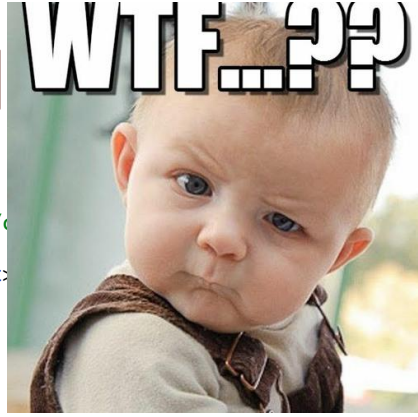
06 – Frontend Development

Web Technology Project (International Computer Science)
Summer semester 2025
Prof. Dr. Felix Schwägerl

We know a few things about HTML, CSS and JavaScript now. Let‘s have a look at the Mensa frontend!

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1" />
  <meta name="color-scheme" content="light dark" />
  <link rel="stylesheet" href="https://cdn.jsdelivr.net/npm/@picocss/pico@2/
  <title>Mensa</title>
  <script type="module" crossorigin src="/assets/index-CRQ5pUdE.js"></script>
  <link rel="stylesheet" crossorigin href="/assets/index-kB8aWevh.css">
</head>
<body>
<div id="root"></div>
</body>
</html>
```

That‘s all? Where are my HTML tags?

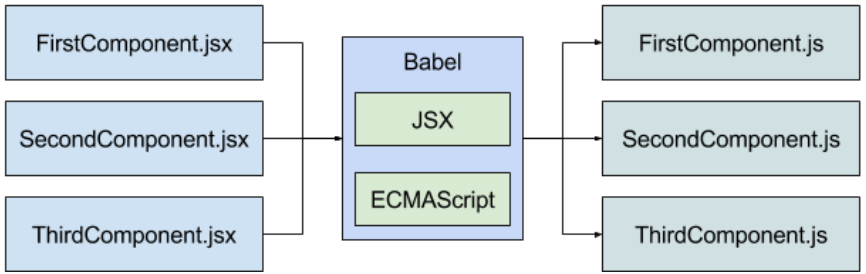


What the fact – Who created this ugly JavaScript code?

```
var Wd;function B0(){return Wd||(Wd=1,function(S){function q(R,K){var Q=R.length;R.push(K);l:for(;0<Q;){var
ol=Q-1>>>1,s=R[ol];if(0<D(s,K))R[ol]=K,R[Q]=s,Q=ol;else break l}}function p(R){return R.length===0?null:R[0]}
function o(R){if(R.length===0)return null;var K=R[0],Q=R.pop();if(Q!==K){R[0]=Q;l:for(var ol=0,s=R.length,z=
s>>>1;ol<z;){var X=2*(ol+1)-1,Y=R[X],j=X+1,tl=R[j];if(0>D(Y,Q))j<s&&0>D(tl,Y)?(R[ol]=tl,R[j]=Q,ol=j):(R[ol]=
Y,R[X]=Q,ol=X);else if(j<s&&0>D(tl,Q))R[ol]=tl,R[j]=Q,ol=j;else break l}}return K}function D(R,K){var Q=R.
sortIndex-K.sortIndex;return Q!==0?Q:R.id-K.id}if(S.unstable_now=void 0,typeof performance=="object"&&typeof
performance.now=="function"){var _=performance;S.unstable_now=function(){return _.now()}}else{var V=Date,I=V
.now();S.unstable_now=function(){return V.now()-I}}var M=[],g=[],O=1,Z=null,B=3,U=!1,el=!1,Ol=!1,bl=typeof
```

Standard JavaScript is generated by a set of tools on top.

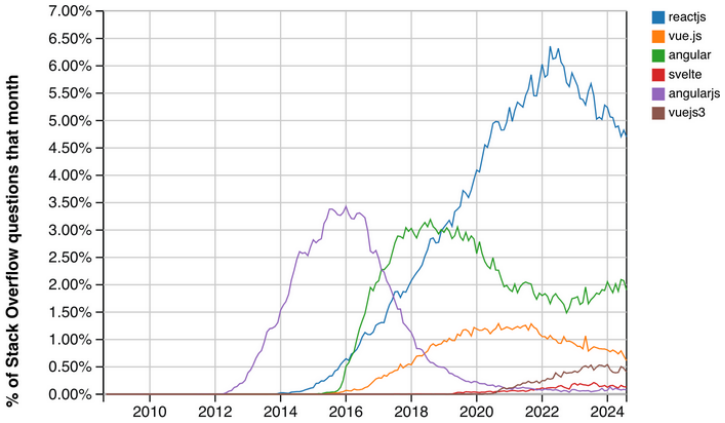
- **Babel** transpiler (converts next-generation JS into browser-compatible JS)
  - Modern browsers guarantee to execute *ECMA6* JavaScript, but JS still evolves.
  - Babel may also convert, e.g., *TypeScript* or *JSX* (→) into browser-compatible JS.
  - Includes a *minify* step, which compresses and obfuscates (←) source code
  - By default, source code is contained in a folder `src`, and the transpile output in `dist`
- **npm** (node package manager)
  - Adds dependency management and built automation (think: Maven for JS)
  - The build parameters and dependencies are specified in a file `package.json`
  - Dependencies are automatically managed in a folder `node_modules`
- **ESLint** (static code analyzer):
  - Adds development-time code analysis, so problems are detected before JS execution
  - Configuration file: `eslint.config.js`



[Boduch 2018]

React is a JavaScript library for building user interfaces.

- **Declarative syntax**:
  - Developers need not specify DOM manipulation operations manually, but they are derived from declarative and state-based specifications.
- **Component-based:**
  - The frontend application (or parts thereof) is decomposed into re-usable components, from which complex UIs can be composed.
- **Cross-platform**:
  - React source code may also be compiled into native desktop or mobile applications.
- Widespread adoption in industry →
- Open source project, created upon initiative by Meta Inc.

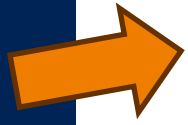


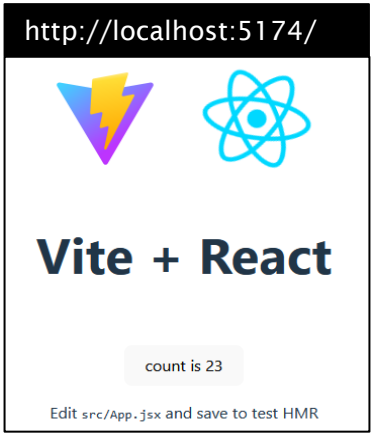
[https://trends.stackoverflow.co/?tags=reactjs%2Cvue.js%2Cangular%2Csvelte%2Cangularjs%2Cvuejs3]

Vite is a development server for React (and other JS libraries).

- Initializing and starting a Vite/React project from the command line (here: Windows PowerShell)

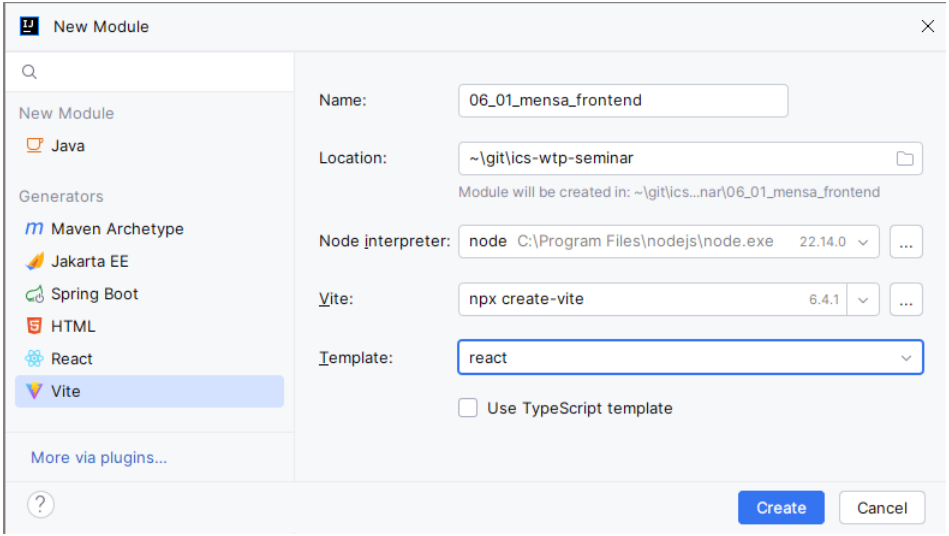
```
PS C:\Users\scx> npm create vite@latest my-react-app -- --template vue

> npx
> create-vite my-react-app vue

◇  Select a framework:
│  React
│
◇  Select a variant:
│  JavaScript
│
◇  Scaffolding project in C:\Users\scx\my-react-app...
│
└  Done. Now run:

  cd my-react-app
  npm install
  npm run dev

PS C:\Users\scx> cd .\my-react-app\
PS C:\Users\scx\my-react-app> npm install

added 152 packages, and audited 153 packages in 8s

32 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
PS C:\Users\scx\my-react-app> npm run dev

> my-react-app@0.0.0 dev
> vite

Port 5173 is in use, trying another one...

  VITE v6.3.4  ready in 203 ms

  ➜  Local:   http://localhost:5174/
  ➜  Network: use --host to expose
  ➜  press h + enter to show help
```

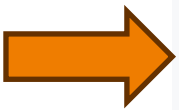


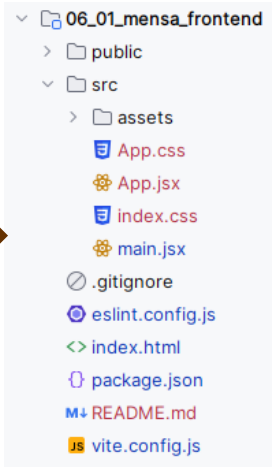


- Application reloads automatically when its source code is updated.
- Browser console logs Vite activity and React problems:

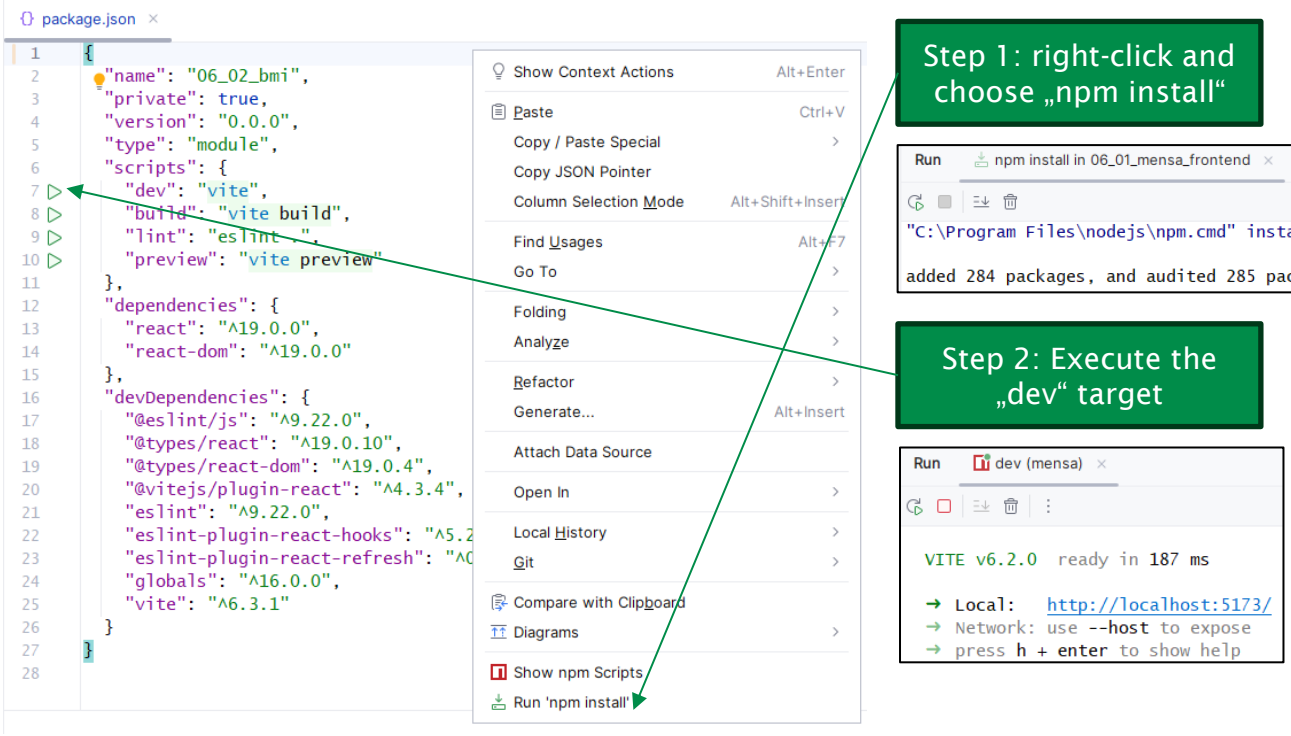
New → Module → Vite





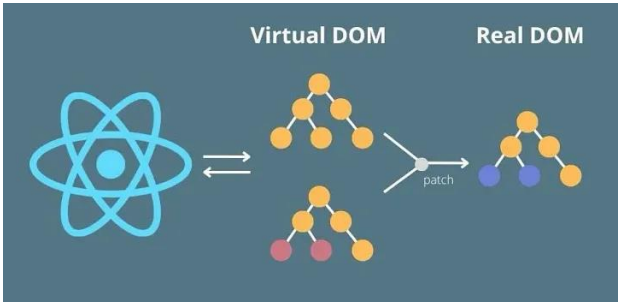


Opening `package.json` in IntelliJ, you may run all `npm` commands without using the command line.



[https://levelup.gitconnected.com/inside-react-the-virtual-dom-7d801f6c8e8f]

React is based on *components* and a *virtual DOM.*



- **Virtual DOM**: Instead of manipulating the DOM directly, the developer defines a virtual DOM, which is kept in sync with the actual DOM by the React library.
  - More declarative code, better performance
- **JSX**: The virtual DOM elements are described using a JavaScript syntax extension that looks like HTML in JavaScript. JSX is, however, not 100% compatible with HTML.
- **Components**: React applications are built from re-usable components, each defining a small piece of the DOM. Components are just JavaScript/JSX functions.
- **Props**: Components share information through special variables. Every component is parameterized with a set of named props.
- **Event listeners**: Browser events can be handled by event listeners, which are more flexible and less error prone than in standard JavaScript.
- **Hooks**: Pre-defined patterns for integrating data, view, and external systems:
  - *State hooks*: Variables based on whose change a component is re-rendered
  - *Ref hooks*: References to elements of the DOM
  - *Effect hooks*: Connections to external systems, e.g., API calls
  - *Context hooks*: Global state components may subscribe to

JSX looks like HTML with templates.

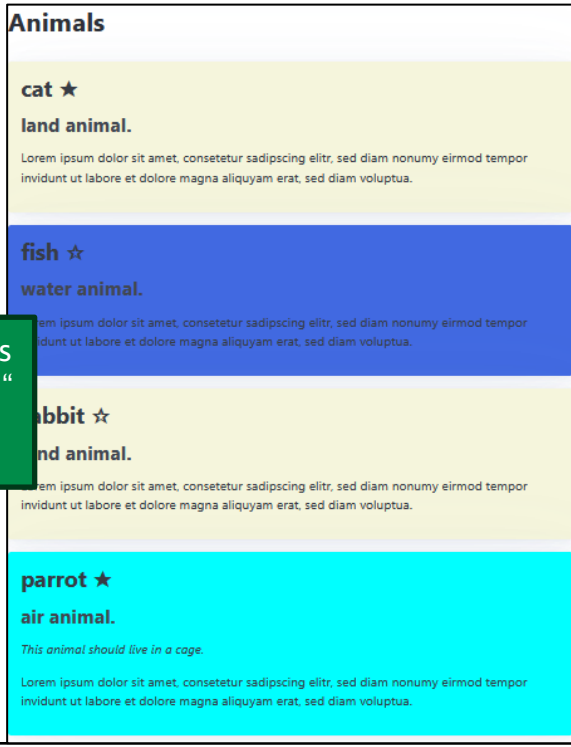
```
const animal1 = {name: "cat", type: "land", favorite: true};
…
const animal4 = {name: "parrot", type: "air", …};
const copy = "Lorem ipsum dolor sit …";
```

```
<>
    <header>
        <h1>Animals</h1>
    </header>
    <body>
        <article className={animal1.type}>
            <h2>{animal1.name} ★</h2>
            <h3>{animal1.type} animal.</h3>
            <p>{copy}</p>
        </article>
        …
        <article className={animal4.type}>
            <h2>{animal4.name} ★</h2>
            <h3>{animal4.type} animal.</h3>
            <p><em>…in a cage.</em></p>
            <p>{copy}</p>
        </article>
    </body>
</>;
```

Access contents of variables in scope

Set CSS class („className“ instead of „class“)

Unnamed root element containing JSX structure



JSX offers generalized control structures.

- If/else: {*condition ? conditionalContent : alternativeContent*}
- If without else: {*condition && conditionalContent*}
- Loop over array: {*array.map(a => repeatedContent(a))*}
- Nested content may contain JSX (and nested content again)

```
const animals = [
    {name: "cat", type: "land", favorite: true}, …
    {name: "parrot", type: "air", favorite: true},
];
const copy = "Lorem ipsum dolor sit amet, …";

return <>
    <header>
        <h1>Animals</h1>
    </header>
    <body>{animals.map(a =>
        <article className={a.type}>
            <h2>{a.name} {a.favorite ? "★" : "☆"}</h2>
            <h3>{a.type} animal.</h3>
            {a.type === "air" && <p><em>…in a cage.</em></p>}
            <p>{copy}</p>
        </article>
    )}
    </body>
</>;
```

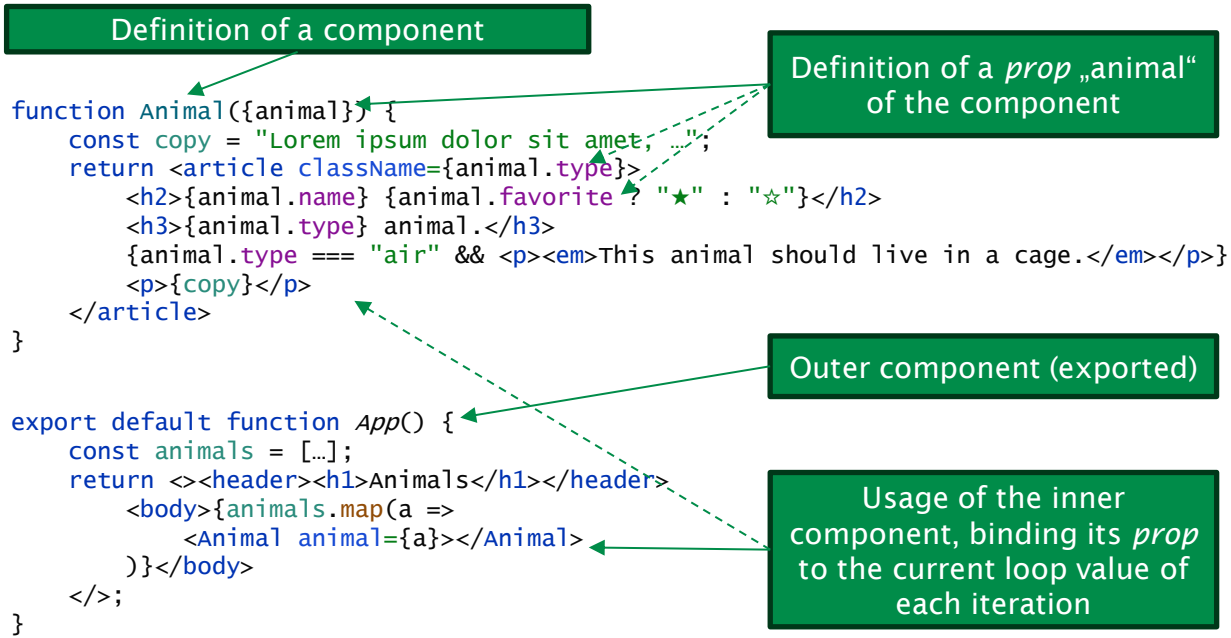
Loop over objects of an array

If/else case distinction based on current loop element

Conditional content for current element

A React component is a function returning JSX.

- Components' UIs are typically nested within each other.
- A re-used component looks like HTML (but has a first-upper-case name!)




```
function Animal({animal}) {
    const copy = "Lorem ipsum dolor sit amet, …";
    return <article className={animal.type}>
        <h2>{animal.name} {animal.favorite ? "★" : "☆"}</h2>
        <h3>{animal.type} animal.</h3>
        {animal.type === "air" && <p><em>This animal should live in a cage.</em></p>}
        <p>{copy}</p>
    </article>
}

export default function App() {
    const animals = […];
    return <><header><h1>Animals</h1></header>
        <body>{animals.map(a =>
            <Animal animal={a}></Animal>
        )}</body>
    </>;
}
```

Components use *props* to exchange information along their hierarchy.

- In the component definition, props are always passed as one object in the first parameter.

```
export default function Component(props) {
    return <div>{props.prop1}</div>;
}
```

- *Object destructuring* makes the props directly available inside the component:

```
export default function Component({prop1, prop2, prop3}) {
    return <div>{prop1}</div>;
}
```

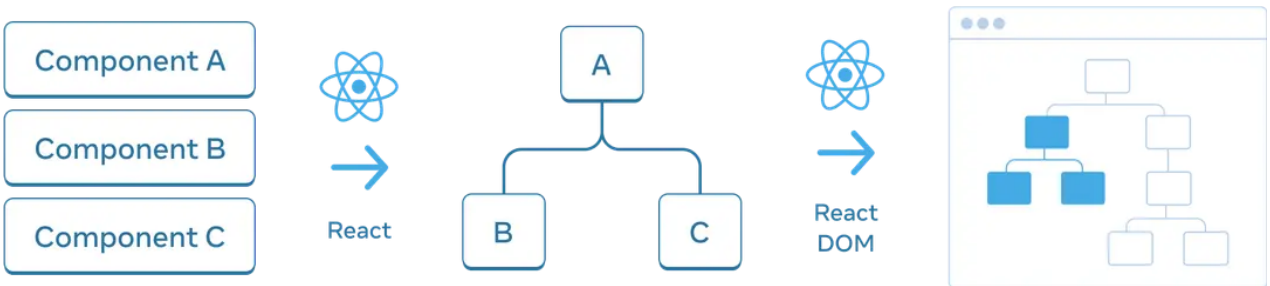
- When creating the component in JSX, props are passed as XML-like attributes:

```
<Component prop1={value1} prop2={value2} prop3={value3}/>
```

- Anything can be a prop, even *hooks* (→)

Use react components to …

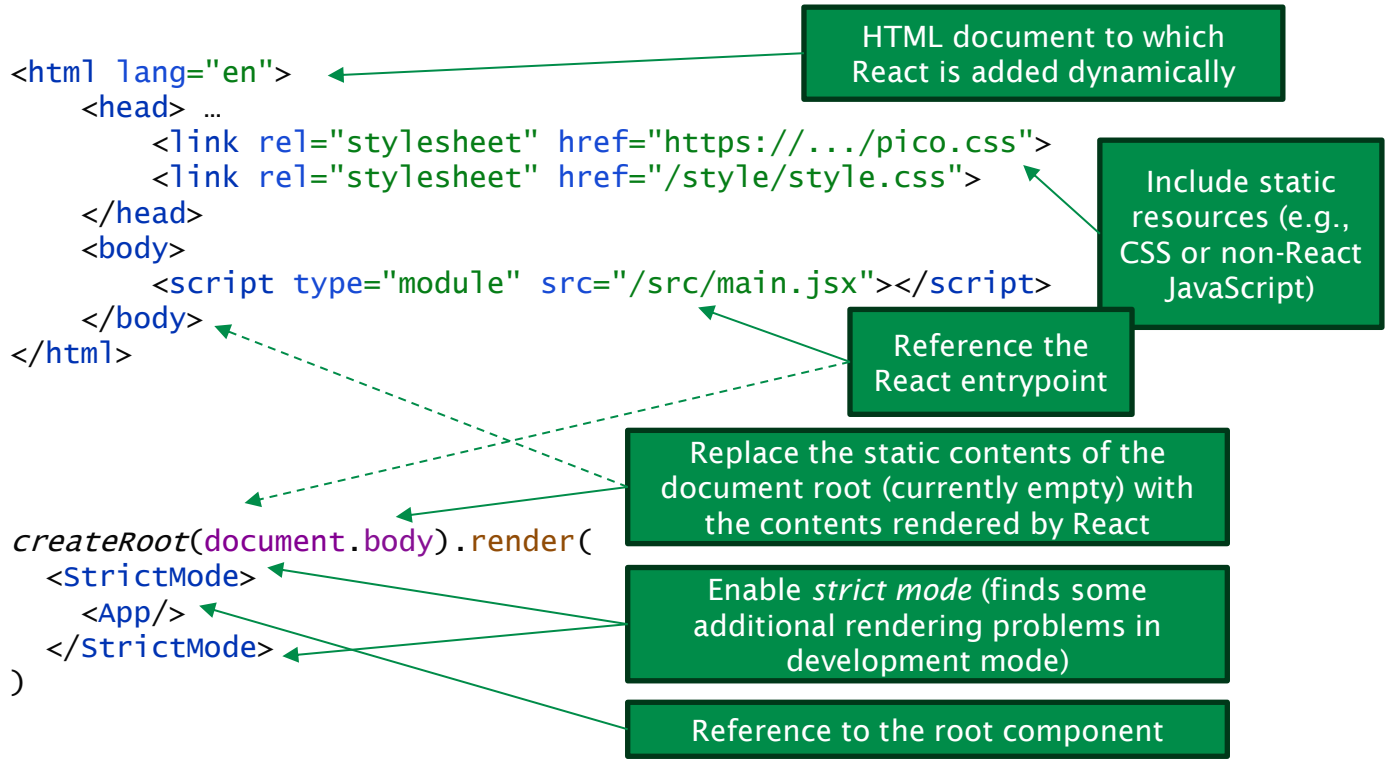
- Decompose your application into re-usable pieces.
- Represent repeatable elements.
- Encapsulate UI elements belonging together (e.g., inputs and buttons of a form).
- Define units of re-rendering on demand.
- Keep your code base clean and modularized.



[https://react.dev/learn/understanding-your-ui-as-a-tree]

Binding the React application to the DOM

The actual DOM is initially empty and filled by React.

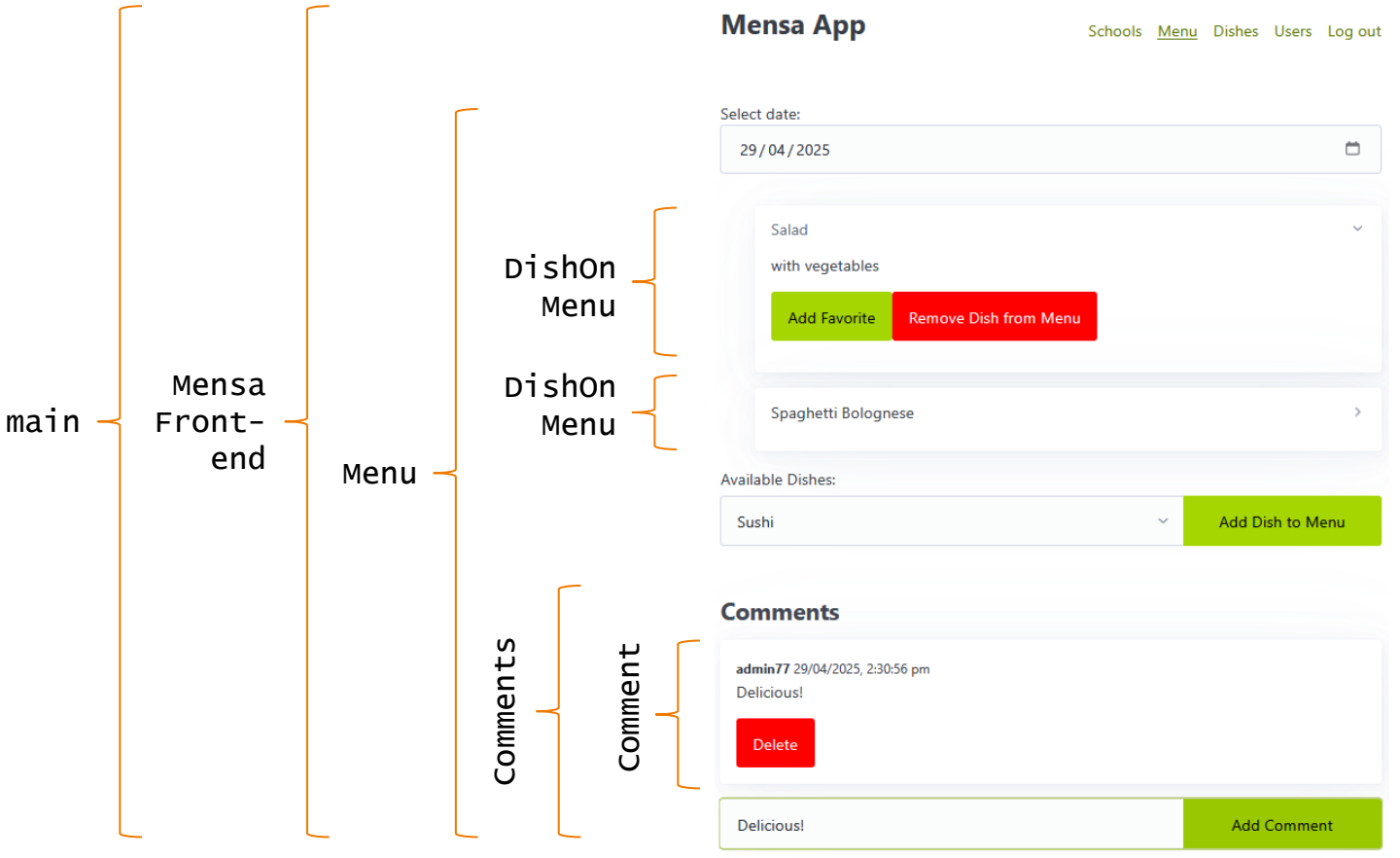


Disabling react prop type validation

- By default, React projects created with Vite have prop type validation enabled.
  - This is a complicated mechanism required for, e.g., interoperability with TypeScript.
- We disable prop type validation as follows (`eslint.config.js`):

```
export default [
  { ignores: ['dist'] },
  {
    files: ['**/*.{js,jsx}'],
    languageOptions: { … },
    settings: { react: { version: '18.3' } },
    plugins: {
      react,
      'react-hooks': reactHooks,
      'react-refresh': reactRefresh,
    },
    rules: {
      …,
      'react/prop-types': 0
    },
  },
]
```

Mensa example: React components visualized



React event listeners

```
<button onClick={callback}></button>
```

- Recall: In plain JavaScript, in-line event listeners are considered bad style.
- In React, they are re-implemented, fixing the plain JavaScript listeners' disadvantages.
  - E.g., multiple event listeners can still be registered to elements.
- Syntactic differences to plain JavaScript event listeners:
  - camelCase notation, e.g., `onClick` instead of `onclick`
  - The value must be surrounded by {}
  - The value is a reference to a function rather than arbitrary JavaScript code.
    - If the event must be captured, the following notation can be used:

```
onClick={e => callback(e)}
```

- The referenced functions are *typically nested* functions of the function returning the JSX of the component:

```
export default function App() {
    function callback() { … }
    return <> … <button onClick={callback}></button> … </>;
}
```

State hooks

```
const [state, setState] = useState(initialValue)
```

- A *state hook* is a React hook that lets you add a *state variable* to your component.
- It consists of a getter (`state`) and setter (`setState`).
- The getter can be referenced like an arbitrary variable, e.g. in JSX:

```
<p>{state}</p>
```

- The setter has two main purposes:
  - Update the state to the specified value.
  - Trigger a re-rendering of the JSX structure in case the state is referenced from there.

```
function update(newState) {
    setState(newState);
}
```

- In React, updates to the DOM should never be manipulated using plain JavaScript DOM selectors and manipulation functions.
- Rather, make the JSX structure produced by a component depend on `state`, which causes an automatic re-render upon `setState`.

Ref hooks

```
const refHook = useRef(initialValue);
```

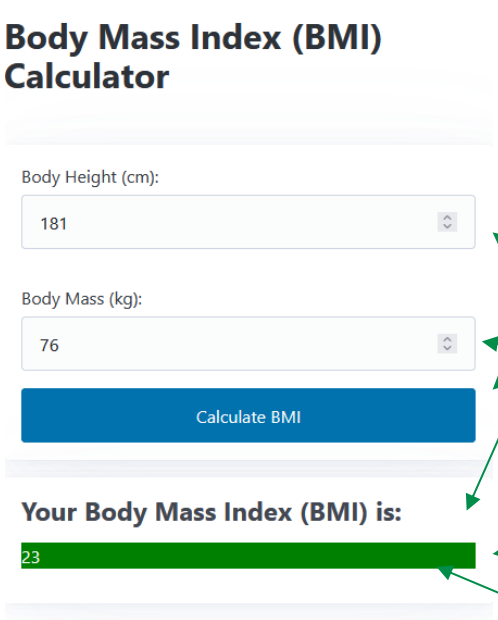
- A *ref hook* is a React hook that lets you reference a value not needed for rendering.
  - Typical use case: Referencing an input value.
- Often, a ref is bound to a DOM element:

```
<input ref={refHook}/>
```

- In contrast to state hooks, updating a ref value does not trigger a re-render of the component.
- The current value of a ref hook may be accessed within the component:

```
const val = refVal.current;
```

BMI example with React: Specification



Initial contents before button is clicked: „Please fill the form above to calculate the result."

When this button is clicked …

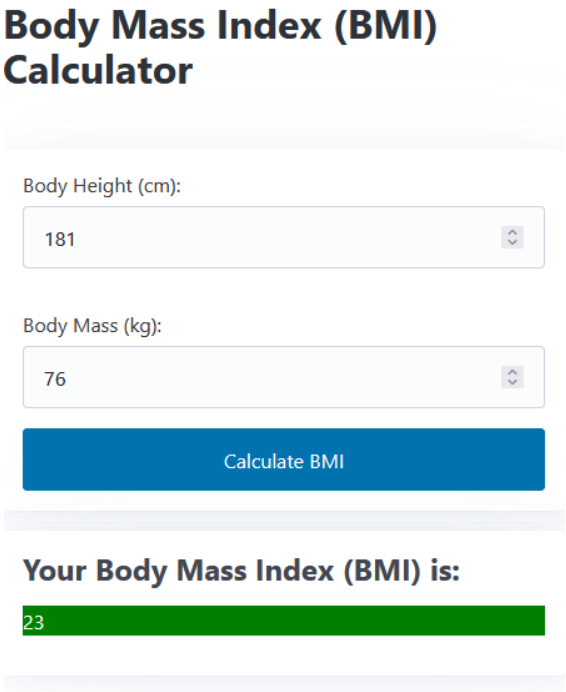
… the BMI should be calculated from these inputs …

… and the calculated result should be shown in this paragraph.

The background should be updated according to the BMI class.

BMI example: A minimal React application (1)

```
import {useRef, useState} from "react";

export default function App() {
    const height = useRef(undefined);
    const mass = useRef(undefined);
    const [bmi, setBmi] = useState(undefined);

    function updateBmi() {
        const m = parseInt(mass.current.value);
        const h = parseInt(height.current.value);
        const b = Math.round((m * 10000) / (h * h));
        setBmi(b);
    }

    function getBMIClass(bmi) {
        if (bmi < 16) return "blue";
        else if (bmi < 19) return "cyan";
        else if (bmi < 25) return "green"
        else if (bmi < 30) return "yellow";
        else if (bmi < 35) return "orange";
        else return "red";
    }
…
}
```



BMI example: A minimal React application (2)

```
export default function App() { …
    return <><header><h1>Body Mass Index (BMI) Calculator</h1></header>
        <main><article>
                <div className="grid">
                    <div>
                        <label htmlFor="body-height">Body Height (cm):</label>
                        <input id="body-height" type="number" min="100" max="220"
                               defaultValue="170" ref={height}/>
                    </div>
                    <div>
                        <label …>Body Mass (kg):</label>
                        <input id="body-mass" type="number„
                               min="30" max="200" defaultValue="70" ref={mass}/>
                    </div>
                </div>
                <div className="grid">
                    <button id="calculate" onClick={updateBmi}>Calculate BMI</button>
                </div>
            </article>
            <article>
                <h3>Your Body Mass Index (BMI) is:</h3>
                {bmi === undefined ?
                    <p>Please fill the form above ….</p> :
                    <p className={getBMIClass(bmi)}>{bmi}</p>
                }
            </article></main></>;
}
```

Effect hooks

`useEffect(() => {setup}, [dependencies]);`

- An *effect* is a React hook that lets you synchronize a component with an external system.
  - The effect is applied initially before the component is rendered.
  - Frequent use case: Call an API to fetch the initial data of a component
- The dependencies array includes other hooks (e.g., states, context) upon whose change the effect *is re-applied.*
  - If omitted, all parts of the component are considered as dependencies.
  - An effect must not depend on a state variable set by itself (→ infinite recursion)
- The setup function may optionally return a *clean-up* function.
- General pattern:

```
useEffect(() => {
    const connection = createConnection(externalSystem);
    connection.connect();
    return () => {
        connection.disconnect();
    };
}, [externalSystem]);
```

Clean-up function

Context hooks

```
export const Context = createContext(initialValue)
```

- A *context* is a React hook that lets you read and subscribe to data from the environment.
  - Examples: Environment variables (e.g., API URL), theme settings
- The current context data can be accessed via *useContext*:

```
const ctx = useContext(Context);
```

- Context hooks can be used to minimize passing around common props from component to component.

Mensa example

- API URL is passed as context, whose initial value is read from the environment.

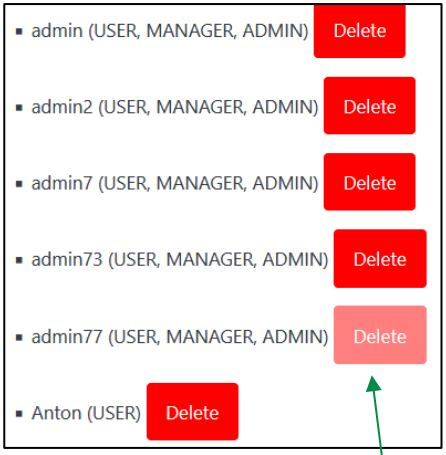
```
export const Api = createContext(import.meta.env.VITE_APP_MENSA_API)
```

- The underlying environment variable is pre-initialized as follows in a `.env` file:

```
VITE_APP_MENSA_API="http://localhost:8080"
```

Mensa example: The users component

```
export default function Users({auth}) {
    const api = useContext(Api);
    const [users, setUsers] = useState([])

    useEffect(() => {
        fetch(api + "/users", {headers: basic(auth)})
            .then(response => {
                if (response.ok) return response.json();
                else throw new Error(response.statusText); })
            .then(result => { setUsers(result); });
    }, [api]);

    function deleteUser(name) {
        fetch(api + "/users/" + name,
            {method: "DELETE", headers: basic(auth)})
            .then(response => {
                if (!response.ok) throw new Error(response.statusText);
            }).then(() => { setUsers(users.filter(u => u.name !== name)); });
    }

    return <ul>{users.map(u =>
        <li key={u.name}>
            <div className="grid">
                <p>{u.name} ({u.roles.join(", ")})</p>
                <button className="delete" disabled={u.name === auth.name}
                        onClick={() => deleteUser(u.name)}>Delete</button>
            </div>
        </li>
    )}</ul>;
}
```



Mensa example: Menu and nested components

```jsx
export default function Menu({auth, schoolId}) {
    const api = useContext(Api);
    const [day, setDay] = useState(new Date().toISOString().slice(0, 10));
    const [menu, setMenu] = useState({dishes: []});
    const [availableDishes, setAvailableDishes] = useState([]);
    const dishIdToAdd = useRef(undefined);

    useEffect(() => {
        fetch(api + "/schools/" + schoolId + "/menu?day=" + day, {headers: basic(auth)})
            … setMenu(…) … }, [api, day]);

    useEffect(() => {
        fetch(api + "/schools/" + schoolId + "/dishes", {headers: basic(auth)})
            … setAvailableDishes(…) … }, [api, day, menu]);


    function updateMenu(newMenu) {…}
    function addSelectedDish() {…}

    return <>
        <p>Select date: <input type="date" defaultValue={day} onChange={d => setDay(d.target.value)}/></p>
        <ul>{menu.dishes.map(d =>
            <DishOnMenu key={d.id} auth={auth} schoolId={schoolId} menu={menu} setMenu={setMenu} dish={d}/>
        )}</ul>
        {auth.roles.includes("MANAGER") && <>
            <label htmlFor="add-dish">Available Dishes:</label>
            <div className="grid">
                <select id="add-dish" ref={dishIdToAdd}>
                    {availableDishes.map(ad => <option value={ad.id} key={ad.id}>{ad.name}</option>)}
                </select>
                <button className="large" onClick={addSelectedDish}>Add Dish to Menu</button>
            </div>
        </>}
        <Comments auth={auth} schoolId={schoolId} day={day}></Comments>
    </>;
}
```

„htmlFor“ instead of „for“

Mensa example: Root navigation

```
export default function MensaFrontend() {
    const [auth, setAuth] = useState({name: null, password: null, roles: [], loggedIn: false});
    const [schoolId, setSchoolId] = useState(undefined);
    const [view, setView] = useState("schools");

    return <><header><nav>
            <h2>Mensa App</h2>
            <ul>
                <li><a href="#" className={view === "schools" ? "current" : "default"}
                   onClick={() => setView("schools")}>Schools</a></li>
                {schoolId !== undefined && auth.roles.includes("USER") && <li>
                   <a href="#" className={view === "menu" ? "current" : "default"}
                       onClick={() => setView("menu")}>Menu</a></li>}
                {schoolId !== undefined && auth.roles.includes("USER") && <li>
                   <a href="#" className={view === "dishes" ? "current" : "default"}
                       onClick={() => setView("dishes")}>Dishes</a></li>}
                {auth.roles.includes("ADMIN") && <li>
                   <a href="#" className={view === "users" ? "current" : "default"}
                       onClick={() => setView("users")}>Users</a></li>}
                <li><a href="#" className={view === "login" ? "current" : "default"}
                       onClick={() => setView("login")}>{auth.loggedIn ? "Log out" : "Log in"}</a></li>
            </ul>
        </nav></header>
        <main>
            {view === "schools" ? <Schools auth={auth} schoolId={schoolId} setSchoolId={setSchoolId}/>
                : view === "menu" ? <Menu auth={auth} schoolId={schoolId}/>
                    : view === "dishes" ? <Dishes auth={auth} schoolId={schoolId}/>
                        : view === "users" ? <Users auth={auth} schoolId={schoolId}/>
                            : <Login auth={auth} setAuth={setAuth} setSchoolId={setSchoolId}/>
            }
        </main>
        <footer>…</footer>
    </>;
}
```

Current view

Current school ID

Current user / credentials

Selected view receives class current

Some nav items have preconditions

Choice of child component depends on current view

Relevant hooks are passed down to child components as props

Mensa example: Authentication

```
export default function Login({auth, setAuth, setSchoolId}) {
    const api = useContext(Api);
    const [createAccount, setCreateAccount] = useState(false);
    const name = useRef(undefined);
    const password = useRef(undefined);
    const role = useRef(undefined);

    function logIn() {
        const newAuth = {name: name.current.value, password: password.current.value}
        fetch(api + "/users/login", {method: "POST", headers: basic(newAuth)}).then(response => {
            if (response.ok) return response.json();
            else throw new Error(response.statusText);
        }).then(result => {
            newAuth.roles = result.roles;
            newAuth.loggedIn = true;
            setAuth(newAuth);
        });
    }

    if (auth.loggedIn) {
        return <>
            <p>Currently logged in as: {auth.name}</p>
            <button onClick={logOut}>Log out</button>
        </>;
    } else {
        return <> …
            {createAccount ? <> … <button onClick={register}>Register</button> … </> :
                             <button onClick={logIn}>Log in</button>}
        </>;
    }
}
```

Authorization header is generated by utility module

Calling setter from parent state hook

Explicit case distinction around returned JSX

Besides React components, a React code base typically contains plain JavaScript utility modules.

```
function makeBasic(auth) {
    return "Basic " + btoa(auth.name + ":" + auth.password);
}

export function basic(auth) {
    return {"Authorization": makeBasic(auth)};
}

export function anonJson() {
    return {"Content-Type": "application/json"}
}

export function basicJson(auth) {
    return {...basic(auth), ...anonJson()}
}
```

Re-usable functions for creating HTTP headers (auth, content type) for fetch function calls

New requirement: Update schools

- We switch back to our *feature branch* `feature/update-schools` from chapter 02
  - We've added a method **`PUT /schools/{id}`**
- We extend our UI by a possibility to *update* schools.
  - The only relevant property so far is the school's *name*. (IDs shouldn't be updated)
- We implement the requirement in the existing component `Schools.jsx`.
  - Updating schools requires `ADMIN` permissions.
- We execute the frontend locally and test manually.

React is a powerful framework for implementing Web frontends

- Declarative, component-based, cross-platform
- *JSX* allows to describe components in a state-based way.
  - A React component is a JS function that returns JSX code.
  - JSX looks like HTML, but it actually describes DOM manipulations.
- *Components* are parameterized by *props*.
- *Hooks* control the (re-)rendering of the UI.
  - State hooks, ref hooks, effect hooks, context hooks
  - React maintains a virtual DOM in the background, which is synchronized with the actual DOM on demand (e.g., when the state of a component is updated).
  - API calls are made on demand, controlled by effect hooks.
- React elements not considered here:
  - Advanced and user-defined hooks
  - Usage of third-party components and libraries
    - Integration of specific UI frameworks (e.g., Bootstrap)
  - Props validation
  - TypeScript (a type-safe variant of JavaScript)
  - React for native and mobile applications
  - Automated *testing* of frontend applications

References

- [Hinkula 2022] Juha Hinkula: Full Stack Development with Spring Boot 3 and React, Packt, 2022
- [Boduch 2018] Adam Boduch: React 16 Tooling, Packt, 2018
- [Mozilla 2025] Mozilla Developer Network (MDN): HTTP web docs, https://developer.mozilla.org/en-US/docs/Web/HTTP
- [React 2025] React documentation: https://react.dev/