JS Frameworks for the web

pdf

Telecom Paris

Slides prepared by Jean-Claude Dufourd

Introduction

- JS is a very fast changing environment, everything changes in 2 years
- jQuery was the first framework, a ground breaker
- Other early frameworks have all died : prototype, mootools, yui, dojo. . .
- Other functions have been integrated in frameworks
  - link to a database, marshalling / unmarshalling
  - responsive design
  - JS module management
  - compilation
  - . . .

jQuery

- Manage the differences between browsers
- Develop faster, less verbose,
- Facilitate DOM manipulations, CSS selectors
- Manage mobile screen/feature differences
- Add UI sugar (including animations), through plugins
- A few tens of Kb
- CONS :
  - *Opinion :* horrible syntax : $
  - Encourages spaghetti code that is hard to debug : e.g. impossible to debug CSS if jQuery changes CSS on-the-fly
  - It is easy not to follow good practices
  - No high level abstraction
  - *Opinion :* No logic in the API, e.g. a return value can be a single value or an array, depending on context
- PROS :
  - Huge success, used everywhere
  - Lots of very vocal fans
  - Lots of variants and refactored versions

Common Notions in Modern Libraries

1. Modules and manifest
2. Compilation, obfuscation, minifier
3. HTML+CSS+JS components
4. Communication and async code
5. Binding
6. JS Dialects
7. Routing

Notion 1 : Modules and manifest

- Need for encapsulation : the notion of private in Java
- Closure :
  - The closure is a function call.
  - The variables and functions defined in a closure are not visible outside.
  - The return value of the closure makes variables and functions accessible outside.
  - The closure call parameters allow you to inject dependencies that can remain hidden.
  - The closure continues to exist as long as there is a pointer to his space.
  - The manifest is a documentation of everything that goes in or out out of a module / closure.

```
function checkscope() {
  var scope = "local scope";
  function f() { return scope; }
           // 'f' is a closure around 'scope'
  return f;
}
```

Example of manifest

```
{
  "name": "mpat",
  "version": "1.0.1",
  "description": "",
  "main": "webpack.config.js",
  "scripts": {
    "dev": "NODE_ENV=dev webpack-dev-server --content-base rea
    "build": "NODE_ENV=production webpack --progress",
...
  },
  "author": "",
  "license": "ISC",
  "dependencies": {
    "axios": "^0.15.3",
    "babel-core": "^6.26.0",
...
    "react": "15.3.1",
...
```

Modules, Node.js and the browser

- Node.js + npm : a lot of very professional modules
  - Node.js : Chrome's JS interpreter, packaged as a python interpreter
  - npm : package manager, equivalent of pip in python or gem in Ruby
- What about the browser ?
  - A module management framework . . . there are many
  - Require.js : import require.js, then a manifest, then everything is loaded (2 script objects)
  - webpack : a kind of compiler + linker that generates a big chunk of JS
- In the scope :
  - ES6, not everywhere . . . sometimes translated into ES5 > ES3
  - babel to handle all dialects
  - JSX to edit in HTML-like syntax
- There is now a big cost of entry into a JS project : module environment, packaging, dialects . . .

Notion 2 : Compilation, Obfuscation, Minifier

- The problem is less serious on the server side/node.js : use require() or import
- Many scripts to import into the HTML page -> 1 only
- Large number of hierarchical dependencies
  - Example of my last project : 655 module dependencies
  - Order of loading
  - Dependency additions, vulnerabilities
  - Single loading
- Reduce the size and the loading time
  - Compress / Minify
  - Remove the useless
- Protect the code (bof)
- Need a manifest that documents the module and its dependencies : package.json
- Many different systems, but it converges

Notion 3 : Components and templates

- A component is a coherent set of HTML + CSS + JS for a function
- More dependencies on other components
- Some frameworks do as much as possible in JS, others separate structure, content, style and code well
- The HTML part can be seen as a template

Notion 4 : Communication and asynchronous code

- As soon as a request goes through the Internet, the answer comes later and you should not wait because waiting would block other browser processes.
- Callback method

Async : Promise

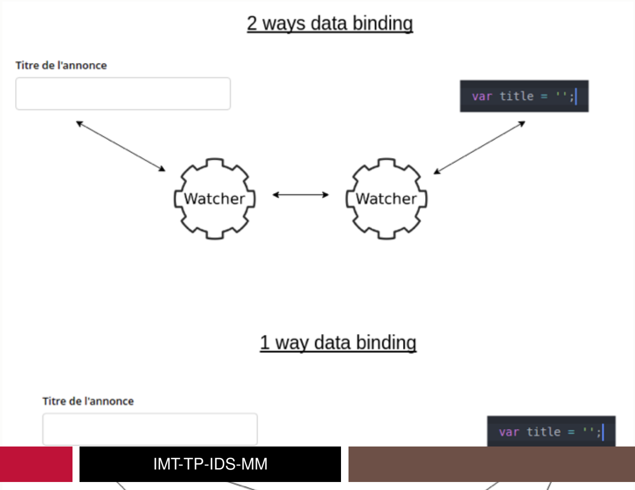
```
function get(url) {// Return a new promise.
  return new Promise(function(resolve, reject) {
    var req = new XMLHttpRequest();
    req.open('GET', url);
    req.onload = function() {
      if (req.status == 200) {
        resolve(req.response);
            // Resolve the promise with the response text
      } else {
        reject(Error(req.statusText));
            // Otherwise reject with the status text
      }};
    req.onerror = function() { // Handle network errors
      reject(Error("Network Error"));
    }
    req.send(); // Make the request
  });
}
```

../tp ../logo-IPP-s

Notion 5 : Binding

One-way vs two-way data binding

- Make the link between model variables and views
- There may be a need for a link in one direction and / or the other



../tp ../logo-IPP-s

Notion 6 : JS Dialects

- JSX
  - HTML-like markup to define the HTML in the JS
  - Allows the UI to be described within the JS code
  - Combines templating and JS
  - Can be translated on the fly
- TypeScript
  - JS with variable and function ***types***
  - OO, enum, generics, any
  - Complicated . . . but in Angular
- CoffeeScript (losing speed)
- All “can” be compiled in ES3 and ES5

Notion 7 : Routing

- Routing is usually on the server side
- Map URLs of a server
  - / routed to the intro
  - /docs routed to a webspace with docs
  - /restServ . . . routed to a REST service with urls like /RestServ/obj1/obj2/param/param2
  - /formResp . . . routed to a service that responds to an HTML form (?par=val&par2=val2)
  - It can be complicated, dynamic . . .
- Angular provides routing between views

Introduction to a few frameworks

- ***Bootstrap :***
  - Origin : Twitter and the need for “responsive” (mobile, tablet, desktop)
  - One page, only predefined components
  - CSS Framework
- React + Redux :
  - Origin : Facebook and their need to have plenty of components active on the screen
  - MV (c) multi component, multi thread
  - A state and a binding update cycle
  - Less framework and more library
  - One-way binding
  - React Native and React Navigation
- Angular :
  - Origin : Google
  - Full MVC, multi component, multi views
  - A rigid project structure (with CLI support)
  - Two-way binding
- D3 [illegible]

Bootstrap

Content

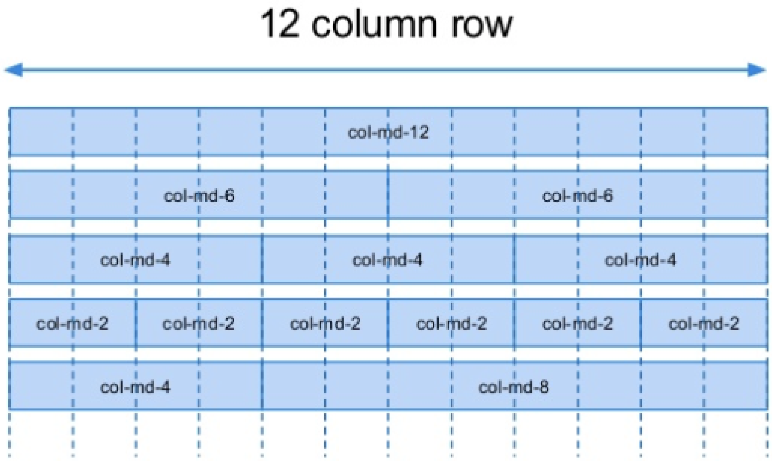
- CSS compatibility between browsers (reset of different defaults)
- 12 columns grid system for layout
- Multi-screen support (responsive design)
- The mobile has priority over the desktop
- Full of cool and easy to use widgets
- Plugins (dialogs, tabs, carousel, tooltips . . .)

Load Bootstrap from a CDN

```
<link href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/cs
<script src="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/j
<meta name="viewport" content="width=device-width, initial-sca
```

../tp ../logo-IPP-s

Layout


12 column row
col-md-12
col-md-6
col-md-6
col-md-4
col-md-4
col-md-4
col-md-2
col-md-2
col-md-2
col-md-2
col-md-2
col-md-2
col-md-4
col-md-8

How to use the grid

```
<div class="row">
    <div class="col-md-4">
        <!-- content -->
    </div>
    <!-- 8 columns remaining -->
</div>
```

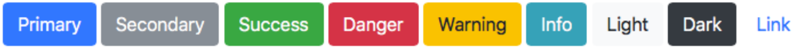
- col-xs-[num] no max size
- col-sm-[num] up to 750px
- col-md-[num] up to 970px
- col-lg-[num] up to 1180px

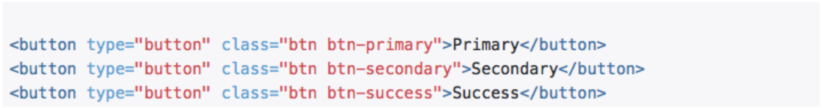
```
<div class="row">
    <div class="col-md-4 col-xs-6">
        <!-- content -->
    </div>
    <div class="col-md-8 col-xs-6">
        <!-- content -->
    </div>
```

Components

Existing Components

- defines easy to reuse classes





```
<button type="button" class="btn btn-primary">Primary</button>
<button type="button" class="btn btn-secondary">Secondary</button>
<button type="button" class="btn btn-success">Success</button>
```

Examples

Introduction to a few frameworks

- Bootstrap :
  - Origin : Twitter and the need for “responsive” (mobile, tablet, desktop)
  - One page, only predefined components
  - CSS Framework
- ***React + Redux :***
  - Origin : Facebook and their need to have plenty of components active on the screen
  - MV (c) multi component, multi thread
  - A state and a binding update cycle
  - Less framework and more library
  - One-way binding
- Angular :
  - Origin : Google
  - Full MVC, multi component, multi views
  - A rigid project structure (with CLI support)
  - Two-way binding
- D3.js : data visualization, origin : Stanford

React

- Assumes DOM is slow (apparently true)
- At **each** change, React recreates a rendering tree (in JS)
- Makes a diff with the previous tree and applies the diff to the DOM
- Data flows from parent component to children by props
- Component life cycle : component[Will|Did][Unm|M]ount()
- As soon as it is necessary to modify data, use the state / Redux
- In theory, React is usable alone
- In practice, use with Redux (or another state manager)
- Management of the state of the component : setState() (asynchronous and managed as an event)
- It is possible to insert React in existing HTML and even having lots of little bits of React in HTML, bits all connected to some piece of the Redux store

React+Redux

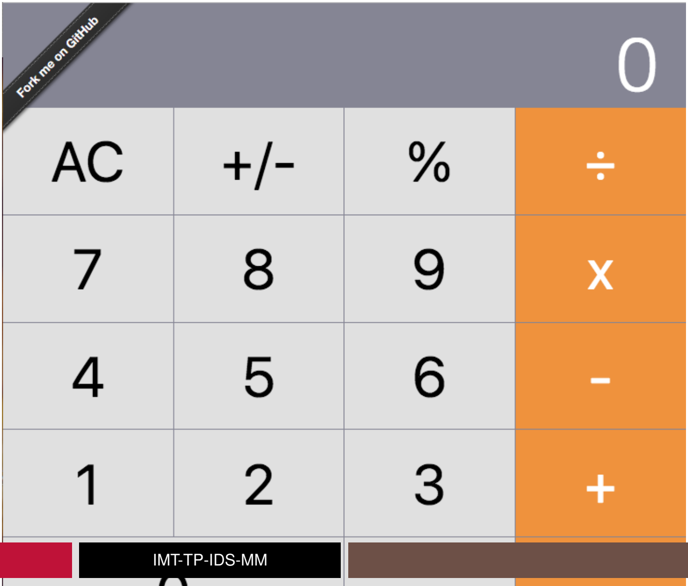
- React : the components
- Redux : the data flow and the state
  - Actions : React components that need to change something call (dispatch) actions (like events)
  - Reducers : actions are processed (asynchronously) by reducers : state + action –> new state
  - Store : the state and the reducers form the store (unique), to which the components subscribe to have subsets of the state
- Do not keep data out of the store
- The state of the components is ephemeral, unlike the state of the store
- Do not try to change the DOM, the DOM will be modified at the next change of state of the store
- Do not try to understand how it works (poorly documented)
- Do not modify a “props” (parameters of the component)
- Do not reorder the objects in the state, this breaks the optimizations
- Y[illegible] c[illegible] less
- The performances are very good despite everything

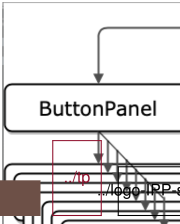
React and JSX

- JSX is designed so that everything can be done in the JS, even the HTML
- Parametric content : insert variables / expressions between {}
- Loops / tables : build a JS chart
- A component receives a list of parameters : props
- and passes some props to subcomponents
- An example of JSX :

```
export function BooleanInput({ label, label2, onChange,
                value = '', placeholder = '' }) {
  return (
    <div className="boolean-input">
      {label} {label && <br />}
      {label2} {label2 && <br />}
      <input type="checkbox"
                placeholder={placeholder}
```

React example





Redux : One-Way Binding

Tons of asynchronous components that all change the state in // = hell to debug

- The choice of Redux : component –> action –> reducer –> store –> component
  - The store is a single object, an array of model objects (MVC)
  - An action is an event whose semantics belong to the application, with parameters
  - A reducer takes a type of action and a state / store and makes a new state (copy). It is therefore necessary to have one reducer per type of action existing in the application
- Flow :
  - Interaction in the DOM
  - A listener JS reacts and launches an action (queuing wait)
  - The actions are handled by the reducers and the state is updated
  - The DOM is updated from the state
  - The starting DOM is not the same as the updated DOM
- To note :
  - The first choice of Flux was in the same direction (a store)

React Native and React Navigation

- React Native
  - extension of React to create native apps for major stores (iOS, Android)
  - uses native widgets/components when possible
  - mostly achieves similar experience on all phones
  - a server observes your working directory and feeds your phone as you design and debug the app
  - a React Native tool compiles the app for the target system, removing the need for a server, then the target system platform is used to compile the native app
- React Navigation
  - manages multiple pages with the same webapp (change page, tabs, drawers, etc)

Introduction to a few frameworks

- Bootstrap :
  - Origin : Twitter and the need for “responsive” (mobile, tablet, desktop)
  - One page, only predefined components
  - CSS Framework
- React + Redux :
  - Origin : Facebook and their need to have plenty of components active on the screen
  - MV (c) multi component, multi thread
  - A state and a binding update cycle
  - Less framework and more library
  - One-way binding
- ***Angular :***
  - Origin : Google
  - Full MVC, multi component, multi views
  - A rigid project structure (with CLI support)
  - Two-way binding
- D3.js : data visualization, origin : Stanford

Design Pattern : Decorator @name

- Function to add properties to an object
- Used to modify the object, for example to add properties
- To be used on a simple object, a function, a class . . .

```
function superhero(target)
  target.isSuper = true;
  target.power = "flight";
}
@superhero
class SuperMan() {}
console.log(SuperMan.isSuper) //true
```

Design Pattern : Dependency Injection

Fragility of certain classes

```
export class Car {
  public engine: Engine;
  public tires: Tires;
  constructor() {
    this.engine = new Engine();
    this.tires = new Tires();
  }
...}
```

Stronger code : I can change the engine type without modification of Car

```
export class Car {
  public engine: Engine;
  public tires: Tires;
  constructor(public engine: Engine, public tires: Tires) {
```

Dependency Injection

Pattern Factory OK but heavier and not OO

```
createCar() {
  let car = new Car(this.createEngine(), this.createTires());
  car.description = 'Factory';
  return car;
}
```

Simplification

```
export class Car {
  constructor(public engine: Engine, public tires: Tires) {}
...}
```

Usage of Dependency Injection

Service creation and declaration as injectable

```
import { Injectable } from '@angular/core';
import { HEROES }     from './mock-heroes';
@Injectable()
export class HeroService {
  getHeroes() { return HEROES; }
}
```

Declaration of use within the application

```
import { Component } from '@angular/core';
import { HeroService } from './hero.service';
@Component({...
  providers: [ HeroService ],
  ...})
export class HeroesComponent { }
```

Angular

- Origin : Google (2009+)
- JS Dialect : TypeScript
- Claims to do as well as native web apps
- A command line helps to create the structure :
  - components, modules, services
  - a node.js server monitors your files and recompiles everything after every change
- A component manages a page end / screen
- A template is the HTML view on the component
- A service can be a lot of things : logger (provides a functionality), data (provides data), encrypt (provides calculation)
- Directives add `if`, `for` and `switch` to HTML
- Decorators are used to indicate metadata for Angular
- Dependency Injection simplifies reuse of elements
- A life cycle to the Android for components
- Can be compiled to “native” iOS and Android applications

Template / HTML++

- Expression :

```
{{hero.name}}
```

- Statement :

```
<button (click)="onSave($event)">Save</button>
<button *ngFor="let hero of heroes">{{hero.name}}</button>
<div *ngIf="existsLetter">...</div>
<button [style.color]="isSpecial ? 'red' : 'green'">
<img [src]="heroImageUrl">
<form #heroForm (ngSubmit)="onSubmit(heroForm)"> ... </form>
```

Binding

- from model to view :

```
{{hero.name}}
<button [disabled]="isUnchanged">...</button>
```

- from view to model, through event :

```
(click)="add(hero.name)"
on-click="add(hero.name)"
```

- both ways, for use in a form :

```
[(ngModel)]="hero.name"
bindon-ngModel="hero.name"
```

Angular Routing

```
const routes: Routes = [
{ path: '', component: HomeComponent },
{ path: 'path/:routeParam', component: MyComponent },
{ path: 'staticPath', component: ... },
{ path: '**', component: ... },
{ path: 'oldPath', redirectTo: '/staticPath' },
{ path: ..., component: ..., data: { message: 'Custom' } }
]);

const routing = RouterModule.forRoot(routes);

<a routerLink="/path">
<a [routerLink]="[ '/path', routeParam ]">
<a [routerLink]="[ '/path', { matrixParam: 'value' } ]">
<a [routerLink]="[ '/path' ]" [queryParams]="{ page: 1 }">
<a [routerLink]="[ '/path' ]" fragment="anchor">
```

Component Life Cycle

```
ngOnChanges(changeRecord) { ... }
  // Called after every change to input properties and
  // before processing content or child views.
ngOnInit() { ... }
  // Called after the constructor, initializing input
  // properties, and the first call to ngOnChanges.
ngDoCheck() { ... }
  // Called every time that the input properties of a componen
  // or a directive are checked. Use it to extend change detec
  // by performing a custom check.
ngAfterContentInit() { ... }
  // Called after ngOnInit when the component's or directive's
  // has been initialized.
ngAfterContentChecked() { ... }
  // Called after every check of the component's or directive
ngAfterViewInit() { ... }
  // Called after ngAfterContentInit when the component's view
  // initialized. Applies to components only.
```

Angular Component : typescript

```
import { async, ComponentFixture, TestBed } from '@angular/co
import { HeroesComponent } from './heroes.component';

describe('HeroesComponent', () => {
  let component: HeroesComponent;
  let fixture: ComponentFixture<HeroesComponent>;
  beforeEach(async(() => {
    TestBed.configureTestingModule({
      declarations: [ HeroesComponent ]
    })
    .compileComponents();
  }));
  beforeEach(() => {
    fixture = TestBed.createComponent(HeroesComponent);
    component = fixture.componentInstance;
    fixture.detectChanges();
  });
  it('should create', () => {
```

Introduction to a few frameworks

- Bootstrap :
  - Origin : Twitter and the need for “responsive” (mobile, tablet, desktop)
  - One page, only predefined components
  - CSS Framework
- React + Redux :
  - Origin : Facebook and their need to have plenty of components active on the screen
  - MV (c) multi component, multi thread
  - A state and a binding update cycle
  - Less framework and more library
  - One-way binding
- Angular :
  - Origin : Google
  - Full MVC, multi component, multi views
  - A rigid project structure (with CLI support)
  - Two-way binding
- ***D3.js :*** data visualization, origin : Stanford

D3.js

- Easily display data in a web page
- Retrieve geographic data to draw on a map
- Retrieve encrypted data to display in graphs
- Display editable graphs
- [Examples] (https ://github.com/d3/d3/wiki/Gallery)
- All in JS, à la jQuery : we chain the calls which always return the object

```
var svg = d3.select("svg"),
    width = +svg.attr("width"),
    height = +svg.attr("height");

var simulation = d3.forceSimulation()
    .force("link", d3.forceLink().id(function(d) { return d.ic
    .force("charge", d3.forceManyBody())
    .force("center", d3.forceCenter(width / 2, height / 2));
```

Other frameworks

- VueJS :
  - by a former Google employee
  - seems lighter in many ways, more progressive
  - looks like Angular without TypeScript
- Polymer.js :
  - Google (again)
  - Component library, not a complete framework
  - Two-way binding
  - Would look more like React
- Meteor.js :
  - integrated with PhoneGap / Apache Cordova
  - like the others + dev server
  - can integrate React, Angular . . .
- Aurelia.js : (Microsoft)
- Ember.js : by the author of jQuery (?)

Which one ?

- If you work at Google : ***Angular***
- If you love TypeScript : ***Angular*** (or React)
- If you love object-orientated-programming (OOP) : ***Angular***
- If you need guidance, structure and a helping hand : ***Angular***
- If you work at Facebook : ***React***
- If you like flexibility : ***React***
- If you love big ecosystems : ***React***
- If you like choosing among dozens of packages : ***React***
- If you love JS & the "everything-is-Javascript-approach" : ***React***
- If you like really clean code : ***Vue***
- If you want the easiest learning curve : ***Vue***
- If you want the most lightweight framework : ***Vue***
- If you want separation of concerns in one file : ***Vue***
- If you are working alone or have a small team : ***Vue*** (or React)
- If your app tends to get really large : ***Angular*** (or React)

How to choose

- How mature are the frameworks / libraries ?
- Are the frameworks likely to be around for a while ?
- How extensive and helpful are their corresponding communities ?
- How easy is it to find developers for each of the frameworks ?
- What are the basic programming concepts of the frameworks ?
- How easy is it to use the frameworks for small or large applications ?
- What does the learning curve look like for each framework ?
- What kind of performance can you expect from the frameworks ?
- Where can you have a closer look under the hood ?
- How can you start developing with the chosen framework ?
- How old is the information on which I base my decision ? (> 2 years == trash)
- You have to identify the "hard" constraints of your project

Summary of the lesson

- JS libraries, history, jQuery
- Common notions : modules, manifest, compilation, components, async, binding, dialects, routing
- Frameworks : Bootstrap, React, Angular, D3
- Other frameworks, how to decide