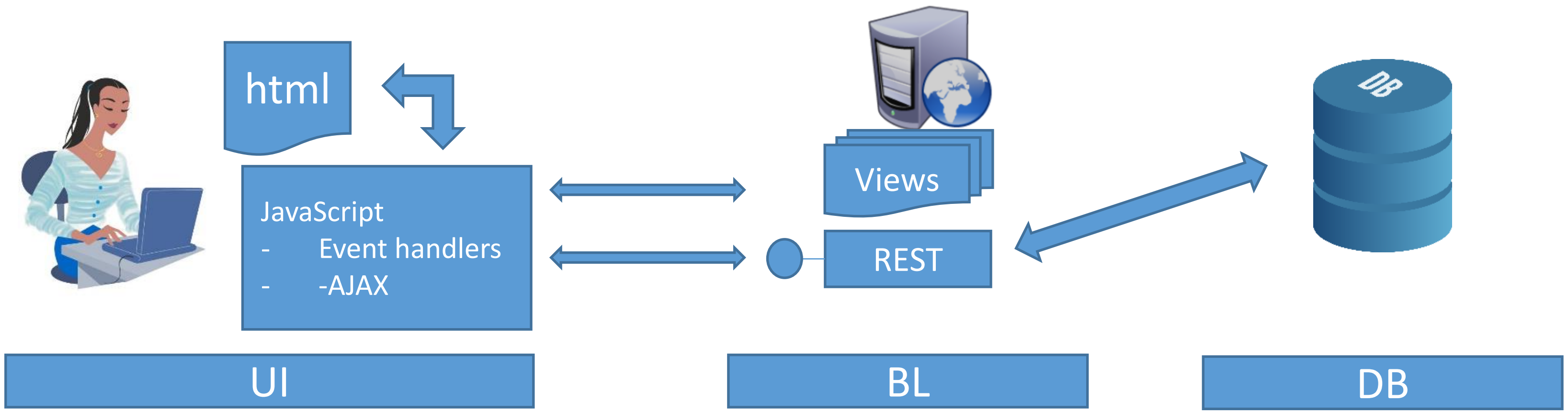
ReactJS

ReactJS-techniques for implementing SPA-applications

Background

- ReactJS is a framework for building SPA-applications
  - Initially released to public in 2013
  - Current version 18.x.y (before 15 there was 0.14.x)
- Originally developed by Facebook
- Currently one of the most popular JavaScript-frameworks
  - Very light weight
  - Easy to get started with
  - Yet powerful and versatile

Single Page Applications



- Application is built with html, css and JavaScript
- JavaScript handles events caused by user actions
  - Loads and updates data with AJAX
  - Changes views
  - Manipulates UI

- Web-server hosts the application
  - Html-page
  - Images
- Views are served by web-server
  - Html-fragments
  - Forms
  - Listings
- Service interface to data
  - Data validations
  - Security

- Data-storage

How do you build your UI

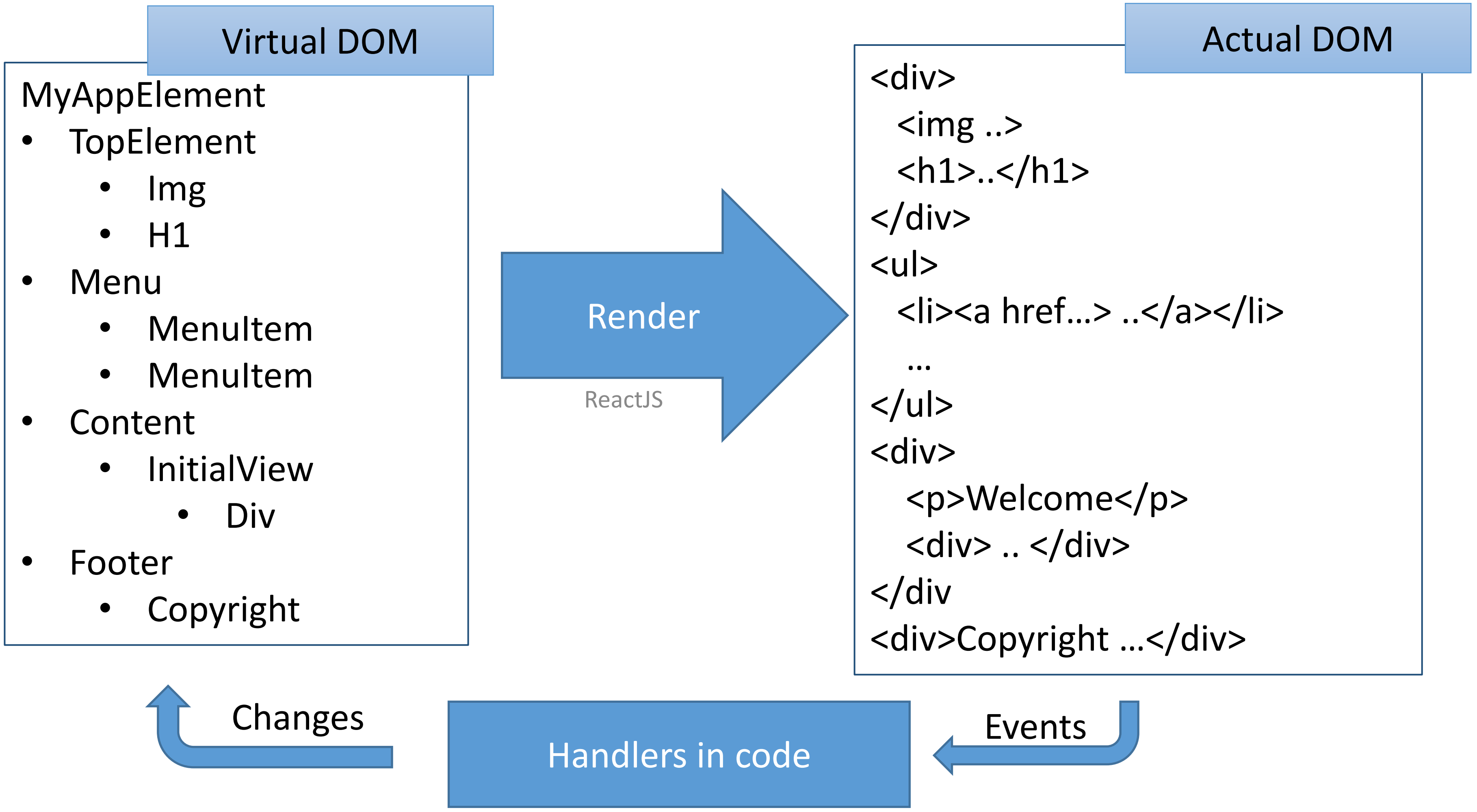
UI is traditionally built from reusable UI-components

- A component may be very simple
  - Display data
  - Allow editing with some intelligence
  - Display list
  - Etc
- Component may be a container
  - Container holds other components
  - Adds intelligence to behavior of the component group
- Container may be a full window
  - Form with components and intelligence for as specific use-case

This is basically also the approach introduced in ReactJS

ReactJS Architecture

In our code we create and manipulate a hierarchy of ReactElements.



MyAppElement
- TopElement
  - Img
  - H1
- Menu
  - MenuItem
  - MenuItem
- Content
  - InitialView
    - Div
- Footer
  - Copyright

```
<div>
  <img ..>
  <h1>..</h1>
</div>
<ul>
  <li><a href...> ..</a></li>
   ...
</ul>
<div>
   <p>Welcome</p>
   <div> .. </div>
</div
<div>Copyright ...</div>
```

ReactJS -features

- Basically just a pattern for describing UI
  - Elements, components
  - Props, state
  - “Data-binding” (or something that resembles it)
  - Styles
- With add-ons
  - Routing between views
    - Route configuration
    - What to show to the user depending on the url-pattern
  - Animations
    - Css transitions under React
  - Helpers
    - Testing, cloning etc
  - And huge number of other add-on libraries for different purposes

Hello world

```
<html>
<head>
   <script src="libs/jquery.js"></script>
   <script src="libs/react/react.js"></script>
   <script src="libs/react/react-dom.js"></script>
   <script src="https://. . ./babel-core/. . ./browser.min.js"></script>
   <script type="text/babel">
      $(document).ready(function () {
         ReactDOM.render(<h2>Hello there</h2>,
                     document.getElementById('hello’)
         );
      });
   </script>
</head>
<body>
   <h1>React-demo</h1>
   <div id="hello">Soon to disappear</div>
</body>
</html>
```

JSX-notation is not understood by browser, babel is required

”Hello there” is rendered here as the document-ready –handler is run

ReactJS-project

ES5 or ES6

JavaScript or JSX

Modularisation

Node and npm

Gulp

Some design considerations

- How will you write your code?
  - Pure JavaScript (perhaps in some rare cases)
  - JSX with ES5 (why would you?)
  - JSX with ES6 (recommendation)
- Which modularization system you will use
  - None, just load separate scripts from html (small projects)
  - AMD (not likely)
  - CommonJS (Good choise for ES5)
  - ES6 (Obviously for ES6)
- What other libraries will you use
  - JQuery
    - direct dom manipulation should NOT be used, benefits are questionable
  - Axios, or some other Ajax-library
  - React Bootstrap, Material UI or some other UI library

Separate JSX-files, compiling

- Using babel in browser might be convenient for some cases but
  - Adds extra overhead
  - Syntax errors are found only at testing
- Better approach is to precompile jsx into JavaScript
  - Compiler locates (at least some of) the errors
  - A possibility to bundle and minify JavaScript
  - Less overhead

Gulp-task compiling jsx into a single bundledapp.js

Babel at work here too

```
function buildApp(){
    let file="./jsx/app.jsx";
    let bundle="app.bundle.js";
    return browserify({
            entries: file,
            extensions: ['.jsx','.js'],
            debug: true
        })
        .transform(babelify.configure({presets: ["@babel/env","@babel/react"]}))
        .bundle()
        .on("error",err => {
            console.log("ERROR:",err.message);
            console.log(err.codeFrame);
        })
        .pipe(source(bundle))
        .pipe(gulp.dest("wwwroot/app"));
}
```

Gulp

- Gulp automatizes build-time tasks
- JavaScript environment
  - Gulp itself provides very simple (and limited) API
  - Node-modules can be used
- Integrated into many IDEs
  - Can be used from command-line also
- gulpfile.js contains tasks
  - Tasks may be bound to different stages of build-process
    - Clean, before, after

WebPack

- WebPack is a module bundler
  - Creates a single JavaScript –file from the entire solution
- The bundle may also contain
  - CSS-files
  - Images
- Quite easy to get started with
  - Configuration on large scale may become somewhat ”extensive”
- Below is a sample of webpack.config.js for building a react application bundle

```
module.exports = {
    entry: './jsx/demo.app.jsx’,
    mode: 'development’,
    output: { path: __dirname+'/wwwroot/app', filename: 'demo.webpack.bundle.js'
},
    module: {
    rules: [{
        test: /.jsx?$/,
        exclude: /node_modules/,
        use:{
            loader: 'babel-loader’,
            options:{ presets:["@babel/env","@babel/react"] }
        }
    }]
  },
  resolve:{ extensions: ['.js','.jsx’] }
};
```

Exercise

- Build the demo app from jsx-folder with both gulpfile and webpack
  - npm run gulp
  - npm run wp
- Study the bundles created

- View the applications
  - http://localhost:9000/demo.gulp.html
  - http://localhost:9000/demo.wp.html

- Can you figure out what the compilation does to the code
- Try also
  - npm run watch

ReactJS Basics

Top-level API

Basic concepts

Top-level API - ReactDOM

v. 17.xxx

- render
  - Kind of an application launcher
  - Display selected ReactElement as contents for specified DOMElement
- unmountComponentAtNode(DOMElement)
  - Removes the mounted ReactComponent from the given DOMElement
- findDomNode(ReactComponent)
  - Returns the native DOM-element associated with given ReactComponent

```
render(
   ReactElement element,
   DOMElement container,
   [function callback]
)
```

React.createElement

document.getElementById

Callback when rendered

Top-level API – ReactDOM.Client

v. 18.xxx

- Methods described on previous page are depricated
- Instead
  - createRoot (typically used)
  - hydrateRoot (used with server rendering)

```
const root = ReactDOM.createRoot(document.getElementById('appcontent'));
root.render(
        <React.StrictMode>
          <MainContent />
        </React.StrictMode>
      );
```

Strict mode just logs extra information about ”unsafe” or deprecated operations

Top-level API - React

Api is seldom used, we use JSX-notation instead

- createElement
  - Creates a new renderable element with specified props and children
  - The user interface is constructed by (hundreds) of calls to createElement
  - Seldom needed now-a-days, use JSX-notation instead. The compilation produces the calls to createElement

These you might want to keep in mind

- cloneElement
  - Creates a copy of existing ReactElement possibly replacing some of the props and children
- isValidElement
  - Is the object passed in as a parameter a valid ReactElement
- createFactory
  - Returns a factory-function for creating elements of speficied type

createElement

React.createElement(type,props,...children)

- This is the function we will be using all the time (though jsx may hide it)
- Type is the name of html-element or a React Component type
- Props is an object holding parameters (“attribute values”) given to the element
- Rest parameters are ReactElements that will become dom-children of the holding element
- ReactElements are stateless and immutable

```
React.createElement("div", null,
    React.createElement("p", null, "Hello, world!"),
    React.createElement(Person, { name: “John" })
);
```

```
<div>
    <p>Hello World</p>
    <Person name="John" />
</div>
```

Person would be a name of React-component

code tutors

Virtual DOM

- ReactElements created in the application form an in-memory hierarchy called Virtual DOM
  - We basically work all the time with ReactElements and React then renders the Virtual DOM into the DOM-tree of the html-page
- Virtual DOM contains ReactNodes
  - ReactElements
  - String or number content
  - Array of ReactNodes (as children for element)

Virtual DOM root element

Element having several children

```
React.createElement("div", {className:’default’},
    React.createElement("p", null, "Hello, world!“,4,6,”Hi”),
    React.createElement(Person, { name: “John" })
);
```

The div has two children

This is not the way you want to construct the entire HTML-page

Basic concept - props

- props-object holds the parameters passed to the element
  - Consider them as attributes given in the html-markup
  - Container passes props to its children
  - May be something to be presented in the UI
  - May be something that affects the state of the component
  - May be something that affects the ui of the component
  - May also be a callback for event handling
- Passing props to html-elements
  - Basically setting the attribute values
  - Use camel-casing in attribute naming (onClick)
  - Some variations (class -> className, for -> htmlFor)
  - Boolean values must be explicitely set (disabled="true")

Basic concept - events

- Traditional html-events wrapped in browser independent container, still similar interface
  - Described as function properties of props-object
- Event hander takes the event object as parameter
- Remember camel-casing: onBlur, onMouseDown
- Event objects (the parameter to the callback) are pooled & reused -> no asynchronous use

```
let simpleWithProps=React.createElement("p",
        {  // Props-oject
        style:{color:'green’},
            onClick:function(ev){  // Event-handler
                alert("I was clicked")
            }
     },
    "I have props");
```

- Enough with the API, wouldn’t you rather:

```
let simpleWithProps=<p style={   {color:'green’}      }
       onClick={ev => alert('I was clicked')}>I have props</p>
```

Basic concepts - Component

Component is declared as an ES6-class extending React.Component

- Take props as constructor parameter, pass them to base class
- Constructor may also initialize the state for the component instance
- Implement render-methods that returns the react element that describes the ui for the component
- Implements event-handlers and life-cycle methods

```
class Clickable extends React.Component{
    constructor(props){
        super(props);
        this.state={clicks:0};
        this.clicked=this.clicked.bind(this);
    }

    clicked(){
        this.setState({clicks:this.state.clicks+1});
    }

    render(){
        return    <p style={{color:'green’}}
             onClick={this.clicked}>
                 I have props {this.state.clicks}</p>


;
    }
}
```

Render-method is a must, everything else is case by case.

Render returns a single React-element

Basic concept - state

- Component instance holds a state (this.state)
- The constructor initializes the state, possibly based on the props
  - this.state={property:value, …};
- When state is later altered setState(modifications)-method should be called
  - This causes rerendering of the component
- Typical methods where state is manipulated
  - componentDidMount (load external data, ajax)
  - Event-handlers from the UI, onChange
  - componentWillReceiveProps - transfer new props into state

Component API

Each component may use methods

- setState(object|function,[callback])
  - Typically you pass in object that holds properties that should be changed in the current state (shallow merge)
  - The first parameter may also be a function taking current state and current props as parameter and returning a new state
  - Also updates the UI
- replaceState
  - Similar to setState but the previous state is replaced entirely with the object passed in
- forceUpdate
  - Just updates the UI

Exercise

- Load http://localhost:9000/app.html
  - It loads app.bundle.js which is compiled from code in app-directory by gulp build
- Create the Clickable component to your app
  - Add clickable.jsx
  - export class Clickable
  - Implementation demonstrated few slides back
- Render Clickable at root.render()
- Rebuild and reload app.html

- Modify webpack.config.js so that it builds the code from app-directory
  - npm run wp
  - Test also npm run watch

JSX

JavaScript with XML…

JSX

- JavaScript with XML-literals
- With JSX-notation you avoid using createElement-extensively
- Must be compiled to “standard” JavaScript
    - With babel at browser
    - At command line before distribution

JSX

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

Compiled into

JS

```
ReactDOM.render(
         React.createElement("h1", null, "Hello world!"),
         document.getElementById('hello')
);
```

Using ES6

- Current recommendation is to use ES6
  - Cleaner syntax
  - Understandable modularization
  - Jsx is compiled, you can set the compiler target to be ES5
- Component class extends React.Component
  - Constructor takes care of props and state
  - this-reference needs to be bound for event-handlers

Study ES6, especially
- New features for object literals
- Object match
- Object spread
- Arrow-functions

```
export class MyComp extends React.Component{
    constructor(props){
        super(props);
        this.state={clicks:0};
        // this.clicked=this.clicked.bind(this);
    }

    render(){
        return <div><p onClick={() => this.clicked()}></div>
    }
    clicked() { . . . }
}
```

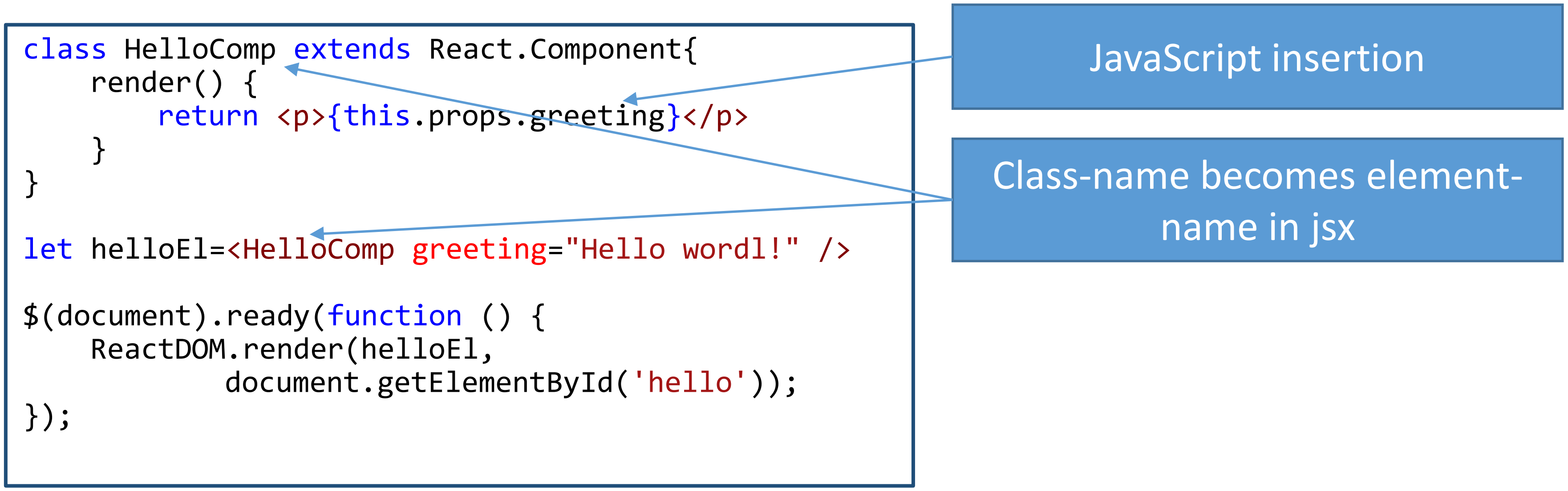
Constructor takes props as parameter and passes to base-class constructor

state-property is set for the object

When using arrow functions you don’t need to “bind” the event handlers. This remains to point to current context.

JSX Features

- Single root element for xml-fragment
  - Not allowed: <p>Hello</p><p>World</p>
  - Must be: <div><p>Hello</p><p>World</p></div>
  - Or <React.Fragment><p>Hello</p><p>World</p></React.Fragment>
- Variables may refer to elements
  - Variable name should start with lowercase letter
- Component class-names are used as elements
- JavaScript insertion with curly-brackets {}



```
class HelloComp extends React.Component{
    render() {
        return <p>{this.props.greeting}</p>
    }
}

let helloEl=<HelloComp greeting="Hello wordl!" />

$(document).ready(function () {
    ReactDOM.render(helloEl,
            document.getElementById('hello'));
});
```

JavaScript insertion {}

- Curly brackets contain JavaScript-expression that evaluates to the value to be inserted
  - Single value
  - Call to a function
  - Conditional statement ?: or ||
- Insertion may be used for
  - Attribute value
  - Element content

```
<div className="{this.props.css}">
    {this.state.data
                ? formatToHtml(this.state.data)
                : <p>No data available</p>}
</div>
```

Some nuances

- Html-attributes
  - Attribute-names use camel-casing: autoFocus, onClick…
  - Two special cases
    - Instead of class-attribute you must use className
    - Instead of for-attribute you must use htmlFor
- Boolean attributes
  - Most often you need to specify the value like: disabled={this.state.isDisabled}
- Entities
  - JavaScript insertion cannot contain html-entities
  - You need to figure out a work-around

Let's just see how often we can forget these two simple rules during the course…

Exercise

- Let’s create app/calculator.jsx
  - export Calculator-component
  - Two input fields
  - Display sum of the figures entered into input fields
  - Initialize contents of input fields with props
    - Transfer values from props to state
  - Create event hander to catch input change-event
    - Call setState save current values of fig1 or fig2 to state
  - Modify app.jsx to display Calculator at root.render()
  - Test
- If onResultChange-prop is passed use it indicate a new result to the container
- Create CalculatorContainer that holds the Calculator and catches the resultChange event
  - You can place CalculatorContainer into app.calculator.jsx
  - Container should display the latest result
  - In the app.jsx render the CalculatorContainer instead of Calculator

ReactJS Components

Child-components

Lifecycle

So once more, props

- Props are parameters given to the component instance by its parent
  - Appear as attributes on template markup where component is used
  - Read-only object
- Props can be of any type
  - Basic types, objects - Pass information to the component
  - Functions - Pass a callback, possible event handler, to the component
- this.props-object may also contain some automatic data members
  - Especially when using router: History, params etc
  - If content is defined for component the children-property contains the child-elements (as ReactElements)
- Props are received by
  - Constructor, very clearly seen in ES6-classes
  - componentWillReceiveProps-method (only after initial render)
  - So these are locations where you might want to modify state based on props given
- getDefaultProps may be implemented to return an object that holds the default values for props not otherwise defined

Factory-functions, Function components

- In many cases component may be implemented as Factory-function instead of Component class
  - Factory-function takes props-object as parameter and returns a react-element based on the props
  - Kind of half-way between a variable referencing to react-element and actual component holding state
- Name of the function may be used as component name in JSX
- Arrow-functions are often used

```
export function Component1(){
    return <p>This is a very simple component</p>;
}
//Pretty much the same as:
let element1=<p>This is a very simple element</p>;



export function Component2(props){
    return <p>This component has props: {props.name}</p>
}

export const Component3 = (props) => <p>Component with {props.data}-function</p>

export const Component4 = ({greeting,target}) => <p> {greeting}, {target}</p>
```

Function component with state

- Since React 16.8 factory components implemented with factory functions may also hold state

```
function FunctionSample(){
    const [data,dataChange]=React.useState('Data');
    return <div>
        <p>Some label</p>
        <input value={data} onChange={ev => dataChange(ev.target.value)} />
        <p>{data}</p>
    </div>
}
```

- React.useState stores the given value to the state and returns an array that holds the value and function that can be called to change the value

Exercise

- Create a new version of calculator (FuncCalculator) into app.calculator.jsx
  - This calculator should be implemented as factory function
- Modify CalculatorContainer to display both calculators
  - Can you pass the result of FuncCalculator to CalculatorContainer
- Allow the user to select the calculator
  - Show “links” to Calculator and FuncCalculator at the container
  - Only show the calculator whose link is clicked

References

- Element may hold a ref-attribute
  - Should point to a callback that takes the reference as parameter
- A mechanism to get a reference to a child element without querying the element from DOM
  - Shouldn't be the first-choice technique for passing data to children
  - Props and rerendering are always the preferred way

```
render() {
    return <div>
             <p onClick={() => this.clicked()}>Click me</p>
             <p ref={r => this.refer=r}>And I'll change</p>
             </div>
}
clicked() {
    this.refer.innerHTML="See!!!";
}
```

Exercise

- Create a third version of calculator (RefCalculator)
  - Two input-fields and Calculate-button
  - Use defaultValue-prop to set contents for the input field
  - Catch the ref to each input field
  - When button is clicked read the value of input fields through the ref and calculate the sum
- Make CalculatorContainer hold also the RefCalculator versions of the calculator
  - Again pass the result to the container from RefCalculator

code tutors

Components and children

- Component may of course describe child-elements itself
  - render returns an element that contains child-elements
- Or the user of the component may describe content to the component
  - Component may choose to display that information or ignore it
  - props.children will automatically contain the contects given to the component

```
export class Child extends React.Component{
    constructor(props){
        super(props);
    }

    render(){
        return <div><h2>Some children from parent</h2>
            {this.props.children}</div>
    }
}

export const Container=() => <div>
    <Child>
        <p>Child will display this</p>
        <p>...and this</p>
    </Child>
</div>
```

propTypes

- When creating a component class propTypes-object may be used to describe possible and required props
  - Object holds members whose name matches the actual prop-member, value is one of the predefined values defining the type for the prop

Import PropTypes from ‘prop-types’

```
export class Class6 extends React.Component{
    constructor(props){
        super(props);
    }

    render(){
        return <div><p>{this.props.value}</p>{this.props.children}</div>
    }
}
Class6.propTypes={
    children: PropTypes.element.isRequired,
    value: PropTypes.string
}
```

Exactly one child is required

Value-prop is optional but it must be string

Constructing an array of child elements

- Component's state may contain an array of data
  - A separate component exists that may display single item
  - Array.map is very convenient, but of course not the only solution
- When creating the array of child-elements a unique value for the key-property should be given to each child
  - Avoid problems when re-rendered (child reconciliation)

```
render(){
    let persons=this.state.persons; // array of person objects
    let rows=persons.map(person => <PersonRow person={person} key={person.id} />);
    return <table><tbody>
            {rows}
        </tbody></table>
}
```

Component life-cycle

Component may hold lifecycle methods

- componentWillMount
  - Executed once before initial render, safe to call setState
- componentDidMount
  - Executed once after initial render, actual DOM exists for children, safe to call asynchronous functionality to setState
- shouldComponentUpdate(newProps,newState,newContext)
  - Implement to return false if you are certain the state change with setState doesn't require rerender
  - Most often you just take new props, possibly transfer them to state and return true
- componentDidUpdate(prevProps,prevState,prevContext)
  - Called right after rerender (not initial), again DOM is updated for children
- componentWillUnmount
  - Executed right before the component will be removed from the DOM

Deprecated:

- componentWillReceiveProps(newProps,newContext)
  - Called before rerenders (not initial), this.props still holds old props, props have not necessarily changed
- componentWillUpdate(newProps,newState,newContext)
  - Called just before rerender (not initial)

Lifecycle with Factory Function

- Factory functions don't have lifecycle methods
  - But we can use React.useEffect to "simulate" componentDidMount and componentWillUnmount
- If the Function Component has state and effect, you may encounter peculiar behavior when the state changes
  - Function is re-executed on each state change

```
const EffectSample = () => {
    React.useEffect(() => {
        console.log("Kind of componentDidMount");
        return () => console.log("Kind of componentWillUnmount");
    })
    return <div>
        <p>useEffect</p>
    </div>
}
```

Exercise, create_react_app

- Run
  - npx create_react_app book-app
  - cd book-app
  - npm start
- Study the files at book-app directory
- Modify app.jsx
  - Npm i –save bootstrap
  - Copy books.gif from ”old” wwwroot/images
  - Replace styles of App.css with ones from wwwroot/styles/styles.css

```
import logo from './books.gif';
import 'bootstrap/dist/css/bootstrap.css';
import './App.css';

function App() {
  return (
    <div className="container">
      <header className="App-header">
        <img src={logo}  />
        <h1>Book application</h1>
      </header>
      <main>
        <p>Hello</p>
      </main>
      <footer>
        Copyright (c) Acme Coding Corp
      </footer>
    </div>
  );
}

export default App;
```

Exercise

- Create a BookList-component (class component)
  - Holds an array of Books in its state
    - Each book should have id,title,author,price,published,description
    - Peek server/bookdao.js
  - Display books in an html-table
- Create BookRow-component (function component)
  - Display singe book as <tr>
  - Gets a book to display from props
- Map the contents of book-array into array of BookRow objects when rendering the BookList
- Modify app.js to render BookList

Context

- Context may be used to pass information through several tiers of children
  - In most cases a global value imported from separate script could be used (KISS)
  - Use context
    - if you encounter a situation where you actually should pass props through several tiers
    - And Main component needs to pass different props to different sets of same child components

```
const valueContext=React.createContext("Hello");

class Sub2 extends React.Component{
    render(){
        return <div>
            <p>Sub2</p>
            <p>{this.context}</p>
        </div>
    }
}
Sub2.contextType=valueContext;

const Sub1=() => <div><p>Sub 1</p><Sub2></Sub2></div>

const Main=() => <div><valueContext.Provider value="Provided Greeting">
        <Sub1 />
    </valueContext.Provider>
</div>
```

Working with input-fields

- Input-, textarea- and select-fields behave in same manner
  - Set the value-prop based on item in the state and implement change-event handler
    - In the event handler change the state so that the UI is updated
    - This creates a “controlled component”
  - You may also create “uncontrolled” component by setting defaultValue
    - Now you don’t create event handler but most likely read data when button is clicked
    - Refs might come in handy….
- With checboxes and radiobuttons we have checked -prop
  - These fields also signal change-event

Exercise

- Add input-fields to the BookList header row on title and author rows
  - Filter the books-array to contain only items that (partially) match contents of input fields
- Add a combobox to the BookList allowing selection of sort-order (title or author)
  - Before rendering sort the books to the correct order

Using styles

The styles may be used in various ways

- Most of the styles should come from css-files
- The style-attribute in the template elements applies
  - You may define an object containing style-definitions
  - In the object the style keys are camel-cased: backgroundImage
- The styles from loaded css-files apply
  - The className-attribute may be used on the template elements

```
const styleEl=<p style={  {color:'red’}    }>Appears in red</p>

const eventProps={
    onClick:function(ev){ alert("Element was clicked");},
    style: {cursor:'pointer'}
};

const otherEl=<p {...eventProps}>But props-object can also be used</p>
```

Transitions

- react-transition-group -module is required
- Items whose transitions are to be “controlled” must be surrounded by

```
<Transition in=[true/false] timeout=[ms]>
```

- Basically you have four states to whom you may declare styles
  - Entering, entered
  - Exiting, exited

```
const transitionStyles = {
    entering: { color:'white',fontStyle:'italic',transform:'scale(1.0)' },
    entered: { color:'white',transform:'scale(1.0)' },
    exiting: { color:'green',fontStyle:'italic',transform:'scale(0.5)' },
    exited: { color:'green',transform:'scale(0.5)' },
};


const TransitionDemo = ({ in: inProp }) => {
    return <Transition in={inProp} timeout={duration}>
        {(state) => {
            var st=Object.assign({},baseStyle,transitionStyles[state]);
            return <div style={st}>
                Some transitions here!!!
            </div>}
        }
    </Transition>
};
```

Transition: all {duration}ms linear;

Exercise

- In the BookRow
  - Display prices < 12 in red, prices > 14 in green
  - Helper-function priceStyle(price) that returns the style object might be handy

Routing

Routing overview

With routing module we automatize changing the view based on the url-pattern

- react-router-dom -module needs to be loaded and configured
- HashRouter, traditional SPA urls with hashes
  - http://myserver.com/index.html#listview
  - No server configuration
  - We load the same page, just “navigate” to a bookmark
- BrowserRouter
  - http://myserver.com/listview
  - Requires server configuration so that regardless of the URL the same page is served

The router has gone through major changes between versions 5 and 6

ReactRouter

- react-router-dom provides us with some elements
  - The Router itself, HashRouter or BrowserRouter
    - Start with HashRouter
  - Switch (v5) or Routes (v6) may be used as a container for Route-objects
  - Route is a description of single url-pattern and associated component/element
  - Link is used to create a hrefs to routes

```
import {HashRouter as Router} from 'react-router-dom'; // or BrowserRouter

// For version 5
import {Route,Link,Switch} from 'react-router-dom’;


// For version 6
import {Route,Link,Routes} from 'react-router-dom';
```

Using router

V5

- We render the Router-element at ReactDOM-render
  - In most cases the Router is the outermost element of the virtual dom
- Main-component displays content based on the current route
- It is possible that the configuration displays several items
  - To avoid this surround Route-elements with Switch-element

```
let C1=() => <p>Child 1</p>
let C2=() => <p>Child 2</p>
let C3=() => <p>Child 3</p>


let Main=(props) => <Router><div><h1>Top</h1>
    <Link to="/">First</Link> <Link to="/second">Second</Link> <Link
to="/third">Third</Link>
    <p>Route content</p>
        <Route path="/" component={C1} />
        <Route path="/second" component={C2} />
        <Route path="/third" component={C3} />
    <p>End route content</p>
</div></Router>
```

Using router

V6

- We render the Router-element at ReactDOM-render
  - In most cases the Router is the outermost element of the virtual dom
- Main-component displays content based on the current route
- It is possible that the configuration displays several items
  - To avoid this surround Route-elements with Switch-element

```
let C1=() => <p>Child 1</p>
let C2=() => <p>Child 2</p>
let C3=() => <p>Child 3</p>


let Main=(props) => <Router><div><h1>Top</h1>
    <Link to="/">First</Link> <Link to="/second">Second</Link><Link to="/third">Third</Link>
    <p>Route content</p>
      <Routes>
        <Route path="/" element={<C1 />} />
        <Route path="/second" element={<C2 />} />
        <Route path="/third" element={<C3 />} />
      </Routes>

    <p>End route content</p>
</div></Router>
```

Link-component

- To-attribute describes the target route
  - String or object
- If to is is described as an object it may hold
  - pathname: target path
  - search: string of query parameters (?some=thing&other=param)
  - hash: bookmark appended to path (#mark)
  - state: object persisted to location.state (target component may access this.props.location.state)

```
<Link to="/cars">Cars</Link>
<Link to={
       {
          pathname:`/books/${car.id}`,
          state:{data:car}
       }
    }>
    {car.make}
<Link>
```

Exercise

- You already should have main-component with layout:
  - <header>
  - <nav> // Display navigation here
  - <main> // Display actual content here
  - <footer>
- Surround the entire main-component with Router-element
  - It may only hold a single child….
  - Use HashRouter to start with
- Add Links to the nav-section
  - Booklist
  - Calculator
    - Copy code for calcultar from the earlier project
- And Switch/Routes to the main-section
  - Holding Route-elements to BookList and Calculator

Catching url-parameters

Version 5

- Url-pattern describing the route may contain variable parts
  - Marked with :, followed by parameter name
- Component processing the route catches the url-parameters from the match-object
  - Match.params is an object whose properties match the url-parameters
- Router places the match-object to the props of the component it displays

```
var C5=({match}) => <p>Child 5 : {match.params.data}</p>

var C3=({match}) => <div><p>Child 3</p>
       <Route exact path={`${match.url}`} component={C4} />
       <Route path={`${match.url}/:data`} component={C5} />
    </div>
```

:data
->
params.data

Catching url-parameters

Version 6

- Version 6 doesn't add anything to props
- Function components may use hooks to query data from router
  - If you need to access router data in class components it is easiest to create a function component wrapper around the class component

```
export class ChildRoutingOrig extends React.Component{
    constructor(props){
        super(props);
    }

    render(){
        let id=this.props.params.id;
        return <div><h3>Child routing</h3>
            <div>
                <Link to="">Basic </Link>
                <Link to='extra'>Extra </Link>
            </div>
            <p>Got parameter {id}!</p>
            <Routes>
                <Route exact path="" element={<BasicChild />} />,
                <Route path="extra" element={<ExtraChild />} />,
            </Routes>
        </div>
    }
}

export const ChildRouting=() => {

    return <ChildRoutingOrig params={useParams()} />
}
```

Navigating

- Basically you just push items to the props.history
  - push(url)
  - goBack()
  - go(n)
- Redirect is a little more complicated
  - You need to render the Redirect-element
  - Taking string or “location”-object as to-props

With version six you must use useNavigate() –hook that returns navigate-function. Parameter may be
- Route to which navigate
- Number of items to ”skip” in history (-1 =back one step)

```
// GO-button navigates to route “/second”
let C1=({history}) => <p>Child 1
       <button onClick={() => history.push("/second") }>GO</button>
    </p>
// BACK goes back to previous page
let C2=({history}) => <p>Child 2
<button onClick={() => history.goBack()}>BACK</button>
</p>

// Redirecting
let C4=({match,location}) => <Route
    render={props => <Redirect to={{
           pathname:`${match.url}/redirected`,
           state:{from:location} }}  />
} />
```

Exercise

- Implement BookDetail (function component)
  - When item is clicked from BookList navigate to BookDetail to edit it
    - ID -> Link to
    - Title -> navigate (this is not as straightforward as you think…)
  - Implement Back-button to BookDetail
- You will also need to implement bookService-object
  - Holds the array of books
  - Provide DAO-methods
    - getAll
    - get(id)
    - create(book)
    - delete(book)
    - update(book)
  - Implement methods with Promise-patterns where the return value is given by the resolve function
    - Client calls the method like bookService.getAll().then(books => doSomethingWith(books));
- Both BookList and BookDetail should use bookService when accessing the data

- Extra: Implement Authors-component
  - Extend your menu with option allowing navigation to Authors-component
  - The component should just display a select field of author and then list of books for the selected author
    - Again links to BookDetail

Route match

- Component that processes the route gets the match-property in its props
  - This allows child-routing
- The example below extends the sample from earlier slides
  - C3 catches the match property
  - With url "/third" C# displays also C4
  - With url "/third/fifth" C5 is displayed

V5 Example,
With V6 you may define routes to any component without any extra work

```
var C4=() => <p>Child 4</p>
var C5=() => <p>Child 5</p>
var C3=({match}) => <div><p>Child 3</p>
       <Route exact path={`${match.url}`} component={C4} />
       <Route path={`${match.url}/fifth`} component={C5} />
    </div>
```

Exercise

- Create PrintableDetail
  - Displays book data without inputs
- Create DetailContainer
  - Holds links to BookDetail and PrintableDetail
  - Holds routing configuration to select one of the above
- In the ”main-routing” display DetailContainer when navigating to book detail

Localization

Translations

Formattings

Localization

- Localization is about
  - Translating the string constants
  - Formatting data according to locale
    - Date-formats
    - Number-formats
    - Currency?
- React provides no support for localization
  - react-localization –module offers some methods for working with translations
- ES6 offers Intl-object to support localization
  - Collators
  - Date-formattings
  - Number and currency formattings

Translating strings

- Basic idea is to define a replaceable object that holds the the translations
  - On object for each supported language
- Select which object to use when the language changes
- Instead of using constant strings in the ui use members of the selected translation object
  - The object must be globally available for all components
  - Or you might want to place it into context in the ”main”-component so that the components that need translations may query it

Intl-object (ES6)

- Can you trust that the browsers support this feature or do you need to implement a replacement for browsers that don’t support it?

```
var fiCollator=new Intl.Collator("fi");
console.log(fiCollator.compare("ä","z"));

var fiNumberFormat=new Intl.NumberFormat("fi-
FI",{minimumFractionDigits:2,maximumFractionDigits:2});
console.log(fiNumberFormat.format(1234567.56789));

var fiCurrency=new Intl.NumberFormat("fi-FI",
                      {style: "currency", currency: "EUR" });
console.log(fiCurrency.format(1234567.89));

var fiDate=new Intl.DateTimeFormat("fi-FI");
console.log(fiDate.format(new Date()));
```

Exercise

Your wwwroot/translations holds couple of translations.json-files

- Implement similar object to the application holding the ”defaults”
  - let translations={}
- In the render methods you may
  - let tx=translations;
  - …. {tx.buttons.create} …

- On main-components useEffect load the selected translations file from the server
  - Demo app holds HTTP-object you might want to use
  - When the file is loaded just replace the defaults with loaded contents

- Modify the menu so that you may change the translations on a fly

- Experiment with Intl-object, display the price of the books in the booklist in localized manner

```
{
"title":"Book application",
  "buttons": {
    "delete": "Delete",
    "create": "Create",
    "back": "Back"
  },
  "book": {
    "title": "Title"
  }
}
```

Add

"proxy": "http://localhost:9000",

To package.json

RESTful-Service

Using data from the server

Designing rest

- Implementing RESTful services is rather easy
  - However some aspects require some thought
- Design consistent url-patterns
  - GET /cars - Return list
  - GET /cars/:id - Return single item
  - POST /cars - Create item
  - PUT /cars/:id - Update item
  - DELETE /cars/:id - Delete item
- What amount of data is returned
  - Basic information first, detailed information with separate request
  - How is binary data accessed
  - Do the listings require paging
  - How should the contained entities be handled
- Should put and post return data

Using REST from ReactJS

- When to make calls?
  - componentDidMount
- How to use ajax
  - XMLHttpRequest or Fetch
    - Self-implemented helpers
  - jQuery
    - Simple gets are simple
    - Put and post may be more complicated
  - Extra library (Axios)

```
$.getJSON("/api/cars", function (data) {
    // data holds the response as JavaScript object
    // or array
});
```

When to make the calls

- Again you need to design your approach
  - How much data can you keep in local memory
  - How to keep local data in sync with server data
  - When populating a list do you fetch all data and filter/sort it in client or do you make separate requests to get filtered/sorted data from the server
- But basically
  - componentDidMount is the correct place to fetch the data
  - And when data arrives the state of the component should be modified and UI updated
  - Data should be sent to server on a separate event-handler (not only componentWillUnmount) to be able to display possible errors on the same view

Exercise

- Study jsx/demo.ajaxhelpers.js
  - doAjax-function
  - HTTP-object
- Study RESTful implementation at server/webapi.js

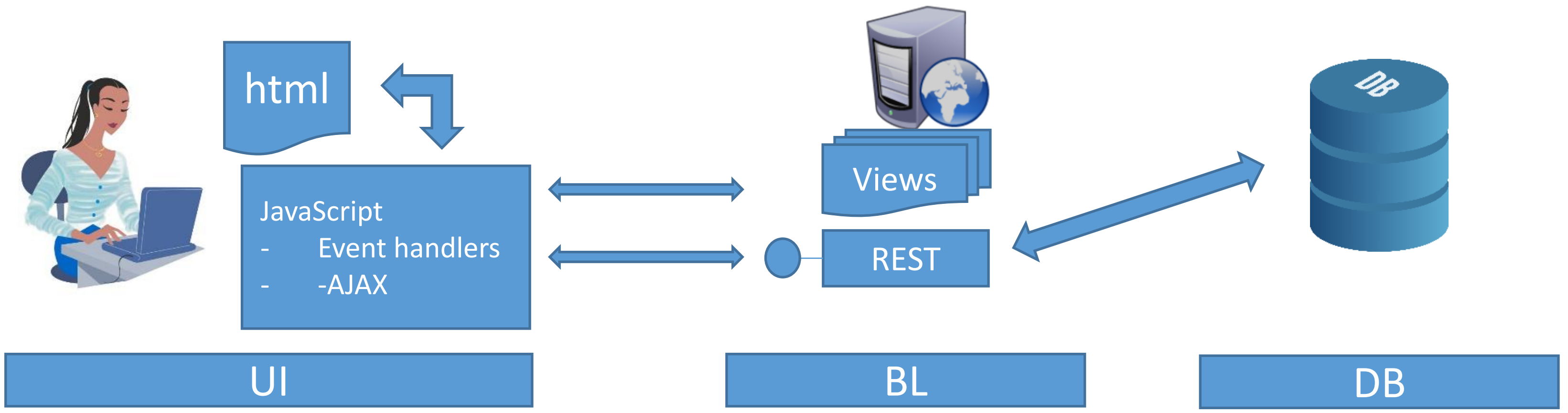
- Use the HTTP-object in the bookService to query and manipulate server data with your Book Application

Security of SPA-application

Security

- You want to protect your data
  - Especially secure the RESTful interface
  - But also
    - Credentials in application logic
    - Secure the server (technical credentials)
- You want to protect the data communication
  - HTTPS
- You might even want to protect the application
  - Logic
  - Templates
  - Resources
- Security measures you choose must come from the requirements of your solution
  - Traceabilty
  - Undeniability
  - Usability

Single Page Applications, security



- Security considerations?
- Security considerations?
- Security Considerations?

Discussion: what kind of security considerations?

User must be able to trust that he sees current and valid information at all times, and he can only access information that is meant for him/her

Protecting the RESTful services

- You have to select the authentication method depending on
  - The possibilities your server environment gives you
  - Your application and its requirements
- In your code
  - You may pass the Authorization-header with your request
  - You may use your own custom headers
  - You may pass authorization information in a cookie generated in user login
  - You may pass authorization information in the body of each request
- NEVER store technical credentials for authentication into the JavaScript-code loaded to the browser

Implementing authentication

- Do you want to maintain your own user database
    - Login credentials stored in your database
    - Rather easy to implement
    - Just don’t store the password unencrypted
        - MD5
- Third-party authentication (OAuth)
    - User identity is verified by some one else
        - Microsoft
        - Google
        - Facebook
    - Most likely you still need to store some profile information locally
        - But not the password

Exercise

- Implement Login component into the Book-application
  - Input fields for usename and password
  - Button to login
- Modify the Main-component
  - Displays Login-component until it get’s a valid user from the login component
- Implement userapi
  - Array of user-objects {email:’xxx’,psw:’xxxx’,psw_reset:’xxx’}
  - Post ‘/api/user/login’, body.email, body.psw
    - Either returns user-object (without password)
    - Or {error:’Not logged in’}

code tutors

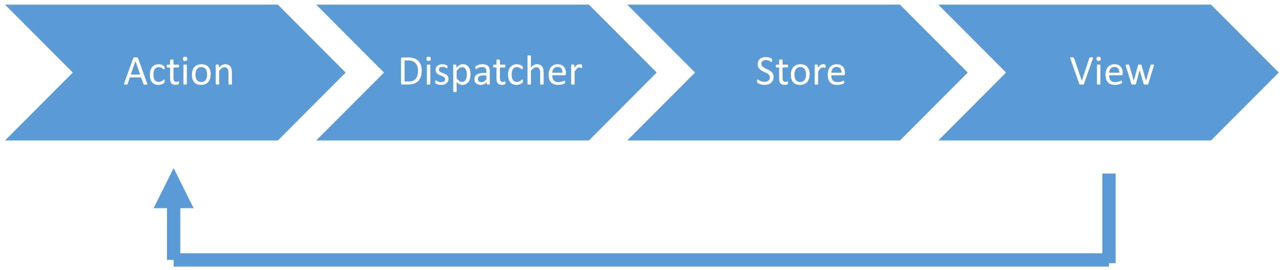
FLUX-Architecture

Managing the Application State

And briefing to Redux

FLUX overview

- FLUX-architecture is used by Facebook on client side development
    - Not a framework, just a pattern
    - Can be implemented with convenient techniques
- Unidirectional dataflow
    - Store(s) hold the data and know how to process different actions against it
        - No getters and setters, just callbacks through which the dispatcher passes the action to the store
    - When the store manipulates data it creates an event which can then be processed by the view(s)
    - The view then may generate new action(s)



What is Redux

- Redux is a state-container for JavaScript applications
  - Can rather well be used as a Store in FLUX-architecture
- Lightweight and simple to use
  - redux npm-module
  - Actual data that is held by the container may be anything
  - We just implement a function that takes state and action as parameter
    - And returns the new state based on the action
  - Elsewhere we may subscribe to the changes in the store
  - And dispatch actions to the store
- There are also extensions for ReactJS
  - react-redux -module
  - Especially targets state of visible components

How to use Redux with React

- Redux.js and react-redux.js are required
- Actions are still just unintelligent “holders”
  - A factory function should be implemented to create an action
  - A helper may also be implemented to dispatch action
- No need to implement dispatcher, it comes from the store
- Design the state object so that it holds the state for the application
  - Consider what data belongs to application state
    - And what data belongs to the compoeenents
  - Of course, this should be somewhat hierarchical for different use cases
- Implement reducers to process actions against the state
  - And return the **new** state-object

Simple sample

```
/* Provide function(s) to create an action(s) */
export function changeSort(sort){
    return{
        type:"CHANGE_SORT",
        data:sort
    };
}


/* Implement a reducer capable of handling an action */
export function bookReducer(previousState,action){
    if (!previousState) return {books:[],bookSort:'title'};
    if (action.type=="CHANGE_SORT"){
        return Object.assign({},previousState,{bookSort:action.data});
    }
    return previousState;
}

export var bookStore=Redux.createStore(bookReducer);

/* You may also create action-functions doing dispatch automatically */
export const dispatchChangeSort = (sort) => bookStore.dispatch(changeSort(sort));
```

```
/* Elsewhere */
var unsubs = bookStore.subscribe(
    () => console.log("State",
            bookStore.getState())
  )
dispatchChangeSort('author');
unsubs();
```

Exercise

- Create app.bookstore.js
  - Initial-state holds an empty array of books
  - Describe action ”GOT_BOOKS”, where data is the array of books
    - Implement the reducer that handles the action
  - Modify your bookservice so that it causes the action when the books are loaded from the server

- Subscribe the changes in the store at the booklist componentDidMount
  - By replacing the array of the books in the booklist
  - Unsubscibe the action in componentWillUnmount

ReactRedux

- Introduces a new kind of container-component
  - Container-components become connected to the state of the store
- State information is automatically passed to connected container components
  - Through props

Main-component provides the store

```
var Provider=ReactRedux.Provider;
return <Provider store={bookStore}><div>
    <header>
        <img src="images/books.gif"/>
        <h1>{tx.title}</h1>
    </header>
    <main>
        <MainMenu menus={menuDescr} />
        {this.props.children || <BookList />}
    </main>
    <footer>Copyright Acme Coding Ltd</footer>
</div></Provider>
```

State-aware component

- Idea should be that UI-components are implemented in a manner that they receive all their data through props
  - Not fetch it independently

```
const mapStateToProps = (state) => {
    return {
        books: getActualBooks(state.books, state.titleFilter,
                              state.authorFilter,state.bookSort)
    }
}

const mapDispatchToProps = (dispatch) => {
    return {
        onChangeAuthor: (auth) => {
            dispatch(changeAuthor(auth))
        },
        onChangeTitle: (title) =>{
            dispatch(changeTitle(title));
        },
        onChangeSort: (sort) => {
            dispatch(changeSort(sort));
        }
    }
}

const ActualBookList = connect(
  mapStateToProps,
  mapDispatchToProps
)(BookList)

export default ActualBookList;
```

Original BookList should use books-prop.

getActualBooks is just a helper function that returns sorted and filtered array of books.

Map event-handlers to action dispatching

ActualBookList is the connected version of original BookList

This replaces original BookList

Thank you!

Any remaining questions?