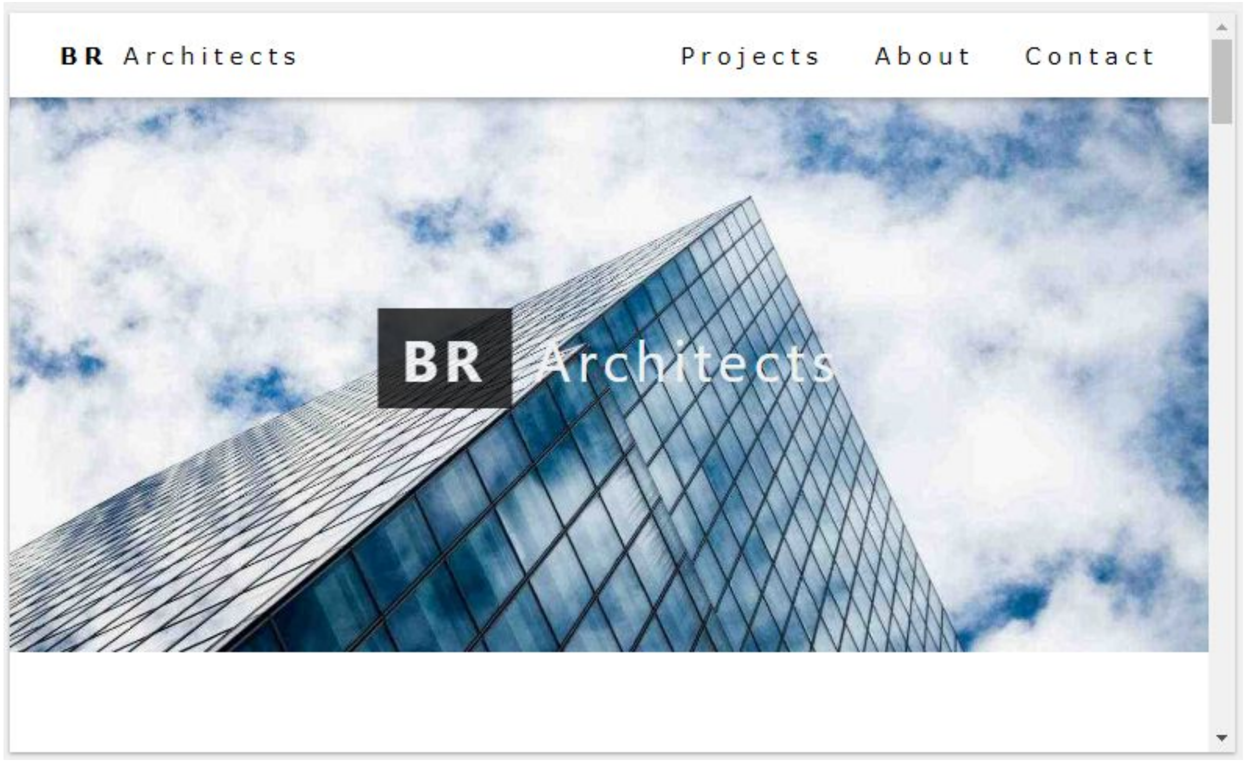
JavaScript for Enterprise Development

Week 4-1

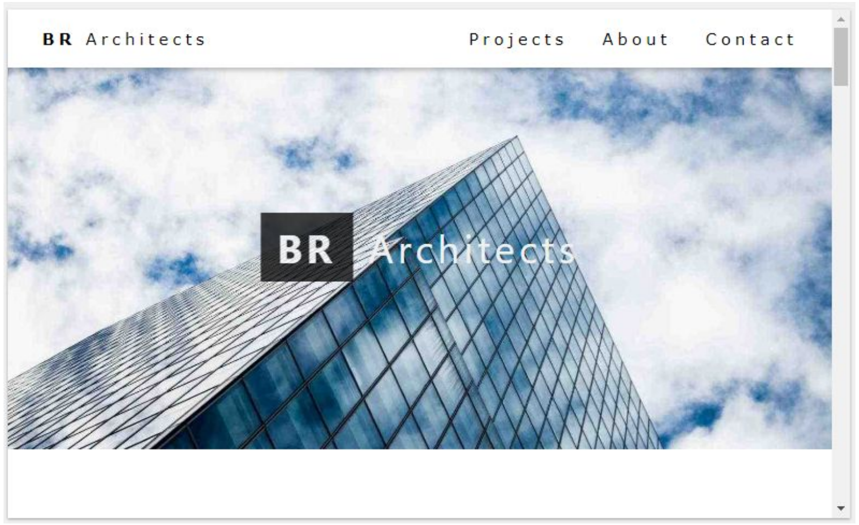
CONTENTS

Component Styling

Simple landing page example

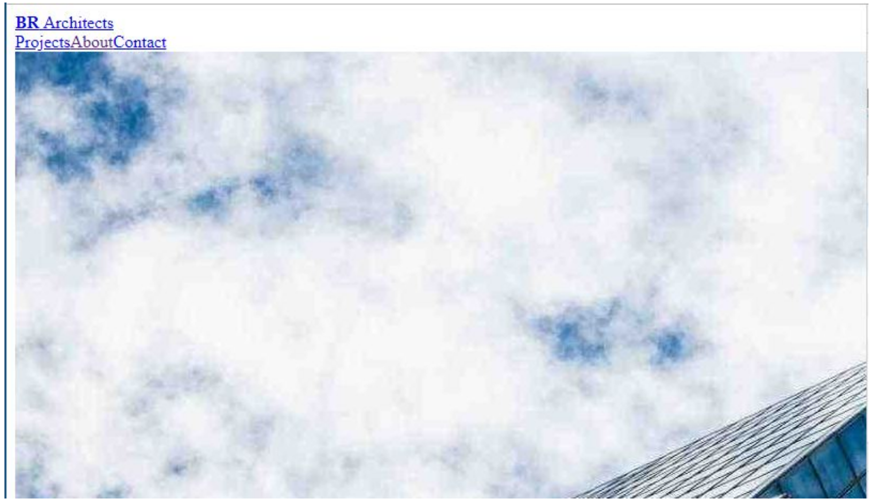


Base semantic markup



```
<main>
 <header>
   <a href="/main"><b>BR</b> Architects</a>
   <nav>
     <a href="/projects">Projects</a>
     <a href="/about">About</a>
     <a href="/contact">Contact</a>
   </nav>
 </header>
 <img src="https://goo.gl/2LxxtC" />
</main>
```

Base semantic markup


BR Architects
ProjectsAboutContact

```
<main>
 <header>
   <a href="/main"><b>BR</b> Architects</a>
   <nav>
     <a href="/projects">Projects</a>
     <a href="/about">About</a>
     <a href="/contact">Contact</a>
   </nav>
 </header>
 <img src="https://goo.gl/2LxxtC" />
</main>
```

1. Inline styles in components

```
<img src="https://..." style={{ maxWidth: '100%' }}/>
```

All native html elements in react have ***style*** prop (ordinary object expected).

This object translates into ***style*** attribute.

All “camelCase” keys casts into kebab-case notation

e.g. ***maxWidth*** became ***max-width***

2. className prop

```
<style>
   .app-header { background-color: #f3f3f3; }
   .nav-link { font-family: Verdana, sans-serif; font-size: 14px;}
</style>
```

```
<main>
 <header className="app-header">
   <a href="/main" className="nav-link">
     <b>BR</b> Architects
   </a>
   ...
</main>
```

2. css imports

styles.css

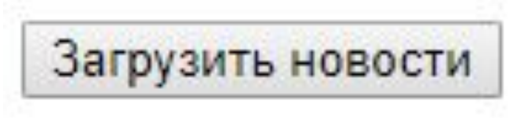
```
.link {
   font-family: Verdana, sans-serif;
   font-size: 14px;
}
```

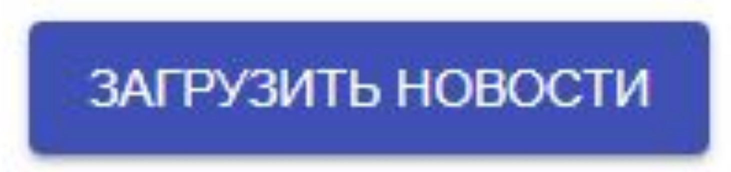
```
import styles from './styles.css'

...

<header className={styles.head}>
```

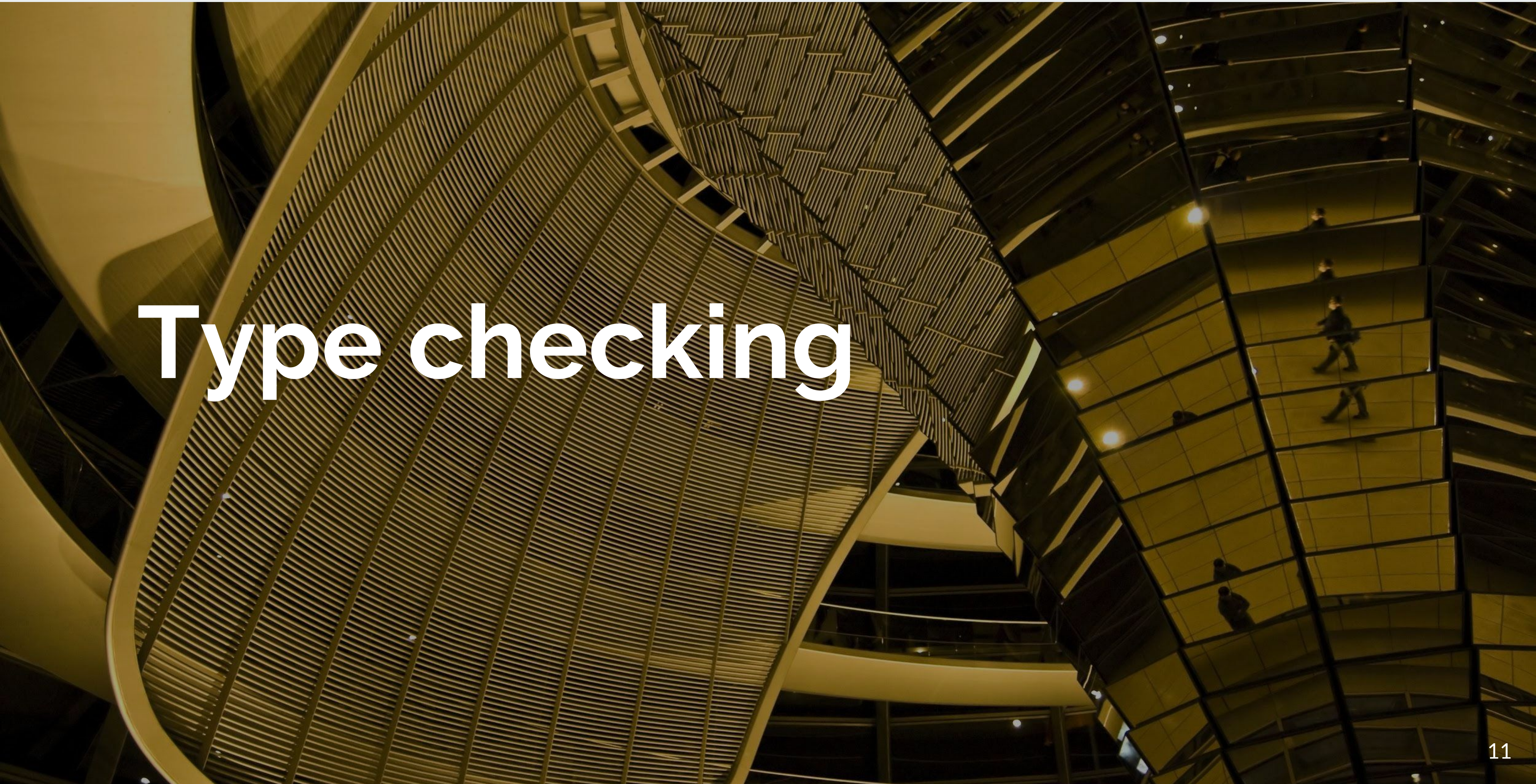
2. UI libraries

```
npm install @material-ui/core
```

```
import Button from "@material-ui/core/Button/Button"

<Button onClick={this.loadData} variant="contained" color="primary">
  Загрузить новости
</Button>
```







Type checking

Component API

Every component could be considered as a black box with some inputs - ***props***.

JavaScrips is a weakly-typed programming language, therefore there is no strict contracts on arguments, passing to your component.

To help you keep in order all your component contracts you can use some of additional tools.

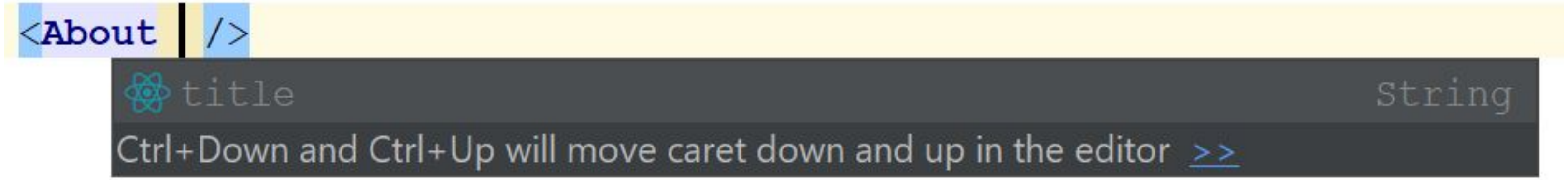
One of them is ***PropTypes*** library.

To run typechecking on the props for a component, you can assign the special ***propTypes*** property.

Definition

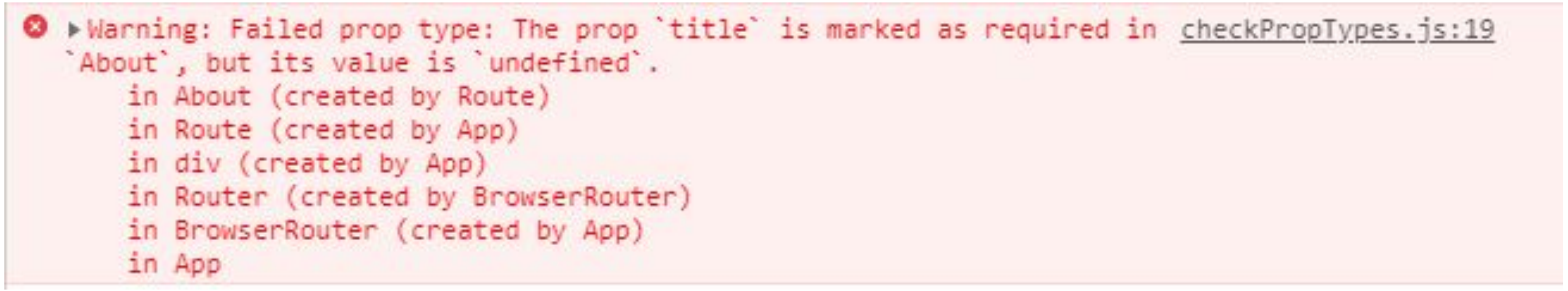
```
import React from 'react'
import PropTypes from 'prop-types';

export const About = (props) => (
 <h3>This is an {props.title} page</h3>
)

About.propTypes = {
 title: PropTypes.string
};
```

Code assist



```
<About  />
title                                         String
```

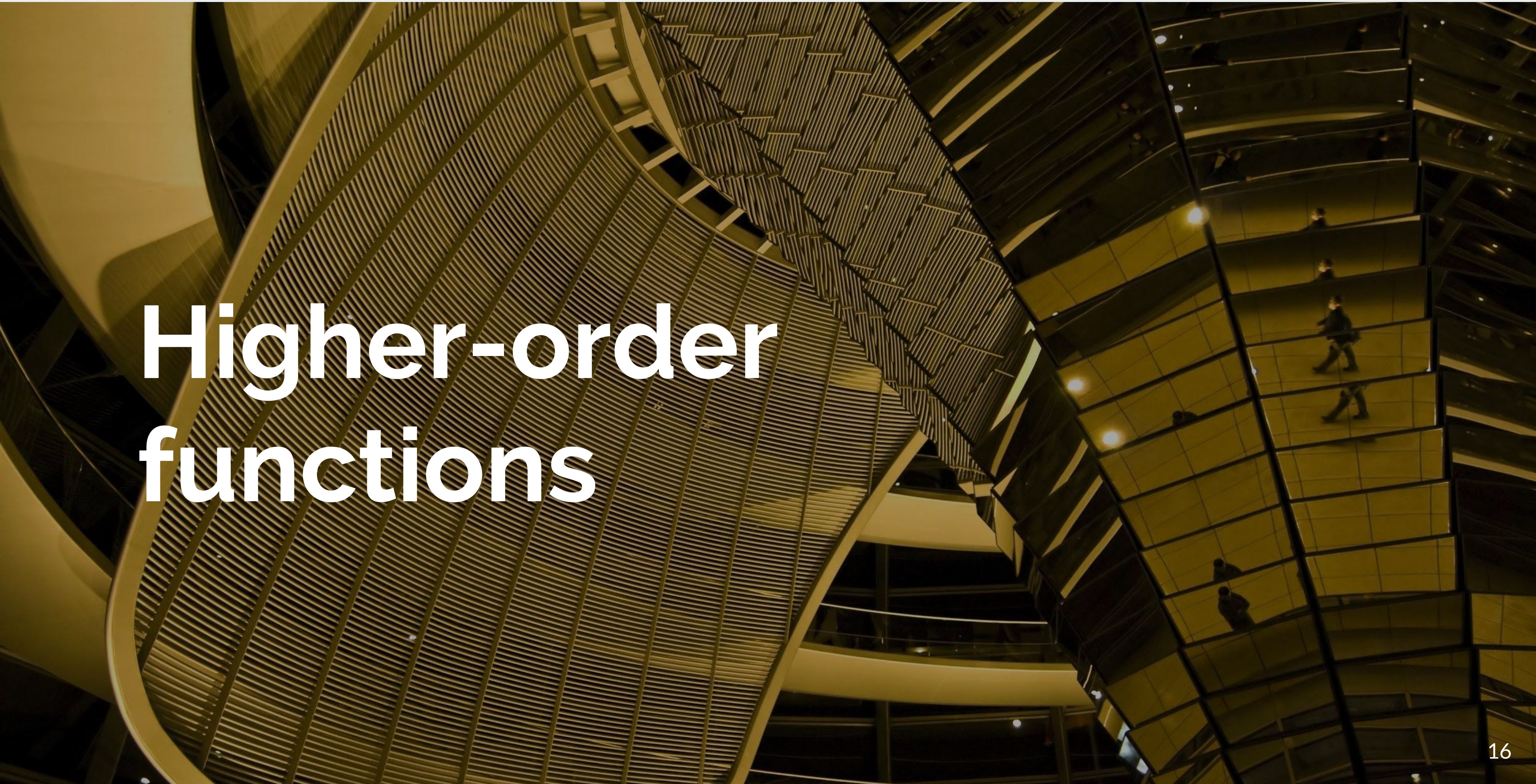
Ctrl+Down and Ctrl+Up will move caret down and up in the editor >>



```
Warning: Failed prop type: The prop `title` is marked as required in    checkPropTypes.js:19
`About`, but its value is `undefined`.
    in About (created by Route)
    in Route (created by App)
    in div (created by App)
    in Router (created by BrowserRouter)
    in BrowserRouter (created by App)
    in App
```

Variations

```
About.propTypes = {
 title: PropTypes.string.isRequired,
 children: PropTypes.node,
 news: PropTypes.oneOfType([PropTypes.array, PropTypes.object]),
 theme: PropTypes.oneOf(['dark', 'light'])
};

About.defaultProps = {
 title: 'News app'
};
```



Higher-order functions

Why?

React emphasizes composition over inheritance, making it easy to compose larger components by reusing smaller ones.

To make composition much easier there is a technique called ***Higher-order functions***.

Higher-order function is a function which returns another (modified, enhanced) function.

Example - News list

```
{this.state.documents.map((doc) => (
    <Article onClick={???}>{doc.title}</Article>
  </li>
))}
```

Naïve approach

```
{this.state.documents.map((doc, id) => (
    <Article onClick={(data)=> { onDocClick(id, data) }}>
      {doc.title}
    </Article>
  </li>
))}
```

Injecting information about which exactly doc was clicked

With HOF

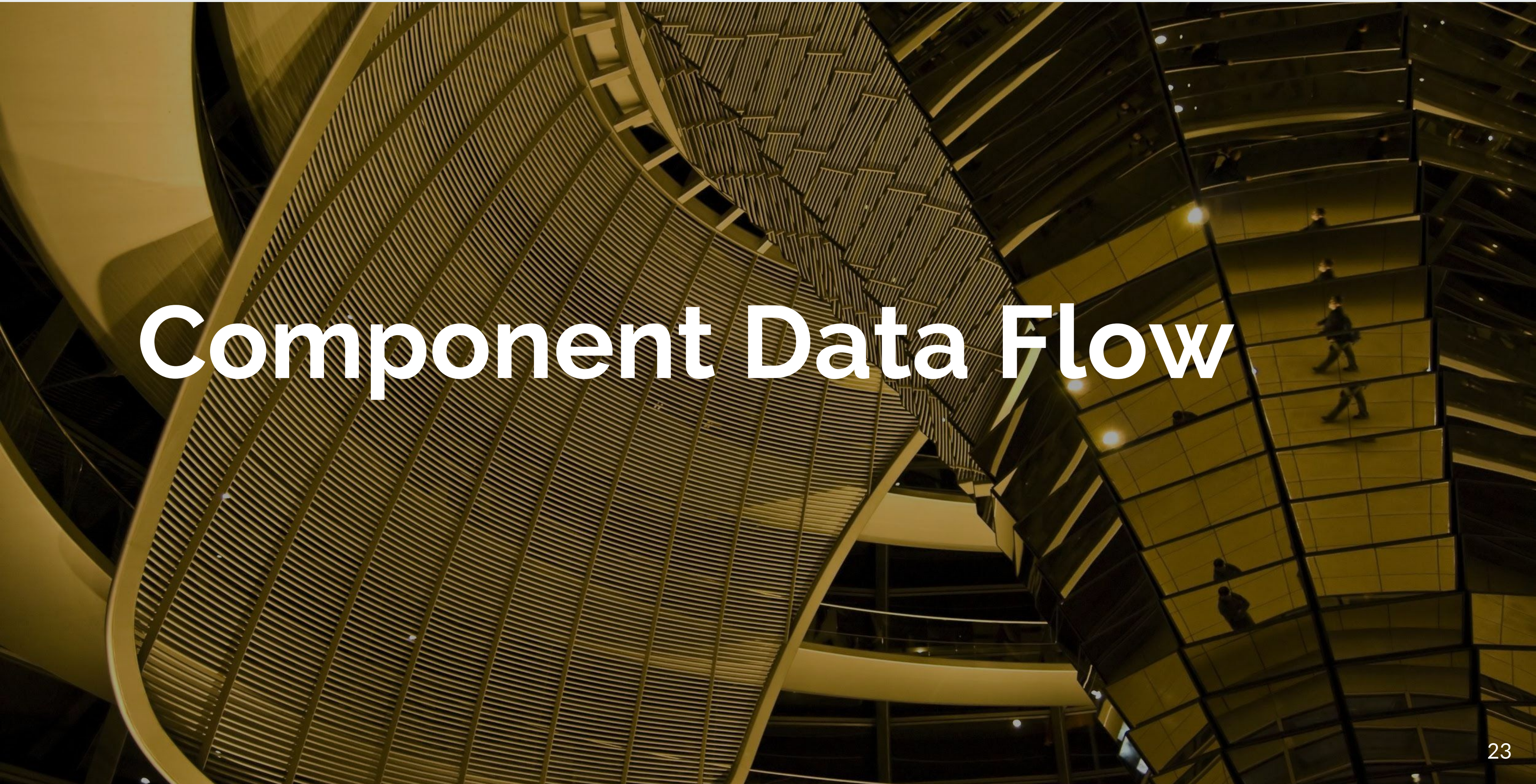
```
createDocClickHandler = (id) => {
  return (data) => onDocClick(id, data);
}

{this.state.documents.map((doc) => (
     <Article onClick={createDocClickHandler(id)}>
       {doc.title}
     </Article>
   </li>
 ))}
```

HOC - Higher-order Components

```
const EnhancedComponent = higherOrderComponent(WrappedComponent);
```

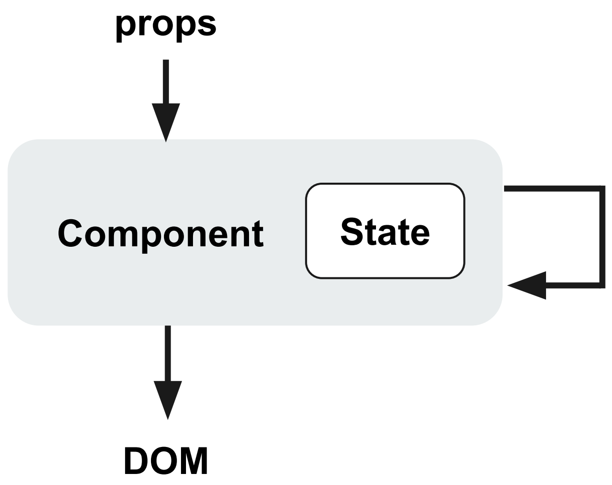
HOC - Higher-order Components

```
const makeRed = (Component) => {
 return (
   <div style={{ backgroundColor: 'red' }}>
     <Component/>
   </div>
 )
}

const MyComponent = () => (
 <span>My component</span>
)

const EnhancedComponent = makeRed(MyComponent);
```

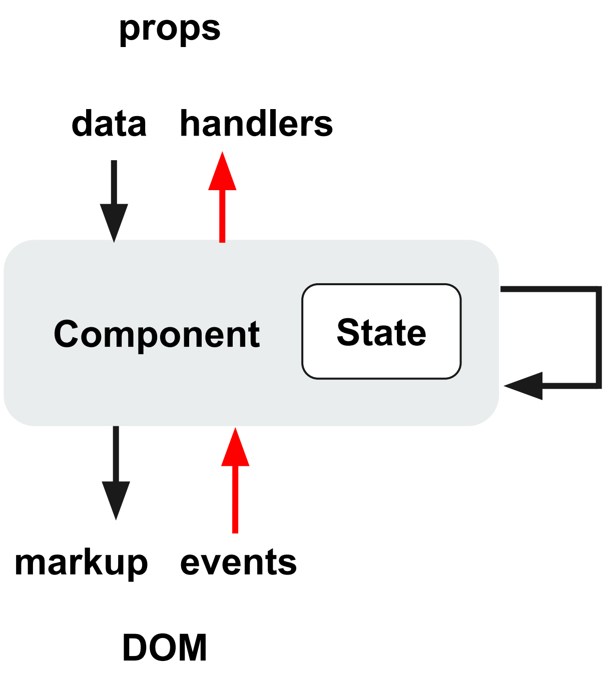


Component Data Flow

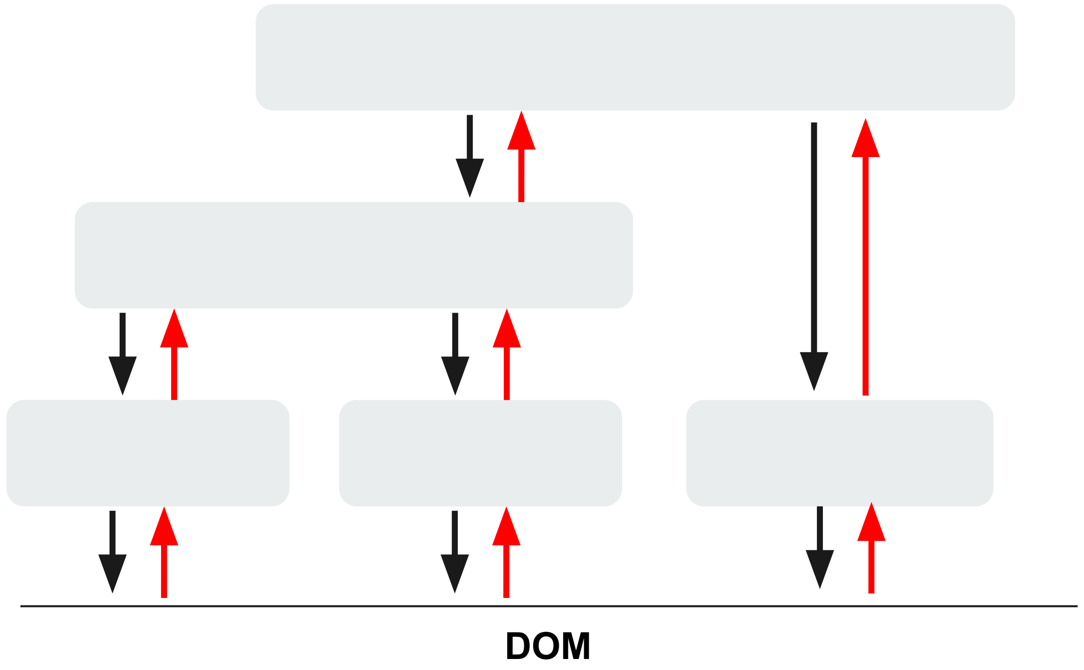
Component data flow

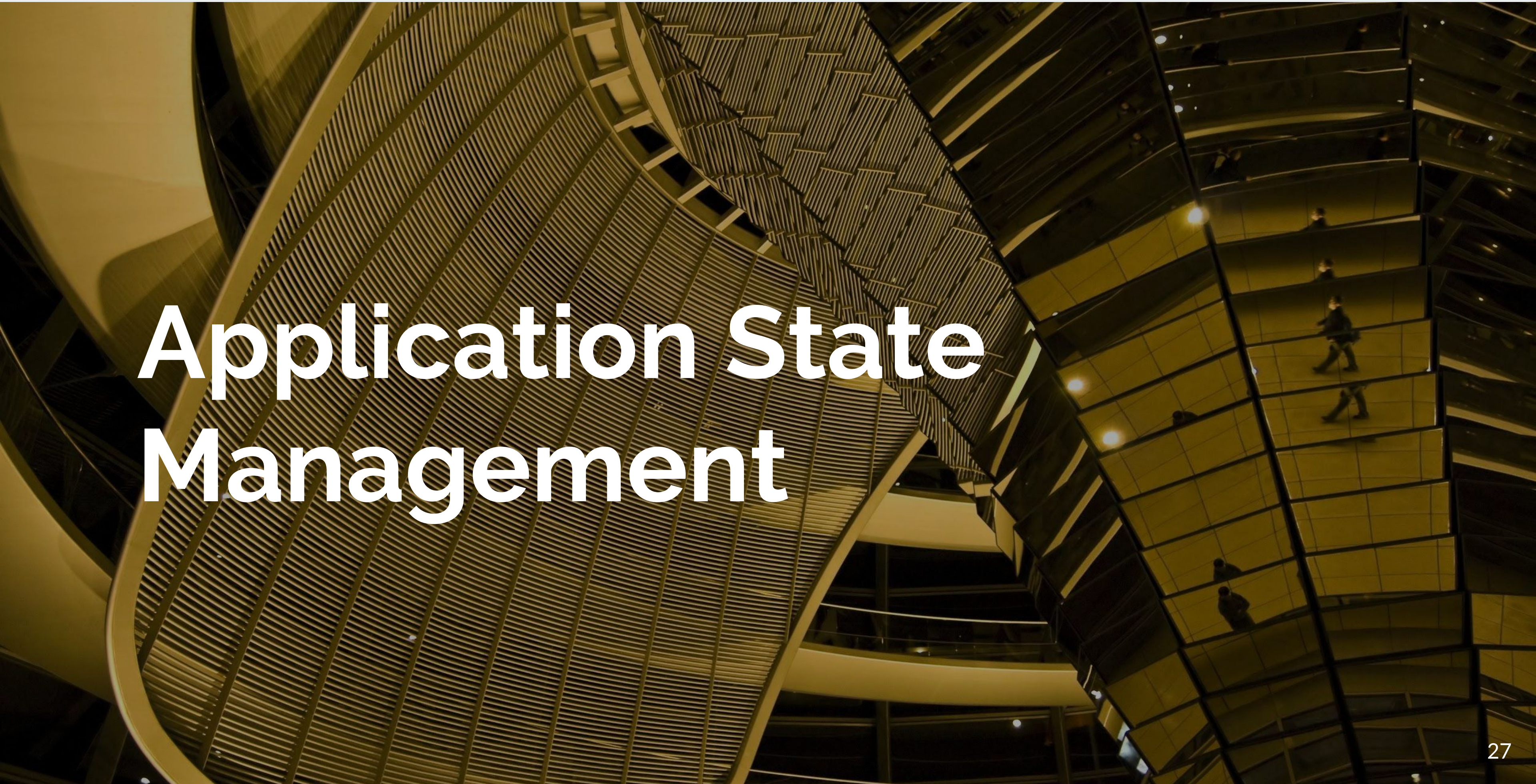


Component props and handlers



Props and callbacks hierarchy





Application State Management

Idea

What if we could have an instrument that allows us to make an global events calls inside of deep-nested components?

What if we could store global application state and share it among our components?

Idea

It would be **HELL**

Tight coupling, race conditions and implicit behaviour.

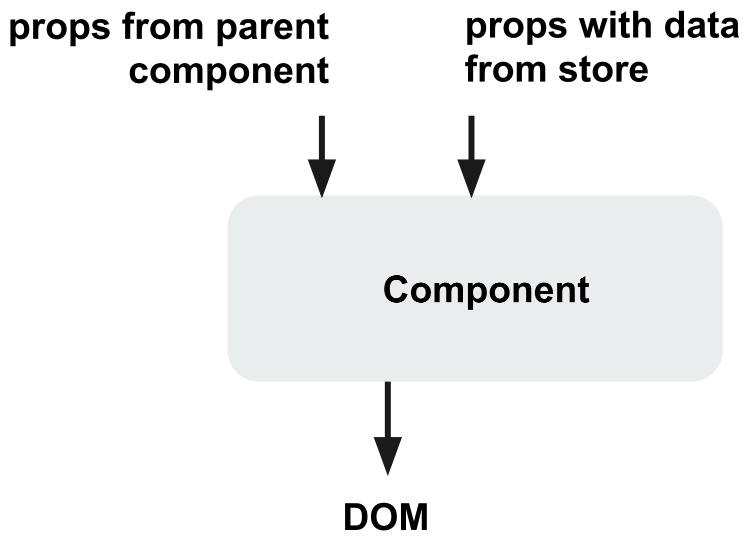
Idea

Nevertheless an approach is unavoidable.

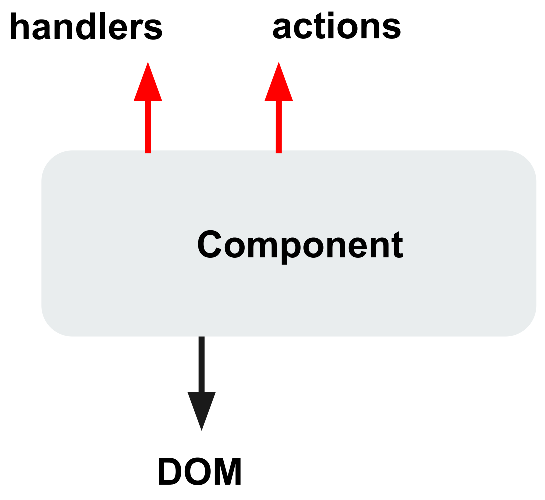
To make things more predictable we have to accept some rules:

1. Only one source of truth about application state - ***Store***
2. Store is read-only
3. The only way to make changes in store is to call ***Action***

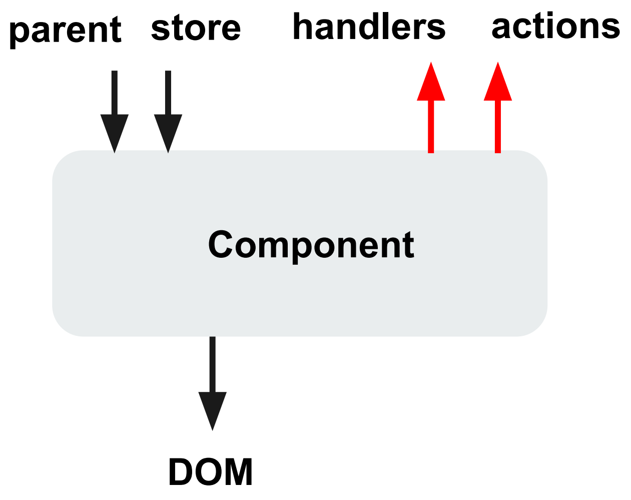
Component data flow with store



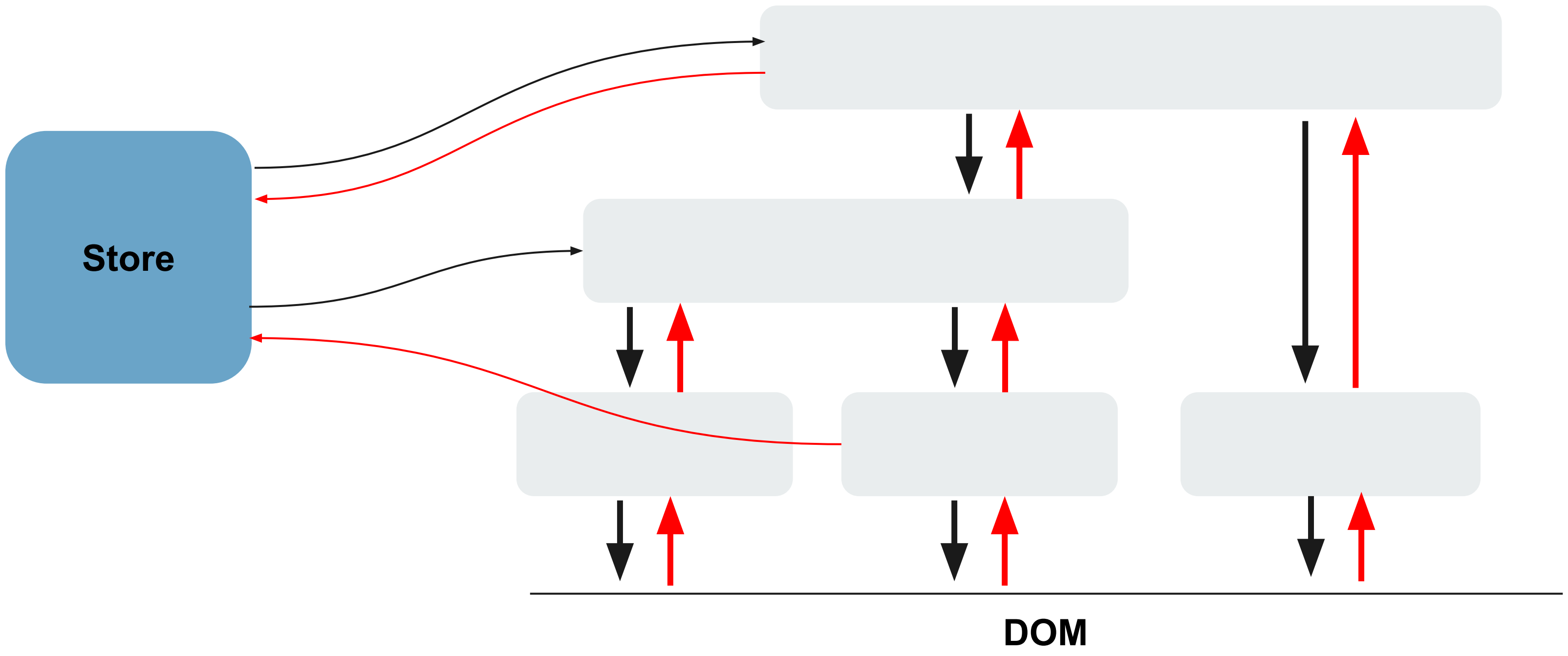
Component data flow with store



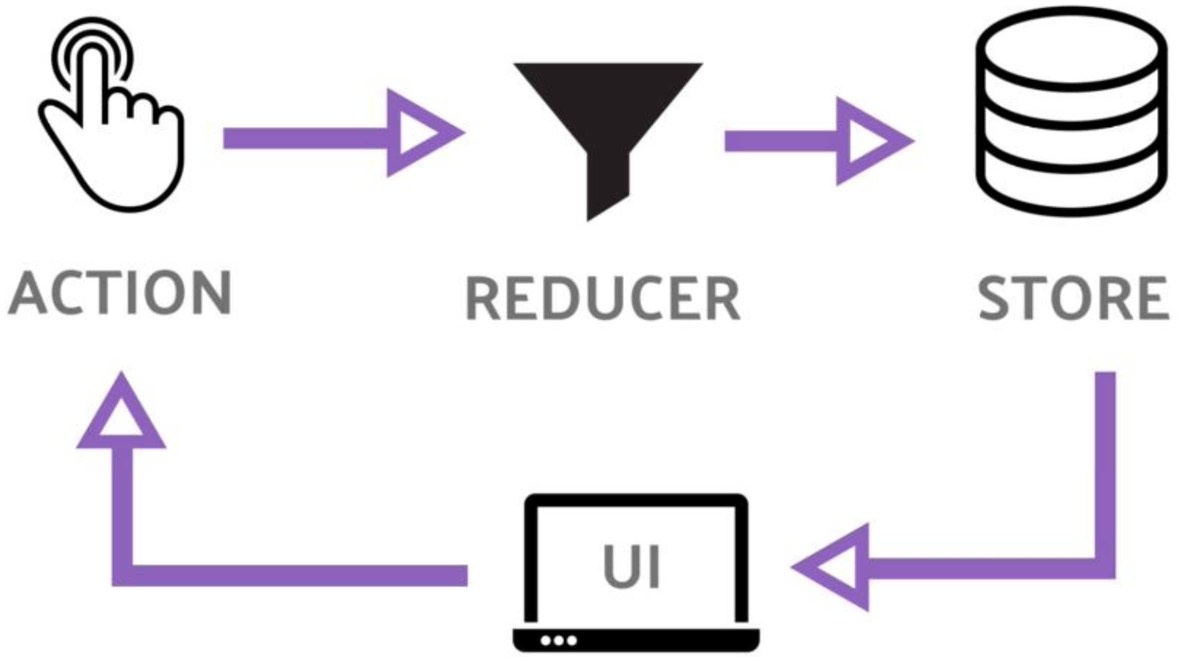
Component data flow with store



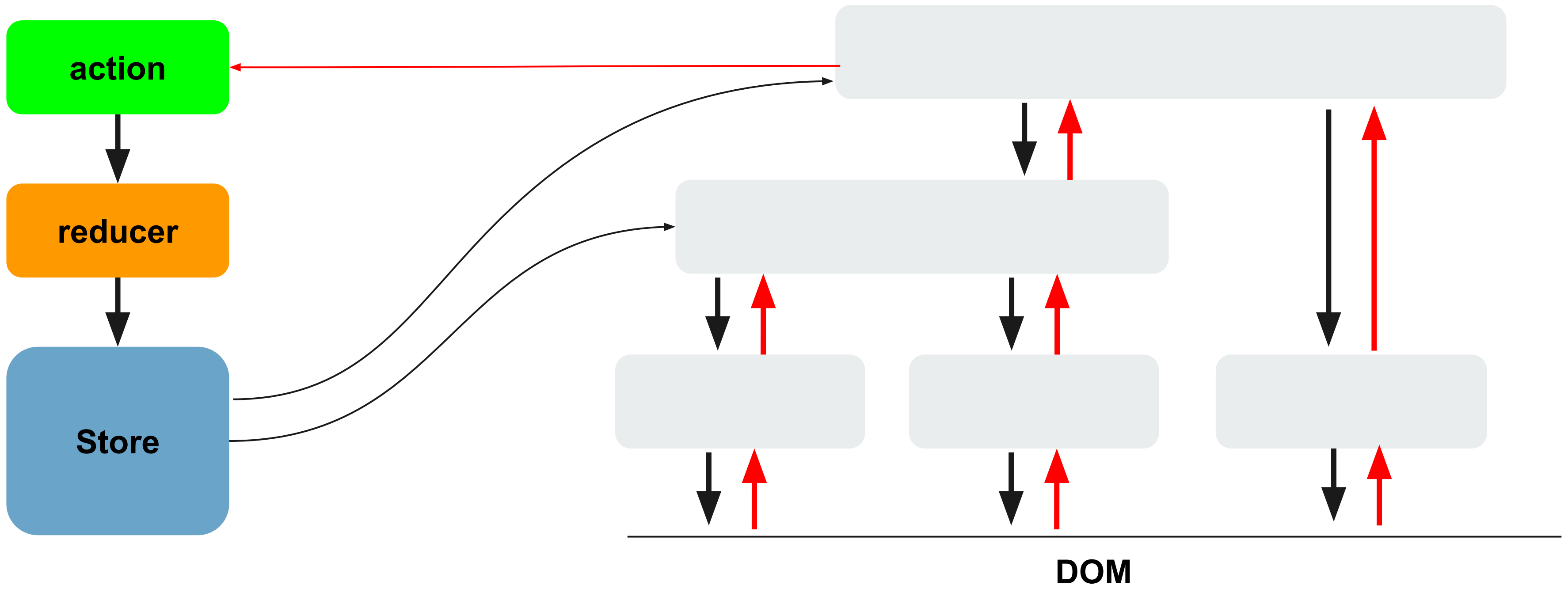
Props and callbacks hierarchy



One of implementations - Redux



Props and callbacks hierarchy



Store

The store has the following responsibilities:

- Holds application state;
- Allows access to state via getState();
- Allows state to be updated via dispatch(action);
- Registers listeners via subscribe(listener);
- Handles unregistering of listeners via the function returned by subscribe(listener).

```
import { createStore } from 'redux'

import myApp from './reducers'

const store = createStore(myApp)
```

Actions

Actions are payloads of information that send data from your application to your store. They are the only source of information for the store. You send them to the store using *store.dispatch()*.

```
const ADD_ARTICLE = 'ADD_ARTICLE'
function addArticle(text) {
  return {
     type: ADD_ARTICLE,
     article: text
  }
}
```

Sending actions to store

```
import store from  './store'


store.dispatch(addArticle('Some Article Text'))
```

Reducers

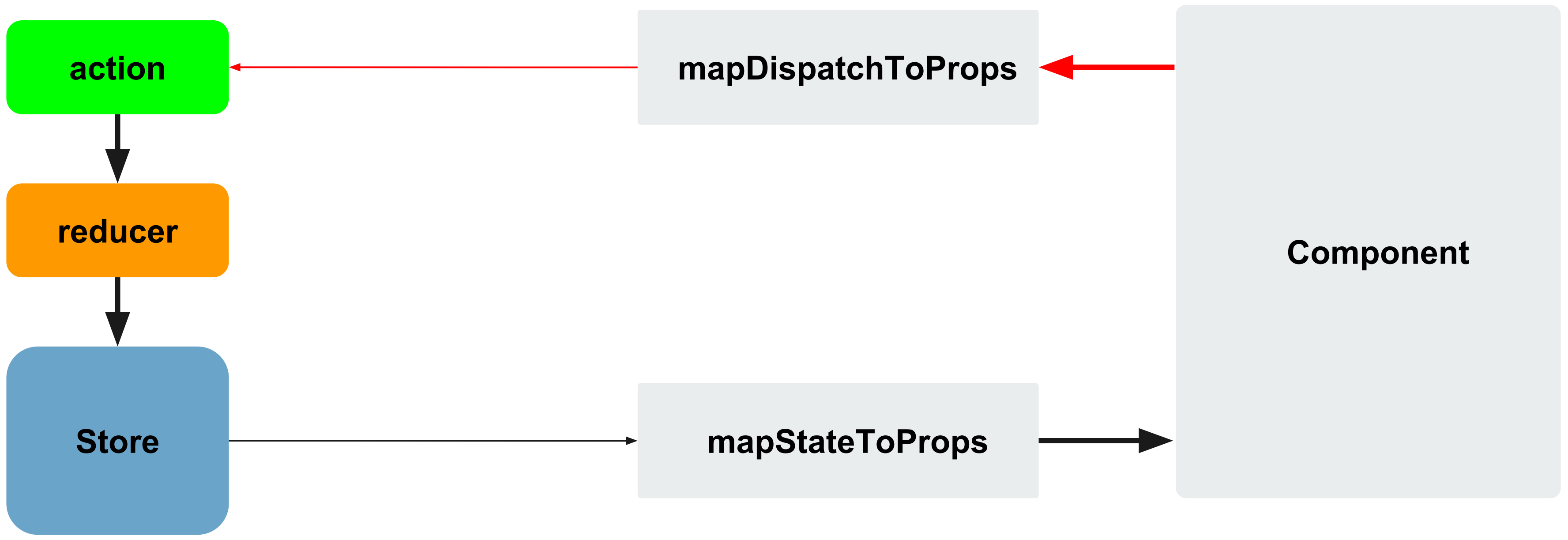
Reducers specify how the application's state changes in response to actions sent to the store. Actions only describe what happened, but don't describe how the application's state changes.

```
function myAppReducer(state = [], action) {
  switch (action.type) {
     case ADD_ARTICLE:
       return store.concat(action.text)
     default:
       return state
  }
}
```

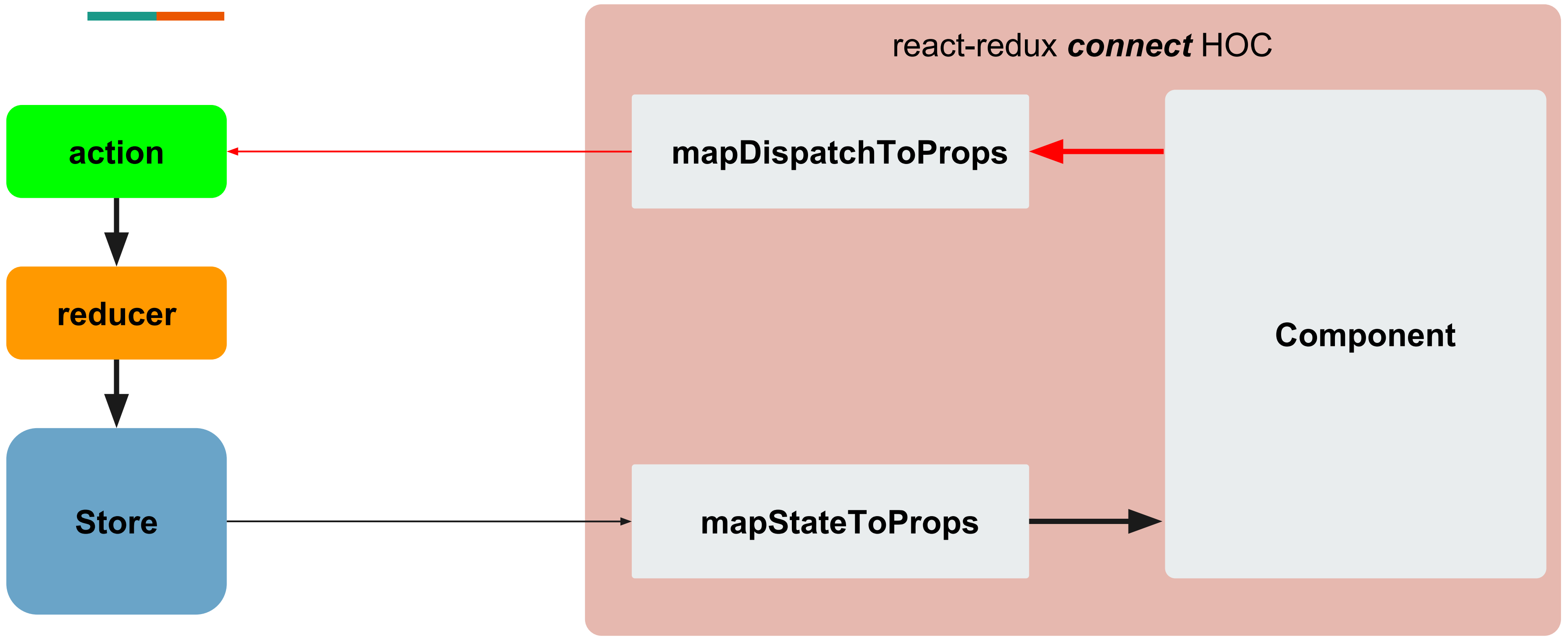
Listen for changes

```
store.subscribe(() =>
 console.log(store.getState())
)

store.dispatch(addArticle('LEL KEK'))

// [ 'LEL KEK' ] in console
```

React + Redux


action
reducer
Store
mapDispatchToProps
mapStateToProps
Component

React + Redux


action
reducer
Store
react-redux *connect* HOC
mapDispatchToProps
mapStateToProps
Component

Practice with Redux

Assignment

Assignment for the next week

- Think through your application data layer and it structure
- Add state management library to your project
- Make an AJAX request in your app and store response data in store
- Make a PR

Resources

- Redux simple app: https://codesandbox.io/s/github/tylerbuchea/my-simple-app/tree/master
- Redux handbook: https://redux.js.org/
- Lectures, resources and course project requirements: https://github.com/kos33rd/web-developer-course
- Our telegram group: https://t.me/JSforEntDev
- Github accounts to send PRs with complete tasks: @kos33rd, @AVVlasov