JIT Frontend Bootcamp

Karol Kowalczuk, Łukasz Wiśniewski

October 2018

Agenda

1. JavaScript basics
2. JavaScript ecosystem tools
3. Introduction to react.js
4. React.js – (more) advanced concepts
5. App state management with redux
6. Advanced debugging techniques and testing*

JavaScript is a single-threaded scripting programming language.

OOP? Functional programming?

Why not both?!

Execution Environments

Web

Both client- (~~interpreted~~ compiled by browsers to CLR nowadays) and server-side (executed by node.js environment)

Non-web

* Embedded scripting language (e.g. browsers extensions)
* General purpose scripting language
* Used as an application platform

OOP

- Creating objects directly from other objects
- Behaviour delegation
- Composability
- Extending objects dynamically

OLOO* („can do”)

* objects-linked-to-other-objects

- Tightly couples base class with inherited ones
- Must know the desired properties of inherited object before instantiation

Inheritance („is a”)

Behaviour delegation

Include a utility method if needed

```
var Task = {
  setID: function (ID) { this.id = ID },
  outputID: function () { console.log(this.id) },
}

// make `XYZ` delegate to `Task`
var XYZ = Object.create(Task)

XYZ.prepareTask = function (ID, Label) {
  this.setID(ID)
  this.label = Label
}

XYZ.outputTaskDetails = function () {
  this.outputID()
  console.log(this.label)
}
```

Credits to: Kyle Simpson

Functional programming

Pure functions

Given the same input always gives the same output; makes composition easy as pie

No shared state

Do not pass the properties of the object between scopes

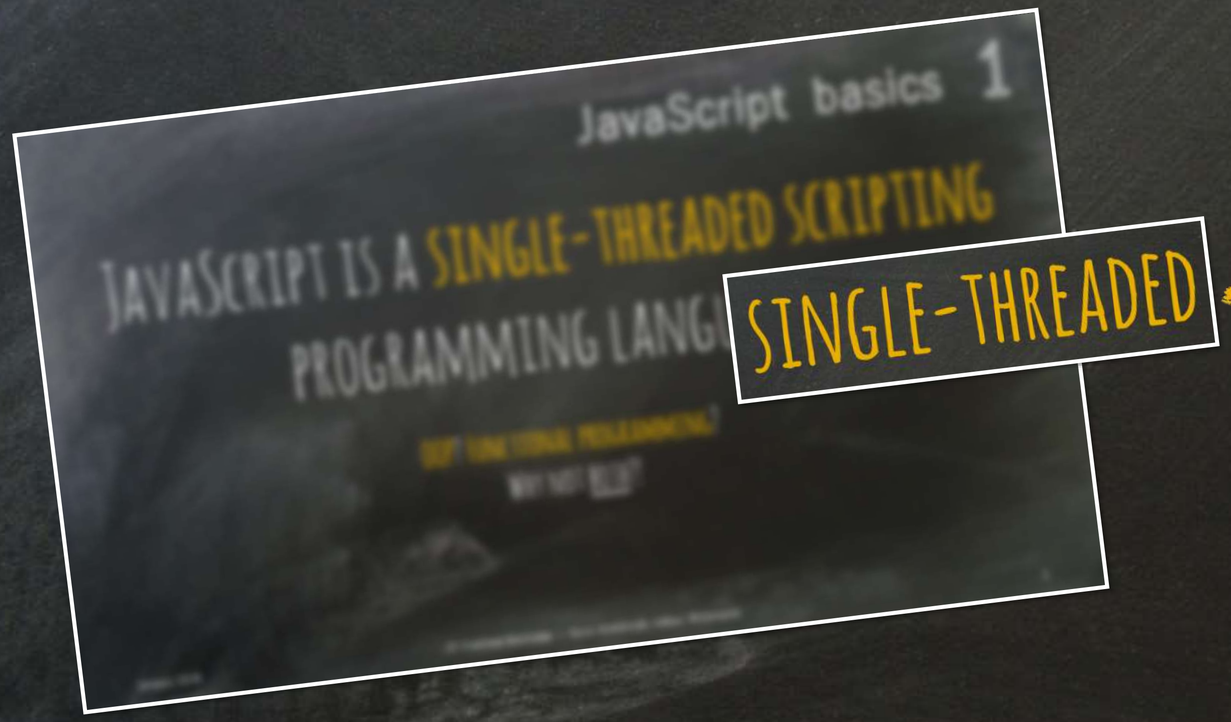
No state mutations

Allows to track the transfomations and do *time-traveling*

No side effects

Just be deterministic

Asynchronous calls



Wait, what?

ONE THREAD == ONE STACK == ONE THING AT A TIME

EXCEPT NOT REALLY.

JavaScript basics 8

```
console.log("Hello")

setTimeout(function cb() {
  console.log(":)")
}, 5000)

console.log("World!")
```

stack

threads (WebAPIs)

console

event loop

tasks queue

JavaScript basics 8

```
console.log("Hello")

setTimeout(function cb() {
  console.log(":)")
}, 5000)

console.log("World!")
```

stack

main()

threads (WebAPIs)

console

event loop

tasks queue

JavaScript basics 8

```
console.log("Hello")

setTimeout(function cb() {
  console.log(":)")
}, 5000)

console.log("World!")
```

stack

log("Hello")

main()

threads (WebAPIs)

console

Hello

event loop

tasks queue

JavaScript basics 8

```
console.log("Hello")

setTimeout(function cb() {
  console.log(":)")
}, 5000)

console.log("World!")
```

stack

main()

threads (WebAPIs)

console

Hello

event loop

tasks queue

JavaScript basics 8

```
console.log("Hello")

setTimeout(function cb() {
  console.log(":)")
}, 5000)

console.log("World!")
```

stack

setTimeout(cb)

main()

threads (WebAPIs)

console

Hello

event loop

tasks queue

JavaScript basics 8

```
console.log("Hello")

setTimeout(function cb() {
  console.log(":)")
}, 5000)

console.log("World!")
```

stack

setTimeout(cb)

main()

threads (WebAPIs)

timer cb

console

Hello

event loop

tasks queue

JavaScript basics 8

```
console.log("Hello")

setTimeout(function cb() {
  console.log(":)")
}, 5000)

console.log("World!")
```

stack

main()

threads (WebAPIs)

timer

cb

console

Hello

event loop

tasks queue

JavaScript basics 8

```
console.log("Hello")

setTimeout(function cb() {
  console.log(":)")
}, 5000)

console.log("World!")
```

stack

log("World")

main()

threads (WebAPIs)

timer cb

console

Hello
World

event loop

tasks queue

JavaScript basics 8

```
console.log("Hello")

setTimeout(function cb() {
  console.log(":)")
}, 5000)

console.log("World!")
```

stack

main()

threads (WebAPIs)

timer cb

console

Hello
World

event loop

tasks queue

JavaScript basics 8

```
console.log("Hello")

setTimeout(function cb() {
  console.log(":)")
}, 5000)

console.log("World!")
```

stack

threads (WebAPIs)

timer cb

console

Hello
World

event loop

tasks queue

JavaScript basics 8

```
console.log("Hello")

setTimeout(function cb() {
  console.log(":)")
}, 5000)

console.log("World!")
```

stack

threads (WebAPIs)

console

Hello
World

event loop

tasks queue cb

JavaScript basics 8

```
console.log("Hello")

setTimeout(function cb() {
  console.log(":)")
}, 5000)

console.log("World!")
```

stack

cb

threads (WebAPIs)

console

Hello
World

event loop

tasks queue

JavaScript basics 8

```
console.log("Hello")

setTimeout(function cb() {
  console.log(":)")
}, 5000)

console.log("World!")
```

stack

log(“:)”)

cb

threads (WebAPIs)

console

Hello
World
:)

event loop

tasks queue

JavaScript basics 8

```
console.log("Hello")

setTimeout(function cb() {
  console.log(":)")
}, 5000)

console.log("World!")
```

stack

cb

threads (WebAPIs)

console

Hello
World
:)

event loop

tasks queue

JavaScript basics 8

```
console.log("Hello")

setTimeout(function cb() {
  console.log(":)")
}, 5000)

console.log("World!")
```

stack

threads (WebAPIs)

console

Hello
World
:)

event loop

tasks queue

JavaScript basics 9

XHR*

Since 1999

Registering callbacks for particular events (onload, onerror)

Promises

since 2012

Objects representing async actions result; there are three different states of Promises: fulfilled, rejected, pending

Fetch API

Since 2015 (Chrome v40+)

Wraps HTTP requests with a Promise; still not widely supported by browsers

* XMLHttpRequest

ECMAScript 2015

6th version of standardized scripting-language specification

Initially created to standardize JavaScript.

Arrow functions

No need to bind "this" anymore

```
// Previously
var _this = this
button.click(function (event) {
  _this.sendData()
})

// Now
button.click((event) => {
  this.sendData()
})
```

JavaScript basics 12

```
// Previously
var processData = function (length, width, color) {
  // Be careful with zeros!
  var length = length || 100
  var width = width || 20
  var color = color || "blue"

  // function body here
}

// Now
var processData = function (length = 100, width = 20, color = "blue") {
  // function body here
}
```

Default parameters

Get rid of "or" evaluations

String literals

Put variables in a string directly

```
// Previously
var greetings = "Hello " + firstName + " " + lastName
  + "\n" + "Your cart contains: " + itemsNo + " items."

// Now
var greetings = `Hello ${firstName} ${lastName}
  Your cart contains: ${itemsNo} items.`
```

```
// Previously
var response = callApi(params)
var body = response.body,
  status = response.status

var listContent = [1, 5, 10, 12]
var first = listContent[0],
  third = listContent[2]

// Now
var response = callApi(params)
var { body, status } = response

var listContent = [1, 5, 10, 12]
var [first, , third] = listContent
```

Destructuring assignment

It's a kind of magic

Rest/Spread operator

Destructuring made even simpler

```
// Now
var details = {
  firstName: "John",
  lastName: "Doe",
  age: 34,
}

var { age, ...rest } = details
console.log(rest) // { firstName: "John", lastName: "Doe" }
var copiedDetails = { ...details }
```

```
// Previous
var x = 1000
if (true) {
  var x = 200
  console.log(x) // 200
}
console.log(x) // 200

// New
let x = 1000
if (true) {
  let x = 200
  console.log(x) // 200
}
console.log(x) // 1000
```

Block-scoped "let"...

Couples variables with the scope

...and "const"

Disallows reassignments

```
// Previous
var x = 1000
if (true) {
  var x = 200
  console.log(++x) // 201
}
console.log(x) // 201

// New
const x = 1000
if (true) {
  const x = 200
  console.log(++x) // TypeError: Assignment to constant variable.
}
console.log(x)
```

```
// Previous
  // module.js
  var port = 3000

  // main.js
  var service = require("module.js")
  console.log(service.port)

// New
  // module.js
  export var port = 3000

  // main.js
  import { port } from "module"
  console.log(port)
```

Modules

Import only what's really needed

For-of loops

Because every language has them :)

```
// Previous
var iterable = [1, 2, 3, 4, 5, 7]
for (var i = 0; i < iterable.length; ++i) {
  console.log(iterable[i])
}

// New
var iterable = [1, 2, 3, 4, 5, 7]
for (var i of iterable) {
  console.log(i)
}
```

JavaScript ecosystem tools 1

Choice

taken to the

extreme

(chaos)

Package managers:
NPM vs YARN

JavaScript ecosystem tools 3

NPM

* Package manager
* Public package repository
* Private packages
* self-hosting repository (npm Enterprise)

Yarn

* "just" a Package manager
* Optimized for performance, package caching
* Lock file

JavaScript ecosystem tools 5

Both use

package.json

```
{
  "name": "test",
  "version": "1.0.0",
  "description": "description",
  "main": "index.js",
  "scripts": {
    "test": "echo 1"
  },
  "dependencies": {
    "jquery": "3.2.1"
  },
  "license": "ISC"
}
```

Task Runners:

Grunt, Gulp, DIY scripts

Grunt

* Declarative configuration
* File-based processing
* Relies on plugins

JavaScript ecosystem tools 8

Gulp

* Configuration = Code
* In-memory, stream-based processing
* Still relies on plugins

DIY (NPM) SCRIPTS

* Just your code
* Or a single command line
* Executed by NPM

Compilers/Transpilers:

Millions of them

(but mainly Babel)

Babel

Cutting edge JS

\/

Ordinary ECMAscript

TypeScript

Cutting edge JS (with types)

\/

Ordinary ECMAscript

The rest

* 'alternative' JS syntax languages (ex. – CoffeeScript)
* Full blown runtime environments (Scala.js, Elm)

Bundlers:

The 'pack it for the browser' gizmo

Before

* <script>, <link>, <style> managed by hand
* Should A.js be before, or after B.js?
* Global variables

After (Webpack)

* Look Ma, a single JS file!
* And a cryptic Declarative + Code based configuration

Hands-on

Basic webpack configuration

source available: 00-base

Tests:

need a guide to choose a tool to **just run them**

Testing

* test runners (mocha, jasmine, jest, karma)
* The 'actual' tests (mocha, jasmine, jest)
* assertions (chai, jasmine, jest)
* snapshot testing (jest, ava)

Testing

* Mocks, spies (sinon, jasmine, jest)
* code coverage (istanbul)
* Browser automation & headless browser environments (PhantomJS, Protractor, testcafe)

Linting & formatting:

Optional, but handy

JavaScript ecosystem tools 22

Linting: ESLint

formatting: Prettier

Introduction to react.js 1

React.js is **NOT** a framework. It is **ONLY** a view library.

Introduction to react.js 2

* No controllers
* No digest loops
* No directives
* No view models

Just Components.

In react.js world every rendered block is a component.

Introduction to react.js 3

Separation of ~~concerns~~ components

(only) if they are designed and built as self-contained entities.

* Composability and flexibility
* Reusability
* Maintainability
* Ability to be tested independently

Introduction to react.js 4

JSX

XML-LIKE SYNTAX EXTENSION FOR ECMASCRIPT

```
return (
  <div className="wrapper s12">
    <p>How the hell is it possible?!</p>
  </div>
)
```

HTML!?

Introduction to react.js 5

Functional components

Define simple and stateless components.

```
const SimpleComponent = (props) => {
  return (
    <div className="wrapper">
      <p>Simple, huh?</p>
    </div>
  )
}
```

Class-based components

Used for more advanced entities maintaining their own state.

```
const ClassComponent = React.createClass({
  // ...
  render: () => {
    return (
      <div className="wrapper">
        <p>I can have my own state!</p>
      </div>
    )
  },
})
```

Data flow

No library

All-to-all

Any component can communicate directly with any other component.

Angular 2+

Two-way data binding

Compares values representing application state (in all components) before and after browser events.

React.js

One-Way data binding

Model is the *single source of truth*. More deterministic than two-way binding – side effects are not possible.

Introduction to react.js 7

```
const Parent = (props) => {
  return (
    <div><Child name="firstComponent" /></div>
  )
}

const Child = (props) => {
  processName = () => {
    doProcessing(props.name)
  }
  // ...
}
```

Data in child component is accessed via "**PROPS**" object.

PROPS within children are **IMMUTABLE** (cannot be overwritten).

Introduction to react.js 8

```
const Child = React.createClass({
  // ...
  disableButton: () => {
    this.setState({ disabled: true })
  },

  render: () => {
    return <button type="submit"
      disabled={this.state.disabled} />
  },
})
```

Each component can have its own isolated **state**.

The state can be accessed by its owner only. It **can be mutated**.

By default react.js rerenders components whenever their **props** or **state** change.

Data flows down

```
const Grandparent = React.createClass({
  // ...
  render: () => {
    return (
      <Parent
        onEnabledClick={this.handleItemClick}
        item={this.currentItem}
      />
    )
  },
})
```

```
const Parent = (props) => {
  return (
    <Child
      onEnabledClick={props.onEnabledClick}
      item={props.item}
    />
  )
}
```

```
const Child = (props) => {
  return (
    <div className="itemContainer">
      <label>{props.item.name}</label>
      <input type="check" onClick={props.onEnabledClick} />
    </div>
  )
}
```

Introduction to react.js 10

EVENTS FLOW UP

```
const Grandparent = React.createClass({
  // ...
  render: () => {
    return (
      <Parent
        onEnabledClick={this.handleItemClick}
        item={this.currentItem}
      />
    )
  },
})
```

```
const Parent = (props) => {
  return (
    <Child
      onEnabledClick={props.onEnabledClick}
      item={props.item}
    />
  )
}
```

```
const Child = (props) => {
  return (
    <div className="itemContainer">
      <label>{props.item.name}</label>
      <input type="check" onClick={props.onEnabledClick} />
    </div>
  )
}
```

Introduction to react.js 10

Data flows down*

Events flow up*

```
const Grandparent = React.createClass({
  // ...
  render: () => {
    return (
      <Parent
        onEnabledClick={this.handleItemClick}
        item={this.currentItem}
      />
    )
  },
})
```

```
const Parent = (props) => {
  return (
    <Child
      onEnabledClick={props.onEnabledClick}
      item={props.item}
    />
  )
}
```

```
const Child = (props) => {
  return (
    <div className="itemContainer">
      <label>{props.item.name}</label>
      <input type="check" onClick={props.onEnabledClick} />
    </div>
  )
}
```

Data flows down

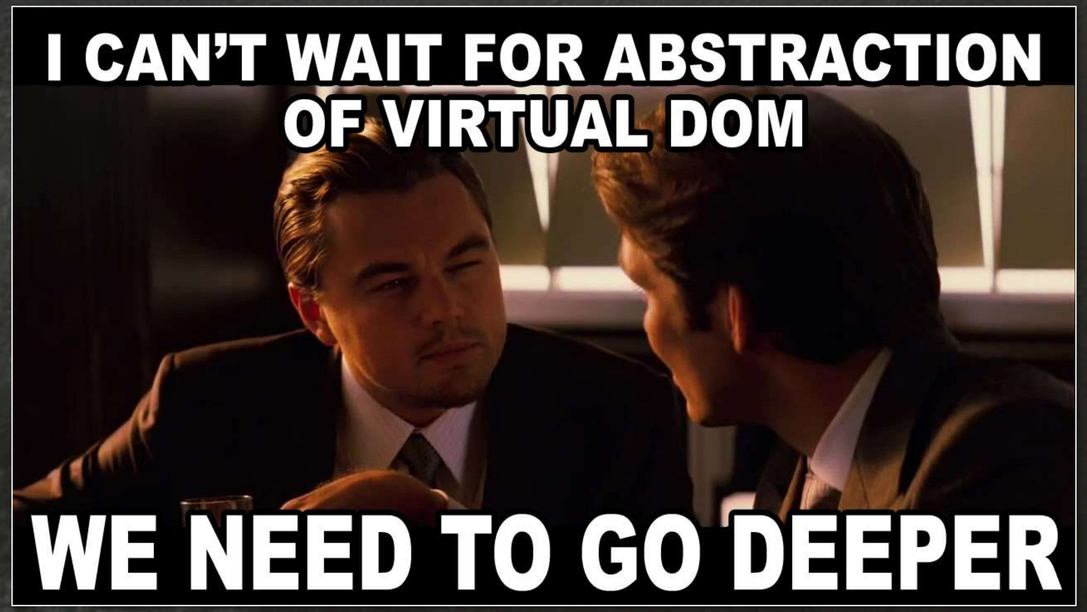
events flow up

Just like in DOM.

Virtual DOM

* Pure JavaScript, in-memory **abstraction of real DOM**
* Whenever render() fires (which means that state or props have been updated), react.js updates the in-memory DOM representation
* If anything has changed, react.js modifies the real DOM to match

Because **performance** matters.



Hands-on

First react component

source available: 01-react-base

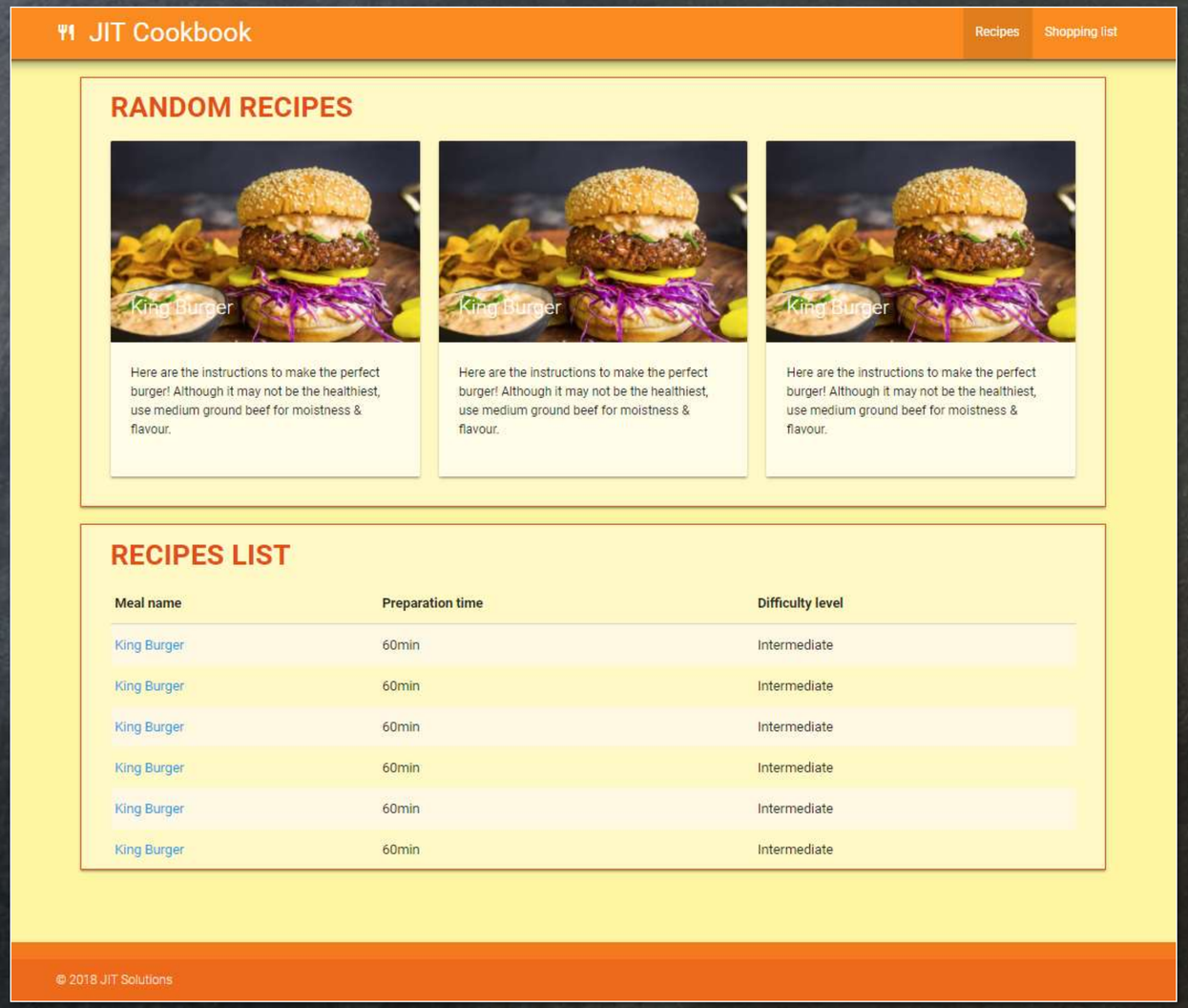
Component split

Has to be done based on the application UI design

&

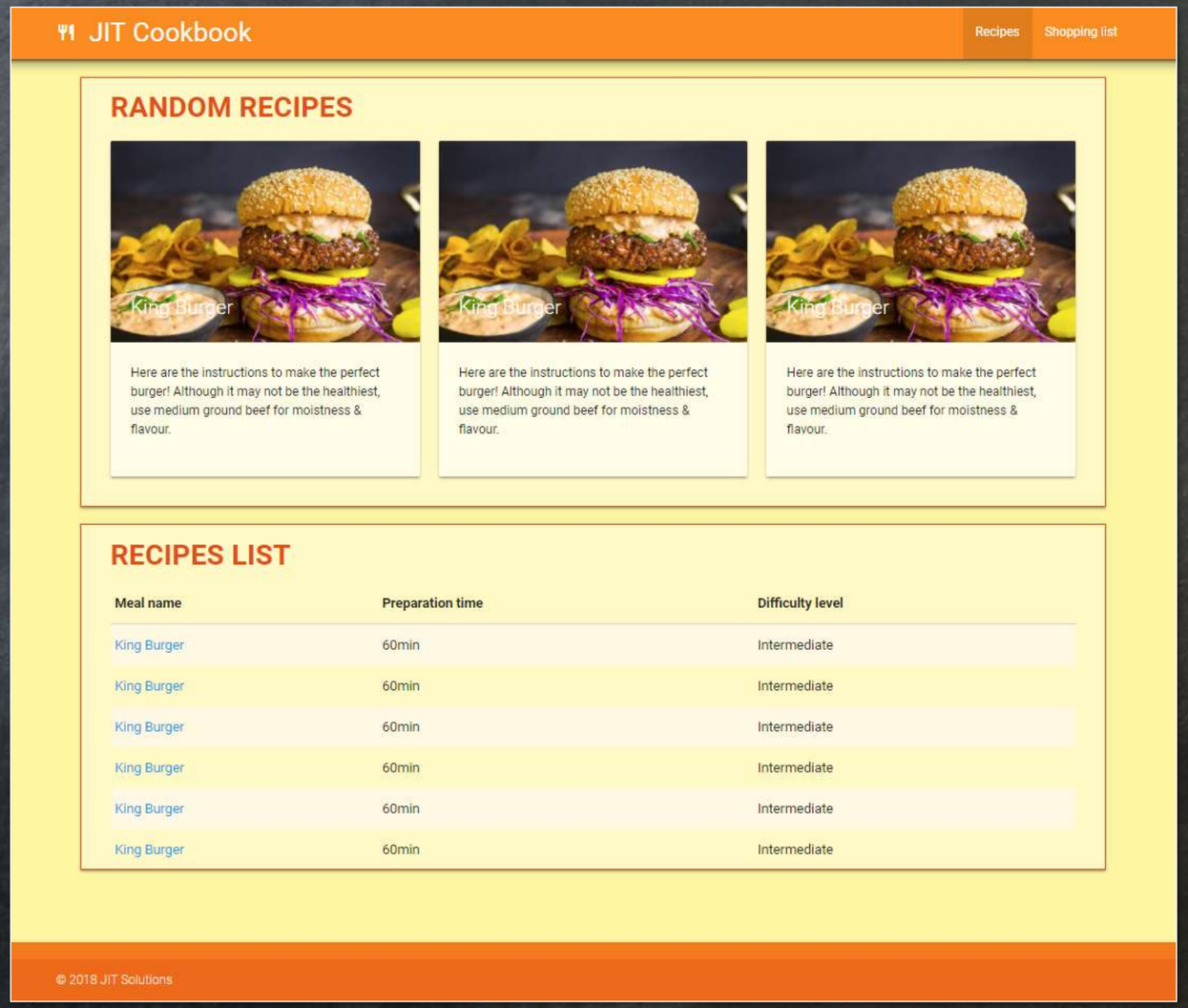
business requirements analysis

Introduction to react.js 15



JIT Cookbook

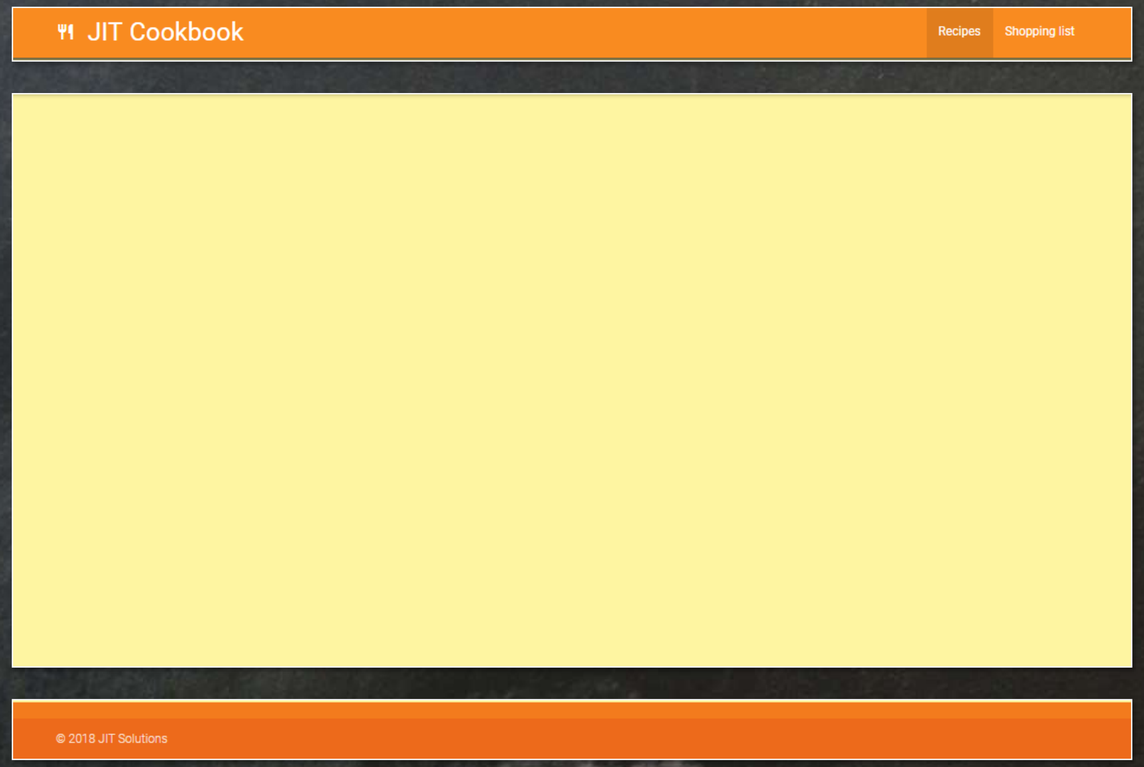
* Navigation bar at the top should show the application name with icon (left side) and navigation panel between dashboards (right side)
* Currently active dashboard tab is indicated with darker color
* User should be able to switch between the dashboards by clicking on particular tabs
* By default Recipes dashboard should be shown
* Footer contains only the copyrights section with company name



JIT Cookbook – recipes dashboard

* Recipes dashboard's main screen is responsible for displaying three random recipes in the top panel and full list of recipes in the bottom panel
* After clicking on any name from the recipes list, the user should see a single recipe view (the whole page should be replaced with page representing single recipe)
* A single recipe view should contain detailed description of the meal and list of ingredients needed in order to prepare it

Introduction to react.js 16

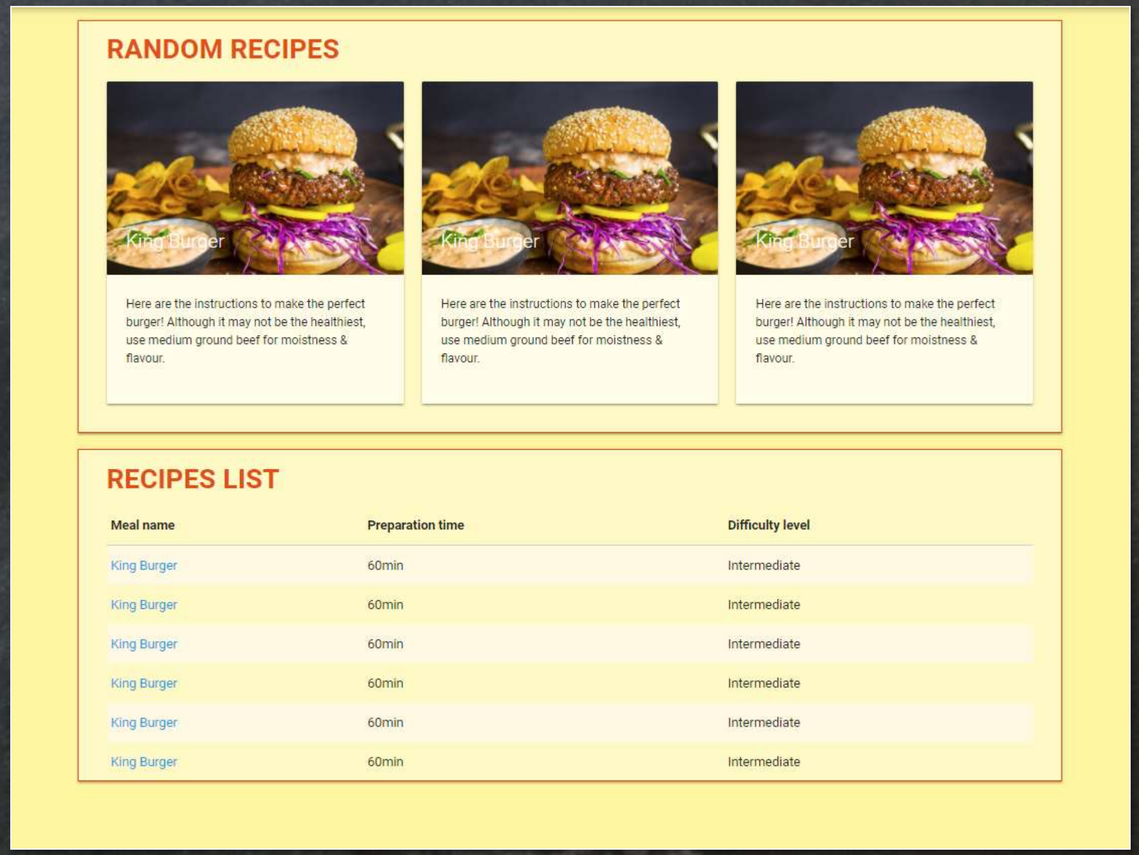


Navbar

Content

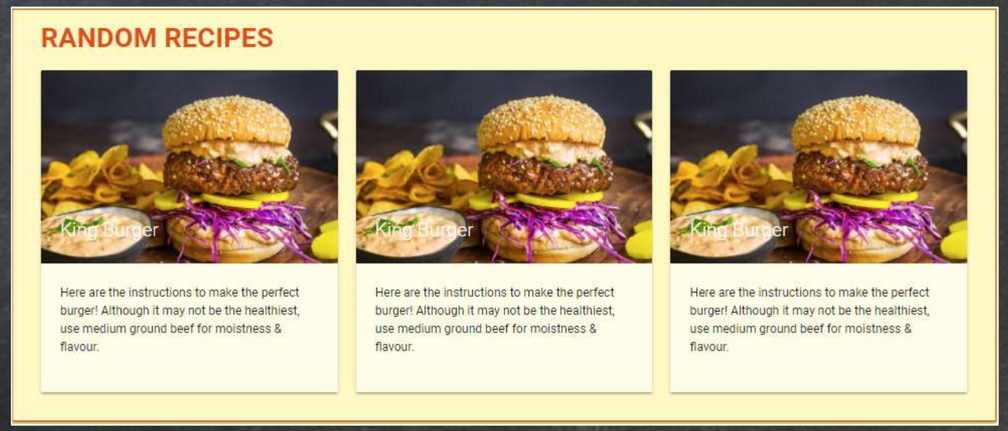
Footer

Introduction to react.js 16

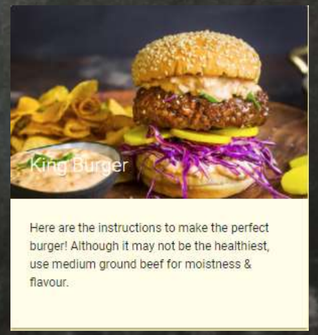


RecipesContainer

Introduction to react.js 16



RandomRecipesList



RecipeCard

Introduction to react.js 16

RECIPES LIST

Meal name	Preparation time	Difficulty level
King Burger	60min	Intermediate
King Burger	60min	Intermediate
King Burger	60min	Intermediate
King Burger	60min	Intermediate
King Burger	60min	Intermediate
King Burger	60min	Intermediate

RecipesTable

King Burger	60min	Intermediate

RecipesTableEntry

Hands-on

Base component split

source available: 02-react-base-layout

PropTypes

* When the application grows it's quite important to ensure that all objects passed to component as **props** match the desired types
* **PropTypes** is a react.js built-in mechanism to run typechecking
* All you have to do is to assign a specific property to a component

Hands-on

Props typechecking

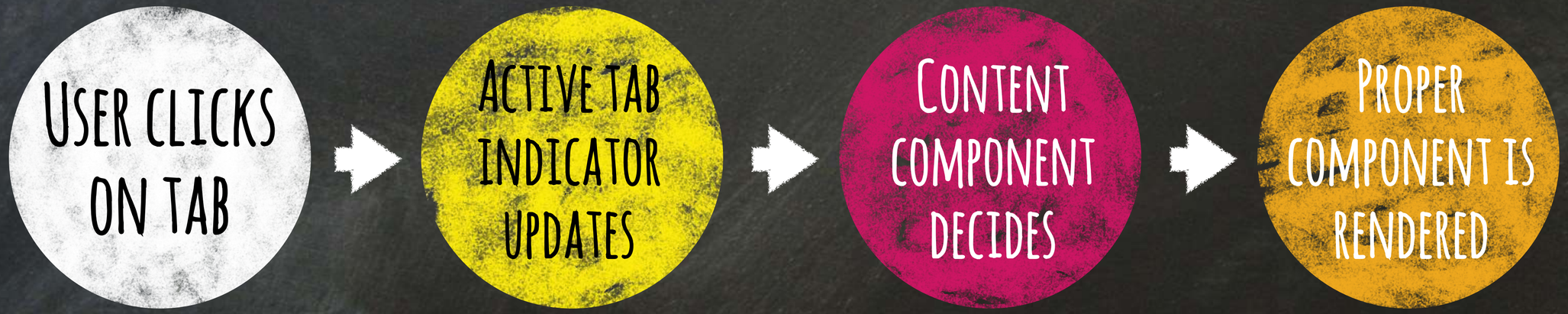
(let's convert some of the components to classes as well)

source available: 03-class-based-components

Action binding

Making application interactive!

Introduction to react.js 21



Conditional rendering

Decide what to render based on current conditions

```
render = () => {
  const userLoggedIn = !isGuest()

  return userLoggedIn
    ? <button name="logOut " type="submit" />
    : <p>Please log in first</p>
}
```

```
render = () => {
  return (
    <div className="profileCard">
      {
        shouldShowAvatar && (
          <img src={this.props.user.imgUrl}
            alt={this.props.user.name} />
        )
      }
      <p>{this.props.user.name}</p>
    </div>
  )
}
```

Hands-on

Bringing some life to the application

source available: 04-state-action-binding

react.js more advanced concepts 1

Lists rendering

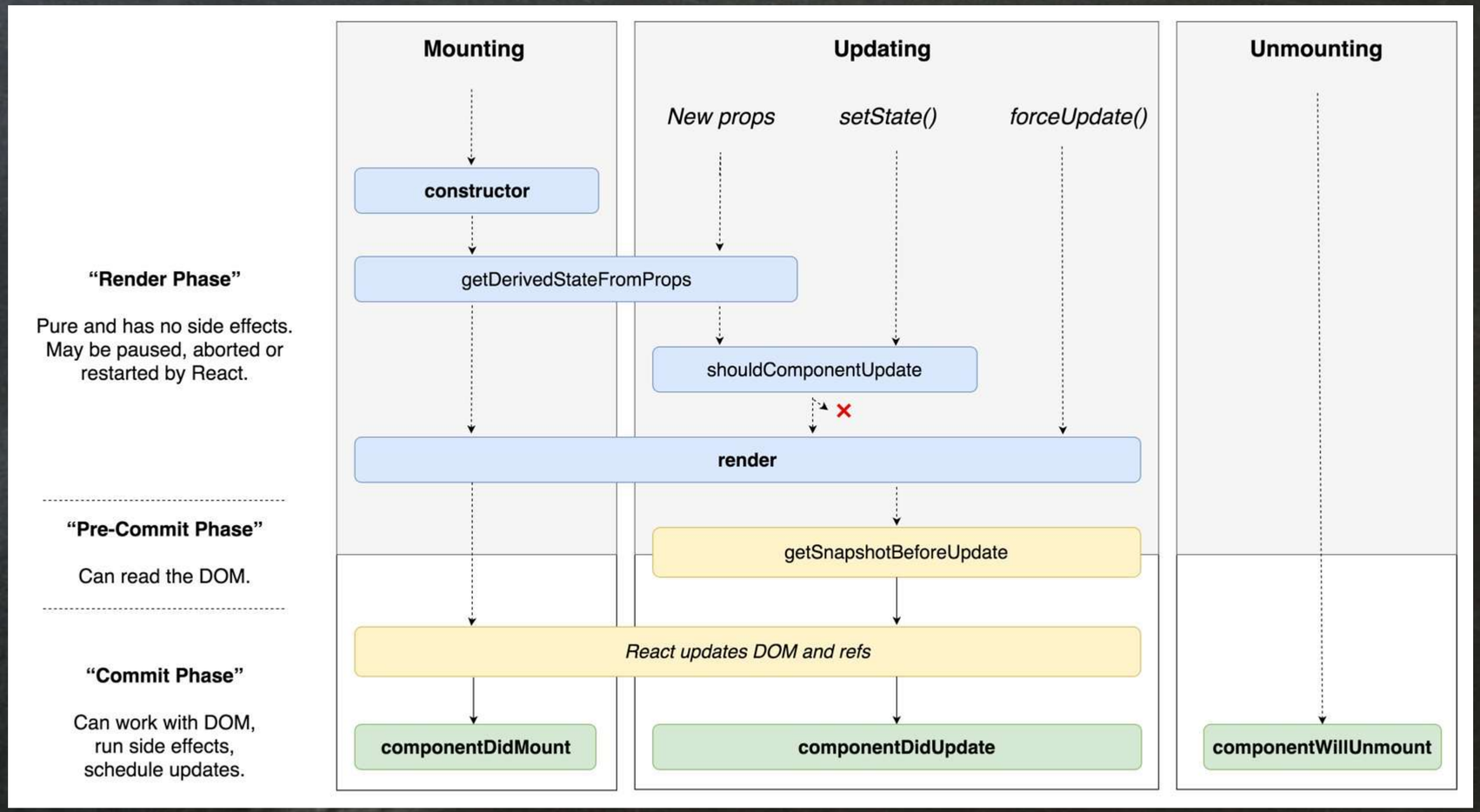
```
render = () => {
  return (
    <div className="usersList">
      <ul>
        {this.props.users.map((user) => {
          return <li key={user.id}>{user.id}. {user.name}</li>
        })}
      </ul>
    </div>
  )
}
```

"KEY" PROPERTY IS CRUCIAL.

Component lifecycle

* React.js allows us to intercept each phase of components lifecycle
* Useful when we want to update component configuration based on the calculation on new props or the difference between new and old ones
* Hooks may be necessary when you're dealing with 3rd party libraries

react.js more advanced concepts 3



Hands-on

Enhance lists renderings and inspect lifecycle hooks

source available: 05-list-renderings-and-hooks

PureComponent

Special subtype of react.js component. Reduces number of wasted renderings.

Functional components equivalent: React.memo(function(props))

Hands-on

Optimize number of renderings by inheriting from React.PureComponent

source available: 06-pure-components

Error handling

During runtime, when exception happens within component code (either constructor, render method or lifecycle methods), the application will crash - the whole component tree will not be rendered.

Error handling

We can catch those errors and render fallback UI instead.

Additional hooks

componentDidCatch(error, info)

* Called during commit phase
* Side effects are allowed
* Will not work with server-side rendering

static getDerivedStateFromError(error)

* Called during render stage
* State changes affecting the render phase should be placed here

Hands-on

Sample error handling for runtime exceptions

source available: 07-error-handling

App state management with redux 1

Redux is a library that brings **state management** to another level.

It was strongly inspired by **flux** architecture.

App state management with redux 2

The only source of state updates; contains a type and often additional context

Routes particular actions to registered stores (via reducers)

Stores the state tree of the whole application

Rendered based on the store (state) updates; can trigger actions too

App state management with redux 3

Why redux produces so deterministic state?

* The store is a single source of truth; the whole application state comes from the store only.
* State of the application is read-only. All modifications should be triggered consciously by direct action calls. They are centralized and synchronized by a dispatcher.
* Changes are applied with pure ~~functions~~ reducers. Reducers are functions which combine a previous state and incoming action into the new state.

App state management with redux 4

Action creator

A state update intention

```
export const addToFavorites = (videoId) => {
  return {
    type: "VIDEO_ADD_TO_FAVORITES",
    videoId,
  }
}
```

App state management with redux 5

```
const initialState = {
  favorites: [],
}

export default reducer = (state = initialState, action) => {
  switch (action.type) {
    case "VIDEO_ADD_TO_FAVORITES":
      return {
        favorites: [ ...state.favorites, action.videoId ]
      }
  }

  return state
}
```

Reducer

Converts previous state and executed action into new state

App state management with redux 6

Store

Always keeps the most recent state

```
import { combineReducers, createStore } from "redux"

import videoReducer from "services/video/reducer"

const reducers = combineReducers({
  videosState: videoReducer,
})

const store = createStore(reducers)
export default store
```

App state management with redux 7

React and Redux

```
const globalState={
  username: "HotBlonde69",
  credentials: [ "viewer" ],
  isAdmin: false,
}

render(
  <Provider state={globalState}>
    <App />
  </Provider>,
  document.getElementById("app")
)
```

"globalState" will be available on each level of component tree without passing it explicitly.

App state management with redux 8

Components

Responsible for displaying the data only

Containers

Fetches the data from the store and stores it in the local state; acts as a proxy between store and components

App state management with redux 9

```
const mapStateToProps = (store) => {
  return {
    favorites: store.videosState.favorites,
  }
}

const mapDispatchToProps = (dispatch) => {
  return bindActionCreators({
    addToFavorites: VideoActions.addToFavorites,
  }, dispatch)
}

export default connect(
  mapStateToProps,
  mapDispatchToProps
)(VideosContainer)
```

Three steps needed to combine react component with redux

App state management with redux 10

Hands-on

Managing application state with redux

sources available: 08-redux-store, 09-redux-recipes-store, 10-selected-recipe-screen

Redux thunk

* Allows to return functions from action creators
* Used for dispatching actions only when certain conditions are met (mostly when the data fetch from external resource completes)

App state management with redux 12

Store with thunks support

That's what middleware is for

```
import { combineReducers, createStore } from "redux"
import thunk from "redux-thunk"

import videoReducer from "services/video/reducer"

const reducers = combineReducers({
  videosState: videoReducer,
})

const store = createStore(reducers, applyMiddleware(thunk))
export default store
```

App state management with redux 13

```
const fetchNewVideosError = (error) => {
  return {
    type: "VIDEOS_FETCH_NEW_ERROR",
    error,
  }

const fetchNewVideosAsync = () => {
  return (dispatch) => {
    dispatch(fetchNewVideos())

    return videosApi.fetchNew().then(
      (data) => dispatch(fetchNewVideosSuccess(data)),
      (error) => dispatch(fetchNewVideosError(error)),
    )
  }
}
```

Thunk creation

Nesting action dispatching

Hands-on

Fetching data using external API

source available: 11-redux-thunk

Redux form

* HTML Input elements are not easy to maintain in react
* That library automatically links input of various types with the auto-generated store of a very specific structure
* It also provides a wrappers for raw HTML form fields with intuitive API

App state management with redux 16

Exposing form component

Allows to use redux form API with

```
export default reduxForm({
  form: "userRegistrationForm",
})(UserRegistrationForm)
```

App state management with redux 17

```
render() {
  return (
    <div className="formWrapper">
      <UserRegistrationForm initialValues={{
        yearOfBirth: 2000,
        group: "Unassigned",
      }}
      onSubmit={(data) => this.submitData(data)}/>
    </div>
  )
}
```

Injected API

No need to reinvent the wheel

Hands-on

Utilizing forms to update the data on server

source available: 12-redux-forms