REACT

AGENDA

- Introduction
- React
- Tooling
- Rendering
- Hooks
- Context

LOGISTIC

- Agenda
- Lunch and breaks
- Other questions ?

INTRODUCTION

AGENDA

- *Introduction*
- React
- Tooling
- Rendering
- Hooks
- Context
- Http requests

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

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;
```

REACT

AGENDA

- Introduction
- *React*
- Tooling
- Rendering
- Hooks
- Context
- Http requests

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: **17**
- 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`)

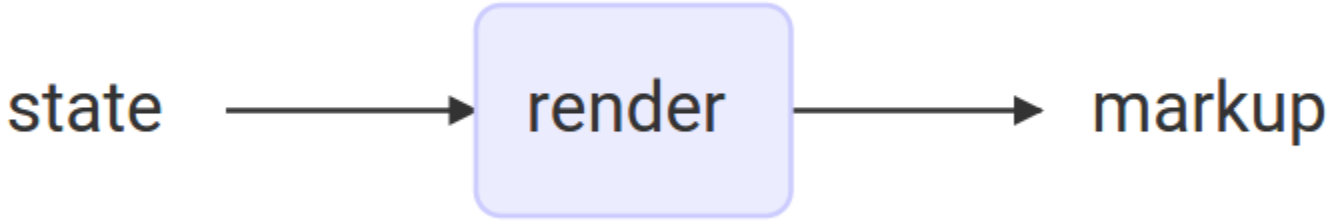
https://reactjs.org/docs/react-component.html

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`)

TOOLING

AGENDA

- Introduction
- React
- *Tooling*
- Rendering
- Hooks
- Context
- Http requests

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.

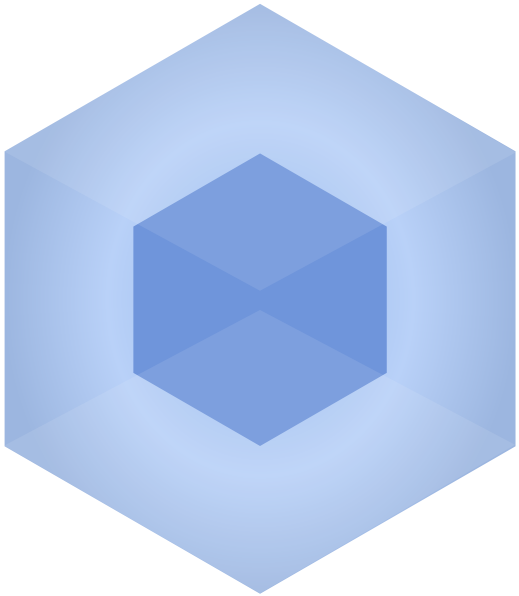
→ Using polyfills or transpilers from different languages (CoffeeScript, TypeScript, ...) to JavaScript to fill in the needs.

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`

Labs prerequisites

RENDERING

AGENDA

- Introduction
- React
- Tooling
- *Rendering*
- Hooks
- Context
- Http requests

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:

```
function Hello(props)
  const name = props.name || 'world'

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

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:

```
function HelloTeam() {
  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

```
function HelloTeam() {
  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:

```
function Hello(props) {
  const name = props.name || 'world'

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

JSX

JSX expression can be used as a variable

→ It is a `ReactElement`!

```
function HelloTeam(props) {
  const { name } = props

  const defaultNode = <Hello name="World" />
  const nameNode = <Hello name={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'}} />
```

LISTENING TO EVENTS

- React events are named using ***camelCase***, rather than lowercase.

- You must pass a ***function*** as the event handler

```
<button type="button" onClick={() => console.log('clicked')}>
  Click me
</button>
```

PROPS VALIDATION

In React, you have the ability to validate the props you receive in your component thanks to the `prop-types` library.

- ***Type***
- ***Object shape***
- ***Value***

```
import { string } from 'prop-types'

Greeting.propTypes = {
  name: string
}
function Greeting(props) {
  return (
    <h1>Hello {props.name}</h1>
  )
}
```

Lab 1

HOOKS

AGENDA

- Introduction
- React
- Tooling
- Rendering
- *Hooks*
- Context
- Http requests

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 USE 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 USE 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 USE 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

FIRST PEEK

useState is a Hook:

- Accept an initial value as argument
- Returns a pair: **current** state value & function to update it

```
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(count + 1)}>
  Click me
</button>
```

EFFECT HOOK

Let you perform side effects in components:

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

EFFECT HOOK

An example

```
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

Second parameter: an array of dependencies

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

Lab 2

HTTP REQUESTS

AGENDA

- Introduction
- React
- Tooling
- Rendering
- Hooks
- *Http requests*
- Context

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

REACT & HTTP REQUESTS

React does ***not*** provide a tool to do HTTP calls.

Making a http request in React is just making an http request in Javascript

- Can use `XMLHttpRequest` API.
- Can use `fetch` API.
- Can use any Javascript library allowing to create an http request.

FETCH

Make an http request

```
fetch('https://jsonplaceholder.typicode.com/todos/1')
```

Get the response and transform it to json format

```
fetch('https://jsonplaceholder.typicode.com/todos/1')
  .then(response => response.json())
```

Log the json data

```
fetch('https://jsonplaceholder.typicode.com/todos/1')
  .then(response => response.json())
  .then(data => console.log(data))
```

FETCH

Example in a React component

```
import React from 'react';

export default function TodosView() {
  const [todos, setTodos] = useState([])

  useEffect() {
    fetch('/todos')
      .then(response => response.json())
      .then(data => setTodos(data))
  }, [])

  return (
    ...
  )
}
```

Lab 3

CONTEXT

AGENDA

- Introduction
- React
- Tooling
- Rendering
- Hooks
- *Context*
- Http requests

CONTEXT API - OVERVIEW

- Context provides a way to pass data through the component tree without having to pass props down manually at every level
- Context is primarily used when some data needs to be accessible by **many** components at different nesting levels
- Use the Provider / Consumer pattern
- Use cases:
    - Theme
    - User locale
    - Data cache

CONTEXT API - EXAMPLE

```
// Create context with a default value
const ThemeContext = React.createContext('light');

const App = () => (
  // Use a Provider to pass the current theme to the tree below.
  // Any component can read it, no matter how deep it is.
  <ThemeContext.Provider value="dark">
    <ThemedButton />
  </ThemeContext.Provider>
)

const ThemedButton = () => (
  // Use a context consumer
  // React will find the closest theme Provider above and use its value.
  <ThemeContext.Consumer>
    {theme => <button className={theme === 'dark' ? 'dark-btn' : 'light-btn'}
/>}
  </ThemeContext.Consumer>
)
```

CONTEXT API - USING HOOK

- Avoid having to nest consumers by using the *useContext* hook.

```
// Create context with a default value
const ThemeContext = createContext('light');

const App = () => (
  // Use a Provider to pass the current theme to the tree below.
  // Any component can read it, no matter how deep it is.
  <ThemeContext.Provider value="dark">
    <ThemedButton />
  </ThemeContext.Provider>
)

const ThemedButton = () => {
  // Use the useContext hook to get the context value at render time
  const theme = useContext(ThemeContext)

  return <button className={theme === 'dark' ? 'dark-btn' : 'light-btn'} />
)
```

CONTEXT API - CAVEATS

- Reference identity is used to determine when to re-render

```
const App = () => (
  // This code will trigger a re-render of every consumer everytime the
provider rerenders
  <MyContext.Provider value={{ something: 'something' }}>
    <Toolbar />
  </MyContext.Provider>
)
```

- Component composition is often a simpler solution than context

CONTEXT API - UPDATE VALUE

```
const ThemeContext = createContext({ theme: 'dark', updateTheme: () => {} });

const App = () => {
  // Store theme in state
  const [theme, updateTheme] = useState('dark')
  // Create a memoized value for context, keeping reference across renders
  const themeContextValue = useMemo(() => ({ theme, updateTheme }), [theme,
updateTheme])

  return (
    <ThemeContext.Provider value={themeContextValue}>
      <ThemeUpdater />
    </ThemeConytext.Provider>
  )
}

const ThemeUpdater = () => {
  const { theme, updateTheme } = useContext(ThemeContext)
  return ( ... )
}
```

Lab 3