ReactJS

Agenda

- Introduction
- Tooling
- React
- Rendering
- State
- Lifecycle
- Hooks
- Tests
- Redux
- REST Architecture
- Routing
- Forms
- Performances
- Isomorphism
- Going further
- Conclusion

Logistic

- Agenda
- Lunch and breaks
- Other questions ?

Introduction

Agenda

- *Introduction*
- Tooling
- React
- Rendering
- State
- Lifecycle
- Hooks
- Tests
- Redux
- REST Architecture
- Routing
- Forms
- Performances
- Isomorphism
- Going further
- Conclusion

JavaScript

- JavaScript was initially created to dynamize HTML pages
- It becomes more and more used:
  - AJAX queries
  - Libraries (jQuery, Lodash, ...)
  - Single Page App (Angular, Backbone, Ember, ...)
  - Server side (Node.js, NoSQL, etc...)

Language history

Time	Editor	Event
Dec. 1995	Sun/Netscape	JavaScript announcement (a.k.a. LiveScript)
Mar. 1996	Netscape	JavaScript coming in Netscape 2.0
Aug. 1996	Microsoft	JScript release in Internet Explorer 3.0
Nov. 1996	Netscape	JavaScript standardization by Ecma
Jun. 1997	Ecma	ECMAScript is universally adopted
1998	Adobe	ActionScript

- Ecma is a private european standards organization
- Not limited to IT standards. Examples : electronic, hardware

ECMAScript

Versions of the **ECMAScript** standard

Ver	Time	Evolution
1	Jun. 1997	ECMAScript 1 adoption
2	Jun. 1998	Standard rewriting, first version of the JavaScript we know today
3	Dec. 1999	RegExp, `try`/`catch`, `Error`, ... Adopted everywhere
4	Discarded	
5	Dec. 2009	Fixes V3 edge-cases, new methods like `Array::map` Most widespread support
6	Jun. 2015	Lots of new concepts and syntactic sugar like `class`, template strings, `const`/`let`, ...
7	Jun. 2016	Exponentiation operator `**` and `Array::includes`
8	Jun. 2017	`async`/`await`, `Object::values`, string padding, ...

Node.js

- Open-source project created by Ryan Dahl.
- First release in 2011
- V8 JavaScript engine
  - Use to drive asynchronous system API (filesystem, network, ...)
- Non-exhaustive list of usages:
  - Web server
  - Command line tools

npm

- Node Package Manager
- Come with Node.js installation
- More than 600.000 packages

A lot of command line tools are available:

```
$ npm install -g chalk-cli
$ ls -l | chalk blue

// $ ls -l | npx chalk-cli blue
```

JavaScript modules

- Node.js has popularized the concept of code-splitting in JavaScript.
  - CommonJS modules
  - No more global variables
  - The dependencies of a file have to be imported at the top of the file
  - has spread on the browser thanks to Webpack/Browserify

```
// display.js
module.exports = function display(text) {
    console.log(text);
};

// main.js
const display = require('./display');

display('Zenika');
```

JavaScript modules

CommonJS has inspired the ES2015 modules.

```
// display.js
export default function display(text) {
    console.log(text);
};

// main.js
import display from './display';

display('Zenika');
```

- Not supported yet
- Widely used though, thanks to Babel

JavaScript classes

Useful for React development:

```
class Contact {

  constructor() {
    this.firstName = '';
    this.lastName = '';
  }

  toString() {
    return `${this.firstName} ${this.lastName}`;
  }
}

const c = new Contact();
```

JavaScript classes: inheritance

```
class Parent {
  toString() {
    return `${this.firstName} ${this.lastName}`;
  }
}

class Contact extends Parent {
  constructor() {
    super();
    this.firstName = 'John';
    this.lastName = 'Doe';
  }
}

const contact = new Contact();

console.log(contact.toString());
```

Array spreading

The spread operator (three dots `...`) has been introduced by ES2015.

Usage in array literals:

```
const christmasEve = [2017, 11, 24];
const christmasEveDinner = [...christmasEve, 19, 30, 0];

console.log(christmasEveDinner); // shows [2017, 11, 24, 19, 30, 0]
```

It can be used to spread arguments in function calls too:

```
const christmasEve = [2017, 11, 24];
const date = new Date(...christmasEve);

// is the same as

const date = new Date(2017, 11, 24);
```

Object spreading

- Planned in ES2018.
- Copy the key/value pairs from an object to another.
- Replacement of `Object::assign`

```
const coordinates = {
  address: '59 New Bridge Road',
  zipCode: '059405',
  country: 'Singapour'
}

const employee = {
  firstName: 'John',
  lastName: 'Doe',
  ...coordinates
}
```

Both array and object spreading are widely used in React development to preserve immutability

Short object notation

Short object property assignments from a variable

```
const date = { year, month, day }

console.log(date.year);  // 2017
console.log(date.month); // 11
console.log(date.day);   // 24
```

In combination with the spread operator, it allows to make immutable object transformations in a very consise way

```
const obj1 = { a: 1, b: 2, c: 3 }
const b = 5
const a = 4

const obj2 = { ...obj1, b, a } // The order is important !

console.log(obj1) // shows { a: 1, b: 2, c: 3 }
console.log(obj2) // shows { a: 4, b: 5, c: 3 }
```

Destructuring

- Short variable assignments from an object or an array.
- Spread operator used to assign the **rest** of the element (last position only)

```
const [year, month, day, ...time] = christmasEveDinner;

console.log(year);  // 2017
console.log(month); // 11
console.log(day);   // 24
console.log(time);  // [19, 30, 0]
```

```
const {employee, office, ...otherProps} = this.props;
```

Tooling

Agenda

- Introduction
- *Tooling*
- React
- Rendering
- State
- Lifecycle
- Hooks
- Tests
- Redux
- REST Architecture
- Routing
- Forms
- Performances
- Isomorphism
- Going further
- Conclusion

JavaScript ecosystem

The emergence of JavaScript leaded to the creation of a lot of tools

- Quality
- Tests
- Project generators
- Tasks runner

JavaScript transpiler

For a long time, the Web had evolved but the language remained unchanged:

- 6 years between ES5 and ES2015.
- Legacy browsers that don't support ES2015 still have to be taken care of.

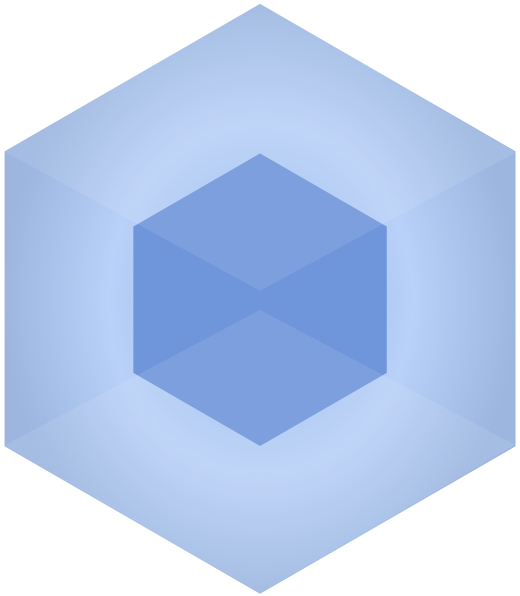
→ Using polyfills or transpilers from different languages (CoffeeScript, TypeScript, ...) to JavaScript to fill in the needs.

JavaScript transpiler

- Parse the JavaScript code and transform ES2015+ instructions into ES5 ones
- Modular configuration through presets:
  - ES versions: `es2015`, `es2016`, `es2017`
  - TC39 proposals: `stage-3`, `stage-2`, `stage-1`, `stage-0`
  - JSX: `react`
- His creator, Sebastian McKenzie, has been hired by Facebook.

Bundlers

- Web side: JavaScript code organization is difficult (global variables...).
- Bundling tools transform multiple CommonJS/ES2015 modules into a unique source file.



Webpack

Browserify

Webpack

- Crawl all the `import` from an entry point.
- Aggregate all exported objects into a bundle.
- Extensible with plugins / loaders:
  - Babel loader
  - CSS bundle
  - Icons bundle
  - ...
- Developer tools: HTTP server, live-reload, Hot Module Replacement.

Webpack dev server

`webpack-dev-server` is a tool built upon Webpack that:

- Watches any sources modifications.
- Starts a lightweight web server that loads all files into the browser.
- Creates a new bundle after each code modification.
- Refreshes the page when a new bundle is created.

create-react-app

- CLI to bootstrap a `React` application
  - Babel configuration
  - Webpack configuration
  - Test configuration
  - Linting
  - Production build
  - ...
- Official `React` bootstrapper
- Uses `yarn` by default if available

create-react-app: commands

- Installation : `npm install -g create-react-app` or `yarn global add create-react-app`
- Create an application : `create-react-app my-app` (you can also use `npx create-react-app my-app`)
- Start : `npm start` or `yarn start`
- Test : `npm test` or `yarn test`
- Build : `npm run build` or `yarn build`

create-react-app: ejection

- The goal of `create-react-app` is to start the development of a React project very quickly and easily.
- At any stage of the development, the abstraction brought by the tool can be removed:
  - `npm run eject` or `yarn eject`
  - ⚠ Irreversible operation
  - Produce battle-documented configuration files for Babel, Webpack, ESLint, Jest

Labs prerequisites

React

Agenda

- Introduction
- Tooling
- *React*
- Rendering
- State
- Lifecycle
- Hooks
- Tests
- Redux
- REST Architecture
- Routing
- Forms
- Performances
- Isomorphism
- Going further
- Conclusion

History

- Library created by Jordan Walke (Facebook) en 2011
- JavaScript implementation of XHP, a PHP extension created in 2009
- Open-sourced in 2013 by Pete Hunt (Instagram)
  "Rethinking best practices"
  https://www.youtube.com/watch?v=x7cQ3mrcKaY

React

- Current release: **16**
- Website: https://reactjs.org
- Documentation: https://reactjs.org/docs
- Sources: https://github.com/facebook/react
- Actively maintained by Facebook (and the community)

Description

> *Lots of people use React as the V in MVC.*

React is a **component-oriented library**.

Cannot build a full application by only using React:

→ Flux (alternative to MVC) (Chapitre 9)

→ Routing (Chapitre 10)

API

React has a simple, concise and consistent API.

→ Low learning curve

- Component API
  - Handle of rendering (`render` method)
  - Handle of lifecycle (`componentDidMount`, `componentDidUpdate`, etc.)
  - Component state (`this.state`, `this.props`)

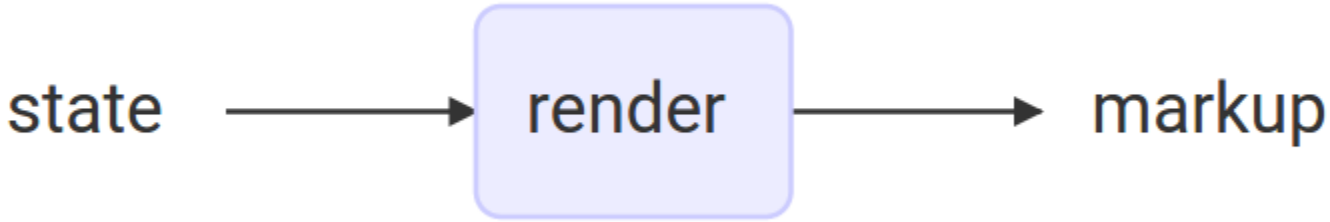
https://reactjs.org/docs/react-component.html

API

- Global API: React
  - Component declaration: ES2015 class inheriting from `React.Component`
  - Virtual DOM declaration: `React.createElement`
- Specific API relative to the DOM: ReactDOM & ReactDOMServer
  - Render a component to a DOM node (`ReactDOM.render`)
  - Render a component to a String (`ReactDOMServer.renderToString`)
  - Get DOM element corresponding to a component instance (`ReactDOM.findDOMNode`)

https://reactjs.org/docs/react-api.html

React ive programming



Output state depends only of input state

- Input state:
  - Properties (`this.props`)
  - Internal state (`this.state`)
- Output state:
  - HTML Markup (output of `render`)

React ive programming

Re-rendering a component is only triggered by changes on `props` or `state`.

→ Make it easy to debug components.

Virtual DOM

- React doesn't handle the DOM directly
- Virtual DOM saved in memory
- Compute differences while re-rendering (`diff`)
- Optimized updating of DOM (`reconciliation`)

Virtual DOM: Example

- Updating a list of elements
- Example:

Before	After
• AngularJS • Backbone • Ember	• React • AngularJS • Ember

- Comparing two processes:
  - Removing old node and adding a new one.
  - Updating element without modifying the DOM tree (React).

Virtual DOM: Example

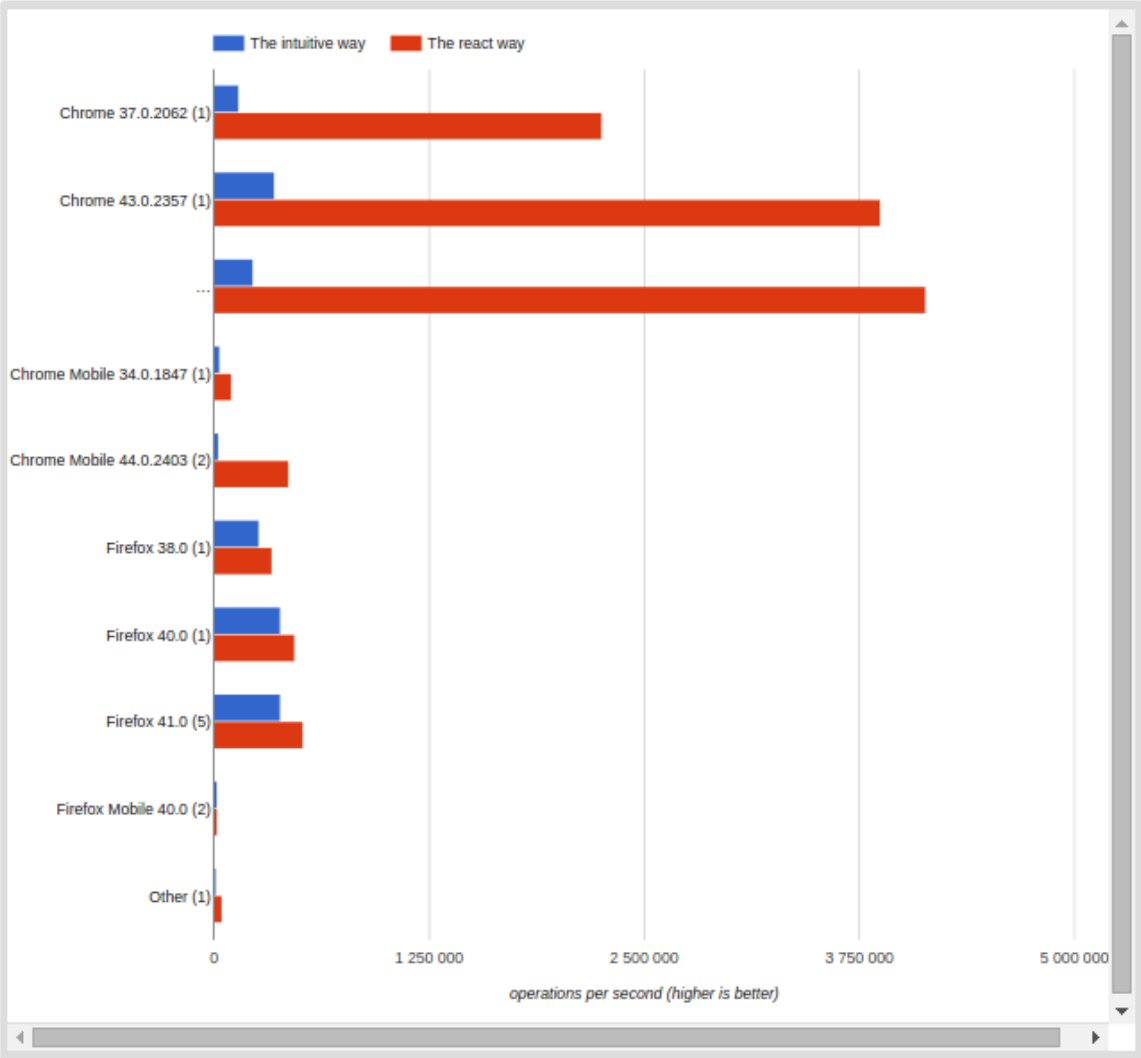
- JSPerf: http://jsperf.com/virtual-dom-optim-demo/2
- Code:

Testing in Chrome 44.0.2403.157 on Linux x86_64		
Test		**Ops/sec**

The intuitive way	`// Remove the "Knockout" child`
	`listElement.removeChild(listElement.children[1])`
	
	`// Create the "React" child`
	`var newChild = document.createElement('li');`
	`newChild.textContent = 'React';`
	
	`// Insert it in first position`
	`listElement.insertBefore(newChild, listElement.firstChild);`
		ready

The react way	`// Reuse the nodes by changing the content`
	`listElement.children[0].textContent = stateToRender[0];`
	`listElement.children[1].textContent = stateToRender[1];`
		ready

Virtual DOM: Example



Rendering

Agenda

- Introduction
- Tooling
- React
- *Rendering*
- State
- Lifecycle
- Hooks
- Tests
- Redux
- REST Architecture
- Routing
- Forms
- Performances
- Isomorphism
- Going further
- Conclusion

First component

Declaring a React component with a class:

```
import React from 'react';

class HelloWorld extends React.Component {

  render() {
    return React.createElement('div', null, 'Hello World !');
  }

}
```

Declaring a React component with a function:

```
import React from 'react';

function HelloWorld() {
  return React.createElement('div', null, 'Hello World !');
}
```

First component

Component rendering:

```
React.createElement(HelloWorld);
```

Generate the following HTML code:

```
<div>Hello World !</div>
```

Global API

`React.createElement` creates an instance of a component or HTML tag

```
ReactElement React.createElement(
    type,
    [object props],
    [children ...]
)
```

`type` could be either a :

- HTML tag (string)
- Component definition (class or function).

Properties

A component may have properties:

```
class Hello extends React.Component {

  render() {
    return React.createElement('div', null, `Hello ${this.props.name} !`);
  }

}

Hello.defaultProps = {
  name: 'World'
};
```

Properties

Component rendering:

```
React.createElement(Hello);

// or

React.createElement(Hello, {name: 'Paul'});
```

Generate the following HTML:

```
<div>Hello World !</div>

<!-- or -->

<div>Hello Paul !</div>
```

DOM rendering

`ReactDOM.render` renders a component inside an existing node

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

Example:

```
const reactElement = React.createElement(HelloWorld);
const domElement = document.getElementById('placeholder');

ReactDOM.render(reactElement, domElement);
```

DOM rendering

Be aware, React needs **total control** over the DOM node.

Avoid `document.body` because it can be altered by other libraries:

- Modals
- Google Font

Rendering multiple elements

Creation of a whole page can be painful:

```
class HelloTeam extends React.Component {

  render() {
    return React.createElement('div', null,
      React.createElement(Hello, {name: 'Paul'}),
      React.createElement(Hello, {name: 'Ben'}),
      React.createElement(Hello, {name: 'Pete'}),
      React.createElement(Hello, {name: 'Sebastian'})
    );
  }

}
```

JSX

- Define `ReactElement` with a declarative syntax looking like HTML
- Tags may be either HTML tags or component definitions
- Attributes are properties given to the component (`this.props`)
- Must be transpiled to plain JavaScript

```
class HelloTeam extends React.Component {
  render() {
    return (
      <div>
        <Hello name="Paul" />
        <Hello name="Ben" />
        <Hello name="Pete" />
        <Hello name="Sebastian" />
      </div>
    );
  }
}
```

JSX placeholders

Use "curly braces" to bind to component variables / methods:

```
class Hello extends React.Component {

  render() {
    return <div>Hello {this.props.name}</div>;
  }

}

Hello.defaultProps = {
  name: 'World'
};
```

JSX

JSX expression can be used as a variable

→ It is a `ReactElement`!

```
class HelloTeam extends React.Component {

  render() {
    const {name} = this.props

    const defaultNode = <Hello name="World" />
    const nameNode = <Hello name={this.props.name} />

    return (
      <div>
        {name ? nameNode : defaultNode}
      </div>
    );
  }

}
```

JSX: Render a list of elements

A list of items can be created by mapping over a collection

Each item element in an array should have a unique key prop.

```
() => (
  <div>
    {list.map(item =>
      <p key={item.id}>{item.text}</p>
    )}
  </div>
)
```

With React < v16, you always have to return a single root element.

JSX: Use of Fragment

- **With React > v16**, you return an array of element or a `Fragment`.
- An array of element must always have unique `key` props

```
() => list.map(i => <p key={i.id}>{i.text}</p>)
```

- `React.Fragment` are just "empty" JSX wrapper

```
() => (
  <Fragment>
    <p>Hello</p>
    <p>World</p>
  </Fragment>
)
// or
() => (
  <>
    <p>Hello</p>
    <p>World</p>
  </>
)
```

JSX

- Norm: https://facebook.github.io/jsx/
- REPL: https://babeljs.io/repl/ ("react" preset must be enabled)
- Good support (Babel, ESLint, IDEs, etc.)

To be clear, all these statements are true:

- It is possible to use React without JSX
- It is possible to use React without a build pipeline (bundler + transpiler)

But it is totally counter-productive

JSX: Tips and tricks

- Variables referencing a component must start by an uppercase.
- Ternary operator is the best way to do conditionals:

```
<span>{this.props.gender === 'H' ? 'Mr': 'Mme'}</span>
```

- Boolean props do not need values

```
<Switch active />
```

- HTML attributes `class` and `for` are JavaScript keywords `className` and `htmlFor` in JSX:

```
<label className="my-class" htmlFor="input-name" />
```

- `style` attribute is an object:

```
<div style={{height: '100%', 'marginTop': '20px'}} />
```

Lab 1

State

Agenda

- Introduction
- Tooling
- React
- Rendering
- *State*
- Lifecycle
- Hooks
- Tests
- Redux
- REST Architecture
- Routing
- Forms
- Performances
- Isomorphism
- Going further
- Conclusion

Inner state

> *Shared mutable state is the root of all evil (Pete Hunt)*

- Props are immutable inside a component.
- `this.state`: Contains mutable state of component (still isolated !).
- Examples:
  - Active / inactive button state
  - Fold / unfold state of accordion
  - Selected tab in tab manager
  - Percentage of progress bar
- When reusing a component, new props are injected, but basically props are immutable.

state or props ?

- The more the component are stateless, the easier the maintenance will be.
- `this.state` must contain modifications caused by:
  - user actions (events)
  - time (`setTimeout`, `setInterval`)
  - props value **only to know mutable initial state**

The state must contain only raw data.

- No components
- No computed data

state or props ?

- In most cases, state can be defined by props.
- Emergence of a pattern:
  → ***stateful*** component wrapper which gives props to the "real" ***stateless*** component
- Separation of concerns:
  - stateful component: contains state creation / transformation
  - stateless component: displays elements depending on props

Stateless component

To encourage the usage of stateless component, React proposes a simple way to declare one:

```
function Hello(props) {

  return <div>Hello {props.name}</div>;

}

Hello.defaultProps = {
  name: 'World'
};
```

State updating

- Use the `setState` method:
  - First mandatory parameter: next state or a function returning the next state.
  - Second optional parameter: callback called when state has been updated and component beeing re-rendered.
- The new state is **merged** with the old one.
- Start the `rendering` process.

State updating

- Any update of `state` must use `setState`.
  - Forbidden: `this.state.counter = 0;`
- Be aware, `state` updating may not be synchronous!
- Example:

```
class MyComponent extends React.Component {
  constructor(props) { ... }

  updateCounter() {
    this.setState({ counter: this.state.counter + 1 });
    // Or:
    this.setState((state, props) => ({ counter: state.counter + 1 }));
  }

  render() { ... }
}
```

Class attributes

Can save variables as attributes.

- Help to keep references
- Access to `this`

Inner state

```
class ProgressBar extends React.Component {
  state = { progression: 0 }

  componentDidMount() {
    let interval = this.props.duration / 100;
    this.timer = setInterval(this.updateProgression, interval);
  }

  componentWillUnmount = () => clearTimeout(this.timer)

  updateProgression = () => this.setState({
    progression: this.state.progression + 1
  });

  render() {
    const width = `${this.state.progression}%`;
    return <div style={{width}} />;
  }
}
```

Props validation

- Possibility to :
  - describe type of props.
  - define if a prop is required or not.
- Help to ensure the component is well used
  - Bad usage displays a warning in the console (only in dev mode).
- Formerly a part of React, it is now a separated package

```
npm install prop-types
```

- https://reactjs.org/docs/typechecking-with-proptypes.html

Props validation

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

class MyComponent extends React.Component { ... }

MyComponent.propTypes = {
    name: PropTypes.string.isRequired,
    address: PropTypes.shape({
        zipcode: PropTypes.number.isRequired,
        country: PropTypes.string
    })
};
```

Lab 2

Lifecycle

Agenda

- Introduction
- Tooling
- React
- Rendering
- State
- *Lifecycle*
- Hooks
- Tests
- Redux
- REST Architecture
- Routing
- Forms
- Performances
- Isomorphism
- Going further
- Conclusion

API introduction

React component lifecycle is divided into 3 main stages:

- **Mount:** First component rendering
- **Update:** Called after any modifications of props or state
- **Unmount:** While removing the component

Each step has several lifecycle methods that you can override.

Mounting

Mounting a component consists in a 4-steps workflow:

- `constructor()`
- `static getDerivedStateFromProps()`
- `render()`
- `componentDidMount()`

Mouting

`contructor(props)`

- initialize the state or bind methods
- must always call `super(props)`
- only in class component

```
constructor(props) {
  super(props);
  this.state = { counter: 0 };
  this.handleClick = this.handleClick.bind(this);
}
```

Mouting

`static getDerivedStateFromProps()`

- Return an object to update the state, or null to update nothing
- Rarely used, for example usefull with animation transitions
- It's a static method

Mouting

`componentDidMount()`

- Called immediately after a component is mounted.
- Used to apply side effects:
  - Fetch data
  - Initialize other JS frameworks
  - DOM manipulation
- **Not called** when doing server side rendering.

```
componentDidMount() {
  fetch(`/api/user/${this.props.id}`)
    .then(response => response.json())
    .then(user => this.setState({ user }))
}
```

Updating

Updating component is done after any `state` or `props` modification:

- `static getDerivedStateFromProps()`
- `shouldComponentUpdate()`
- `render()`
- `getSnapshotBeforeUpdate()`
- `componentDidUpdate()`

Updating

`static getDerivedStateFromProps(nextProps, prevState)`

- Return an object to update the state, or null to update nothing
- Rarely used, for example usefull with animation transitions
- It's a static method

```
static getDerivedStateFromProps(nextProps, prevState) {
  if (nextProps.id !== prevState.id) {
    return {
      id: nextProps.id;
    };
  }
  return null;
}
```

Updating

`shouldComponentUpdate(prevProps, prevState)`

- It helps to avoid unnecessary `render`.
- It improves **performances** a lot.
- Return `true` by default (the component is always rendered).
- Return `false` when we know that the component doesn't need to be rendered.
- Can access to previous `state` and `props` in arguments.
- Can access to next `state` and `props` with `this`.

Updating

`getSnapshotBeforeUpdate(prevProps, prevState)`

- Enables your component to capture some information from the DOM, before it is potentially changed.
- Example: catch the scroll position before it changes.
- The returned value will be avaiblable at third argument of `componentDidUpdate`.
- To use only in specific use-cases.

```
getSnapshotBeforeUpdate(prevProps, prevState) {
  if (prevProps.list.length < this.props.list.length) {
    const list = this.listRef.current;
    return list.scrollHeight - list.scrollTop;
  }
  return null;
}
```

Updating

`componentDidUpdate(prevProps, prevState, snapshot)`

- like `componentDidMount`.
- Can access to previous `state` and `props` in arguments.
- Can access to next `state` and `props` with `this`.

Unmount

- Before removing a component ***componentWillUnmount*** is called.
- Most of the time, it is used to remove DOM modification made with:
  - `componentDidMount`
  - `componentDidUpdate`
- Examples:
  - Clearing `setInterval` timer.
  - Removing DOM events listener.

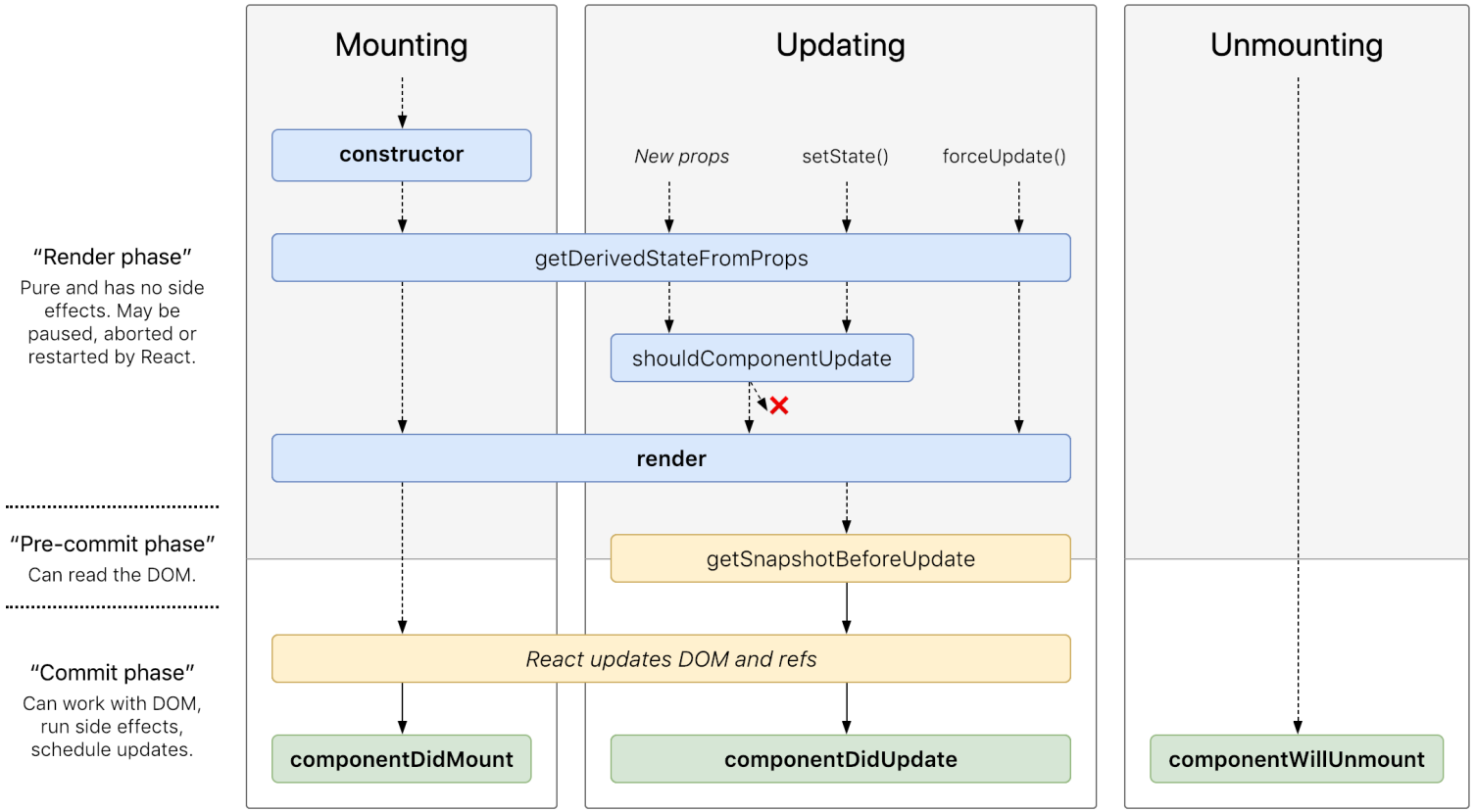
Error handling

`componentDidCatch(error, info)`

- Lifecycle method handling errors.
- Catch javascript error in child component tree.
- The component doesn't catch an error within itself.
- Act like a javascript `catch {}`.

```
componentDidCatch(error, info) {
  // Display fallback UI
  this.setState({ hasError: true });
  // You can also log the error to an error reporting service
  logErrorToMyService(error, info);
}
```

Summary



Source: http://projects.wojtekmaj.pl/react-lifecycle-methods-diagram/

Hooks

Agenda

- Introduction
- Tooling
- React
- Rendering
- State
- Lifecycle
- *Hooks*
- Tests
- Redux
- REST Architecture
- Routing
- Forms
- Performances
- Isomorphism
- Going further
- Conclusion

API introduction

New addition in **React 16.8**

Let you use React features in Function Components.

No breaking changes:

- **Completely opt-in**
- **100% backwards-compatible**

You can adopt hooks gradually.

They don't replace your knowledge of React, only provide a direct API.

You can make you own hooks and combine them.

Why using Hooks

Allow you to reuse stateful logic between components:

- Helps avoid the onion effect of wrappers
- Share stateful logic without changing the components hierarchy

Why using Hooks

Complex components are harder to understand:

- Unrelated actions done during the same cycle step
- Split components into smaller functions based on functionality with Hooks

Why using Hooks

Classes are more confusing than functions:

- The case of **this** in JavaScript
- Code is more verbose
- Disagreements about Classes/Functions usage in React, even between experienced developers
- Even compilers struggle with classes more than with function

Gradual adoption

You can keep your existing code as is.

No rush to migrate to Hooks.

Avoid big rewrites, do it gradually starting with non critical components.

Class components will stay supported by React for the foreseeable future.

First peek

useState is a Hook:

- Accept an initial value as argument
- Returns a pair: **current** state value & function to update it
- Similar to **this.useState**, but doesn't merge old and new

```
import React, {useState} from 'react';

function MyComponent() {
  // declare state variable count
  const [count, setCount] = useState(0);

  return (
    <button onClick={() => setCount(count + 1)}>
      You clicked me {count} times !
    </button>
  )
}
```

First peek

useEffect is another Hook:

- Access to props and state
- By default runs useEffect after each render (including the first)

```
import React, {useEffect, useState} from 'react';

function MyComponent() {
  const [count, setCount] = useState(0);

  // Similar to componentDidMount and componentDidUpdate:
  useEffect(() => {
    document.title = `You clicked ${count} times`;
  });

  return (
    <button onClick={() => setCount(count + 1)}>
      You clicked me {count} times !
    </button>
  )
}
```

Rules of Hooks

2 important rules:

- Call hooks at **top level** only. Don't call them in loops, conditions or nested functions
- Only call Hooks from **React function components** (The only other valid place is another custom Hook)

You can enforce those rules through the plugin **eslint-plugin-react-hooks**.

```
// Your ESLint configuration
{
  "plugins": [
    // ...
    "react-hooks"
  ],
  "rules": {
    // ...
    "react-hooks/rules-of-hooks": "error", // Checks rules of Hooks
    "react-hooks/exhaustive-deps": "warn" // Checks effect dependencies
  }
}
```

State Hook

Declaring a state variable without Hooks

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      count: 0,
    };
  }
}
```

Declaring a state variable with Hooks

```
import React, {useState} from 'react';

function MyComponent() {
  const [count, setCount] = useState(0);
}
```

State Hook

Declares a **state variable**

Use the argument as initial value for the state (doesn't have to be an object)

Returns a pair of values as a tuple:

- current state
- update method

If we want 2 values in our state, we call useState twice

State Hook

Reading State

In a class:

```
<p>You clicked {this.state.count} times!</p>
```

You need to access **this.state.count**.

In a function:

```
<p>You clicked {count} times!</p>
```

You can access **count** directly.

State Hook

Updating State

In a class:

```
<button onClick={() => this.setState({ count: this.state.count + 1 })}>
  Click me
</button>
```

We need to use **this.setState**.

In a function:

```
<button onClick={() => setCount(cnt => cnt + 1)}>
  Click me
</button>
```

We already have access to **count** and **setCount**, so **this** is unnecessary.

Effect Hook

Let you perform side effects in components:

- Data fetching
- Setting up subscriptions
- Changing DOM manually
- etc...

2 types of Effects in React:

- Don't require cleanup
- Require cleanup

Effect Hook

Effects without cleanup

In React classes we have to put effects in life-cycle functions:

```
class MyComponent extends React.Component {
  componentDidMount() {
    document.title = 'This is my value: ' + this.props.myValue;
  }

  componentDidUpdate() {
    document.title = 'This is my value: ' + this.props.myValue;
  }
}
```

Notice how we have to duplicate the code.

And now the same example with hooks:

```
function MyComponent(props) {
  useEffect(() => {
    document.title = 'This is my value: ' + props.myValue;
  });
}
```

Effect Hook

useEffect register the **Effect** and execute it after each render.

Component state and props are in the Effect scope. **No API required**.

React guarantees the DOM has been updated before running the effect.

Effect Hook

Effects with cleanup

Using classes:

```
class MyComponent extends React.Component {
  componentDidMount() {
    API.subscribeToChanges(this.props.idToObserve, this.handleChanges);
  }

  componentWillUnmount() {
    API.unsubscribeToChanges(this.props.idToObserve);
  }

  handleChanges() {
    // TODO: Handle changes
  }
}
```

We have to split the logic even though it's related.

Effect Hook

Effects with cleanup

Using functions:

```
function MyComponent(props) {
  useEffect(() => {
    function handleChanges() {
    // TODO: Handle changes
    }
    API.subscribeToChanges(props.idToObserve, handleChanges);

    return () => {
      API.unsubscribeToChanges(props.idToObserve);
    }
  });
}
```

useEffect is designed to keep the logic together.

The return part is an optional cleanup mechanism.

Helps keep the logic in the same effect.

The cleanup is executed when React unmounts the component.

Effect Hook

Tips

- Separate concerns in multiple effects
- useEffect accept an additional argument to optimize performances

```
useEffect(() => {
    document.title = 'This is my value: ' + props.myValue;
}, [props.myValue]) // run again only if props.myValue has changed;
```

Custom Hooks

Sometimes different components can share the same logic:

```
const MyComponent = (props) => {
  const [value, setValue] = useState(0);
  useEffect(() => {
    const handleChanges = (arg) => setValue(arg);
    API.subscribeToChanges(props.idToObserve, handleChanges);
    return () => {
      API.unsubscribeToChanges(props.idToObserve);
    }
  });
}

const MyOtherComponent = (props) => {
  const [value, setValue] = useState(5);
  useEffect(() => {
    const handleChanges = (arg) => setValue(arg);
    API.subscribeToChanges(props.idToObserve, handleChanges);
    return () => {
      API.unsubscribeToChanges(props.idToObserve);
    }
  });
}
```

Custom Hooks

Creating our own custom hook:

```
import React, { useState, useEffect } from 'react';

const useMyValue = (valueId, initialValue) => {
  const [myValue, setMyValue] = useState(initialValue || 0);

  useEffect(() => {
    const handleChanges = (arg) => setValue(arg);
    API.subscribeToChanges(valueId, handleChanges);
    return () => {
      API.unsubscribeToChanges(valueId);
    }
  });

  return myValue;
}
```

The logic stays the same as the one from the components.

There is no specific signature, you can customize the arguments and the return values.

A hook should **always** start with **use**.

Custom Hooks

We can now use our custom hook:

```
function MyComponent(props) {
  const value = useMyValue(props.idToObserve);
}

function MyOtherComponent(props) {
  const value = useMyValue(props.idToObserve, 5);
}
```

The code works exactly as before.

The state of the hook is **not** shared between the 2 components.

To share information between hooks, you need to pass the data in the arguments, as you would with a regular function.

Remember to respect the **use** convention when naming your hooks.

Tests

Agenda

- Introduction
- Tooling
- React
- Rendering
- State
- Lifecycle
- Hooks
- *Tests*
- Redux
- REST Architecture
- Routing
- Forms
- Performances
- Isomorphism
- Going further
- Conclusion

Concepts

- Module `react-dom/test-utils` to ease writing tests.
- Need a test runner:
  - ***Jest*** provided by Facebook.
  - Karma (used with Angular).
- Assertions libraries:
  - Jasmine.
  - Jest provides an API based on Jasmine.

Jasmine

- Tests framework maintained by Pivotal Labs
- http://jasmine.github.io/
- https://github.com/jasmine/jasmine

Jasmine

API:

- `describe`: tests suite.
- `beforeAll` / `afterAll`: code executed once before and after all test suite.
- `beforeEach` / `afterEach`: code executed before and after each test.
- `it`: a test.
- `xdescribe` / `xit`: ignore test suite / test.
- `fdescribe` / `fit`: execute only test suite / test.

Jasmine

Assertions:

- `expect('foo').toBeDefined()`
- `expect(null).toBeNull()`
- `expect(undefined).toBeUndefined()`
- `expect('foo').toBe('foo')`
- `expect([1, 2, 3]).toEqual([1, 2, 3])`
- `expect([1, 2, 3]).toContain(1)`
- `expect('foo').toMatch(/foo/)`
- etc.

Jasmine

Example:

```
describe('Test Suite', function() {
  beforeEach(function() {
    // Setup
  });

  afterEach(function() {
    // Tear down
  });

  it('should pass', function() {
    expect(true).toBe(true);
    expect(true).not.toBe(false);
  });
});
```

Jest

- Developed by Facebook.
- Based on Jasmine.
- Official React test runner.
- https://facebook.github.io/jest/
- https://github.com/facebook/jest

Jest

- Execute tests located under any `__tests__` folder.
- Tests are executed using an in-memory DOM (using `jsdom`).
- Built-in mocking mechanism.

- Fast.
- Don't need browser to be run (thanks to`jsdom`).
- Interactive mode `--watch`:
  - Re-run tests when modifications
  - Ease the selection of tests to run

Jest

- Based on Jasmine:
  - `beforeEach`, `afterEach`
  - `describe`, `it`
  - `xdescribe`, `xit`
  - `fdescribe`, `fit`
  - `expect(value).toBe(value)`
- Few differences:
  - `expect(mockFunc).toBeCalled();`
  - `expect(mockFunc).toBeCalledWith(arg1, arg2);`
  - `expect(mockFunc).lastCalledWith(arg1, arg2);`

Individual mock

- Mock a module by calling `jest.mock`.
- Functions exported by the module become mocked functions
  - Return `undefined` by default
  - Customizable behavior

```
// sum.js
export default function (num1, num2) {
  return num1 + num2;
}

// test file:
import sum from './sum';

jest.mock('./sum');

sum(2, 3); // return undefined

sum.mockReturnValue(10);

sum(2, 3); // return 10
```

Global mock

- Jest scans `__mocks__` folders.
- If there is a mock matching the module name, it will be used.
- Possibility to get the initial implementation by using `require.requireActual`.
- The mock is accessible for siblings and their subdirectories.
- Useful for mocking 3rd-party libraries.

Automocking

- Jest can be configured to mock ***all*** modules.
  - Even 3rd-party library like React
  - Even the module to test !

```
// __tests__/my-module.test.js
import myModule from '../my-module';

jest.unmock('../my-module');

...
```

- Activated by default prior to version 15.
- Force to make real ***unit*** tests.
- Require additional configuration.
- May slow down the test execution.

Jest

- Jest configuration is in the `package.json` file or a separated file:
  - `automock`: Activate the automocking feature or not
  - `unmockedModulePathPatterns`: useful to avoid mocking some libraries (React).
  - `globals`: global variables accessible while running tests.
  - `testRegex`: Pattern applied on file path to detect if it is a test suite.
  - `setupFiles`: useful to add plugins for instance.
- Complete reference: https://facebook.github.io/jest/docs/en/configuration.html

Jest

Configuration example:

```
{
  "name": "my-app",
  "version": "0.1.0",
  "scripts": {
    "test": "jest"
  },

  "dependencies": { ... },
  "devDependencies": { ... },

  "jest": {
    "setupFiles": ["./prepare-jest-env"],
    "automock": true,
    "unmockedModulePathPatterns": [
      "react"
    ],
    "globals": {
      "DEBUG": true
    }
  }
}
```

Jest

Example:

```
// foo.js
export default function(name) {
  return 'Hello ' + name;
};
```

```
// __tests__/foo-test.js
import foo from '../foo';

describe('foo', function() {
  it('should say hello', function() {
    expect(foo('World')).toBe('Hello World');
  });
});
```

Karma

- Test runner mainly used with Angular.
- Developed by Google.
- http://karma-runner.github.io
- https://github.com/karma-runner/karma

Karma

- Need a configuration file to define:
  - The test framework (Jasmine, Mocha).
  - Browsers to use.
  - Scripts to load.
- Extensible with plugins:
  - ***karma-babel-preprocessor***
  - ***karma-webpack***
  - karma-coverage
  - etc.

Karma

Example:

```
// karma.conf.js
module.exports = function(config) {
  config.set({
    singleRun: true,
    autoWatch: true,

    files: [
      '**/*-test.js'
    ],

    browsers: [
      'PhantomJS',
      'Chrome'
    ],

    preprocessors: {
      'src/**/*.js': ['babel'],
      'test/**/*.js': ['babel']
    }
  });
};
```

React Test Utilities

- Module developed by React teams.
- API to test components:
  - `renderIntoDocument`
  - `isElement` / `isElementOfType`
  - `findRenderedDOMComponentWithTag` (1 element)
  - `scryRenderedDOMComponentsWithTag` (multiple elements)
  - `findRenderedComponentWithType` (1 element)
  - `scryRenderedComponentsWithType` (multiple elements)
  - `Simulate.{eventName}`
  - etc.
- https://reactjs.org/docs/test-utils.html

React Test Utilities

Example:

```
// __tests__/MyComponent-test.js

import React from 'react';
import ReactDOM from 'react-dom';
import TestUtils from 'react-dom/test-utils';

import MyComponent from '../MyComponent';

describe('A suite', function() {
  it('should render component', function() {

    const component = <MyComponent />;

    expect(TestUtils.isElement(component)).toBe(true);
    expect(TestUtils.isElementOfType(component, MyComponent)).toBe(true);

    const rendered = TestUtils.renderIntoDocument(myComponent);
    const child = TestUtils.findRenderedDOMComponentWithClass(rendered, 'foo');

    expect(child.textContent).toEqual('Hello World');
  });
});
```

React Test Utilities

- Use `TestUtils.Simulate.{eventName}` to simulate user interactions.
- Example: click

```
// __tests__/MyComponent-test.js

import React from 'react';
import ReactDOM from 'react-dom';
import TestUtils from 'react-dom/test-utils';

import MyComponent from '../MyComponent';

describe('My Component', function() {
  it('handle click', function() {
    const component = <MyComponent />;
    const instance = TestUtils.renderIntoDocument(component);
    const node = ReactDOM.findDOMNode(instance);

    TestUtils.Simulate.click(node);
    // ... assertions
  });
});
```

React Test Utilities

There are two ways to test component rendering:

- With `renderIntoDocument` method:
  - Need the DOM.
  - Assertions test "rendering" (maybe verbose and complicated).

- Shallow Rendering with `react-test-renderer` module:
  - Test component without DOM.
  - Isolation testing of `render`.
  - Test the component directly (`props`, `state`, etc.).

```
npm install react-test-renderer --save-dev
```

React Test Utilities

Example:

```
// __tests__/MyComponent-test.js

import React from 'react';
import ReactDOM from 'react-dom';
import TestUtils from 'react-dom/test-utils';
import ShallowRenderer from 'react-test-renderer/shallow';

import MyComponent from '../MyComponent';

describe('A suite', function() {
  it('should render component', function() {
    let shallowRenderer = new ShallowRenderer();
    let element = <MyComponent className="foo" />;

    shallowRenderer.render(element);

    let component = shallowRenderer.getRenderOutput();

    expect(component.props.className).toBe('foo');
    expect(component.props.children).toEqual(<p>Hello World</p>);
  });
});
```

react-testing-library

- Render a React component
- Avoid Memory leaks
- Debug the DOM state
- Test React Component Event Handlers
- Assert rendered text

Render a React component

Example:

```
import 'jest-dom/extend-expect'
import React from 'react'
import {render} from 'react-testing-library'
import MyComponent from './MyComponent'

test('renders MyComponent', () => {
  const {getByLabelText} = render(< MyComponent />)
  const input = getByLabelText(/favorite number/i)
  expect(input).toHaveAttribute('type', 'number')
})
```

Avoid Memory leaks #1

Example:

```
import React from "react";
import { cleanup } from "@testing-library/react";
import "@testing-library/jest-dom/extend-expect";
import MyComponent from "./ MyComponent ";

describe(" MyComponent", () => {
  afterEach(cleanup);
  test("should increment counter", () => { });
});
```

Avoid Memory leaks #2

Example:

```
import React from "react";
import { fireEvent, render } from "@testing-library/react";
import "@testing-library/jest-dom/extend-expect";
import "@testing-library/react/cleanup-after-each";
import MyComponent from "./ MyComponent ";

describe(" MyComponent", () => {
  test("should increment counter", () => { });
});
```

Debug the DOM state

Example:

```
test("should increment counter", () => { });
  const {getByLabelText, debug} = render(<MyComponent />);
  const input = getByLabelText(/favorite number/i);
  expect(input).toHaveAttribute('type', 'number');
  debug(input);
})
```

Test React Component Event Handlers

Example:

```
import { cleanup, fireEvent, render } from "@testing-library/react";

describe("LikeBtn", () => {
  test("should increment counter", () => {
    const { getByTitle } = render(<LikeBtn type={"up"} counter={0} />);
    const likeButtonElement = getByTitle("+1");
    fireEvent.click(likeButtonElement);
  });
});
```

Assert rendered text

Example:

```
test("should increment counter", () => {
  onst {getByLabelText, getByTestId} = render(<MyComponent />);
  const input = getByLabelText(/favorite number/i);
  expect(getByTestId('error-message')).toHaveTextContent(/the number is
invalid/i);
})
```

Lab 3

Redux

Plan

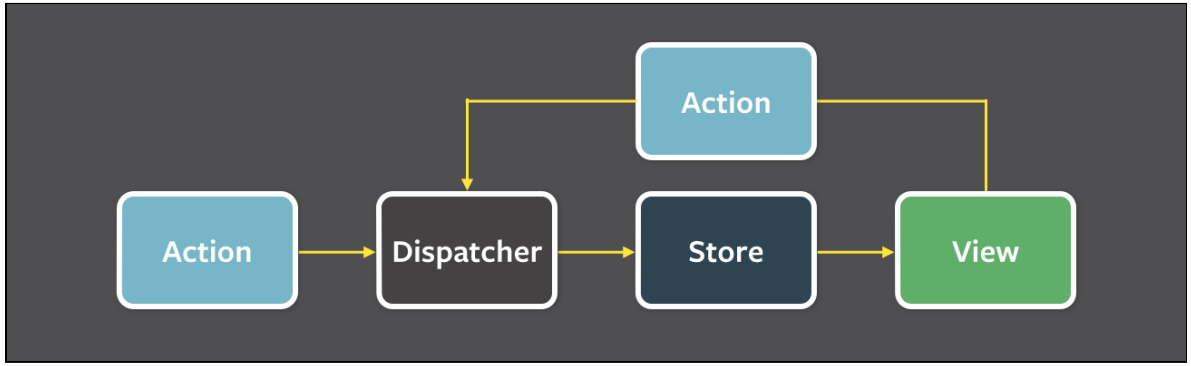
- Introduction
- Tooling
- React
- Rendering
- State
- Lifecycle
- Hooks
- Tests
- *Redux*
- REST Architecture
- Routing
- Forms
- Performances
- Isomorphism
- Going further
- Conclusion

Flux - Introduction

- Architecture to build applications.
  - Not a framework !
  - Not a library !
- Used by Facebook but not related to React.
- https://facebook.github.io/flux

Flux - Concepts

- Application structured around three components:
  - `dispatcher`
  - `store`
  - `view`
- **Unidirectional** Flow.



https://facebook.github.io/flux/docs/overview.html

Flux - Actions

- Identify an event in your application.
  - Triggered by user actions.
  - Triggered programmatically.
- Examples:
  - Click to increment a counter.
  - Form submission.
  - Websocket notification.
  - Data loading.
  - Opening / closing modal.

Flux - Dispatcher

- One role: propagate actions (events bus)
  - Does not contain business logic !
  - **Stateless**.
  - Singleton.
- Examples:
  - When a user click, the view call the `dispatcher` to propagate the action.
  - When receiving a Websocket notification, the `dispatcher` propagate the action.
- Facebook provides its own implementation: https://github.com/facebook/flux

Flux - Store (1/2)

- Contains application state.
  - Implements the business logic.
  - Contains all the ***business logic***.
  - May contain UI data(modal state, tab selected, etc.).
- Registers to the `dispatcher`: The store updates when an action is propagated by the dispatcher.
- Propagates an event to the view when the store has been updated.

Flux - Store (2/2)

- Be aware to not ***give access to data***:
  - No public "setters".
  - Be aware with mutable structures (objects, arrays)
  - Use immutable structures: https://facebook.github.io/immutable-js/
- The stores are the ***single source of truth***.

Redux - Principles

- Single source of truth (one ***store*** => one ***state***).
- ***State*** read only (immutable)
- Changes made by ***pure functions***

Redux - Actions Creators

Functions that create actions

```
export const ADD_TODO = 'ADD_TODO';

function addTodo(text) {
  return {
    type: ADD_TODO,
    text
  };
}

...

store.dispatch(addTodo("foo"));
```

Redux - Reducers

Pure functions updating the store: ***(previousState, action) => newState***

```
import { ADD_TODO } from './actions';

const initialState = { todos: [], foo: {} };

function todoApp(state = initialState, action) {
  switch (action.type) {
    case ADD_TODO:
      return {
        ...state,
        todos: [
          ...state.todos,
          {
            text: action.text,
            completed: false
          }
        ]
      };
    default:
      return state;
  }
}
```

Redux - Splitting Reducers

Split into smaller business domain using ***combineReducers***

```
function todoApp(state = {}, action) {
  return {
    todos: todos(state.todos, action),
    visibilityFilter: visibilityFilter(state.visibilityFilter, action)
  };
}

function todos(state = [], action) {
  switch (action.type) {
    case ADD_TODO:
      return { ... };
    default:
      return state;
  }
}

function visibilityFilter(state = {}, action) { ... }
```

Redux - Store

Helps to bind actions and reducers

```
import { addTodo, toggleTodo, setVisibilityFilter, VisibilityFilters } from
'./actions';
import { createStore } from 'redux';
import todoApp from './reducers';
const store = createStore(todoApp);

// Log the initial state
console.log(store.getState());

// Every time the state changes, log it
// Note that subscribe() returns a function for unregistering the listener
let unsubscribe = store.subscribe(() =>
  console.log(store.getState())
);

// Dispatch some actions
store.dispatch(addTodo('Learn about actions'));
store.dispatch(addTodo('Learn about reducers'));

// Stop listening to state updates
unsubscribe();
```

Redux + React

	Presentational Components	Container Components
	Presentational Components	Container Components
Purpose	How things look (markup, styles)	How things work (data fetching, state updates)
Aware of Redux	No	Yes
To read data	Read data from props	Subscribe to Redux state
To change data	Invoke callbacks from props	Dispatch Redux actions
Are written	By hand	Usually generated by React Redux

Redux + React

Provide the store to components

```
import React from 'react';
import { render } from 'react-dom';
import { Provider } from 'react-redux';
import { createStore } from 'redux';
import rulesApp from './reducers';
import RulesList from './rules-list';

let store = createStore(rulesApp);

render(
  <Provider store={store}>
    <RulesList />
  </Provider>,
  document.getElementById('main')
);
```

Redux + React

```
import { connect } from 'react-redux';
import { selectTodo } from '../actions';
import Todo from '../components/Todo';

const mapStateToProps = (state, props) => {
  return {
    isSelected: props.id === state.todos.selectedTodoId
  };
};

const mapDispatchToProps = (dispatch, props) => {
  return {
    onClick: () => {
      dispatch(selectTodo(props.id))
    }
  };
};

const TodoContainer = connect(
  mapStateToProps,
  mapDispatchToProps
)(Todo);

export default TodoContainer;
```

Redux - Going further

- Selectors with ***reselect***
- React Developer Tools
- Redux Developer Tools
- ImmutableJS

Lab 4

REST Architecture

Agenda

- Introduction
- Tooling
- React
- Rendering
- State
- Lifecycle
- Hooks
- Tests
- Redux
- *REST Architecture*
- Routing
- Forms
- Performances
- Isomorphism
- Going further
- Conclusion

REST Architecture

- REST: REpresentational State Transfer
  - Architecture for distributed hypermedia systems
  - Invented by Roy Fielding in 2000
- Architecture
  - Client / Server
  - Stateless: each request contains all needed information to be processed
  - Based on manipulation (creation, modification, deletion) of resources identified by their URI
  - A system based on that architecture is said RESTful

REST Architecture

- Actions

Resource	GET	PUT	POST	DELETE
Collection	Get a collection of elements	Replace all elements	Create a new entry in the collection	Remove the collection
Object	Get one element	Update one element	-	Remove the element

- Return code

Status	Type
200 - 299	Success
300 - 399	Redirection
400 - 499	Client errors
500 - 599	Server errors

REST Architecture

- Benefits
  - Decoupling client/server
  - Server scalability
  - Cache resources
- Drawbacks
  - More network traffic
  - Security is more difficult (client side)

React & REST Architecture

- No API to do HTTP calls.
  - Can use `XMLHttpRequest` API.
  - Can use `fetch` API (still in draft version).
  - Can use a library based on `XMLHttpRequest`.
- ***Be aware*** of libraries working only in the browser:
  - Use them only in `componentDidMount` (isomorphism).
  - If available, prefer a Node.js compatible library.

React & REST Architecture

- HTTP call example with `XMLHttpRequest` & `Promise` ES2015:

```
export default function(options) {
  return new Promise(function (resolve, reject) {
    var xhr = new XMLHttpRequest();
    xhr.open(options.method, options.url);

    xhr.readystatechange = function() {
      if (this.statechange === 4 && this.status >= 200) {
        if (this.status < 400) {
          resolve(this.responseText);
        } else {
          reject(this.responseText);
        }
      }
    };

    xhr.send(options.data || null);
  };
})
```

React & REST Architecture

- using with React component:

```
import React from 'react';
import ajax from './ajax';

export default class TodosView extends React.Component {
  constructor() { ... }

  componentDidMount() {
    this.rq = ajax({ method: 'GET', url: '/todos' }).then(data => {
      this.setState({
        todos: JSON.parse(data);
      });
    });
  }

  componentWillUnmount() {
    if (this.rq) {
      this.rq.abort();
    }
  }

  render() { ... }
}
```

React & REST Architecture

- Be aware:
  - In case of isomorphic application (chap. 13),using browser API (`XMLHttpRequest`, `fetch`) must be done in `componentDidMount`.
  - An HTTP call is ***asynchronous***:
    - Update component `state` when getting data => re-render the component.
    - Tip: cancel asynchronous call in `componentWillUnmount`.

React & REST Architecture

- Known libraries:
  - `superagent` (http://visionmedia.github.io/superagent/).
  - `node-fetch` (https://github.com/bitinn/node-fetch)
  - `isomorphic-fetch` (https://github.com/matthew-andrews/isomorphic-fetch): `fetch` API implementation for server and browser.
- `http` module available with Node.js is usable client side:
  - Webpack modify it by a library working in the browser
  - Works thanks to `http-browserify`: https://github.com/substack/http-browserify

React & REST Architecture

- Examples with `superagent`:

```
import React from 'react';
import superagent from 'superagent';

export default class TodosView extends React.Component {
  constructor() { ... }

  componentDidMount() {
    this.rq = superagent.get('/todos')
      .end((err, res) => {
        if (res.ok) {
          this.setState({
            todos: res.body
          });
        }
      });
  }

  componentWillUnmount() {
    if (this.rq) { this.rq.abort(); }
  }

  render() { ... }
}
```

Axios

- Performing a GET request
- Performing a POST request
- Backend API

Performing a GET request

- Examples with `axios`:

```
const getUser = () => {
  try {
    const response = await axios.get('/user/1');
    console.log(response);
  } catch (error) {
    console.error(error);
  }
}
```

Performing a POST request

- Examples with `axios`:

```
const getUser = () => {
  try {
    const response = await axios.post('/user', {
      firstName: 'Fred',
      lastName: 'Flintstone'
    });
    console.log(response);
  } catch (error) {
    console.error(error); }
  }
```

Backend API

- Features to Backend API

User Feature	Backend API
List all users	axios.get('/users');
Get one user detail	axios.get('/users/1');
Create new user	axios.post('/users', { ... });
Update user	axios.put('/users/1', { ... });
Delete user	axios.delete('/users/1');

Redux - redux-thunk

- Handles asynchronous actions
- Manual dispatch
- Access to `dispatch` by returning a function

```
export const RULES_LOADED = 'RULES_LOADED';

function rulesLoaded(rules) {
  return {
    type: RULES_LOADED,
    rules
  }
}

export function fetchRules() {
  return function(dispatch) {
    fetch('/api/rules')
      .then(function(rules) {
        dispatch(rulesLoaded(rules))
      })
  }
}
```

Redux - redux-thunk configuration

- Example

```
import thunkMiddleware from 'redux-thunk';
import { createStore, applyMiddleware } from 'redux';
import rulesReducer from './reducers';

const store = createStore(
  rulesReducer,
  applyMiddleware(
    thunkMiddleware,
  )
);
export default store;
```

Lab 5

Routing

Agenda

- Introduction
- Tooling
- React
- Rendering
- State
- Lifecycle
- Hooks
- Tests
- Redux
- REST Architecture
- *Routing*
- Forms
- Performances
- Isomorphism
- Going further
- Conclusion

Routing

- No official router provided by Facebook.
- `react-router` project widely adopted by the community.

- https://github.com/ReactTraining/react-router
- https://reacttraining.com/react-router

React Router

- Ease routing configuration.
- Bring all needed components:
  - Routers: `BrowserRouter`, `HashRouter`, `NativeRouter`, ...
  - `Route`
  - `Link`
  - `Switch`
  - etc.

Core concepts

- Each route is bound to a component.
- The component can also declare one or more routes (nested routes).
- Routing is dynamic: the routing plan is not loaded once and for all. Routes are read and evaluated on the fly:
  - Easy to reason about when developing.
  - Allows very flexible patterns (recursive routes, routing based on screen size, etc.).
  - Could be hard to maintain: no place where all the routes are declared.

Core concepts

- Example (1/2):

```
// index.js
import React from 'react';
import ReactDOM from 'react-dom';
import { BrowserRouter, Route } from 'react-router-dom';
import App from './App';

const el = document.getElementById('root');

ReactDOM.render((
  <BrowserRouter>
    <Route path="/" component={App} />
  </BrowserRouter>
), el);
```

Core concepts

- Example (2/2):

```
// App.js
import React from 'react';
import { Route } from 'react-router-dom';
import Home from './Home';
import About from './About';

export default class App extends React.Component {
  render() {
    const { url } = this.props.match;
    return (
      <div>
        <h1>App</h1>
        <Route path={`${url}home`} component={Home} />
        <Route path={`${url}about`} component={About} />
      </div>
    );
  }
}
```

Home page

- `Route` can have an `exact` property.
- It defines the page to display when navigate on a root path (for instance `/`).
- Example:

```
// index.js
ReactDOM.render((
  <BrowserRouter>
    <Route path="/" component={App} />
  </BrowserRouter>
), el);

// App.js
render() {
  return (
    <div>
      <h1>App</h1>
      <Route exact path="/" component={Home} />
      <Route path="/about" component={About} />
    </div>
  );
}
```

Nested routes

- Can define nested routes.
- Can have parameters.
- Example :
  - `/users`
  - `/users/:id`

Nested routes

- Example (1/2):

```
// index.js
ReactDOM.render((
  <BrowserRouter>
    <Route path="/" component={App} />
  </BrowserRouter>
), el);

// App.js
render() {
  return (
    <div>
      <h1>App</h1>
      <Route path="/home" component={Home} />
      <Route path="/users" component={UserList} />
    </div>
  );
}
```

Nested routes

- Example (2/2):

```
// UserList.js
render() {
  return (
    <div>
      <h2>UserList</h2>
      <Route path={`${this.props.match.url}/:id`} component={UserForm}/>
    </div>
  );
}

// UserForm.js
render() {
  return (
    <div>
      <h3>User #{this.props.match.params.id}</h3>
    </div>
  );
}
```

React Router props

- Some properties are provided to components that:
  - Are linked in a `Route` definition.
  - Are wrapped by the React Router HOC using `withRouter`.
- Examples of useful properties:
  - URL parameters: `this.props.match.params`.
  - Query string parameters: `this.props.location.search` (must be parsed by an other library though).
  - History function to get back programmatically: `this.props.history.goBack()`.

Link components

- The `Link` component helps to generate a link to the route:
  - Support either a string or a location object.
  - In case of parametrized URL, build the `pathname` manually: use ES2015 templates string!
- The `NavLink` component adds some styling options:
  - Possibility to add CSS class when the route is activated (`activeClassName` attribute).
  - `exact` and `strict` options provides more flexibility.

NavLink example

```
// menu.js
export default class Menu extends React.Component {
    render() {
      const { id } = this.props.match.params;

      return (
        <ul>
          <li>
            <NavLink exact activeClassName="active" to="/users">Users</NavLink>
          </li>
          <li>
            <NavLink activeClassName="active" to={`/users/${id}`}>User</NavLink>
          </li>
        </ul>
      );
    }
}
```

URL handling

- Two ways to use browser history:
  - Use HTML5 `pushState` API (a.k.a. `history`).
  - Use URL hash (after #).
- To be defined when using the router:
  - `BrowserRouter` for a history-based navigation
  - `HashRouter` for a hash-based navigation

URL handling

- Example with "#":

```
import ReactDOM from 'react-dom';
import {Route} from 'react-router';
import { HashRouter } from 'react-router-dom';
import App from './App';

ReactDOM.render(
  <HashRouter>
    <Route path='/' component={App} />
  </HashRouter>,
  document.getElementById('app')
);
```

URL handling

- `history` API avoids the `#` hack, but:
  - Doesn't work with older browsers (IE >= 10).
  - Routes must be handled server side.

Manual navigation

Example:

- Using ***withRouter*** HOC since 2.4.0

```
import React, { Component } from 'react';
import { withRouter } from 'react-router';

class MyComponent extends Component {
  render() {
    return (
      <div>
        <div onClick={() => this.props.history.push('/foo')}>Go to foo</div>
      </div>
    )
}

export default withRouter(MyComponent);
```

Lab 6

Forms

Agenda

- Introduction
- Tooling
- React
- Rendering
- State
- Lifecycle
- Hooks
- Tests
- Redux
- REST Architecture
- Routing
- *Forms*
- Performances
- Isomorphism
- Going further
- Conclusion

Forms - Starting with

- Principles:
  - Use classic forms fields (`input`, `textarea`, `select`).
  - Binding between `state` and the field: a modification updates the component `state`.
- Unidirectional Flow: no double binding.
- No validation mechanism is included...

Forms - Starting with

- The value of the field is handled with:
  - `value` properties for "free" fields (`input`, `select`, `textarea`).
  - `checked` for inputs like `checkbox` or `radio`.
- Change detection thanks to the `onChange` event
  - Same behaviour for all fields.
  - Equivalent to `input` event: triggered when field value is updated.

Forms - Starting with

- Example:

```
export default class MyForm extends React.Component {
  state = {
    name: '',
    desc: '',
  };

  updateName = e => this.setState({ name: e.target.value });

  updateDesc = e => this.setState({ desc: e.target.value });

  render() {
    const { name, desc } = this.state
    return (
      <form>
        <input type="text" value={name} onChange={this.updateName}/>
        <textarea value={desc} onChange={this.updateDesc} />
      </form>
    );
  }
}
```

Forms - Starting with

- Other events:
  - Keyboard: `onKeyDown`, `onKeyPress`, `onKeyUp`.
  - Focus: `onFocus`, `onBlur`.
  - Form: `onSubmit`, `onInput`, `onChange`.
  - Clip: `onPaste`.
- Mobile: optional support of `touch` event:
  - Must call `React.initializeTouchEvents(true)` before any rendering.
  - Support `onTouchCancel`, `onTouchEnd`, `onTouchMove`, `onTouchStart`

Forms - Validation

- Validation: nothing is provided by React => manual validation.
- Can use the HTML5 form validation API:
  - `required`, `pattern`, `min`, `max`, etc. properties
  - `checkValidity` method to check fields validity depending on validation properties.
  - Be aware: IE >= 10
  - Be aware: Validation API behaviour may change depending on browsers.

Forms - HTML5 validation

- Properties supported:
  - `input`: `required`, `maxLength`, `pattern`
  - `textarea`: `required`, `maxLength`
  - `select`: `required`
- Specific properties:
  - `input[number]`: `min`, `max`, `step`
  - `input[date]` - `input[datetime]` - `input[time]`: `min`, `max`
- Patterns for `mail` or `url` inputs
- Can create custom validators:
  - Use `setCustomValidity` function

Forms - Validation

- Example:

```
export default class MyForm extends React.Component {
  state = { name: '', canSubmit: false }

  handleSubmit = () => { ... }

  refTitle = element => this._title = element

  validate = () => this.setState({ canSubmit: this._title.checkValidity() });

  render() {
    const { canSubmit, name } = this.state;
    return (
      <form onSubmit={this.handleSubmit}>
        <input ref={this.refTitle} value={name} required
               onChange={this.setName}  onBlur={this.validate} />
        <button type="submit" disabled={!canSubmit}></button>
      </form>
    )
  }
}
```

Forms - Validation

- Forms handling is one of React weakness.
- The most widely used form library: redux-form

- ***Be aware***:
  - Client-side validation must never replace server-side validation.
  - Just to build better GUI and help the user (fast feedbacks, help to fill, etc.).

Redux-form - Core concepts

- Principles:
  - Redux reducer
  - Forms are deeply connected to the Redux store
  - Container (HOC pattern) that provides form handling props and actions to the underlying component

Redux-form - Redux store

Use `combineReducers` to include the reducer provided by Redux-form

```
 import { createStore, combineReducers } from 'redux'
import { reducer as formReducer } from 'redux-form'

const reducers = {
  // ... your other reducers here ...
  form: formReducer     // <---- Create a key 'form' in the state
}
const reducer = combineReducers(reducers)
const store = createStore(reducer)
```

Redux-form - Form decoration

- Provide properties and actions from the forms handled by Redux-form.
- Use the specific Redux-form components (i.e. `Field`) to ease usage of this properties and actions.
- All the decorated forms are connected to the store
- In the following example, it would be `state.form.contact`

Redux-form - Store

```
import React, { Component } from 'react';
import { Field, reduxForm } from 'redux-form';

class ContactForm extends Component {
  render() {
    const { handleSubmit } = this.props;
    return (
      <form onSubmit={handleSubmit}>
        <div>
          <label htmlFor="firstName">First Name</label>
          <Field name="firstName" component="input" type="text"/>
        </div>
        <button type="submit">Submit</button>
      </form>
    );
  }
}

// Decorate the form component, and name it with a unique identifier
export default reduxForm({ form: 'contact' })(ContactForm);
```

Redux-form - Validation

- Form-level validation: we provide a function to the form.
- Field-level validation: we provide a function to each field that needs validation.
- Still backed by HTML5 validation.
- Validation could be:
  - synchronous: when the user is typing,
  - asynchronous: after a call to the backend.
- Errors:
  - Computed then provided to the `Field` components.
  - Can be manipulated for custom usage thanks to the `error` property.

Redux-form - In detail

- For more details, see the official website.
  - very clear, provides a lot of examples,
  - the library is moving fast, so check the documentation regularly,
- The library updates could include breaking changes.
- Website: http://redux-form.com

Redux-form - Advanced components

- Redux-form components to ease complex form creation:
  - `<FormSection>`
    Create a reusable block of multiple inputs.
    Example: an adress input (with number + street + postal code + city).
  - `<FieldArray>`
    Create dynamic form by repeating a same field pattern.
    For example: adding a team leader with all its teammates.

Lab 7

Performances

Agenda

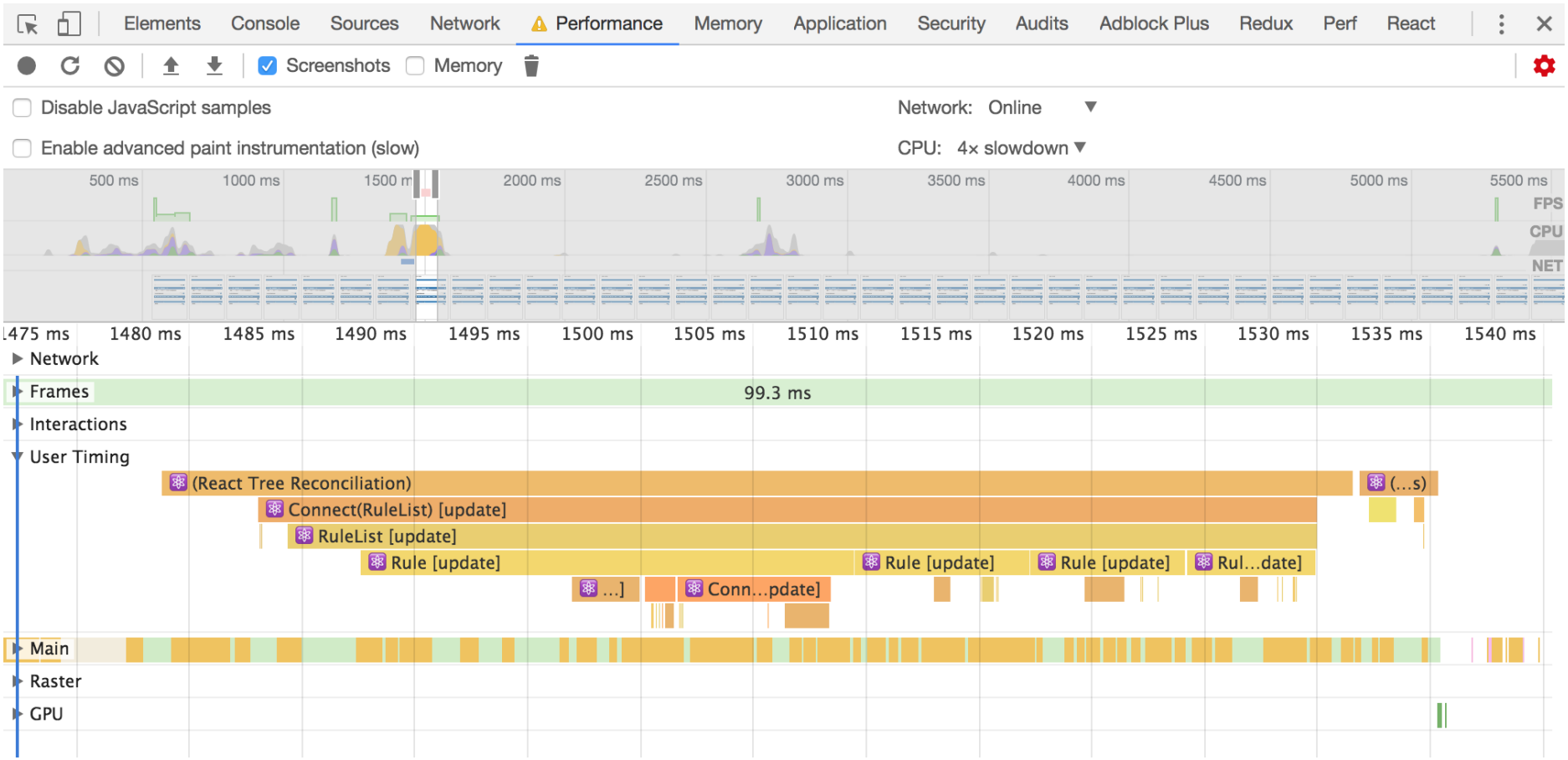
- Introduction
- Tooling
- React
- Rendering
- State
- Lifecycle
- Hooks
- Tests
- Redux
- REST Architecture
- Routing
- Forms
- *Performances*
- Isomorphism
- Going further
- Conclusion

Performances

- React is fast "by default", but:
  - Some components may be expensive to be built.
  - Huge tree DOM comparison may be expensive.
- `shouldComponentUpdate` method is essential to optimize performances.
- The more important is to **measure** thoroughly:
  - Browsers dev-tools

Performances - Tooling

Browser devtools helps to measure components time mounting, updating !



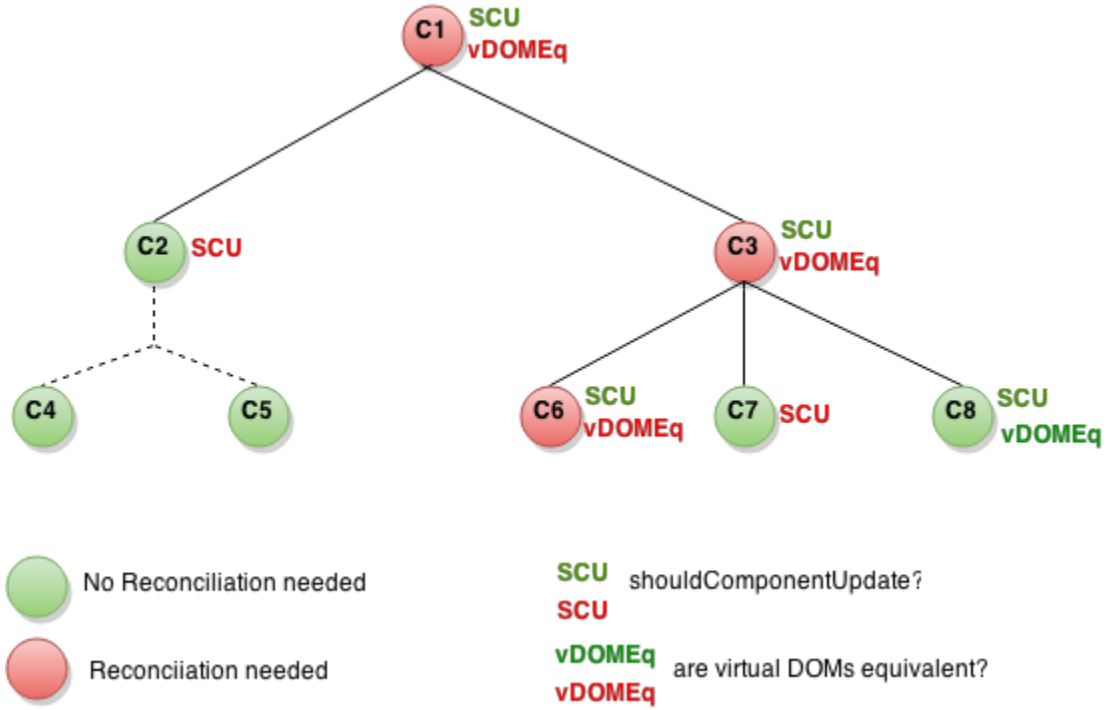
Performances - Optimizations

- All modifications of `state` or `props` start the rendering process and the **reconciliation**:
  - -> The component is rendered (`render` method).
  - -> The result is compared with current rendering (`virtual DOM`).
  - -> All needed changes are done in the DOM.
- **Be aware**: all components rendering (after modifying state/props) start the rendering (call to `render`) of the components tree below, **event if they don't undergo changes**.

Performances - shouldComponentUpdate

- `virtual DOM` concept is efficient, but:
  - Even if there is no changes to apply, `render` method will be executed.
  - Nothing is faster than "doing nothing" !
- Solution: `shouldComponentUpdate` method:
  - Method called **automatically** by React before rendering: component lifecycle.
  - Avoid calling `render` method.
  - Is applied to whole DOM tree !
  - Must return `true` or `false` depending on the component must be updated or not.

Performances - shouldComponentUpdate



Changes propagation with `shouldComponentUpdate`

Performances - shouldComponentUpdate

- Example: Default implementation

```
import React from 'react';

export default MyComponent extends React.Component {
  constructor(props) {
    super(props);
  }

  shouldComponentUpdate(nextProps, nextState) {
    return true;
  }

  render() { ... }
}
```

Performances - shouldComponentUpdate

- Example: ideal implementation, ***pure*** function

```
import React from 'react';

export default MyComponent extends React.Component {
  constructor(props) {
    super(props);
  }

  shouldComponentUpdate(nextProps, nextState) {
    return this.props !== nextProps || this.state !== nextState;
  }

  render() { ... }
}
```

Performances - shouldComponentUpdate

- **Be aware** to not be too much restrictive when blocking rendering
  - It blocks rendering children
  - It may be hard to debug (some components that doesn't update without errors).
  - May happen by using pure implementation on impure component or event on pure component having impure children!

Performances - shouldComponentUpdate

- `shouldComponentUpdate` method is often called by React:
  - Be aware of expensive computation.
  - Be aware when comparing objects: deep comparison is expensive (i.e `dirty checking`)
- In case of object comparison:
  - Immutable structures: comparison with `===` is enough !
  - Facebook provides a library: `immutable-js`.
  - https://facebook.github.io/immutable-js
  - https://github.com/facebook/immutable-js

Performances - PureComponent

- `PureComponent` implements a simplified version of `shouldComponentUpdate`
  - Perform a shallow comparison.
  - False-positives may happen with complex data structures.
  - Skip props updates on the component subtree.
  - Can be written in a simplified syntax

Performances - PureComponent

- ES2015: Full syntax

```
import React from 'react';

export default MyComponent extends React.PureComponent {
  constructor(props) {
    super(props);
  }

  // No need for shouldComponentUpdate. Acts like having:
  // shouldComponentUpdate(nextProps, nextState) {
  //   return this.props !== nextProps || this.state !== nextState;
  // }

  render() { ... }
}
```

Performances - PureComponent

- ES2015: Simplified syntax

```
import React from 'react';

export default ({ ...props }) => {
  ...
};
```

Performances - Redux

- `connect` implements `shouldComponentUpdate`
- Almost no need to implement it
- To avoid computation in containers: use `reselect`

Performances - immutable-js

- `immutable-js` library offers:
  - Immutable collections: `List`, `Stack`, `Map`, etc.
  - ***Lazy*** collections: `Seq`.
- Works well with React:
  - Handle complex data.
  - Support ES2015: `arrow functions`, `class`, `iterators`, etc.
- Not specific to React !

Performances - immutable-js

- Example: `Map`

```
let map1 = Immutable.Map({
  a:1,
  b:2
});

let map2 = map1.set('b', 2);
console.log(map1 === map2); // true

let map3 = map1.set('b', 5);
console.log(map1 === map3); // false
```

Performances - immutable-js

- Example: `List`

```
let list1 = Immutable.List.of(1, 2);

let list2 = list1.push(3);
console.log(list1 === list2); // false
```

Performances - immutable-js

- With `immutable-js`, `shouldComponentUpdate` function needs only strict comparison (`===`) !
- Benefits:
  - Simple.
  - Efficient.
- Drawbacks:
  - Libraries needed specific structures (`lodash`).
  - Be aware when chaining lazy operations (`map`, etc.) and basic operations (`get`, `delete`, etc.).

Lab 8

Isomorphism

Agenda

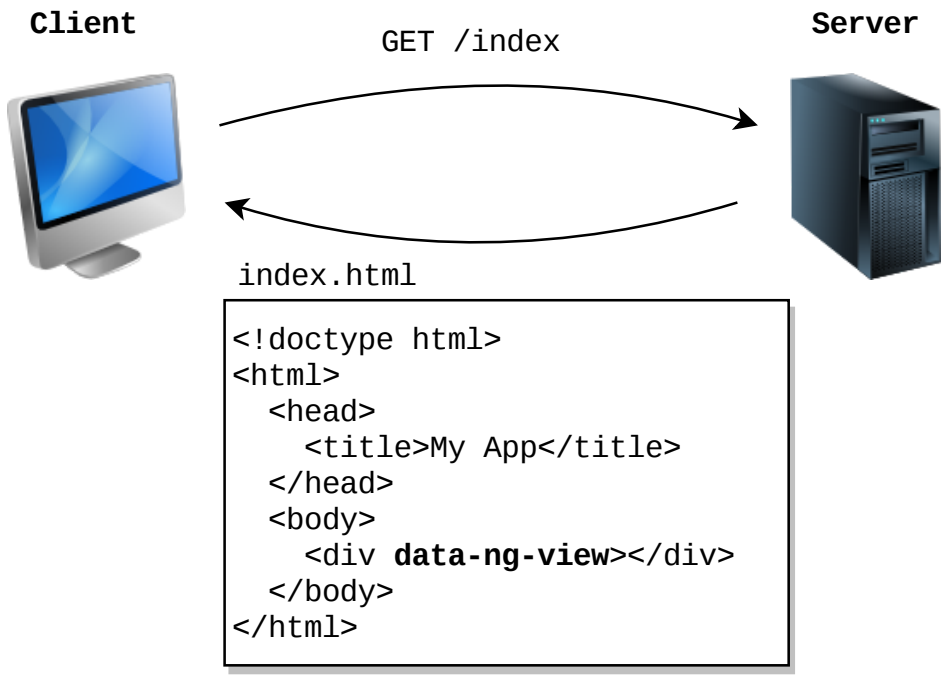
- Introduction
- Tooling
- React
- Rendering
- State
- Lifecycle
- Hooks
- Tests
- Redux
- REST Architecture
- Routing
- Forms
- Performances
- *Isomorphism*
- Going further
- Conclusion

Single Page Application

- Classic Single-Page-Application:
  - The server return an "empty shell".
  - Content is generated by client.
- Issues:
  - Indexation.
  - Accessibility.
  - Performance: The content is not directly available for users.

Single Page Application

- Example with Angular.js application:



Angular.js Application

Isomorphic Single Page Application

- Isomorphic Single-Page-Application:
  - HTML markup is generated by the server.
  - Client side, the browser displays directly the content.
- Benefit: resolve SPA issues while having rich application.
- Drawback: more complex than a classic SPA.
- Not specific to React, other frameworks are isomorphic driven:
  - rendr (https://github.com/rendrjs/rendr)
  - Angular (https://github.com/angular/angular)

Isomorphic React

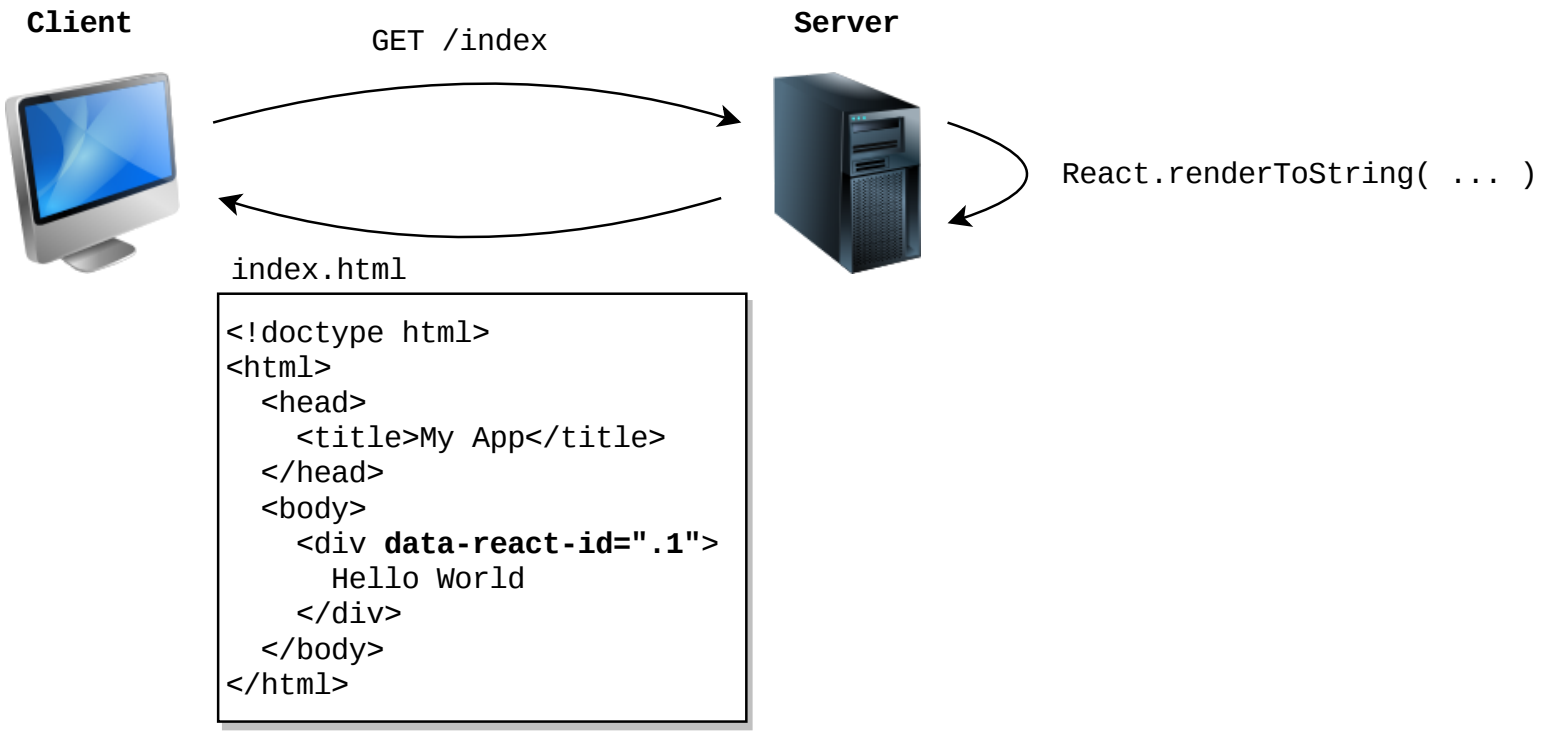
- How it works:
  1. HTML markup generated by the server.
  2. The browser displays the page, including HTML markup.
  3. The browser downloads Javascript files (React + bundle JavaScript).
  4. React spots the markup is there and instantiates the components without rendering them.
  5. The application starts and behaves like classic Single Page Application.

Isomorphic React

- Thanks to Virtual DOM, React doesn't need DOM to work:
  - Ideal to be run server side(Node.js).
  - Ideal to render a component as a `string`.
- `ReactDOMserver.renderToString` method in `react-dom/server` module, helps to build string representation of a component.
- React components are simple ES2015 / Node.js modules, so usable server side and browser side after webpack bundling.

Isomorphic React

- Example of React isomorphic application:



React Application

Isomorphic React

- Example with Express framework:

```
<!doctype html>
<html>
  <head></head>
  <body>
    {{ markup }}
  </body>
</html>
```

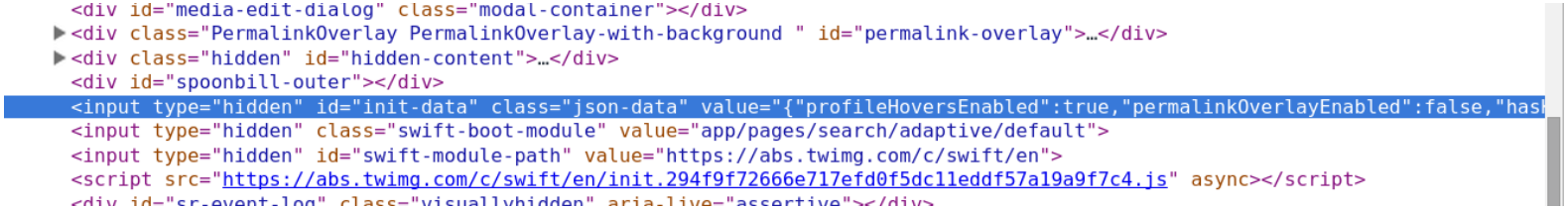
```
const React = require('react');
const ReactDOMserver = require('react-dom/server');
const MyComponent = require('./my-component.js');

// ....

app.get('/index', function (req, res) {
  fetchData(function() {
    res.render('index', {
      markup: ReactDOMserver.renderToString(React.createElement(MyComponent))
    });
  });
});
```

Isomorphic & Data

- `renderToString` method generates only HTML component markup.
- Issue: How to not fetch data twice ?
  - -> When generating the markup by the server.
  - -> When starting the application by the client.
- Solution: put data into HTML markup !
  1. Data serialization by the server (object -> JSON).
  2. Data deserialization by the client (JSON -> object).

Isomorphic & Data

- Example with Twitter:



Twitter

- Data are available inside the page source code.
- When the application starts, Twitter get these data:
  - No extra HTTP call.
  - Easy to deserialize JSON with JavaScript.

Isomorphic & Data

- Reminder: with Flux application, `stores` contain application state.
- When initializing `stores`, they can directly get data from page source code.
- Multiple solutions to get data from HTML markup:
  - Inline JavaScript.
  - Value of `hidden input`.
  - Content of DOM element (hidden or not executable).

Isomorphic & Data

- Inline JavaScript:

```
<html>
  <body>
    ....
  </body>
  <script type="application/javascript">
    const myData = { "id": 1, "name": "foo" };
  </ script>
</html>
```

- Getting data:

```
const myData = window.myData;
```

Isomorphic & Data

- Using an `input`

```
<html>
  <body>
    ....
    <input id="iso-data" type="hidden" value='{ "id": 1, "name": "foo" }' />
  </body>
</html>
```

- Getting data:

```
const myData = JSON.parse(document.getElementById('iso-data').value);
```

Isomorphic & Data

- Content of DOM element (hidden or not executable):

```
<html>
  <body>
    ....
    <script id="iso-app" type="application/json">{ "id": 1, "name": "foo" }</
script>
  </body>
</html>
```

- Getting data:

```
const myData = JSON.parse(document.getElementById('iso-data').textContent);
```

Isomorphic & Data

- Depends on constraints:
  - CSP (Content Security Policy) forbids inline JavaScript.
  - Be aware of formatting when using hidden elements (closing tags, single quote, etc.).
- If CSP is not activated: use inline JavaScript
  - Simpler.
  - More efficient (no access to the DOM).

React Isomorphic - redux (1/2)

- Example with `redux`: backend

```
function handleRender(req, res) {
  fetchData(result => {
    // Create initial state
    let initialState = {
      data: result
    };

    // Create store instance
    const store = createStore(app, initialState);

    // Render the component to a string
    const html = ReactDOMserver.renderToString(<Provider store={store}><App />
</Provider>);

    // Get the state from store (may have been updated by components).
    const finalState = store.getState();

    // Send the rendered page back to the client
    res.send(renderFullPage(html, finalState));
  })
}
```

React Isomorphic - redux (2/2)

- Example with `redux`: front

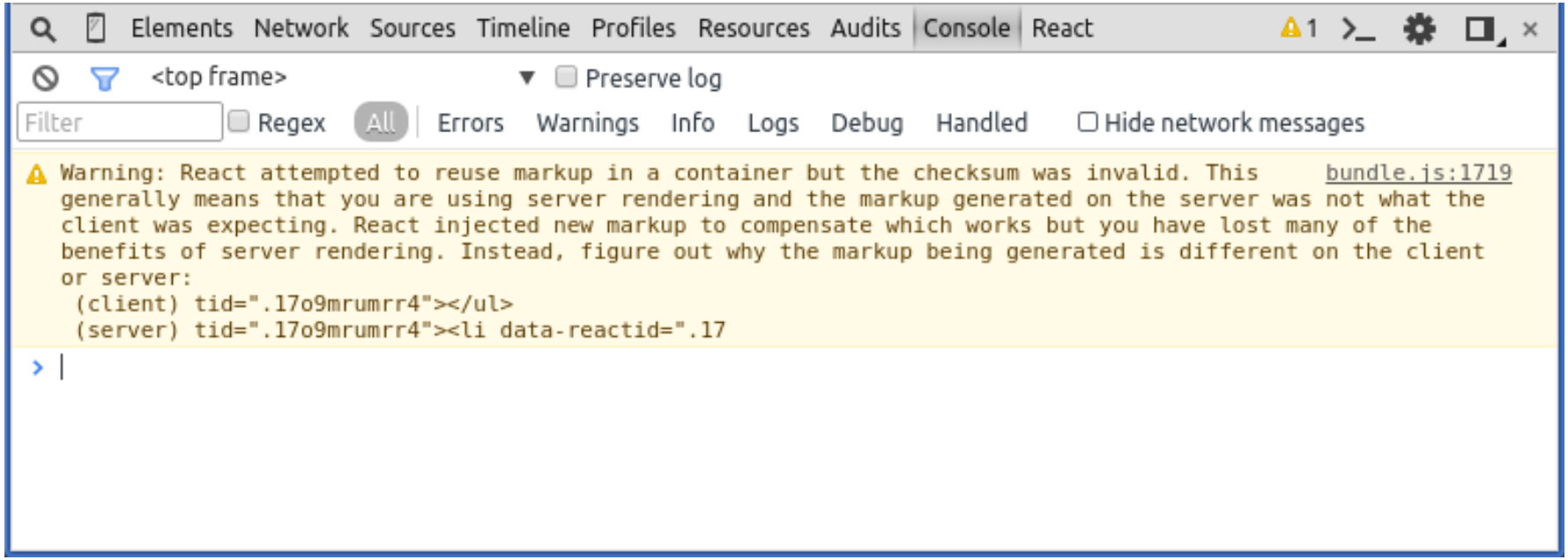
```
import React from 'react';
import { createStore } from 'redux';
import { Provider } from 'react-redux';
import App from './app';
import app from './reducers';

// Grab the state from a global injected into server-generated HTML
const initialState = window.__INITIAL_STATE__;

// Create Redux store with initial state
const store = createStore(app, initialState);

// Render App.
React.render(
  <Provider store={store}><App /></Provider>),
  document.getElementById('root')
);
```

React Isomorphic - Issues

- For the isomorphic application to work well, be aware of some points:
  - Client & server rendering must be equivalent:
  - If the rendering depends on:
    - Platform (screen resolution, browser)
    - Time (Progress bar)
    - Etc...
- Sometime the page cannot be fully isomorphic: use `componentDidMount` to delay client specific rendering.

React Isomorphic - Issues

- Example of warning when client and server rendering are not equivalent:



Going further

Agenda

- Introduction
- Tooling
- React
- Rendering
- State
- Lifecycle
- Hooks
- Tests
- Redux
- REST Architecture
- Routing
- Forms
- Performances
- Isomorphism
- *Going further*
- Conclusion

Going further ?

- Essential library: `lodash`
- using external libraries?
- Internationalisation
- Native Application: `react-native`

Lodash

- Created John David Dalton (team leader on Microsoft Edge JavaScript engine).
- Fork of `underscore`.
- Open Source: https://github.com/lodash/lodash
- Documentation: https://lodash.com/
- One of the most used library.
- Features:
  - Utilities functions (`map`, `reduce`, `pluck`, `indexBy`, etc.).
  - Complex functions : `deepClone`, `debounce`, `memoize`, etc.
- Works well with React !

Lodash

- Example: using debounce with React + Lodash

```
import React from 'react';
import _ from 'lodash';

export default class MyForm extends React.Component {
  constructor(props) {
    this.handleEvent = _.debounce(this.handleEvent, 100).bind(this);
  }

  compute() { ... }

  handleEvent(e) {
    this.compute();
  }

  render() {
    return <input type="text" onChange={this.handleEvent} />
  }
}
```

Internationalisation

- Internationalisation (i18n) is a classic issue for Single Page Application
- Need to format:
  - Dates
  - Numbers
  - Currencies
  - Texts
- No React default component...

Internationalisation

- Can use `react-intl` library
- Developed and maintained by Yahoo
- Open Source: https://github.com/yahoo/react-intl
- Documentation: http://formatjs.io/react/
- Multiple components available: `FormattedNumber`, `FormattedDate`, etc.
- Handle translations with `FormattedMessage` component
  - Pluralisation
  - Dictionary

Using external libraries

- While developing a web project, often need to use external libraries:
  - jQuery plugins (bootstrap, etc.)
  - Essential libraries: `lodash`, `moment`, etc.
- If the library doesn't change the DOM: OK
- If the library changes the DOM, so:
  - Plugin initialisation in `componentDidMount`
  - Update the component in `componentDidUpdate`.
  - Get DOM nodes with `ReactDOM.findDOMNode`.
  - Be aware on React DOM updates (`shouldComponentUpdate`)

Using external libraries

- Example:

```
import React from 'react';
import $ from 'jquery';

export default class MyView extends React.Component {
  constructor(props) { ... }

  componentDidMount() {
    let domNode = this.refs.placeholder;
    $(domNode).tooltip();
  }

  shouldComponentUpdate() {
    // Important, because the component manipulates the DOM.
    return false;
  }

  render() {
    return <button ref="btn" type="button">Help</button>;
  }
}
```

Using external libraries

- prefer using libraries dedicated to React:
  - `react-bootstrap`,
  - `react-google-maps`,
  - `react-modal`,
  - `react-d3`
  - etc.
- React ecosystem evolves every day !

react-native

- Library to use "natives" components
  - iOS: `TabBarIOS`, `NavigatorIOS`, etc.
  - Android: `ProgressBarAndroid`, `ToolbarAndroid`, etc.
- Documentation: https://facebook.github.io/react-native/
- Source: https://github.com/facebook/react-native
- Use React ecosystem in native application !
- Alternative to `phonegap` or `ionic`.

> *Learn once, write everywhere*

react-native

- `react-native` is pretty new:
  - Some incompatibilities exist between platforms.
  - Need to use polyfills for some API / features (geolocation, HTTP call with`XMLHttpRequest`, etc.).
- Still a promising library:
  - Sharing code between Web and native application.
  - Multiples components ready for use.
  - Performances: application fully native, more reactive than webview (ionic or phonegap).

react-native

- Example: Displaying a simple view

```
export default class MoviesView {
  constructor(props) { ... }

  getStyles() { ... }

  render: function(movie) {
    let styles = this.getStyles();
    return (
      <View>
        <Image
          source={{uri: movie.posters.thumbnail}}
          style={styles.thumbnail}
        />
        <View style={styles.rightContainer}>
          <Text style={styles.title}>{movie.title}</Text>
          <Text style={styles.year}>{movie.year}</Text>
        </View>
      </View>
    );
}
```

Conclusion

Agenda

- Introduction
- Tooling
- React
- Rendering
- State
- Lifecycle
- Hooks
- Tests
- Redux
- REST Architecture
- Routing
- Forms
- Performances
- Isomorphism
- Going further
- *Conclusion*

Conclusion

- React evolves steadily:
  - Updates.
  - Be aware of breaking changes...
- Important to check updates:
  - Tests as soon as possible.
  - Fix breaking changes as soon as possible.
- This is true for all libraries (`react-router`)...

Conclusion

- Some tips:
  - Avoid components doing everything.
  - Use JSX.
  - Respects component lifecycle.
  - Avoid DOM manipulation.
  - Tests !