e Class components, Miscellaneous

Class Components

During the course we have only used React components having been defined as Javascript functions. This was not possible without the hook-functionality that came with version 16.8 of React. Before, when defining a component that uses state one had to define it using Javascript's Class-syntax.

It is beneficial to at least be familiar with Class Components to some extent, since the world contains a lot of old React code, which will probably never be completely rewritten using the updated syntax.

Let's get to know the main features of Class Components by producing yet another very familiar anecdote application. We store the anecdotes in the file *db.json* using *json-server*. The contents of the file are lifted from here.

The initial version of the Class Component look like this

```
import React from 'react'

class App extends React.Component {
  constructor(props) {
    super(props)
  }

  render() {
    return (
      <div>
        <h1>anecdote of the day</h1>
      </div>
    )
  }
```

```
}

export default App
```

The component now has a constructor, in which nothing happens at the moment, and contains the method render. As one might guess, render defines how and what is rendered to the screen.

Let's define a state for the list of anecdotes and the currently visible anecdote. In contrast to when using the useState-hook Class Components only contain one state. So if the state is made up of multiple "parts" they should be stored as properties of the state. The state is initialized in the constructor:

```
class App extends React.Component {
  constructor(props) {
    super(props)

    this.state = {
      anecdotes: [],
      current: 0
    }
  }

  render() {
    if (this.state.anecdotes.length === 0 ) {
      return <div>no anecdotes...</div>
    }

    return (
      <div>
        <h1>anecdote of the day</h1>
        <div>
          {this.state.anecdotes[this.state.current].content}
        </div>
        <button>next</button>
      </div>
    )
  }
}
```

The component state is in the instance variable `this.state`. The state is an object having two properties. *this.state.anecdotes* is the list of anecdotes and *this.state.current* is the index of the currently shown anecdote.

In Functional components the right place for fetching data from a server is inside an effect hook, which is executed when a component renders or less frequently if necessary, e.g. only in combination with the first render.

The lifecycle-methods of Class Components offer corresponding functionality. The correct place to trigger the fetching of data from a server is inside the lifecycle-method componentDidMount,

which is executed once right after the first time a component renders:

```
class App extends React.Component {
  constructor(props) {
    super(props)

    this.state = {
      anecdotes: [],
      current: 0
    }
  }

  componentDidMount = () => {
    axios.get('http://localhost:3001/anecdotes').then(response => {
      this.setState({ anecdotes: response.data })
    })
  }

  // ...
}
```

The callback function of the HTTP request updates the component state using the method setState. The method only touches the keys that have been defined in the object passed to the method as an argument. The value for the key *current* remains unchanged.

Calling the method setState always trigger the rerender of the Class Component, i.e. calling the method `render`.

We'll finish off the component with the ability to change the shown anecdote. The following is the code for the entire component with the addition highlighted:

```
class App extends React.Component {
  constructor(props) {
    super(props)

    this.state = {
      anecdotes: [],
      current: 0
    }
  }

  componentDidMount = () => {
    axios.get('http://localhost:3001/anecdotes').then(response => {
      this.setState({ anecdotes: response.data })
    })
  }

  handleClick = () => {
    const current = Math.floor(
```

```
      Math.random() * this.state.anecdotes.length
    )
    this.setState({ current })
  }

  render() {
    if (this.state.anecdotes.length === 0 ) {
      return <div>no anecdotes...</div>
    }

    return (
      <div>
        <h1>anecdote of the day</h1>
        <div>{this.state.anecdotes[this.state.current].content}</div>
        <button onClick={this.handleClick}>next</button>
      </div>
    )
  }
}
```

For comparison here is the same application as a Functional component:

```
const App = () => {
  const [anecdotes, setAnecdotes] = useState([])
  const [current, setCurrent] = useState(0)

  useEffect(() =>{
    axios.get('http://localhost:3001/anecdotes').then(response => {
      setAnecdotes(response.data)
    })
  },[])

  const handleClick = () => {
    setCurrent(Math.round(Math.random() * (anecdotes.length - 1)))
  }

  if (anecdotes.length === 0) {
    return <div>no anecdotes...</div>
  }

  return (
    <div>
      <h1>anecdote of the day</h1>
      <div>{anecdotes[current].content}</div>
      <button onClick={handleClick}>next</button>
    </div>
  )
}
```

In the case of our example the differences were minor. The biggest difference between Functional

components and Class components is mainly that the state of a Class component is a single object, and that the state is updated using the method `setState`, while in Functional components the state can consist of multiple different variables, with all of them having their own update function.

In some more advanced use cases the effect hook offers a considerably better mechanism for controlling side effects compared to the lifecycle-methods of Class Components.

A notable benefit of using Functional components is not having to deal with the self referencing `this`-reference of the Javascript class.

In my opinion, and the opinion of many others, Class Components offer basically no benefits over Functional components enhanced with hooks, with the exception of the so-called error boundary mechanism, which currently (15th February 2021) isn't yet in use by functional components.

When writing fresh code there is no rational reason to use Class Components if the project is using React with a version number 16.8 or greater. On the other hand, there is currently no need to rewrite all old React code as Functional components.

Organization of code in React application

In most applications we followed the principle, by which components were placed in the directory *components*, reducers were placed in the directory *reducers*, and the code responsible for communicating with the server was placed in the directory *services*. This way of organizing fits a smaller application just fine, but as the amount of components increase, better solutions are needed. There is no one correct way to organize a project. The article The 100% correct way to structure a React app (or why there's no such thing) provides some perspective on the issue.

Frontend and backend in the same repository

During the course we have created the frontend and backend into separate repositories. This is a very typical approach. However, we did the deployment by copying the bundled frontend code into the backend repository. A possibly better approach would have been to deploy the frontend code separately. Especially with applications created using create-react-app it is very straightforward thanks to the included buildpack.

Sometimes there may be a situation where the entire application is to be put into a single repository. In this case a common approach is to put the *package.json* and *webpack.config.js* in the root directory, as well as place the frontend and backend code into their own directories, e.g. *client* and *server*.

This repository provides one possible starting point for the organization of "single-repository-code".

Changes on the server

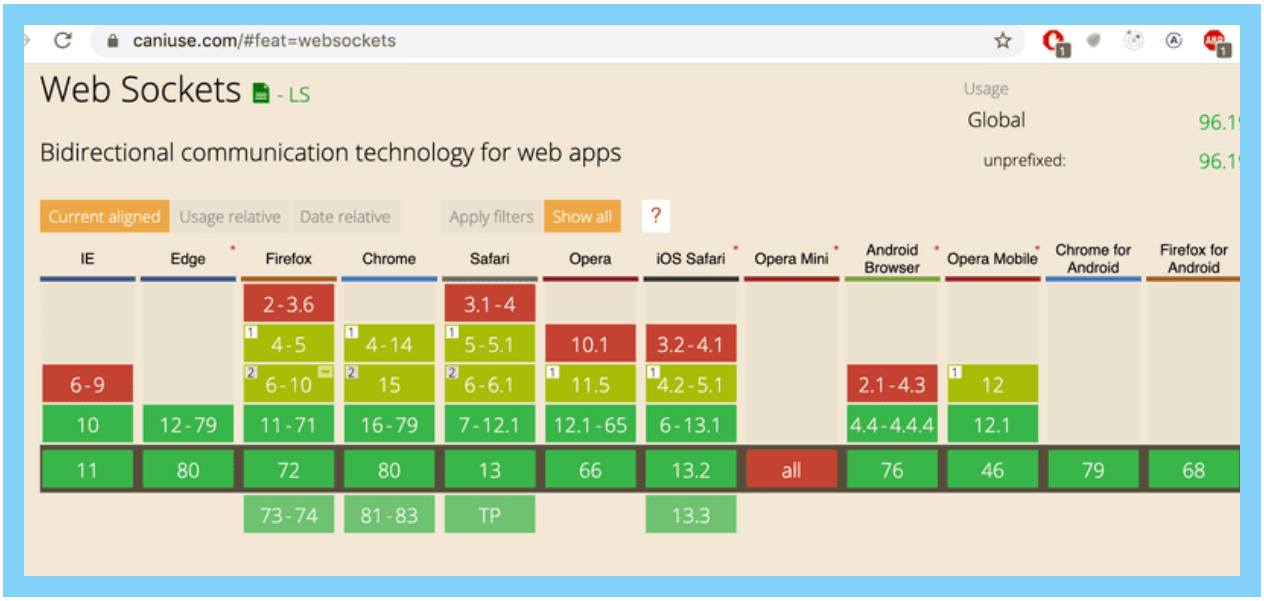
If there are changes in the state on the server, e.g. when new blogs are added by other users to the bloglist service, the React-frontend we implemented during this course will not notice these changes until the page reloads. A similar situation arises when the frontend triggers a time-

consuming computation in the backend. How do we reflect the results of the computation to the frontend?

One way is to execute polling on the frontend, meaning repeated requests to the backend API e.g. using the setInterval-command.

A more sophisticated way is to use WebSockets, using which it is possible to establish a two-way communication channel between the browser and the server. In this case the browser does not need to poll the backend, and instead only has to define callback functions for situations when the server sends data about updating state using a WebSocket.

WebSockets are an API provided by the browser, which is not yet fully supported on all browsers:



Instead of directly using the WebSocket API it is advisable to use the Socket.io-library, which provides various *fallback*-options in case the browser does not have the full support for WebSockets.

In part 8 our topic is GraphQL that provides a nice mechanism for notifying clients when there are changes in the backend data.

Virtual DOM

The concept of the Virtual DOM often comes up when discussing React. What is it all about? As mentioned in part 0 browsers provide a DOM API, using which the JavaScript running in the browser can modify the elements defining the appearance of the page.

When a software developer uses React they rarely or never directly manipulate the DOM. The function defining the React component returns a set of React-elements. Although some of the elements look like normal HTML-elements

```
const element = <h1>Hello, world</h1>
```

they are also just JavaScript based React-elements at their core.

The React-elements defining the appearance of the components of the application make up the Virtual DOM, which is stored in system memory during runtime.

With the help of the ReactDOM-library the virtual DOM defined by the components is rendered to a real DOM that can be shown by the browser using the DOM API:

```
ReactDOM.render(
  <App />,
  document.getElementById('root')
)
```

When the state of the application changes a *new virtual DOM* gets defined by the components. React has the previous version of the virtual DOM in memory and instead of directly rendering the new virtual DOM using the DOM API React computes the optimal way to update the DOM (remove, add or modify elements in the DOM) such that the DOM reflects the new virtual DOM.

On the role of React in applications

In the material we may not have put enough emphasis on the fact that React is primarily a library for managing the creation of views for an application. If we look at the traditional Model View Controller -pattern, then the domain of React would be *View*. React has a more narrow area of application than e.g. Angular, which is an all-encompassing Frontend MVC-framework. Therefore React is not being called a *framework*, but a *library*.

In small applications data handled by the application is being stored in the state of the React-components, so in this scenario the state of the components can be thought of as *models* of an MVC-architecture.

However, MVC-architecture is not usually mentioned when talking about React-applications. Furthermore, if we are using Redux, then the applications follow the Flux-architecture and the role of React is even more focused on creating the views. The business logic of the application is handled using the Redux state and action creators. If we're using redux thunk familiar from part 6, then the business logic can be almost completely separated from the React code.

Because both React and Flux were created at Facebook one could say that using React only as a UI library is the intended use case. Following the Flux-architecture adds some overhead to the application, and if we're talking about a small application or prototype it might be a good idea to use React "wrong", since over-engineering rarely yields an optimal result.

As I mentioned at the end of part 6, the React Context-api offers one alternative solution for centralized state management without the need for third party libraries such as redux. You can read more about this here and here.

React/node-application security

So far during the course we have not touched on information security much. We do not have much

time this for now either, but fortunately the department has a MOOC-course Securing Software for this important topic.

We will, however, take a look at some things specific to this course.

The Open Web Application Security Project, otherwise known as OWASP, publishes an annual list of the most common security risks in Web-applications. The most recent list can be found here. The same risks can be found from one year to another.

At the top of the list we find *injection*, which means that e.g. text sent using a form in an application is interpreted completely differently than the software developer had intended. The most famous type of injection is probably the SQL-injection.

For example, if the following SQL-query would be executed in a vulnerable application:

```
let query = "SELECT * FROM Users WHERE name = '" + userName + "';"
```

Now let's assume that a malicious user *Arto Hellas* would define their name as

```
Arto Hell-as'; DROP TABLE Users; --
```

so that the name would contain a single quote `'` , which is the beginning- and end-character of a SQL-string. As a result of this two SQL-operations would be executed, the second of which would destroy the database table *Users*

```
SELECT * FROM Users WHERE name = 'Arto Hell-as'; DROP TABLE Users; --'
```

SQL-injections are prevented by sanitizing the input, which would entail checking that the parameters of the query do not contain any forbidden characters, in this case single quotes. If forbidden characters are found they are replaced with safe alternatives by escaping them.

Injection attacks are also possible in NoSQL-databases. However, mongoose prevents them by sanitizing the queries. More on the topic can be found e.g. here.

Cross-site scripting (XSS) is an attack where it is possible to inject malicious JavaScript code into a legitimate web-application. The malicious code would then be executed in the browser of the victim. If we try to inject the following into e.g. the notes application

```
<script>
  alert('Evil XSS attack')
</script>
```

the code is not executed, but is only rendered as 'text' on the page:

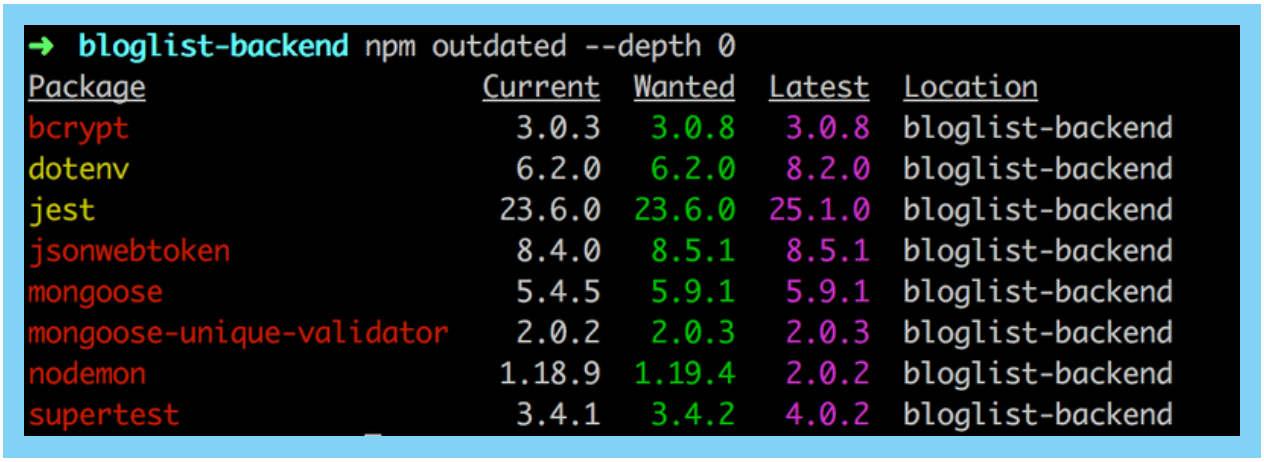
since React takes care of sanitizing data in variables. Some versions of React have been vulnerable to XSS-attacks. The security-holes have of course been patched, but there is no guarantee that there could be more.

One needs to remain vigilant when using libraries; if there are security updates to those libraries, it is advisable to update those libraries in one's own applications. Security updates for Express are found in the library's documentation and the ones for Node are found in this blog.

You can check how up to date your dependencies are using the command

```
npm outdated --depth 0
```

Last year's model answer for the exercises in part 4 already have quite a few outdated dependencies:



The dependencies can be brought up to date by updating the file *package.json* and running the command `npm install`. However, old versions of the dependencies are not necessarily a security risk.

The npm audit command can be used to check the security of dependencies. It compares the version numbers of the dependencies in your application to a list of the version numbers of dependencies containing known security threats in a centralized error database.

Running `npm audit` on an exercise from part 4 of last year's course print a long list of complaints and suggested fixes. Below is a part of the report:

```
$ bloglist-backend npm audit

                       === npm audit security report ===

# Run  npm install --save-dev jest@25.1.0  to resolve 62 vulnerabilities
SEMVER WARNING: Recommended action is a potentially breaking change
┌───────────────┬──────────────────────────────────────────────────────────────┐
│ Low           │ Regular Expression Denial of Service                         │
├───────────────┼──────────────────────────────────────────────────────────────┤
│ Package       │ braces                                                       │
├───────────────┼──────────────────────────────────────────────────────────────┤
│ Dependency of │ jest [dev]                                                   │
├───────────────┼──────────────────────────────────────────────────────────────┤
│ Path          │ jest > jest-cli > jest-config > babel-jest >                 │
│               │ babel-plugin-istanbul > test-exclude > micromatch > braces   │
├───────────────┼──────────────────────────────────────────────────────────────┤
│ More info     │ https://npmjs.com/advisories/786                             │
└───────────────┴──────────────────────────────────────────────────────────────┘

┌───────────────┬──────────────────────────────────────────────────────────────┐
│ Low           │ Regular Expression Denial of Service                         │
├───────────────┼──────────────────────────────────────────────────────────────┤
│ Package       │ braces                                                       │
├───────────────┼──────────────────────────────────────────────────────────────┤
│ Dependency of │ jest [dev]                                                   │
├───────────────┼──────────────────────────────────────────────────────────────┤
│ Path          │ jest > jest-cli > jest-runner > jest-config > babel-jest >   │
│               │ babel-plugin-istanbul > test-exclude > micromatch > braces   │
├───────────────┼──────────────────────────────────────────────────────────────┤
│ More info     │ https://npmjs.com/advisories/786                             │
└───────────────┴──────────────────────────────────────────────────────────────┘

┌───────────────┬──────────────────────────────────────────────────────────────┐
│ Low           │ Regular Expression Denial of Service                         │
├───────────────┼──────────────────────────────────────────────────────────────┤
│ Package       │ braces                                                       │
├───────────────┼──────────────────────────────────────────────────────────────┤
│ Dependency of │ jest [dev]                                                   │
├───────────────┼──────────────────────────────────────────────────────────────┤
│ Path          │ jest > jest-cli > jest-runner > jest-runtime > jest-config > │
│               │ babel-jest > babel-plugin-istanbul > test-exclude >          │
│               │ micromatch > braces                                          │
├───────────────┼──────────────────────────────────────────────────────────────┤
│ More info     │ https://npmjs.com/advisories/786                             │
└───────────────┴──────────────────────────────────────────────────────────────┘

...
```

```
found 416 vulnerabilities (65 low, 2 moderate, 348 high, 1 critical) in 20047 scanned packages
  run `npm audit fix` to fix 354 of them.
  62 vulnerabilities require semver-major dependency updates.
```

After only one year the code is full of small security threats. Luckily there is only 1 critical threat. Let's run `npm audit fix` as the report suggests:

```
$ bloglist-backend npm audit fix

+ mongoose@5.9.1
added 19 packages from 8 contributors, removed 8 packages and updated 15 packages in 7.325s
fixed 354 of 416 vulnerabilities in 20047 scanned packages
  1 package update for 62 vulns involved breaking changes
  (use `npm audit fix --force` to install breaking changes; or refer to `npm audit` for steps to fix tl
```

62 threats remain because by default `audit fix` does not update dependencies if their *major* version number has increased. Updating these dependencies could lead to the whole application breaking down. The remaining threats are caused by the testing dependency jest. Our application has the version 23.6.0 when the secure version is 25.0.1. As jest is a development dependency the threat is actually nonexistent, but let's update it just to be safe:

```
npm install --save-dev jest@25.1.0
```

After the update the situation looks good

```
 $ blogs-backend npm audit

                       === npm audit security report ===

found 0 vulnerabilities
 in 1204443 scanned packages
```

One of the threats mentioned in the list from OWASP is *Broken Authentication* and the related *Broken Access Control*. The token based authentication we have been using is fairly robust, if the application is being used on the traffic-encrypting HTTPS-protocol. When implementing access control one should e.g. remember to not only check a user's identity in the browser but also on the server. Bad security would be to prevent some actions to be taken only by hiding the execution options in the code of the browser.

On Mozilla's MDN there is a very good Website security -guide, which brings up this very important topic:

> Important: The single most important lesson you can learn about website security is to **never trust data from the browser**. This includes, but is not limited to data in URL parameters of `GET` requests, `POST` requests, HTTP headers and cookies, and user-uploaded files. Always check and sanitize all incoming data. Always assume the worst.

The documentation for Express includes a section on security: Production Best Practices: Security, which is worth a read through. It is also recommended to add a library called Helmet to the backend. It includes a set of middlewares that eliminate some security vulnerabilities in Express applications.

Using the ESlint security-plugin is also worth doing.

Current trends

Finally, let's take a look at some technology of tomorrow (or actually already today), and directions Web development is heading.

Typed versions of JavaScript

Sometimes the dynamic typing of JavaScript variables creates annoying bugs. In part 5 we talked briefly about PropTypes: a mechanism which enables one to enforce type checking for props passed to React-components.

Lately there has been a notable uplift in the interest in static type checking. At the moment the most popular typed version of Javascript is Typescript which has been developed by Microsoft. Typescript is covered in part 9.

Server side rendering, isomorphic applications and universal code

The browser is not the only domain where components defined using React can be rendered. The rendering can also be done on the server. This kind of approach is increasingly being used, such that when accessing the application for the first time the server serves a pre-rendered page made with React. From here onwards the operation of the application continues as usual, meaning the browser executes React, which manipulates the DOM shown by the browser. The rendering that is done on the server goes by the name: *server side rendering*.

One motivation for server side rendering is Search Engine Optimization (SEO). Search engines have traditionally been very bad at recognizing JavaScript rendered content, however, the tide might be turning, e.g. take a look at this and this.

Of course, server side rendering is not anything specific to React or even JavaScript. Using the same programming language throughout the stack in theory simplifies the execution of the concept, because the same code can be run on both the front- and backend.

Along with server side rendering there has been talk of so-called *isomorphic applications* and *universal code*, although there has been some debate about their definitions. According to some definitions an isomorphic web application is one that performs rendering on both the front- and backend. On the other hand, universal code is code that can be executed in most environments, meaning both the frontend and the backend.

React and Node provide a desirable option for implementing an isomorphic application as universal code.

Writing universal code directly using React is currently still pretty cumbersome. Lately a library called Next.js, which is implemented on top of React, has garnered much attention and is a good option for making universal applications.

Progressive web apps

Lately people have started using the term progressive web app (PWA) launched by Google.

In short, we are talking about web-applications, working as well as possible on every platform taking advantage of the best parts of those platforms. The smaller screen of mobile devices must not hamper the usability of the application. PWAs should also work flawlessly in offline-mode or with a slow internet connection. On mobile devices they must be installable just like any other application. All the network traffic in a PWA should be encrypted.

Applications created using create-react-app are progressive by default. If the application uses data from a server, making it progressive takes work. The offline functionality is usually implemented with the help of service workers.

Microservice architecture

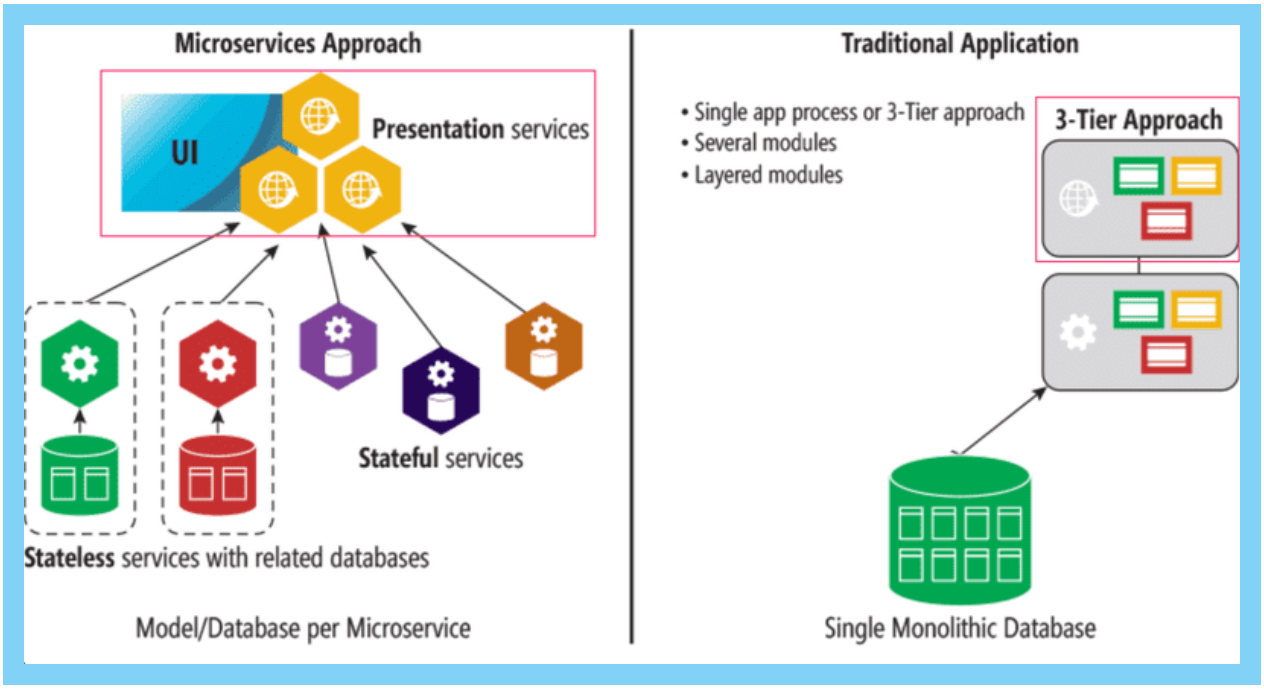
During this course we have only scratched the surface of the server end of things. In our applications we had a *monolithic* backend, meaning one application making up a whole and running on a single server, serving only a few API-endpoints.

As the application grows the monolithic backend approach starts turning problematic both in terms of performance and maintainability.

A microservice architecture (microservices) is a way of composing the backend of an application from many separate, independent services, which communicate with each other over the network. An individual microservice's purpose is to take care of a particular logical functional whole. In a pure microservice architecture the services do not use a shared database.

For example, the bloglist application could consist of two services: one handling user and another taking care of the blogs. The responsibility of the user service would be user registration and user authentication, while the blog service would take care of operations related to the blogs.

The image below visualizes the difference between the structure of an application based on a microservice architecture and one based on a more traditional monolithic structure:



The role of the frontend (enclosed by a square in the picture) does not differ much between the two models. There is often a so-called API gateway between the microservices and the frontend, which provides an illusion of a more traditional "everything on the same server"-API. Netflix, among others, uses this type of approach.

Microservice architectures emerged and evolved for the needs of large internet-scale applications. The trend was set by Amazon far before the appearance of the term microservice. The critical starting point was an email sent to all employees in 2002 by Amazon CEO Jeff Bezos:

> All teams will henceforth expose their data and functionality through service interfaces.
>
> Teams must communicate with each other through these interfaces.
>
> There will be no other form of inter-process communication allowed: no direct linking, no direct reads of another team's data store, no shared-memory model, no back-doors whatsoever. The only communication allowed is via service interface calls over the network.
>
> It doesn't matter what technology you use.
>
> All service interfaces, without exception, must be designed from the ground up to be externalize-able. That is to say, the team must plan and design to be able to expose the interface to developers in the outside world.
>
> No exceptions.
>
> Anyone who doesn't do this will be fired. Thank you; have a nice day!

Nowadays one of the biggest forerunners in the use of microservices is Netflix.

The use of microservices has steadily been gaining hype to be kind of a silver bullet of today, which is being offered as a solution to almost every kind of problem. However, there are a number of challenges when it comes to applying a microservice architecture, and it might make sense to go monolith first by initially making a traditional all encompassing backend. Or maybe not. There are a bunch of different opinions on the subject. Both links lead to Martin Fowler's site; as we can

see, even the wise are not entirely sure which one of the right ways is more right.

Unfortunately, we cannot dive deeper into this important topic during this course. Even a cursory look at the topic would require at least 5 more weeks.

Serverless

After the release of Amazon's lambda-service at the end of 2014 a new trend started to emerge in web-application development: serverless.

The main thing about lambda, and nowadays also Google's Cloud functions as well as similar functionality in Azure, is that it enables *the execution of individual functions* in the cloud. Before, the smallest executable unit in the cloud was a single *process*, e.g. a runtime environment running a Node backend.

E.g. Using Amazon's API-gateway it is possible to make serverless applications where the requests to the defined HTTP API get responses directly from cloud functions. Usually the functions already operate using stored data in the databases of the cloud service.

Serverless is not about there not being a server in applications, but about how the server is defined. Software developer can shift their programming efforts to a higher level of abstraction as there is no longer a need to programmatically define the routing of HTTP-requests, database relations, etc., since the cloud infrastructure provides all of this. Cloud functions also lend themselves to creating well scaling system, e.g. Amazon's Lambda can execute a massive amount of cloud functions per second. All of this happens automatically through the infrastructure and there is no need to initiate new servers, etc.

Useful libraries and interesting links

The JavaScript developer community has produced a large variety of useful libraries. If you are developing anything more substantial, it is worth it to check if existing solutions are already available. One good place to find libraries is https://applibslist.xyz/. Below is listed some libraries recommended by trustworthy parties.

If your application has to handle complicated data lodash, which we recommended in part 4, is a good library to use. If you prefer functional programming style, you might consider using ramda.

If you are handling times and dates, date-fns offers good tools for that.

Formik and redux-form can be used to handle forms easier. If your application displays graphs, there are multiple options to chose from. Both recharts and highcharts are well recommended.

The immutable.js-library maintained by Facebook provides, as the name suggests, immutable implementations of some data structures. The library could be of use when using Redux, since as we remember from part 6: reducers must be pure functions, meaning they must not modify the store's state but instead have to replace it with a new one when a change occurs. Over the past year some of the popularity of Immutable.js has been taken over by Immer, which provides similar functionality but in a somewhat easier package.

Redux-saga provides an alternative way to make asynchronous actions for redux thunk familiar from part 6. Some embrace the hype and like it. I don't.

For single page applications the gathering of analytics data on the interaction between the users and the page is more challenging than for traditional web-applications where the entire page is loaded. The React Google Analytics -library offers a solution.

You can take advantage of your React know-how when developing mobile applications using Facebook's extremely popular React Native -library, that is topic of the part 10 of the course.

When it comes to the tools used for the management and bundling of JavaScript projects the community has been very fickle. Best practices have changed rapidly (the years are approximations, nobody remembers that far back in the past):

- 2011 Bower
- 2012 Grunt
- 2013-14 Gulp
- 2012-14 Browserify
- 2015- Webpack

Hipsters seem to have lost their interest in tool development after webpack started to dominate the markets. Few years ago Parcel started to make the rounds marketing itself as simple (which Webpack absolutely is not) and faster than Webpack. However after a promising start Parcel has not gathered any steam, and it's beginning to look like it will not be the end of Webpack.

Another notable mention is the Rome library, which aspires to be an all-encompassing toolchain to unify linter, compiler, bundler, and more. It is currently under heavy development since the initial commit earlier this year on Feb 27, but the outlook sure seems promising.

The site https://reactpatterns.com/ provides a concise list of best practices for React, some of which are already familiar from this course. Another similar list is react bits.

Reactiflux is a big chat community of React developers on Discord. It could be one possible place to get support after the course has concluded. For example numerous libraries have their own channels.

If you know some recommendable links or libraries, make a pull request!

About course

Propose changes to material

Course contents

Previous part **Part 7d**

Next part **Part 7f**

FAQ

Partners

Challenge


HOUSTON