing setup etc.
- The "React way" of developing often clashes with other frameworks / libraries, e.g. Bootstrap or jQuery, and leads to ugly hacks to pass data around. Mastering the component lifecycle took me quite some while.

Also I had some problems with the global state of my application (I heard that integrating Reflux makes that better, but that means learning yet another paradigm)

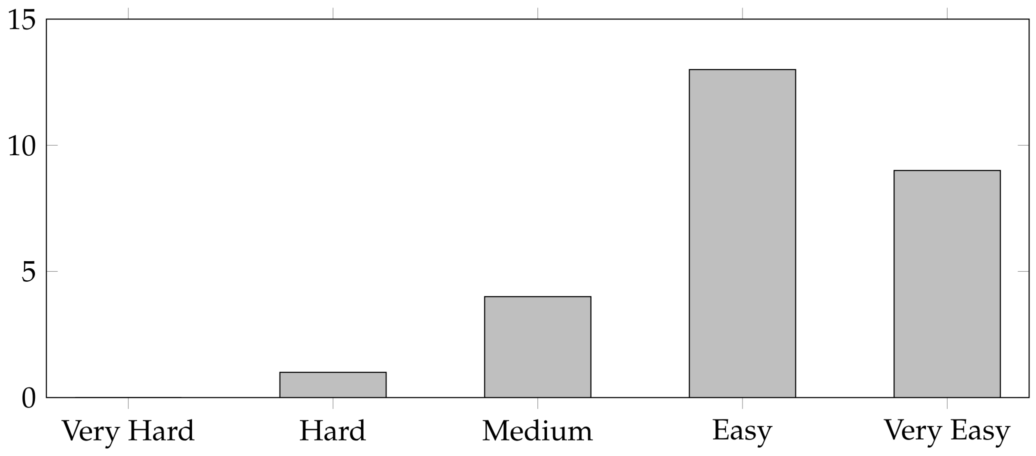
### What is awesome about React?

- the programming model
- Quick prototyping via hot reload, a useful component can always be found on the web, easy to get started
- Composibility, Lifecycle hooks, Structure / Sanity
- very Modularized
- Its Extensibility, simplicity, it's fast, there are lot's of guides and stackoverflow questions
- Easy to use, Big Projects are transparent
- functional, descriptive, chain of responsibility
- The state-driven paradigm and the speed
- the collaboration of components and how they can interact
- Lack of async issues.
- Ecosystem, Functional Programming
- Pretty straight forward, allows minimal and hot reloading.
- Isomorphism + React Native
- composability, JSX
- It feels so simple and straight-forward.
- testability
- If you can manage to do everything in "the React way" and basically just pass dumb data around, handling data flow and building an application is quite nice.

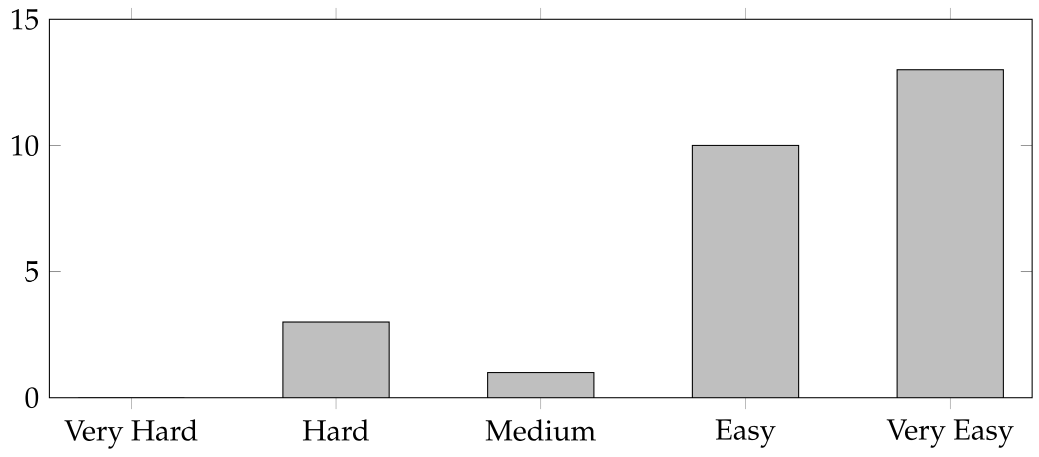
## How long have you used Express?



## How easy is it to get started with Express?



### How easy is it to setup a new Express application?



### When would you use Express?

- when building small-medium applications
- Small backend servers, not too many routes, not too much work in the backend
- any node js server, ReST APIs
- simple API, prototypes
- To optimize
- Whenever I would need a HTTP server.
- So far: Whenever I need a backend. Javascript for the win.
- Hobby Projects, in small companies
- rest apis
- Have used it only for small services yet.
- I use it serve DB routes. I'd use it for non-SPA, rather static sites.
- Small(er) Projects, no extensive testing
- It's my go to technology for server side
- if you need a simple backend

### When would you NOT use Express?

- when building large applications with lots of routes

- Complex server-side logic & complex routing etc.
- any node app that is not a web server
- performance critical applications
- No idea, since I haven't tried any other framework yet.
- If I have to work with people who don't want to code javascript for server-side code.
- Big Projects with Lots of Developers
- static server rendering
- Not enough experience up to now
- These days, never to serve more than the initial page.
- Performance Critical Applications should not be designed with react
- When inheriting a massive .Net codebase
- high performance/much computation, very large project

### What are limitations of Express?

- no idea
- Performance
- Templates are messy. There are no enhancements now that IBM controls the repo. I doubt it'll be upgraded to HTTP/2.
- JavaScript
- Have to deal with a great deal of plumbing

### What is awesome about Express?

- simplicity
- Quickly prototype servers & APIs without too much hassle
- simple routing
- Pretty easy to get started
- Setting up routes is easy
- it is fast and simple
- JavaScript aus backend
- It handles nicely all the messy, non-standard, non-compliant HTTP found in the wild.

- Very easy to start and very compatible
- Rich ecosystem - lots of libraries and extensions. Simple and flexible
- makes building a webserver far more easier compared to plain node.js