Formation Angular

Les fondamentaux

Introduction

Versions d’Angular

- AngularJS est la première version qui a fait la popularité du framework.
- Angular 2 est une réécriture complète du Framework AngularJS. Sa base représente le socle futur du framework
- Angular 4 est un update d’Angular 2. La version 3 a été sautée pour des “problématiques” de versioning
- Angular 7 est la dernière version stable (Octobre 2018)

Angular

- Angular est un framework JavaScript qui permet de créer des **SPAs** réactives.
- Single Page Application
  - Application dont la navigation se fait sans rechargement de la page
  - Html minimaliste composé uniquement de blocs de récupération de code JavaScript (sous forme de bundle(s))
  - Le DOM (HTML) est entièrement manipulé par le JavaScript
  - Consultation du serveur uniquement pour manipuler des ressources
  - Applis Web avec une expérience proche de celle d’applications mobiles

WinDecision

Philosophie d’Angular

Angular est un framework **orienté composant**. Nous allons écrire de petits composants, et assemblés, ils vont constituer une application complète. Un **composant est un groupe d’éléments HTML**, dans un template, dédiés à une tâche particulière. Pour cela, nous aurons besoin d’un peu de **logique métier derrière ce template**, pour peupler les données, et réagir aux événements par exemple.

Architecture orienté Composants

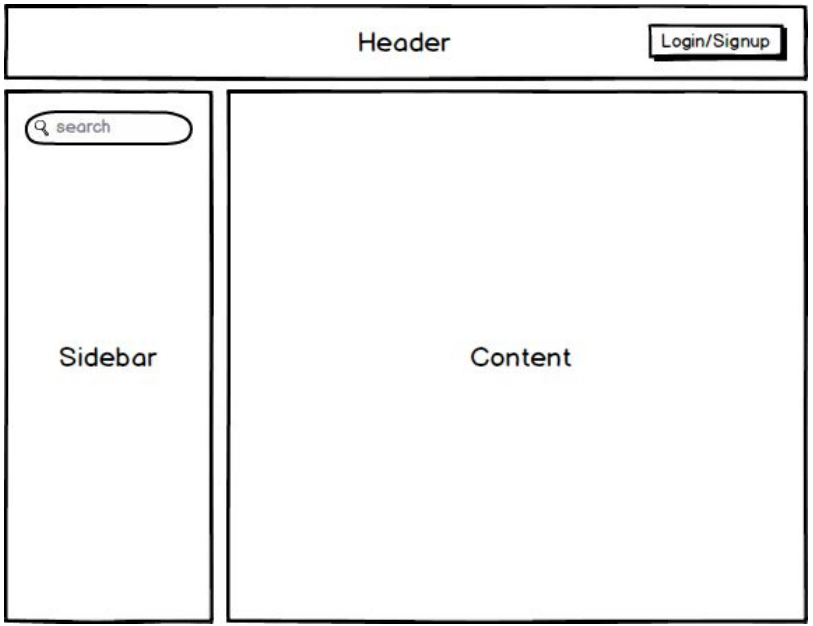
Header Component

Sidebar Component

Content Component

Search Component

Authentication Component



Web components

Un standard a été défini autour de ces composants : le standard Web Component ("composant web"). Même s'il n'est pas encore complètement supporté dans les navigateurs, on peut déjà construire des petits composants isolés, réutilisables dans différentes applications.

Cette orientation composant est largement partagée par de nombreux frameworks front-end : c'est le cas depuis le début de ReactJS. EmberJS et AngularJS ont leur propre façon de faire quelque chose de similaire ; et les petits nouveaux Aurelia ou Vue.js parient aussi sur la construction de petits composants.

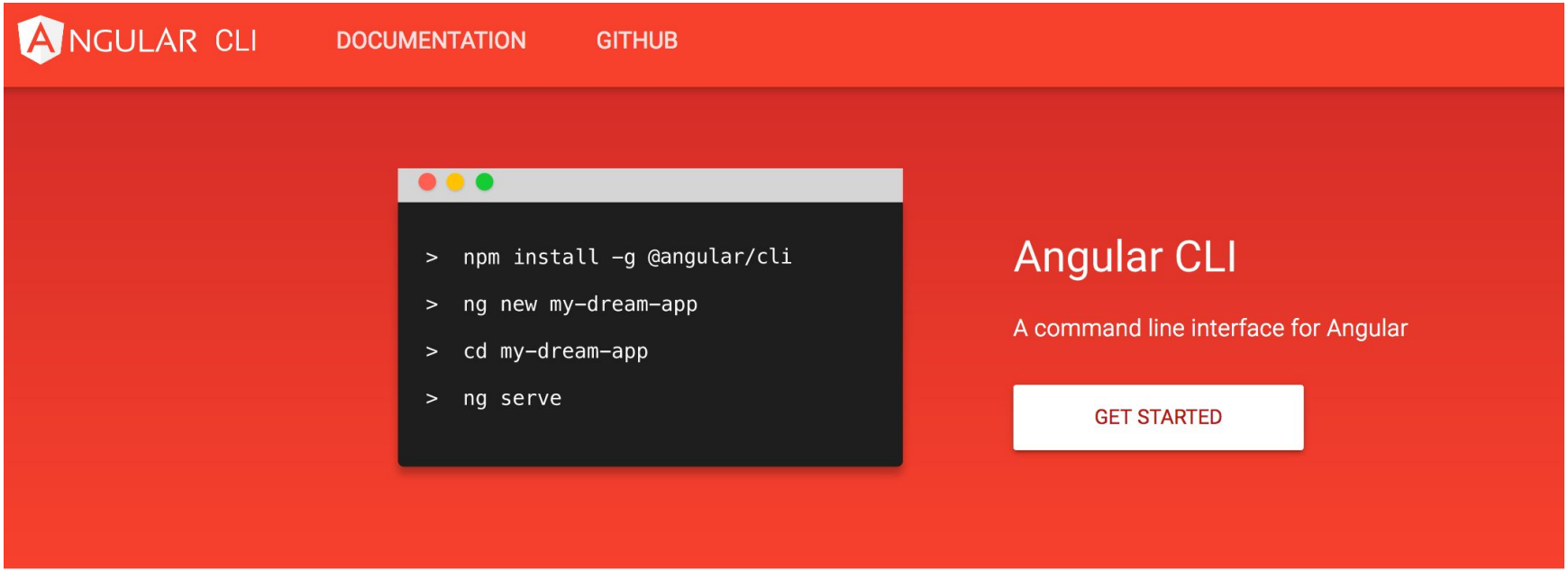
WinDecision.

Setup de projets - Environnement complexe

- Installation des dépendances
- Architecture/Squelette du projet
- File Watching
- Transpilation à la volée
- Sourcemaps
- Création de Bundle optimisé
- Exécution de tests unitaires/e2e
- Serveur de développement

WinDecision.

Setup de projet - Angular CLI



Setup de projets - Angular CLI

Commandes

- `npm install -g @angular/cli`
- `ng new sample-app`
- `cd sample-app`
- `ng serve`

Proxy

- **`npm config set proxy`** `http://proxy.company.com:8080`
- **`npm config set https-proxy`** `http://proxy.company.com:8080`

WinDecision.

Workshop 0

Environnement
30mn

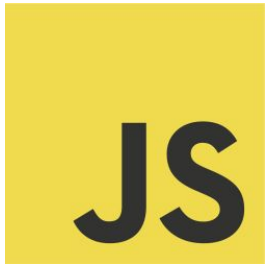
- Installer Node.js®
- Installer Angular CLI
- Initialiser un projet avec Angular CLI
- http://localhost:4200

JavaScript

Histoire de JavaScript

- Java et Java Applet - 1995
  - Développé par Netscape en 1995 (Brendan Eich en 10 jours :)
  - Noms originels : Mocha puis LiveScript
  - JavaScript -> Marketing
- Standardisation ECMAScript - 1996
- Ajax - 2005
- jQuery par John Resig - 2006
- v8 JavaScript Engine par Google - 2008
- NodeJS: v8 + librairies - 2009
- Pléthore de frameworks JS et de modules NPM

Logo non officiel



JavaScript - Frameworks/Librairies

FrontEnd
- Angular
- Ember
- React
- Vue.js
- D3.js
- ...

BackEnd
- Express
- Hapi.js
- Meteor
- Sails
- LoopBack
- ...

Mobile
- Ionic
- React Native
- NativeScript
- ...

Desktop
- Electron
- NW.js

Tools
- Npm
- SystemJS
- Webpack
- Bower
- Grunt
- Gulp

JavaScript - Entreprise

Paypal en 2013:

There’s been a lot of talk on **PayPal moving to node.js** for an application platform. As a continuation from part 1 of Set My UI Free, I’m happy to say that the rumors are true and our web applications are **moving away from Java onto JavaScript and node.js**.
Jeff Harrell

Nature du langage JavaScript

- Typage dynamique
- Paradigme Fonctionnel
- Paradigme Orienté Objet
- Langage interprété (Déploiement de code source)

WinDecision.

Typage dynamique

- JavaScript est un langage à typage dynamique.
- Il n'est pas nécessaire de spécifier le type de données d'une variable lors de sa déclaration.
- Les types de données sont convertis automatiquement durant l'exécution du script.

```
var foo = "bar";
foo = 1;
foo = {};
foo = [];
```

Paradigme Fonctionnel

- First-class functions
- Closures
- Application partielle via bind()
- Built-in map() et reduce()

```
// Fonction
var salam = function salam() { }
function applyHello(hello) {
  hello();
}
// Variable de fonction en paramètre
applyHello(salam);
// Fonction anonyme en paramètre
applyHello(function hola() { });
applyHello(function() { });
applyHello(() => { });
// Currying (using closures)
function addToX(base) {
  return function (val) {
      return val += base;
  }
}
var add2 = addToX(2);
add2(10); //12
```

WinDecision.

Paradigme Orienté Objet

- Polymorphisme
- Encapsulation
- Héritage

```
[1, 2, 3] instanceof Object
[1, 2, 3] instanceof Array

[1, 2, 3].toString() //"1,2,3"
new String("Hello ") instanceof Object
new String("Hello").toString() // Hello

// Abstraction des objets
function Map() {
   this.getValue = function (key) {
       return 'something';
   }
}
var m = new Map();
m.getValue('key');
```

WinDecision.

Paradigme Orienté Objet

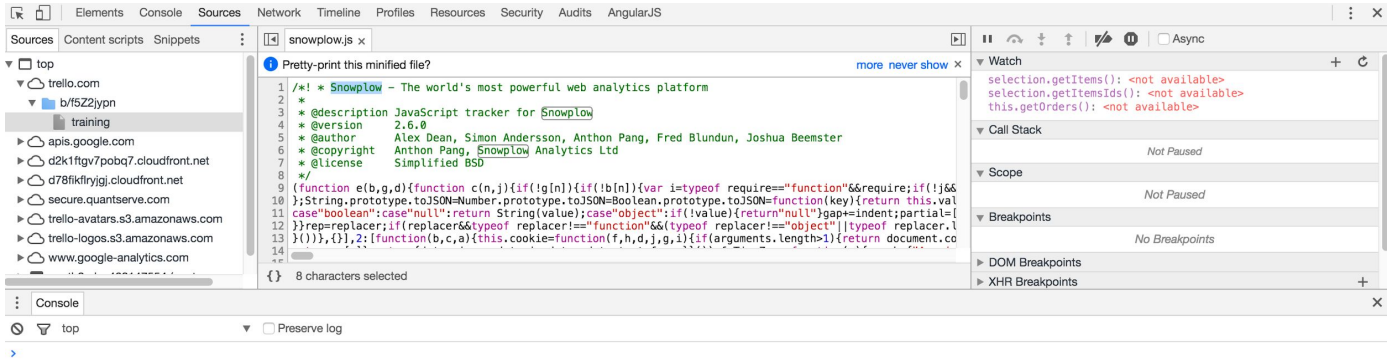
- Polymorphisme
- Encapsulation
- Héritage

```
function Greeter(greeting: string) {
   this.greeting = greeting;
}

Greeter.prototype.greet = function () {
   return "Hello, " + this.greeting;
}

let greeter = new Greeter("world");
greeter.greet();
```

WinDecision.

Déploiement de code source

- Langage interprété avec compilation à la volée (JIT)
- Inspection sur Chrome (DevTools)
- Techniques d'optimisation de taille de code (Minification, Obfuscation, Compression)



JavaScript / EcmaScript

JavaScript versus ECMAScript

- ECMAScript est le nom officiel de JavaScript
- ECMAScript est le nom utilisé dans la spécification du langage
- La version standard actuelle est **ES9** ou **ES2018**

WinDecision.

Versions ECMAScript

Version	Date
ES1,2,3	1997, 1998, 1999
ES4	2006
ES5	Décembre 2009
ES6/ES2015	Juin 2015
ES7/ES2016	Juin 2016
ES8/ES2017	Juin 2017
ES9/ES2018	Juin 2018

WinDecision.

JavaScript - ES6

ES6 ajoute une tonne de fonctionnalités à JavaScript (ES5), comme:

- les classes
- les constantes
- les arrow functions
- les promesses
- ...

WinDecision.

ES6 - Let et const

La déclaration de variable avec var peut être délicate. Dans à peu près tous les autres langages, une variable existe à partir de la ligne contenant la déclaration de cette variable. Mais en JS, il y a un concept nommé **hoisting** ("remontée") qui déclare la variable au tout début de la fonction, même si tu l'as écrite plus loin.

```
function getUserFullName(user) {
 if (user.isChampion) {
   var name = 'Champion ' + user.name;
   return name;
 }
 // Ici name = 'Champion xxx'
 return user.name;
}

// Equivalent à
function getUserFullName(user) {
 var name;
 if (user.isChampion) {
   name = 'Champion ' + user.name;
   return name;
 }
 // Ici name = 'Champion xxx'
 return user.name;
}
```

WildDecision.

ES6 - Let et const

ES6 introduit un nouveau mot-clé pour la déclaration de variable, let, qui se comporte enfin comme on pourrait s'y attendre.

L'accès à la variable name est maintenant restreint à son bloc. **let** a été pensé pour remplacer définitivement var à long terme

```
function getUserFullName(user) {
 "use strict";
 if (user.isChampion) {
   let name = 'Champion ' + user.name;
   return name;
 }
 console.log('toto');
 console.log(name);
 return user.name;
}

> getUserFullName({isChampion: false})
Uncaught ReferenceError: name is not defined
```

WinDecision.

ES6 - Propriétés d'objets

Il y a un nouveau raccourci pour créer des objets, quand la propriété de l'objet que l'on veut créer a le même nom que la variable utilisée comme valeur pour l'attribut.

```
function createSomething() {
 const name = 'Something';
 const color = 'blue';
 return { name: name, color: color };
}

// peut être simplifié en :

function createSomething() {
 const name = 'Something';
 const color = 'blue';
 return { name, color };
}
```

WinDecision.

ES6 - Affectations déstructurées

Il y a maintenant un raccourci pour affecter des variables à partir d'objets ou de tableaux.

```
var httpOptions = { timeout: 2000, isCache: true };
// later
var timeout = httpOptions.timeout;
var isCache = httpOptions.isCache;
```

```
const httpOptions = { timeout: 2000, isCache: true };
// later
const { timeout, isCache } = httpOptions;
```

WinDecision.

ES6 - Paramètres optionnels et valeurs par défaut

En ES5, si l'on passe plus d'arguments que déclarés par la fonction, les arguments supplémentaires sont tout simplement ignorés. Si l'on passe moins d'arguments que déclarés par la fonction, les paramètres manquants auront la valeur **undefined**

```
function getUsers(size, page) {
 size = size || 10;
 page = page || 1;
 // ...
 server.get(size, page);
}
```

WinDecision.

ES6 - Paramètres optionnels et valeurs par défaut

ES6 offre désormais une façon plus formelle de déclarer des paramètres optionnels, dès la déclaration de la fonction

```
function getUsers(size = 10, page = 1) {
 // ...
 server.get(size, page);
}
```

WinDecision.

ES6 - Classes

ES6 introduit les classes en JavaScript ! On pourra désormais facilement faire de l'héritage de classes en JavaScript. C'était déjà possible, avec l'héritage prototypal, mais ce n'était pas une tâche aisée, surtout pour les débutants…

```
class User {
 constructor(age: number) {
   this.age = age;
 }
}
class Student extends User {
 constructor(age, level) {
   super(age);
   this.level = level;
 }
}
const student = new Student(20, 3);
console.log(student.age); // 20
```

WinDecision.

ES6 - Promises

L'objectif des promises est de simplifier la programmation asynchrone. Notre code JS est plein d'asynchronisme, comme des requêtes AJAX, et en général on utilise des callbacks pour gérer le résultat et l'erreur. Mais le code devient vite confus, avec des callbacks dans des callbacks, qui le rendent illisible et peu maintenable.

```
getUser(login, function (user) {
    getRights(user, function (rights) {
        updateMenu(rights);
    });
});
```

```
getUser(login)
    .then(function (user) {
        return getRights(user);
    })
    .then(function (rights) {
        updateMenu(rights);
    })
```

WinDecision.

ES6 - Promises

Une promise est un objet **thenable**, ce qui signifie simplement qu'il a une méthode **then**. Cette méthode prend **deux arguments : un callback de succès et un callback d'erreur.**

Quand la promesse est réalisée, alors le callback de succès est invoqué, avec le résultat en argument. Si la promesse est rejetée, alors le callback d'erreur est invoqué, avec la valeur rejetée ou une erreur en argument.

```
const getUser = function (login) {
   return new Promise(function (resolve, reject) {
      // async stuff, like fetching users from server
      if (response.status === 200) {
        resolve(response.data);
      } else {
        reject('No user');
      }
   });
};
```

ES6 - Promises

Le code peut s'écrire à plat. Si par exemple le callback de succès retourne lui aussi une promise, on peut écrire l'écrire sans imbrication.

```
getUser(login)
    .then(function (user) {
        return getRights(user)
            .then(function (rights) {
                return updateMenu(rights);
            });
    })
getUser(login)
    .then(function (user) {
        return getRights(user);
    })
    .then(function (rights) {
        return updateMenu(rights);
    })
```

WinDecision.

ES6 - Promises

A l'image du then le catch permet de catcher une erreur rejetée dans la chaîne d'appels asynchrones.

```
getUser(login)
    .then(function (user) {
        return getRights(user);
    })
    .then(function (rights) {
        return updateMenu(rights);
    })
    .catch(function (error) {
        console.error('Error');
    })
```

WinDecision.

ES6 - Arrow functions

ES6 introduit une nouvelle syntaxe arrow function ("fonction flèche"), utilisant l'opérateur fat arrow ("grosse flèche") : ⇒. C'est très utile pour les callbacks et les fonctions anonymes !

```
getUser(login)
 .then(function (user) {
   return getRights(user);
 })
 .then(function (rights) {
   return updateMenu(rights);
 })
```

```
getUser(login)
 .then(user => getRights(user))
 .then(rights => updateMenu(rights))
```

WinDecision.

ES5 - this selon le contexte

Prenons un exemple où on itère sur un tableau avec la fonction map pour y trouver le maximum.

Le résultat obtenu n'est pas celui attendu car le this.max à l'intérieur de la closure passée au forEach fait référence aux window.max global !

```
max = 20;
var maxFinder = {
 max: 0,
 setMax: function (numbers) {
   numbers.forEach(
     function (element) {
       if (element > this.max) {
         this.max = element;
       }
     });
 }
};
maxFinder.setMax([2, 3, 4]);
console.log(maxFinder.max) // 0
```

WinDecision.

ES5 - Solution avec self = this

La solution classiquement utilisée pour faire référence à l'objet est d'utiliser une variable conventionnellement nommée self et d'y assigner le this en question.
La closure à l'intérieur du foreach n'aura donc plus à mal "interpreter" le this mais utilisera le self.

```
max = 20;
var maxFinder = {
 max: 0,
 setMax: function (numbers) {
   var self = this;
   // ici self === this === maxFinder
   numbers.forEach(
     function (element) {
       if (element > self.max) {
         self.max = element;
        // maxFinder.max fonctionne aussi
       }
     });
 }
};
maxFinder.setMax([2, 3, 4]);
console.log(maxFinder.max) // 4
```

WinDeation.

ES6 - Arrow functions et this

Mais il y a maintenant une solution bien plus élégante avec les arrow functions

```
max = 20;
const maxFinder = {
 max: 0,
 setMax: function (numbers) {
   numbers.forEach(element => {
     if (element > this.max) {
       this.max = element;
     }
   });
 }
};
maxFinder.setMax([2, 3, 4]);
console.log(maxFinder.max); // 4
```

WinDesign.

ES6 - Arrow functions et this

Mais il y a maintenant une solution bien plus élégante avec les arrow functions

```
var max = 20;
const maxFinder = {
   max: 0,
   setMax: function (numbers) {
       numbers
           .filter(element => element > this.max)
           .forEach(element => this.max = element);
   }
};
maxFinder.setMax([2, 3, 4]);
console.log(maxFinder.max); // 4
```

WinDecision

ES6 - Set et Maps

On **utilisait jusque-là de simples objets JavaScript** pour jouer le rôle de map ("dictionnaire"), c'est à dire un objet JS standard, dont les clés étaient nécessairement des chaînes de caractères. Mais nous pouvons **maintenant utiliser la nouvelle classe Map.**

```
// Map
const employee = { id: 1, name: 'Salman' };
const users = new Map();
users.set(employee.id, employee); // adds a user
console.log(users.has(employee.id)); // true
console.log(users.size); // 1
users.delete(employee.id); // removes the user

// Set
const employee = { id: 1, name: 'Salman' };
const users = new Set();
users.add(employee); // adds a user
console.log(users.has(employee)); // true
console.log(users.size); // 1
users.delete(employee); // removes the user
```

WinDevellop.

ES6 - Template strings

Construire des strings a toujours été pénible en JavaScript, où nous devions généralement utiliser des concaténations.

Les templates de string sont une nouvelle fonctionnalité mineure mais bien pratique, où on doit utiliser des accents graves (backticks `) au lieu des (quotes ') ou (double-quotes "), fournissant un moteur de template basique.

```
const fullname = 'Mr ' + firstname + ' ' + lastname;
const fullname = `Mr ${firstname} ${lastname}`;
```

WinDecision.

ES6 - Modules

Il a toujours manqué en JavaScript une façon standard de ranger ses fonctions dans un espace de nommage, et de charger dynamiquement du code. **NodeJS a été un leader** sur le sujet, avec un écosystème très riche de modules utilisant la convention **CommonJS**. Côté navigateur, il y a aussi l'API **AMD (Asynchronous Module Definition), utilisé par RequireJS**. Mais aucun n'était un vrai standard, ce qui nous conduit à des débats incessants sur la meilleure solution.

ES6 - Modules

ES6 a pour objectif de créer une syntaxe avec le meilleur des deux mondes, sans se préoccuper de l'implémentation utilisée. Le comité Ecma TC39 (qui est responsable des évolutions d'ES6 et auteur de la spécification du langage) voulait **une syntaxe simple.**

ES6 - Modules

```
// Fichier user.service
export function search(user) { // ...
}
export function get(id) { //...
}

// Exemple d'import
import { search, get } from './user.service';
search('someone');
get(1);
```

```
// Fichier user.service
export function search(user) { // ...
}
export function get(id) { //...
}

// Exemple d'import avec alias
import { search as searchUser } from './user.service';
searchUser('someone');

// Exemple de l'ensemble des éléments exportés
import * as userService from './user.service';
userService.get(1);
```

WinDecision.

Aujourd'hui - ES5 ou ES6 ?

- Compatibilité ES5 par les navigateurs
  - http://kangax.github.io/compat-table/es5/
- Compatibilité ES2015 (ES6) par les navigateurs
  - https://kangax.github.io/compat-table/es6/
- Conclusions
  - La version pleinement supportée par les navigateurs aujourd'hui est ES5
  - Le code source déployé de nos applications doit être en ES5

WinDecision.

Transpiler

ES6 vient d’atteindre son état final, il **n’est pas encore supporté par tous les navigateurs**. Et bien sûr, certains navigateurs vont être en retard.

Nous avons donc **besoin d’un Transpiler pour transformer (compiler) notre code ES6 en JavaScript ES5** supporté par les navigateurs du marché.

WinDecision.

Transpiler

Il y a trois outils principaux pour transpiler de l'ES6 :

- Traceur, un projet Google.
- Babel, un projet démarré par Sebastian McKenzie (17 ans) et qui a reçu beaucoup de contributions extérieures.
- TypeScript (ES6 et plus)

WinDecision.

Démo Babel

- Se rendre sur le site babeljs.io
- Cliquer sur Try it out
- Jouer avec ES6
- Optionnel: https://goo.gl/8XrvLq

TypeScript

TypeScript

A partir de la version 2.0 d'Angular, le framework entend tirer parti des nouveautés de EcmaScript 6 et même plus, avec TypeScript, qui rassemble à ES6 et un système de typage et d'annotations (décorateurs).

Les développeurs pourront choisir d'écrire en TypeScript (préconisé) ou en ES6 (très rare) avec une phase de transpilation vers ES5 ou directement en ES5 (extrêmement rare).

WinDecision.

TypeScript > ES6 > ES5

- ES6/ES2015 est l'évolution d'ES5
- TypeScript est un SuperSet d'ES2015/ES6 développé par Microsoft
- Angular 2 est développé en TypeScript
  - Initialement Angular 2 utilisait atScript
  - Rapprochement des équipes de Google et Microsoft

Pourquoi TypeScript

La nature dynamique de JavaScript est efficace en productivité mais elle peut aussi avoir un effet inverse.

Dans le cas de l'utilisation d'une API tierce. Les paramètres d'une fonction JavaScript ne peuvent être décrits que via sa documentation.

Sans les informations de type, les IDEs n'ont aucun indice pour savoir si l'on écrit quelque chose de faux, et les outils ne peuvent pas aider à trouver des bugs dans le code.

WinDecision.

Pourquoi TypeScript

- Tout ES6
- Système de types
  - Les api sont “self documented”
  - Détection des erreurs de types en phase de compilation avant (run)
  - Outillage: IntelliSense, Navigation entre les classes/méthodes, mise en brillance des erreurs…
  - Productivité
- Typage optionnel - Le JavaScript compile
- Le futur de JavaScript maintenant

WinDecision.

Démo TypeScript

- Se rendre sur le site typescriptlang.org
- Cliquer sur PlayGround

Optionnel:

- npm install -g typescript
- Créer un fichier .ts
- tsc test.ts

TypeScript - Les types

La syntaxe pour ajouter des informations de type en TypeScript est basique:

- `let variable: type;`

Les différents types sont simples à retenir :

```ts
const userNumber: number = 0;
const userName: string = 'Salman Yahya';

const printName = (user: User) => console.log(user.name);
printName({name: 'Salman'});
```

WinDecision.

TypeScript - Les types

TypeScript supporte aussi ce que certains langages appellent des types génériques

```ts
const users: Array<User> = [new User()];
users.push('hello'); // error TS2345
// Argument of type 'string' is not assignable to
parameter of type 'User'.
```

WinDecision.

TypeScript - Valeurs énumérées (enum)

TypeScript propose aussi des valeurs énumérées : enum. Par exemple, un utilisateur dans l'application peut être soit Actif, Inactif.

```
enum UserStatus {Active, Inactive}
const user: User = {status: UserStatus.Inactive};
```

WinDecision.

TypeScript - Return types

TypeScript permet de spécifier un type de retour aux fonctions

```ts
function setUser(user: User): User {
  user.status = UserStatus.Active;
  return user;
}
// Si la fonction ne retourne rien, déclarer avec void:
function setUser(user: User): void {
  user.status = UserStatus.Active;
}
```

WinDecision.

TypeScript - Interfaces

Une fonction marchera si elle reçoit un objet possédant la bonne propriété. Cette fonction peut être appliquée à n'importe quel objet ayant une propriété score. TypeScript permet de spécifier cette règle.

```
interface HasScore {
  score: number;
}
function addPointsToScore(player: HasScore, points: number){
  player.score += points;
}
addPointsToScore("erreur", 6); // Erreur de compilation
addPointsToScore({score: 10}, 6); // Ok
```

WinDecision.

TypeScript - Paramètre optionnel

Pour montrer qu'un paramètre est optionnel dans une fonction (ou une propriété dans une interface), on ajoute un ? après le paramètre. Ici, le paramètre points est optionnel.

```
function addPointsToScore
(player: HasScore,
points?: number): void {
  points = points || 0;
  player.score += points;
}
addPointsToScore({score: 10}); // Ok
```

WinDecision.

TypeScript - fonctions en propriété

On peut décrire un paramètre comme devant posséder une fonction spécifique plutôt qu'une propriété

```
interface CanRun {
 run(meters: number): void;
}

function startRunning(user: CanRun): void {
 user.run(10);
}

startRunning({
 run: (meters) => {console.log('started')}
});
```

WinDecision.

TypeScript - Classes

Une classe peut implémenter une interface. Pour nous, un user peut courir, donc on pourrait écrire :

```
class User implements CanRun {
  run(meters) {
      logger.log(`user runs ${meters}m`);
  }
}
```

WinDecision.

TypeScript - Classes

Le compilateur nous obligera à implémenter la méthode run dans la classe. Si nous l'implémentons mal, par exemple en attendant une string au lieu d'un number, le compilateur va crier :

```
class IllegalUser implements CanRun {
  run(meters: string) {
      console.log(`user runs ${meters}m`);
  }
}
// error TS2420: Class 'IllegalUser' incorrectly
implements interface 'CanRun'.
// Types of property 'run' are incompatible.
```

WinDecision.

TypeScript - Classes

Une classe en TypeScript peut avoir des propriétés et des méthodes. Avoir des propriétés dans une classe n'est pas une fonctionnalité standard d'ES6, c'est seulement possible en TypeScript

```
class SpeedyUser {
  speed = 10;
  run() {
      logger.log(`user runs at ${this.speed}m/s`);
  }
}
```

WinDecision.

TypeScript - Classes

Tout est public par défaut. Mais ont peut utiliser le mot-clé private pour cacher une propriété ou une méthode. Ajouter public ou private à un paramètre de constructeur est un raccourci pour créer et initialiser un membre privé ou public :

```
class NamedUser {
  constructor(
      public name: string,
      private speed: number) {
  }

  run() {
      logger.log(`user runs at ${this.speed}m/s`);
  }
}
```

WinDecision.

TypeScript - Décorateurs

C’est une fonctionnalité toute nouvelle, ajoutée seulement en TypeScript 1.5, juste pour le support d’Angular.

Les décorateurs peuvent modifier leur cible (classes, méthodes, etc.) et par exemple modifier les paramètres ou le résultat retourné, appeler d’autres méthodes quand la cible est appelée, ou ajouter des métadonnées destinées à un framework.

```
class UserService {
 @Log()
 getUsers() {
   // call API
 }

 @Log()
 getUser(userId) {
   // call API
 }
}
```

WinDecision.

TypeScript - Décorateurs

Selon ce sur quoi nous voulons appliquer notre décorateur, la fonction n'aura pas exactement les mêmes arguments. Ici nous avons un décorateur de méthode, qui prend 3 paramètres:

- target: la méthode ciblée par notre décorateur
- name: le nom de la méthode ciblée
- descriptor: le descripteur de la méthode ciblée

```
const Log = function () {
 return (target: any, name: string, descriptor: any) => {
   logger.log(`call to ${name}`);
   return descriptor;
 };
};

userService.getUsers();
// logs: call to getUsers
userService.getUser(1);
// logs: call to getUser
```

WinDecision.

Workshop 1

TypeScript
20mn

Définir une interface User ayant comme propriétés, firstName, lastName, email age et une liste de rôles.

L’âge doit être marqué comme propriété optionnelle.

Dans la classe du composant, créer un user et l’afficher sur la page (sans ses rôles).

Composants

Application Bootstrap - Root module

Notre application aura toujours au moins un module, le module racine.

Par la suite nous pourrons avoir d'autres modules, par fonctionnalité.

Par exemple, nous pourrions ajouter un module dédié à la partie Administration de notre application, contenant tous les composants et la logique métier de cette partie. Mais nous reviendrons là-dessus plus tard.

WinDecision.

Composants - Création

- Répertoire du nom du composant
  - Fichier .ts: La classe du composant
  - Fichier .html: Template du composant
  - Fichier .css: Style appliqué sur le composant
- Décorateur du composant
  - selector
  - templateUrl
  - styleUrls
- Ajout du composant aux déclarations du module

```ts
import { Component } from '@angular/core';

@Component({
 selector: 'app-footer',
 templateUrl: './footer.component.html',
 styleUrls: ['./footer.component.css']
})
export class FooterComponent {
}
```

WinDecision.

Composants - Déclarations

Nous devons déclarer les composants qui appartiennent à notre module racine dans l'attribut declarations de son objet de configuration. Ajoutons le composant que nous avons développé : FooterComponent.

```
...
import { FooterComponent } from './footer/footer.component';

@NgModule({
 declarations: [
  AppComponent,
  FooterComponent
 ],
 imports: [...]
})
export class AppModule { }
```

WinDecision.

Composants - Création via Angular Cli

- ng generate component footer
- ng g c footer

Angular CLI se chargera de faire

la création/insertion dans le module

```
import { Component } from '@angular/core';

@Component({
 selector: 'app-footer',
 templateUrl: './footer.component.html',
 styleUrls: ['./footer.component.css']
})
export class FooterComponent {
}
```

WinDesign

Composants - Utilisation

```
import { Component } from '@angular/core';

@Component({
 selector: 'app-root',
 template: `
       app works <br />
       <app-footer />
 `
})
export class AppComponent {
}
```

```
app works

footer component works
```

WinDecision.

Application Bootstrap - Composant Racine

Comme ce module est notre module racine, nous devons également indiquer à Angular quel composant est le composant racine, c'est à dire le composant que nous devons instancier quand l'application démarre. C'est l'objectif du champ **bootstrap** de l'objet de configuration :

```
import { BrowserModule } from
'@angular/platform-browser';
import { NgModule } from '@angular/core';
import { AppComponent } from './app.component';

@NgModule({
 declarations: [
  AppComponent
 ],
 imports: [
  ...
 ]
 bootstrap: [AppComponent]
})
export class AppModule { }
```

Application Bootstrap - Root module

Puisque nous construisons une application pour le navigateur, le module racine devra importer **BrowserModule**.

Ce n'est la seule cible possible pour Angular, nous pouvons choisir de rendre l'application sur le serveur par exemple, et devrions ainsi importer un autre module.

```
import { BrowserModule } from
'@angular/platform-browser';
import { NgModule } from '@angular/core';

@NgModule({
 declarations: [
  AppComponent
 ],
  imports: [
    BrowserModule
  ]
})
export class AppModule { }
```

WinDecision.

Application Bootstrap - Root component

```
import { Component } from '@angular/core';

@Component({
 selector: 'app-root',
 templateUrl: './app.component.html',
 styleUrls: ['./app.component.css']
})
export class AppComponent {
}
```

```
<!doctype html>
<html>
 <head>
  <meta charset="utf-8">
  <title> Angular 2+ - Formation </title>
  <base href="/">
  <meta name="viewport" … >
  <link rel="icon" ....>
 </head>
 <body>
  <app-root>Loading...</app-root>
 </body>
</html>
```

WinDecision.

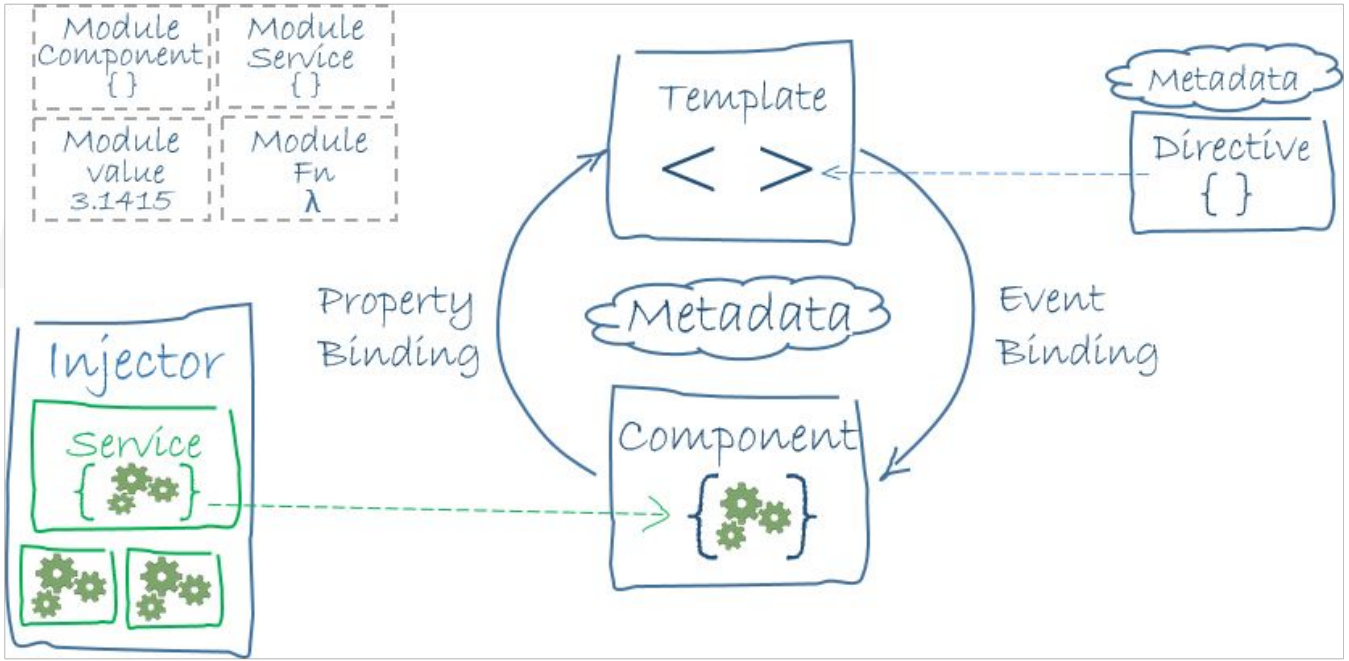
Workshop 2

Premier composant
30mn

- Créer un composant “user” qui affiche les nom, prénom et @ mail d’un utilisateur.
- Créer un composant “users” qui affiche plusieurs fois ce composant

NB: Les données étant statiques dans le template, le même utilisateur sera affiché plusieurs fois

Templates


Module
Component
{}
Module
Service
{}
Module
value
3.1415
Module
Fn
λ
Template
< >
Metadata
Directive
{ }
Property
Binding
Metadata
Event
Binding
Injector
Service
Component

Composants - Template / Url de template

La caractéristique principale d'**un @Component** est d'avoir **un template**. On peut soit déclarer le template en ligne, avec l'attribut template, ou utiliser une URL pour le placer dans un fichier séparé avec templateURL (mais pas les deux simultanément).

WinDecision.

Composants - Templates

- Le template est obligatoire
  - **templateUrl**: externe
  - **template**: inline
  - Obligatoirement **une des deux propriétés**
- Templates Externes:
  - Indiquer une url
- Templates Inline:
  - Uniquement si le code html n'est pas conséquent
  - **Entourer les inlines template de `**

```
import { Component } from '@angular/core';

@Component({
 selector: 'app-user',
 templateUrl: './user.component.html',
 // template: `<h1> Hello from user </h1>"
})
export class UserComponent {
}
```

WinDecision.

Composants - Templates - URLs

En règle générale, si le template est petit (1-2 lignes), c'est parfaitement acceptable de le garder en ligne. Quand il commence à grossir, le déplacer dans son propre fichier est une bonne façon d'éviter l'encombrement du composant.

On peut utiliser un chemin absolu pour l'URL, ou un relatif, ou même une URL HTTP complète.

Quand le composant est chargé, Angular résout l'URL et essaye de charger le template. S'il y parvient, le template deviendra le Shadow Root du composant, et ses expressions seront évaluées.

WinDecision.

Composants - Styles

- Le style est optionnel
  - **styleUrls**: externe
  - **styles**: inline
- Styles Externes:
  - Indiquer les urls relatives des fichiers de style
- Styles Inline:
  - Uniquement si le code css n'est pas conséquent
  - Entourer les inlines style de `

```
import { Component } from '@angular/core';
@Component({
 selector: 'app-user',
 templateUrl: './user.component.html',
 styles: [`
     h1 {
         color: darkred;
     }
 `]
})
export class UserComponent {
}
```

WinDecision.

Composants - Styles - URLs

L'attribut styles reçoit un tableau de règles CSS sous forme de chaînes de caractères. Comme on peut l'imaginer, elles deviennent vite énormes, alors utiliser un fichier séparé et styleUrls est une bonne idée. Comme le nom le laisse deviner, on peut y indiquer un tableau d'URLs

```
import { Component } from '@angular/core';
@Component({
 selector: 'app-user',
 templateUrl: './user.component.html',
 styleUrls: ['./user.component.css']
})
export class UserComponent {
}
```

WinDecision.

Composants - Selector

- Le sélecteur doit être unique
- Types de sélecteur:
  - Element: app-user
  - Attribute: [app-user]
  - Class: .app-user
- En général, nous utilisons uniquement le sélecteur Element pour les composants

```
import { Component } from '@angular/core';

@Component({
   selector: '[app-user]',
   template: `<h1> User </h1>......`,
   styles: [`
      h1 {
          color: darkred;
      }
  `]
})
export class UserComponent {
}
```

WinDecision.

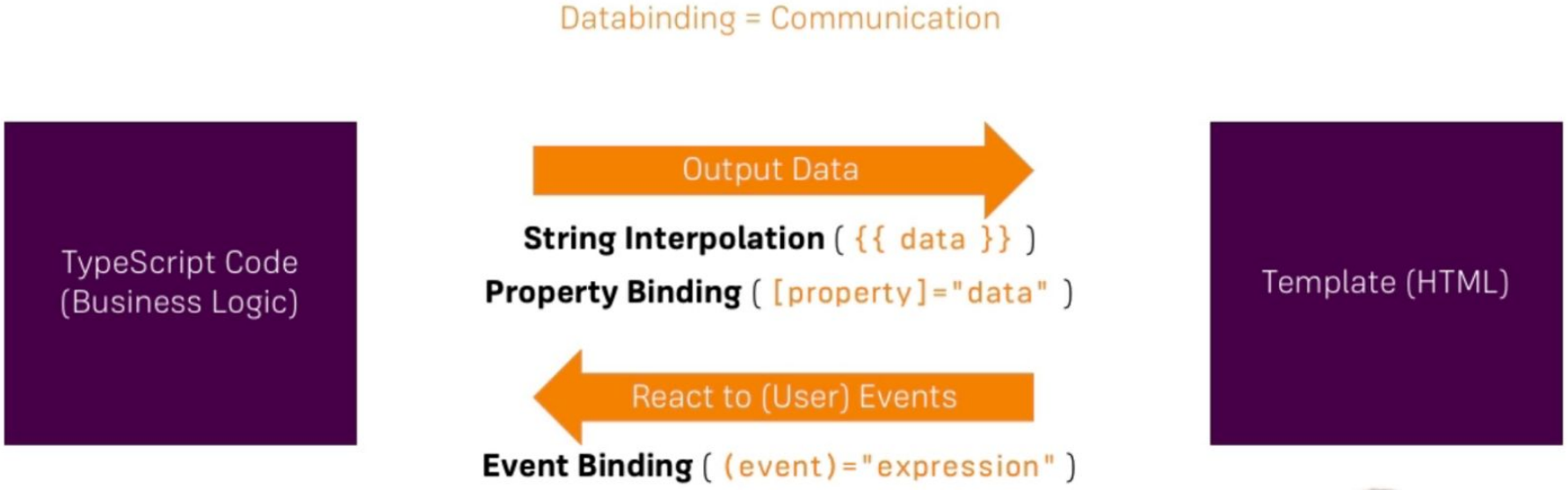
Workshop 3

Components: Templates et Styles
30mn

- Créer un composant alert-success qui affiche un message de success. Les style et template doivent être externes
- Créer un composant alert-error qui affiche un message d’erreur. Les style et template doivent être inline

Composants et Databinding

Databinding - Two way



Databinding - Interpolation

La classe UserComponent a une propriété, firstName. Et le template a été enrichi avec une balise <h2>, utilisant la fameuse notation avec double-accolades (les "moustaches") pour indiquer que cette expression doit être évaluée. Ce type de templating est de l'interpolation.

```
import { Component } from '@angular/core';

@Component({
selector: '[app-user]',
template: `<h1> User : {{firstName}} </h1>......`,
})
export class UserComponent {
   firstName = 'Adil"
}
```

WinDecision.

Databinding - Interpolation

Il faut cependant se rappeler une chose importante : si on essaye d'afficher une variable qui n'existe pas, au lieu d'afficher undefined, Angular affichera une chaîne vide. Et de même pour une variable null.

```
import { Component } from '@angular/core';

@Component({
selector: '[app-user]',
template: `<h1> User : {{firstName}} </h1>......`,
})
export class UserComponent {
}
```

WinDecision.

Databinding - Interpolation

Que se passe-t-il si mon objet user est en fait récupéré depuis le serveur, et donc indéfini dans mon composant au début ? Que pouvons-nous faire pour éviter les erreurs quand le template est compilé ?

```
@Component({
  selector: 'user-app',
  // user is undefined
  // but the ?. will avoid the error
  template: `
  <h1> User </h1>
  <h2> Welcome {{user?.name}} </h2>
`
})
export class UserComponent {
  user: any;
}
```

WinDecision.

Databinding - Binding de propriété

En Angular, on peut écrire dans toutes les propriétés du DOM via des attributs spéciaux sur les éléments HTML, entourés de crochets [].

```
<p> {{user.name}} </p>
<p [textContent]="user.name"></p>
```

WinDecision.

Databinding - Binding de propriété

La balise input ci-dessus a deux attributs : un attribut type et un attribut value. Chaque attribut HTML standard a une propriété correspondante dans un nœud du DOM. Mais un nœud du DOM a aussi d'autres propriétés, qui ne correspondent à aucun attribut HTML. Par exemple : childElementCount, innerHTML ou textContent.

```
<p> {{user.name}} </p>
<p [textContent]="user.name"></p>
```

WinDecision.

Databinding - Binding de propriété

La syntaxe à base de crochets permet de modifier la propriété textContent du DOM, et nous lui donnons la valeur user.name qui sera évaluée dans le contexte du composant courant, comme pour l'interpolation.
Note que l'analyseur est sensible à la casse, ce qui veut dire que la propriété doit être écrite avec la bonne casse.

```
<p> {{user.name}} </p>
<p [textContent]="user.name"></p>
```

WinDecision.

Databinding - Binding de propriété

Bien sûr, une expression peut aussi contenir un appel de fonction

```
<img src="http://localhost/images/{{user.getId()}}" />

<img [src]=" 'http://localhost/images/' + user.getId() " />
```

WinDecision.

Databinding - Événements

Les événements sont un mécanisme permettant de réagir aux interactions de l'utilisateur.

```
@Component({
 selector: 'app-users',
 template: `<h2>Users</h2>
   <button (click)="refreshUsers()">
            Refresh the users list
    </button>
   <p>{{users.length}} users</p>`
})
export class UsersComponent {
 users: any = [];
 refreshUsers() {
   this.users = [{ name: Rabat }, { name: 'Fès' }];
 }
}
```

WinDecision.

Databinding - Conclusion

Dans le premier cas de binding de propriété, **la valeur getSomething() est une expression**, et sera évaluée à chaque cycle de détection de changement pour déterminer si la propriété doit être modifiée.
Dans le second cas de binding d'événement, **la valeur doSomething() est une instruction** (statement), et ne sera évaluée que lorsque l'événement est déclenché.

```
<!--Expressions évalués par Angular à chaque cycle -->
<component property="{{user.name}}"></component>
<component property="{{getSomething()}}"></component>


<component [property]="user.name"></component>
<component [property]="getSomething()"></component>

<!-- Instruction exécutée sur réception d'un event -->
<component (event)="doSomething()"></component>
```

Databinding - Expressions vs instructions

Une expression sera évaluée plusieurs fois, par le mécanisme de détection de changement. **Elle doit ainsi être la plus performante** possible. Pour faire simple, une expression Angular est une version simplifiée d'une expression JavaScript.
Une expression doit être unique et sans effet de bord.

Une instruction est déclenchée par l'événement correspondant.

On peut enchaîner plusieurs instructions, séparées par un point-virgule. **Une instruction peut avoir des effets de bord**, et doit généralement en avoir.
Une instruction peut contenir des affectations de variables, et peut contenir des mot-clés.

Databinding - Variables locales

En analysant le template, Angular va regarder dans l'instance du composant pour trouver une variable et dans les variables locales. Les variables locales sont des variables qu'on peut déclarer dynamiquement dans le template avec la notation #

```
<input type="text" #name>
{{ name.value }}
```

```
<input type="text" #name>
<button (click)="name.focus()">
  Focus the input
</button>
```

WinDecision.

Databinding - Two-way databinding

La directive NgModel met à jour la valeur de l'input à chaque changement du modèle lié user.username, d'où la partie **[ngModel]="user.name"**.

La directive génère un événement depuis un output nommé ngModelChange à chaque fois que l'input est modifié par l'utilisateur, où l'événement est la nouvelle valeur, d'où la partie **(ngModelChange)="user.name = $event"**, qui met donc à jour le modèle user.username avec la valeur saisie.

```
<input type="text" [(ngModel)]="user.name" />
{{ user.name }}

<input type="text"
    [ngModel]="user.name"
    (ngModelChange)="user.name = $event" />
```

WinDecision.

Databinding - Utilisation de ngModel

Afin de pouvoir utiliser la directive ngModel, il faudrait ajouter le module FormsModule aux tableau imports[] défini dans le root module.

Nous aborderons plus en détail le module FormsModule sur la partie des formulaires

```
...
import { FormsModule } from '@angular/forms';
import { AppComponent } from './app.component';

@NgModule({
declarations: [ AppComponent ],
imports: [
  BrowserModule,
  FormsModule
],
 bootstrap: [AppComponent]
})
export class AppModule {  }
```

WinDecision.

Conclusion

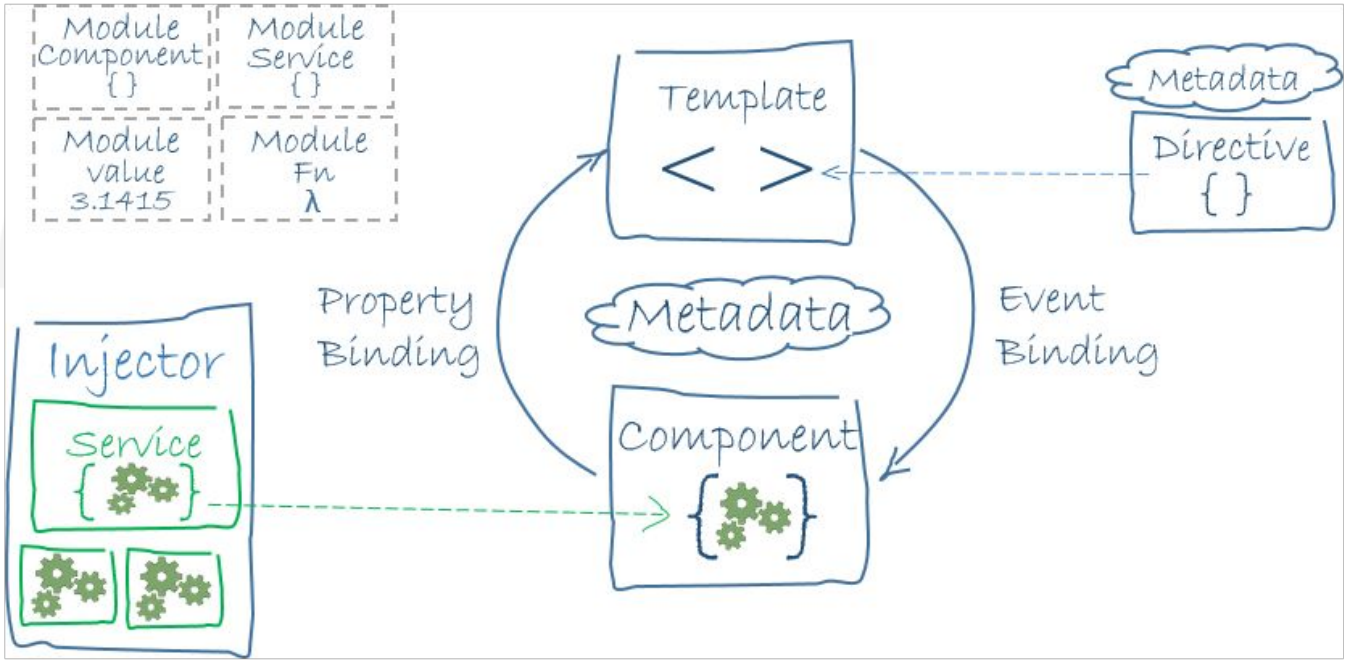
Le système de template d'Angular nous propose une syntaxe puissante pour exprimer les parties dynamiques de notre HTML. Elle nous permet d'exprimer du binding de données, de propriétés, d'événements, et des considérations de templating, d'une façon claire, avec des symboles propres :

- {{}} pour l'interpolation,
- [] pour le binding de propriété,
- () pour le binding d'événement,
- # pour la déclaration de variable locale

WinDecision.

Directives

Directives


Module
Component
{ }
Module
Service
{ }
Module
value
3.1415
Module
Fn
λ
Template
< >
Metadata
Directive
{ }
Property
Binding
Metadata
Event
Binding
Injector
Service
Component

Directives

Les directives sont des instructions dans le DOM. C’est une façon de dire à Angular de rajouter un comportement au DOM à partir d’une implémentation spécifique.

Les directives fournies par le framework sont déjà pré-chargées pour nous, nous n’avons donc pas besoin d’importer et de déclarer NgIf dans l’attribut directives du décorateur @Component.

WinDecision.

Directives - *ngIf

Utilisée pour n'afficher un template (un noeud du DOM) que lorsqu'une condition est réalisée

```
import { Component } from '@angular/core';

@Component({
  selector: 'app-users',
  template: `
        <div *ngIf="users.length">
           <h2>Users</h2>
        </div>`
})
export class UsersComponent {
  users: Array<any> = [];
}
```

WinDecision.

Directives - *ngIf else

Au lieu de mettre une condition sur un template et son contraire pour afficher deux blocs dont un uniquement doit être affiché, la syntaxe est dorénavant plus simplifié

Angular 2

```
<div *ngIf="condition">Mon Bloc OK</div>
<div *ngIf="!condition">Mon Autre Bloc</div>
```

Angular 4

```
<div *ngIf="condition; else elseBlock">
  Mon Bloc OK
</div>
<ng-template #elseBlock>
  Mon Autre Bloc
<ng-/template>
```

WinDecision.

Directives - *ngFor

*ngFor permet d'instancier un template par élément d'une collection

```
import { Component } from '@angular/core';

@Component({
  selector: 'app-users',
  template: `
    <ul>
     <li *ngFor="let user of users">{{user.name}}</li>
    </ul>`
})
export class UsersComponent {
  users = [
      {name: 'ali'}, {name: adil}
   ];
}
```

WinDecision.

Directives de templating - ngStyle

Si on veut changer plusieurs styles en même temps, nous pouvons utiliser la directive ngStyle :

Note que la directive attend un objet dont les clés sont les styles à définir. La clé peut être en camelCase (fontWeight) ou en dash-case ('font-weight').

```
import { Component } from '@angular/core';

@Component({
  selector: 'app-sample',
  template: `
    <ul>
     <div [ngStyle]="style">
    </ul>`
})
export class SampleComponent {
 style = { fontWeight: 'bold', color: 'red' };
}
```

WinDecision.

Directives de templating - ngClass

Dans le même esprit, la directive ngClass permet d'ajouter ou d'enlever dynamiquement des classes sur un élément.
Comme pour le style, on peut soit définir une seule classe avec le binding de propriété :

```
import { Component } from '@angular/core';

@Component({
   selector: 'app-sample',
   template: `
   <ul>
    <div [ngClass]="{'error': hasErrorStatus(),
'success': hasSuccessStatus()}">
   </ul>`
})
export class SampleComponent {
}
```

WinDecision.

Workshop 4

Binding et Directives du Framework
1h30

Créer une mini application affichant une liste d'utilisateurs et permettant d'ajouter un utilisateur à partir d'un formulaire à deux champs “Nom” et “Prénom” et un bouton “Ajouter”

Si un des champs n'est pas renseigné ou que le nombre d'utilisateurs est > 5, une alerte d'erreur est affichée

Si l'utilisateur a été rajouté, une alerte de succès est affichée

Démo

Debogage
10mn

- Interprétation des erreurs
- Utilisation de la web developer console
- Utilisation des sourcemaps
- Utilisation du plugin chrome Augury

Composants - Architecture

Application Planning

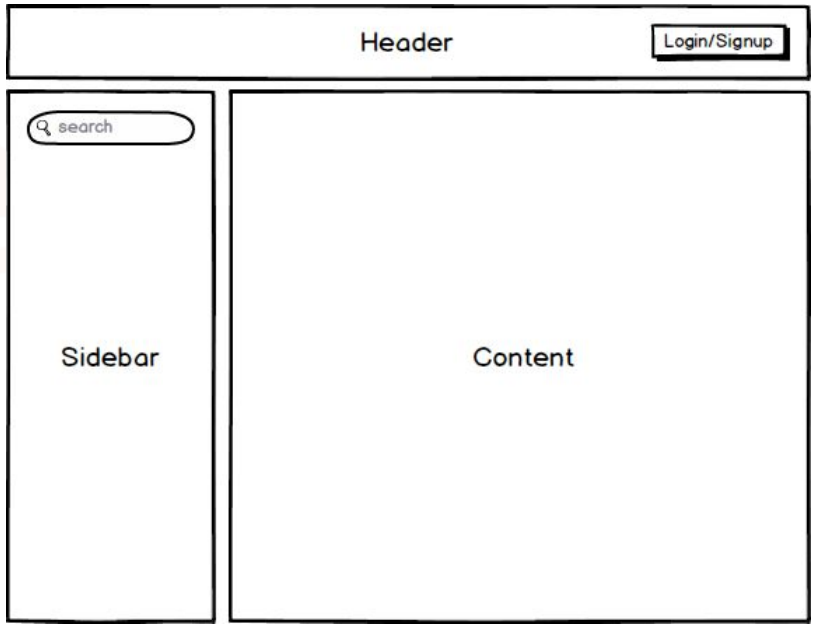
Avant de commencer le développement d'une application Angular, il faut:

- Découper l'application en plusieurs composants
- Décrire la responsabilité de chaque composant
- Spécifier les inputs/outputs de chaque composant

WinDecision.

Application Planning - Exemple

```
Header Component
 - Inputs: aucun
 - Outputs: LoginChanged

SidebarComponent
 - Inputs: aucun
 - Outputs: SearchTermChanged

ContentComponent
 - Inputs: SearchTerm

...
```



Data Flow

Le Data Flow décrit la façon dont les composants sont liés l'un à l'autre et le cheminement de la donnée qui en resulte.

```
<header (loginChanged)="loggedIn = $event"></header>
<sidebar (searchTermChanged)="searchTerm = $event"></sidebar>
<content [searchTerm]="searchTerm"></content>
```

WinDecision.

One-way Data Flow

Dans une approche one-way data Flow, la récupération des **données est centralisée au sein de composants dits intelligents** qui se chargent d'invoquer des services puis est acheminée aux composants dits **bêtes qui ne font que les afficher**. Un **événement dans un composant bête est remonté** au parent intelligent qui a la responsabilité de le traîter.

Nous parlons de **smart/dumb components** ou de **containers/components**

Propriétés et événements

Composants - Inputs

Les entrées sont parfaites pour passer des données d'un élément supérieur à un élément inférieur. Par exemple, si on veut avoir un composant affichant une liste d'utilisateurs, il est probable qu'on ait un composant supérieur contenant la liste, et un autre composant inférieur affichant un utilisateur

Afin de déclarer une entrée dans un composant, nous utilisons le décorateur @Input

Composants - Inputs

```
@Component({
 selector: 'app-users',
 template: `<app-user [user]="selectedUser">
            </app-user>
`
})
export class UsersComponent {
 selectedUser: User = { id: 1, name: 'Salman' }};
}
```

```
@Component({
 selector: 'app-user',
 template: `<div>{{user.name}}</div>`
})
export class UserComponent {
 @Input() user: User;
}
```

WinDecision.

Composants - Inputs / Alias

Si on souhaite exposer la propriété par un terme différent de celui de la propriété, il faudrait ajouter l'alias en paramètre de l'appel du décorateur.

```
@Input('userInput') user: User;
```

```
@Component({
 selector: 'app-user',
 template: `<div>{{user.name}}</div>`
})
export class UserComponent {
 @Input('user') internalUser: User;
}

@Component({
 selector: 'app-users',
 template: `<app-user [user]="selectedUser">
            </app-user>
`

})
export class UsersComponent {
 selectedUser: User = { id: 1, name: 'Salman' }};
}
```

WinDecision.

Composants - Événements

En Angular, les données entrent dans un composant via des propriétés, et en sortent via des événements.

Les événements sont un mécanisme permettant à un composant de notifier son parent et au parent de gérer la notification

Les événements spécifiques sont émis grâce à un EventEmitter, et doivent être déclarés dans le décorateur, via l'attribut outputs. Comme l'attribut inputs, il accepte un tableau contenant la liste des événements que le composant peut déclencher. Et, comme pour les inputs, on préfère généralement le décorateur @Output()

Composants - Événements

Si on veut émettre un événement appelé userSelected. Il y aura trois étapes à réaliser :

- déclarer l'événement en l'annottant du décorateur @Output()
- créer un EventEmitter
- émettre un événement quand le user est sélectionné

Dans l'exemple ci-dessus, chaque fois que l'utilisateur clique sur le nom d'un user, un événement userSelected est émis, avec le user qui en constitue la valeur (le paramètre de la méthode emit()).

Le composant parent écoute cet événement, il appellera sa méthode modifyUser avec la valeur de l'événement $event. $event est la syntaxe qui doit être utilisée pour accéder à l'événement déclenché.

WinDecision.

Composants - Événements

```
import { Component } from '@angular/core';
@Component({
 selector: 'app-users',
 template: `
   <app-user (userSelected)="modifyUser($event)">
   </app-user>`
})
export class UsersComponent {
 modifyUser($event) {
    console.log('Selected user\'s email'+ $event.email);
 }
}
```

```
import { Component, Output, EventEmitter }
     from '@angular/core';
@Component({
 selector: 'app-user',
 template: `
    <button (click)="clickedUser()">
      {{user.name}}
    </button>`
})
export class SelectableUserComponent {
 @Output() userSelected = new EventEmitter<User>();
 clickedUser() {
   this.userSelected.emit(this.user);
 }
}
```

WinDecision.

Composants - Événements / Alias

De la même manière, il est possible de mettre un tag/alias aux événements afin de désigner son utilisation dans le template

WinDecision.

Composants - Événements

```
import { Component } from '@angular/core';
@Component({
 selector: 'app-users',
 template: `
   <app-user (userSelected)="modifyUser($event)">
   </app-user>`
})
export class UsersComponent {
 modifyUser($event) {
    console.log('Selected user\'s email'+ $event.email);
 }
}
```

```
import { Component, Output, EventEmitter }
     from '@angular/core';
@Component({
 selector: 'app-user',
 template: `
    <button (click)="clickedUser()">
      {{user.name}}
    </button>`
})
export class SelectableUserComponent {
 @Output('userSelected') userClicked = new
EventEmitter<User>();
 clickedUser() {
   this.userClicked.emit(this.user);
 }
}
```

WinDecision.

Workshop 5

Binding de propriétés et Événements
1h30

Refactorer l’application de gestion des utilisateurs dans une approche one-way data flow:

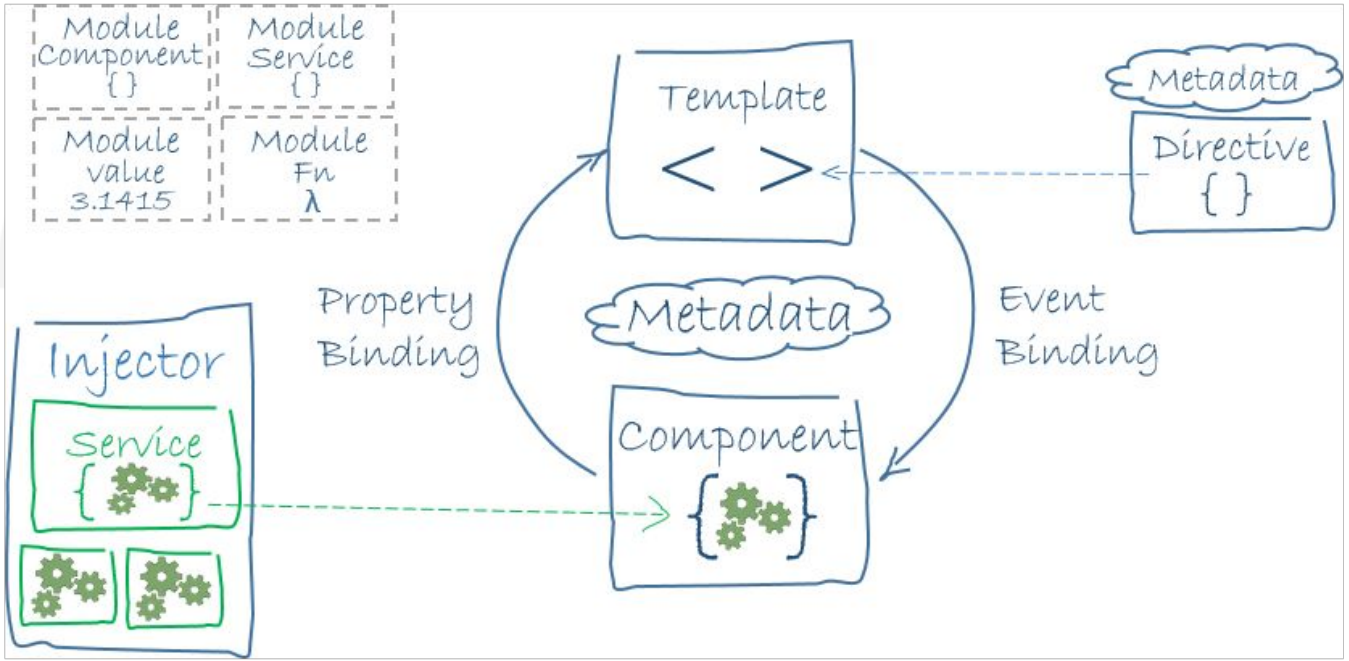
- Les formulaires et listings doivent être mis dans des composants spécifiques
- Les composants d’alerte doivent être paramétrables par le message à afficher.

L’injection de dépendances

Inversion de contrôle

L’injection de dépendances est un design pattern bien connu. Un composant peut avoir besoin de faire appel à des fonctionnalités qui sont définies dans d’autres parties de l’application (un service, par exemple). C’est ce que l’on appelle une dépendance : **le composant dépend du service**. Au lieu de laisser au composant la charge de créer une instance du service, **l’idée est que le framework crée l’instance du service lui-même, et la fournisse au composant** qui en a besoin. Cette façon de procéder se nomme l’inversion de contrôle.

DI


Module
Component
{}
Module
Service
{}
Module
value
3.1415
Module
Fn
λ
Template
< >
Metadata
Directive
{ }
Property
Binding
Metadata
Event
Binding
Injector
Service
Component

Injection de dépendances

L’injection de dépendances (une solution d’inversion de contrôle) apporte plusieurs bénéfices :

- le développement est simplifié, **on exprime juste ce que l’on veut**, où on le veut.
- la configuration est simplifiée, en **permutant facilement différentes implémentations**.
- le test est simplifié, en permettant de **remplacer les dépendances par des versions bouchonnées.**

C’est un concept largement utilisé côté serveur, mais AngularJS 1.x était un des premiers à l’apporter côté client.

WinDecision.

Injection de dépendances - Angular

Pour faire de l'injection de dépendances, on a besoin :

- d'une façon d'**enregistrer une dépendance**, pour la rendre disponible à l'injection dans d'autres composants/services.
- d'une façon de **déclarer quelles dépendances sont requises** dans chaque composant/service.

Le framework se chargera ensuite du reste. Quand on déclarera une dépendance dans un composant, il regardera dans le registre s'il la trouve, récupérera l'instance existante ou en créera une, et réalisera enfin l'injection dans notre composant.

WinDecision.

Les services

Angular propose le concept de services : des classes que l’on peut injecter dans une autre.

Quelques services sont fournis par le framework, certains par les modules communs, et on peut en construire d’autres.

WinDecision.

Créer son propre service

Un service est un singleton, donc la même instance unique sera injectée partout. Cela en fait le moyen idéal pour **partager un état parmi plusieurs composants séparés** !

```
export class UserService {
 list() {
   return [{ name: 'London' }];
 }
}
```

WinDecision.

Injection de dépendances - Déclaration et Utilisation

```
import { UserService } from './user.service';
@NgModule({
 imports: [BrowserModule, HttpModule],
 declarations: [UserComponent],
 providers: [
   UserService
 ],
 bootstrap: [UserComponent]
})
export class AppModule {
}
```

```
import { Component } from '@angular/core';
import { UserService } from './user.service';

@Component({
 selector: 'app-root',
 templateUrl: './app.component.html',
 styleUrls: ['./app.component.css']
})
export class AppComponent {
 title = 'app works!';
 constructor(public userService: UserService) {}
}
```

Injection de dépendances - Déclaration et Utilisation

Nous pouvons décider de ne rendre injectable l'instance du service qu'au sein d'un composant (et de ses composants fils). Ceci est fait en déclarant le service dans la liste des providers du composant.

```
import { Component } from '@angular/core';
import { UserService } from './user.service';

@Component({
 selector: 'app-root',
 templateUrl: './app.component.html',
 styleUrls: ['./app.component.css'],
 providers: [UserService]
})
export class AppComponent {
 title = 'app works!';
 constructor(public userService: UserService) {}
}
```

WinDecision.

Dépendances entre services

Si le service a lui aussi des dépendances, alors il faut lui ajouter le décorateur @Injectable().

Sans ce décorateur, le framework ne pourra pas faire l'injection de dépendances.

```
@Injectable()
export class UserService {
 constructor(public http: Http) {}
 list() {
   return http.get('/api/cars');
 }
}
```

WinDesign.

Dépendances entre services

Depuis Angular 6, il existe une nouvelle façon recommandée d'enregistrer une dépendance: utiliser le décorateur @Injectable en indiquant comme valeur de la propriété providedIn, le module dans lequel le service doit être enregistré.

```
@Injectable({
  providedIn: 'root',
})
export class UserService {
 constructor(public http: Http) {}
 list() {
   return http.get('/api/cars');
 }
}
```

Workshop 6

Services
1h30

Créer un service Users exposant les méthodes suivantes:

- add(User: user): void
- update(User: user): void
- get(id: number): User
- getAll(): Array<User>

Refactorer l'application en centralisant la gestion du tableau des utilisateurs dans le service

Programmation réactive

Programmation réactive - Introduction

La programmation réactive n’est pas quelque chose de fondamentalement nouveau. C’est une façon de construire une application avec des événements, et d’y réagir (d’où le nom). Les événements peuvent être combinés, filtrés, groupés, etc. en utilisant des fonctions comme map, filter, etc.

Windsurfeur.

Programmation réactive - Angular

Angular est construit sur de la programmation réactive, et nous utiliserons aussi cette technique pour certaines parties.

- Répondre à une requête HTTP.
- Lever un événement spécifique dans un de nos composants
- Gérer un changement de valeurs dans un de nos formulaires
- ...

WinDecision.

Programmation réactive - Flux

Dans la programmation réactive, toute donnée entrante sera dans un flux. Ces flux peuvent être écoutés, modifiés (filtrés, fusionnés, …), et même devenir un nouveau flux que l'on pourra aussi écouter. Cette technique permet d'obtenir des programmes faiblement couplés : **on n'a pas à soucier des conséquences de l'appel de méthode, on se contente de déclencher un événement, et toutes les parties de l'application intéressées réagiront en conséquence.**

WinDesign

Programmation réactive - Flux

Dans la programmation réactive, **tout est un flux**. Un flux est une séquence ordonnée d'événements. Ces événements représentent

- des valeurs (une nouvelle valeur !)
- des erreurs (quand il y en a)
- des terminaisons (fin du flux)

WinDecision.

Programmation réactive - Observer Pattern

Tous ces événements sont poussés par un producteur de données, vers un consommateur. Dans notre code on **s'abonne** (**subscribe**) à ces flux, i.e. définir un listener capable de gérer ces trois possibilités. Un tel **listener sera appelé un observer**, et **le flux, un observable**. Ces termes ont été définis il y longtemps, car ils constituent un design pattern bien connu : l'observer.

Programmation réactive - Observables

Les observables sont très similaires à des **tableaux**. Un tableau est une collection de valeurs, comme un observable. Un observable ajoute juste la notion de valeur reportée dans le temps : dans un tableau, toutes les valeurs sont disponibles immédiatement, dans un observable, les valeurs viendront plus tard, par exemple dans plusieurs minutes.

Windesign.

RxJS

La bibliothèque la plus populaire de programmation réactive dans l’écosystème JavaScript est RxJS. Et c’est celle choisie par Angular.

WinDecision.

Programmation réactive - Opérateurs

Tout observable, comme un tableau, peut être transformé avec des fonctions classiques :

- take(n) va piocher les n premiers éléments.
- map(fn) va appliquer la fonction fn sur chaque événement et retourner le résultat.
- filter(predicate) laissera passer les seuls événements qui répondent positivement au prédicat.
- reduce(fn) appliquera la fonction fn à chaque événement pour réduire le flux à une seule valeur unique.
- merge(s1, s2) fusionnera les deux flux.
- subscribe(fn) appliquera la fonction fn à chaque événement qu'elle reçoit.
- ...

WinDecision.

Exemple avec un tableau

Si tu as un tableau de nombres que tu veux tous multiplier par 2, filtrer ceux inférieurs à 5, et les afficher enfin, tu peux écrire :

```
[1, 2, 3, 4, 5]
 .map(x => x * 2)
 .filter(x => x > 5)
 .forEach(x => console.log(x)); // 6, 8, 10
```

WinDecision.

Observable - Exemple avec un tableau

RxJS nous permet de construire un observable à partir d'un tableau. Et nous pouvons ainsi faire exactement la même chose :

```
Observable.from([1, 2, 3, 4, 5])
  .map(x => x * 2)
  .filter(x => x > 5)
  .delay(5000)
  .subscribe(x => console.log(x));

// 6, 8, 10 (au bout de 5 secondes)
```

WinDecision.

Observable - Exemple avec un tableau

RxJS nous permet de construire un observable à partir d'un tableau. Et nous pouvons ainsi faire exactement la même chose :

```
Rx.Observable.from([1, 2, 3, 4, 5])
.map(x => x * 2)
.filter(x => x > 5)
.delay(5000)
.subscribe(
    x => console.log(x),
    (error) => console.log('Erreur'),
    () => console.log ('Fin du flux')
 );

// 6, 8, 10 puis Fin du flux (au bout de 5 secondes)
```

WinDecision.

Observable - Exemple avec un événement de saisie

Mais un observable est bien plus qu'une simple collection. C'est une collection asynchrone, dont les événements arrivent au cours du temps. Les événements dans le navigateur sont un bon exemple : ils arriveront au cours du temps, donc ils sont de bons candidats pour l'utilisation d'un observable. Voici un exemple avec jQuery :

```
const input = $('input');

Observable
 .fromEvent(input, 'keyup')
 .subscribe((key) => console.log('keyup! '));
```

WinDecision.

Observable - Création

Nous pouvons créer nos propres observables.

Observable.create reçoit une fonction qui émet des événements sur son paramètre observer. Ici, elle n'émet qu'un seul événement.

```
const observable =
  Observable.create((observer) => observer.next('hello'));

observable.subscribe((value) => console.log(value));
// logs "hello"
```

WinDecision.

Observable - Gestion d'erreur

Nous pouvons aussi traiter les erreurs. La méthode subscribe accepte un deuxième callback, consacré à la gestion des erreurs.

Ici, la méthode map lève une exception, donc le deuxième callback de la méthode subscribe va le tracer.

```
Observable.range(1, 5)
 .map(x => {
   if (x % 2 === 1) {
     throw new Error('something went wrong');
   } else {
     return x;
   }
 })
 .filter(x => x > 5)
 .subscribe(
     x => console.log(x),
     error => console.log(error)
    // something went wrong
   );
```

WinDecision.

Observable - EventEmitter

Angular utilise RxJS, et nous permet aussi de l'utiliser. Le framework propose un adaptateur autour de l'objet Observable : **EventEmitter**.

EventEmitter a donc également une méthode **subscribe**() pour réagir aux événements, et cette méthode reçoit également les trois paramètres pour réagir aux événements, aux erreurs et à la terminaison.

Un EventEmitter peut émettre un événement avec la méthode **emit().**

```
const emitter = new EventEmitter();

emitter.subscribe(
 value => console.log(value),
 error => console.log(error),
 () => console.log('done')
);

emitter.emit('hello');
emitter.emit('there');
emitter.complete();

// logs "hello", then "there", then "done"
```

WinDecision.

Workshop 7’

Observables
30mn

Au lieu de rafraîchir la liste des utilisateurs par appel explicite, la liste doit se mettre à jour de façon réactive.

Ajouter une méthode dans le service de gestion des users retournant un observable (EventEmitter) des utilisateurs auquel sera souscrit le composant des utilisateurs.

Http

Envoyer et recevoir des données par HTTP

Angular fournit un module http, mais ne te l'impose pas. Il est possible d'utiliser une bibliothèque HTTP différente pour faire des requêtes asynchrones.

Le module s'appelle **HttpClientModule**, pour l'utiliser il faut utiliser les classes du package **@angular/common/http**.

WinDecision.

Envoyer et recevoir des données par HTTP

```
import { NgModule } from '@angular/core';
import { BrowserModule } from
'@angular/platform-browser';
import { HttpClientModule } from
'@angular/common/http';

@NgModule({
 imports: [BrowserModule, HttpClientModule],
 declarations: [UserComponent],
 bootstrap: [UserComponent]
})
export class AppModule {
}
```

```
import { HttpClient } from '@angular/common/http';

@Injectable()
export class UserService {
 constructor(public http: HttpClient) { }
 list() {
   return this.http.get('/api/cars');
 }
}
```

WinDesign.

Envoyer et recevoir des données par HTTP

Par défaut, le service Http réalise des requêtes AJAX avec XMLHttpRequest.

Il propose plusieurs méthodes, correspondant au verbes HTTP communs :

- get
- post
- put
- delete
- patch
- head

```
@Component({
 selector: 'user-app',
 template: `<h1>User</h1>`
})
export class UserComponent {
 constructor(private http: HttpClient) {
 }
}
```

WinDecision.

Envoyer et recevoir des données par HTTP

L'appel à http.get retourne un Observable, on peut s'abonner à cet observable pour obtenir la réponse.

La réponse est un objet Response, avec quelques champs et méthodes bien pratiques.

On peut ainsi facilement accéder au code status, au headers, etc.

```
http.get(`http://localhost:8000/api/cars`)
 .subscribe(response: Response => {
    console.log(response.status); // logs 200
    console.log(response.headers); // logs []
 });
```

WinDecision.

Envoyer et recevoir des données par HTTP

Le corps de la réponse est la partie la plus intéressante. Mais pour y accéder, il faudra utiliser une méthode :

- text() si on s'attend à du texte;
- json() si on s'attend à un objet JSON, le parsing étant fait automatiquement

```
import { map } from 'rxjs/operators';

http.get(`http://localhost:8000/api/users`)
 .subscribe(response => {
   console.log(response.json());
 });

http.get<User[]>('http://localhost:8000/api/users')
.pipe(
   map(response => response),
   filter(response => user.age >= 12)
 )
.subscribe((users: User[]) => {
  this.users = users;
});
```

WinDecision.

Envoyer et recevoir des données par HTTP

Envoyer des données est aussi trivial. Il suffit d'appeler la méthode post(), avec l'URL et l'objet à poster :

```
http.post(
   'http://localhost:8000/api/users',
   {.....}
)
```

WinDecision.

Workshop 8

Http
1h30

Modifier le service Users afin de faire appel au service Http permettant ainsi d'effectuer les mêmes opérations sur le endpoint users fourni

- add(User: user): void
- update(User: user): void
- get(id: number): Observable<User>
- getAll(): Observable<Array<User>>

Les pipes

Les pipes - Utilité

Souvent, les données brutes n’ont pas la forme exacte que l’on voudrait afficher dans la vue. On a envie de les transformer, les filtrer, les tronquer, etc.

Un pipe peut être **utilisé dans le HTML**, **ou dans le code** applicatif.

WinDecision.

Les pipes - json

---

json est un pipe pas tellement utile en production, mais bien pratique pour le débug. Ce pipe applique simplement JSON.stringify() sur tes données. Si nous avons un tableau de users nommé users, et que l’on veut rapidement voir ce qu’il contient, on aurait pu écrire :

Les pipes - Utilisation dans le template

Pas de chance, cela affichera [object Object]…

Mais JsonPipe arrive à la rescousse. On peut l'utiliser dans n'importe quelle expression, au sein de l'HTML :

Et cela affichera la représentation JSON de l'objet :

```
<p>{{ users }}</p>

<p>{{ users | json }}</p>

<p>
  [{"name": "Salman"}, {"name": "Yahya"}]
</p>
```

WinDecision.

Les pipes - slice

Si l'on a envie d'afficher qu'un sous-ensemble d'une collection, slice est fait pour ça. Il fonctionne comme la méthode du même nom en JavaScript, et prend deux paramètres : un indice de départ et, éventuellement, un indice de fin. Pour passer un paramètre à un pipe, il faut lui ajouter un caractère : et le premier paramètre, puis éventuellement un autre : et le second argument, etc.

```
<p>{{ users | slice:0:2 | json }}</p>

<p>{{ 'Ma chaîne' | slice:0:5 }}</p>
```

WinDecision.

Les pipes - slice

On peut utiliser slice dans n'importe quelle expression, on peut aussi l'utiliser avec NgFor :

```
import { Component } from '@angular/core';

@Component({
 selector: 'app-users',
 template: `
   <div *ngFor="let user of users | slice:0:2">
       {{user.name}}
   </div>`
})
export class UsersComponent {
 users: Array<any> = [
   { name: 'User 1' },
   { name: 'User 2' },
   { name: 'User 3' }
 ];
}
```

WinDecision.

Les pipes - lowercase ("minuscule")

Pendant du précédent, ce pipe transforme une chaîne de caractères dans sa version minuscule

```
<p>{{ user.lastname | lowercase }}</p>
```

WinDecision.

Les pipes - uppercase ("majuscule")

---

ce pipe transforme une chaîne de caractères dans sa version MAJUSCULE

```
<p>{{ user.firstname | uppercase }}</p>
```

WinDecision.

Les pipes - titlecase ("titre")

Angular 4 ajoute un nouveau pipe titlecase. Celui-ci change la première lettre de chaque mot en majuscule

```
<p>{{ 'salman yahya' | titlecase }}</p>
<!-- will display 'Salman Yahya' -->
```

WinDecision.

Pipes - Créer ses propres pipes

On peut bien évidemment créer ses propres pipes. C’est parfois très utile. En AngularJS 1.x, on utilisait souvent des filtres customs.

```
<p>{{ 'salman yahya' | customPipe }}</p>
```

WinDecision.

Pipes - Créer ses propres pipes

Tout d'abord, nous devons créer une nouvelle classe. Elle doit **implémenter l'interface PipeTransform**, ce qui nous amène à écrire une méthode **tranform()**, celle qui fait tout le travail.

```
import { PipeTransform, Pipe } from '@angular/core';

@Pipe({ name: 'customPipe' })
export class CustomPipe implements PipeTransform {
 transform(value, args) {
   return value;
 }
}
```

WinDecision.

Pipes - Créer ses propres pipes

Maintenant, nous devons rendre disponible notre pipe dans l'application.

```
@NgModule({
 imports: [...],
 declarations: [...., CustomPipe],
 bootstrap: [....]
})
export class AppModule {
}
```

WinDecision.

Les pipes - Utilisation dans le code

```
import { Component } from '@angular/core';
// you need to import the pipe you want to
use
import { JsonPipe } from '@angular/common';
@Component({
 selector: 'app-users',
 template: `<p>{{formattedUser}}</p>`
})
```

```
export class UsersComponent {
 users: Array<any> =
   [
    { name: 'Adil' },
    { name: 'Issam' }
   ];
 formattedUser: string;
 constructor(customPipe: CustomPipe) {
   this.formattedUser = customPipe.transform(this.users);
 }
}
```

WinDecision.

Workshop 9

Pipes
30mn

Créer le pipe 'fromNow' afin d'afficher depuis n'importe quelle date l'information "depuis x jours" relative à la date sur laquelle le pipe a été appliqué

Le routage

Routeur

Le routeur d'Angular a un objectif simple : **permettre d'avoir des URLs compréhensibles qui reflètent l'état de notre application, et déterminer pour chaque URL quels composants initialiser et insérer dans la page**. Tout cela sans rafraîchir la page et sans lancer de requête auprès de notre serveur : c'est tout l'intérêt d'avoir une Single Page Application.

Routeur - Préparation des routes

Les Routes sont un **tableau d'objets, chacun d'eux étant une route.** Une configuration de route est en général une paire :

- **path** : quelle URL va déclencher la navigation ;
- **component** : quel composant sera initialisé et affiché.

La configuration est définie (idéalement) dans un fichier dédié généralement nommé app.routes.ts

```
import { Routes } from '@angular/router';
import { HomeComponent } from './home/home.component';

export const ROUTES: Routes = [
 { path: '', redirectTo: '/home', pathMatch: 'full' },
 { path: 'home', component: HomeComponent }
 { path: '**', component: NotFoundComponent }
];
```

Routeur - Configuration du module

C’est un module optionnel; il n’est donc pas inclus dans le noyau du framework.

Il faut l’**inclure dans le module racine** en précisant une configuration pour définir les associations entre les URLs et les composants.

```
import { RouterModule } from '@angular/router';
import { ROUTES } from './app.routes';
…
@NgModule({
 imports: [..., RouterModule.forRoot(ROUTES)],
 declarations: [...],
 bootstrap: [...]
})
export class AppModule {
}
```

WinDecision.

Routeur - Point d'entrée dans la vue

Pour qu'un composant soit inclus dans notre application, comme le HomeComponent de l'exemple ci-dessus, il faut utiliser un tag spécial dans le template du composant principal : **<router-outlet>.**

Quand on naviguera, tout restera en place (le header, le main, et le footer ici), et le composant correspondant à la route actuelle sera inséré à la suite de la directive router-outlet

```
<header>
 <nav>...</nav>
</header>
<main>
 <router-outlet></router-outlet>
 <!-- le template du composant se retrouvera ici -->
</main>
<footer> Copyright 2017 </footer>
```

WinDerision.

Routeur - Navigation depuis le template

Dans un template, on peut insérer un lien avec la directive RouterLink pointant sur le chemin où tu veux aller.

On peut utiliser cette directive car notre module racine importe RouterModule, rendant toutes les directives exportées par RouterModule disponibles dans le module racine.

```
<a href="" routerLink="/">Home</a>
<a href="" routerLink="/home">Admin</a>
```

WinDecision.

Routeur - Navigation dans le code

Il est aussi possible de naviguer depuis le code, en utilisant le service Router et sa méthode navigate(). C'est souvent pratique quand on veut rediriger notre utilisateur suite à une action :

```
export class UsersComponent  {
 constructor(private router: Router) {
 }

 saveAndMoveBackToHome () {
   this.router.navigate(['/home']);
 }
}
```

WinDecision.

Routeur - Récupération des paramètres

Il est également possible d'avoir des paramètres dans l'URL, et c'est très pratique pour définir des URLs dynamiques.

Par exemple, on pourrait afficher une page de détail pour un user, et cette page aurait une URL significative comme users/id-du-user/le-nom-du-user.

```
export const routes: Routes = [
 { path: '', component: HomeComponent },
 { path: 'users', component: UsersComponent },
 {
   path: 'users/:userId',
   component: UserComponent
 }
];
```

WinDecision.

Routeur - Récupération des paramètres

On peut alors définir des liens dynamiques avec routerLink :

```
<a href=""
[routerLink]="['/users', user.id]">
 {{user.name}}
</a>
```

WinDecision.

Routeur - Récupération des paramètres

On peut récupérer ces paramètres assez facilement dans le composant cible.

Ici, grâce à l'injection de dépendances, notre composant UserComponent reçoit un objet de type **ActivatedRoute**.

Cet objet peut être utilisé dans **ngOnInit**, et a un champ bien pratique : snapshot. Ce champ contient tous les paramètres de l'URL dans paramMap !

```
export class UserComponent implements OnInit {
 user: User;

 constructor(private userService: UserService,
             private route: ActivatedRoute) {
 }

 ngOnInit() {
   const id = this.route.snapshot.paramMap.get('userId');
   this.userService
       .get(id)
       .subscribe(user => this.user = user);
 }
}
```

WinDecision.

Workshop 10

Routage
1h

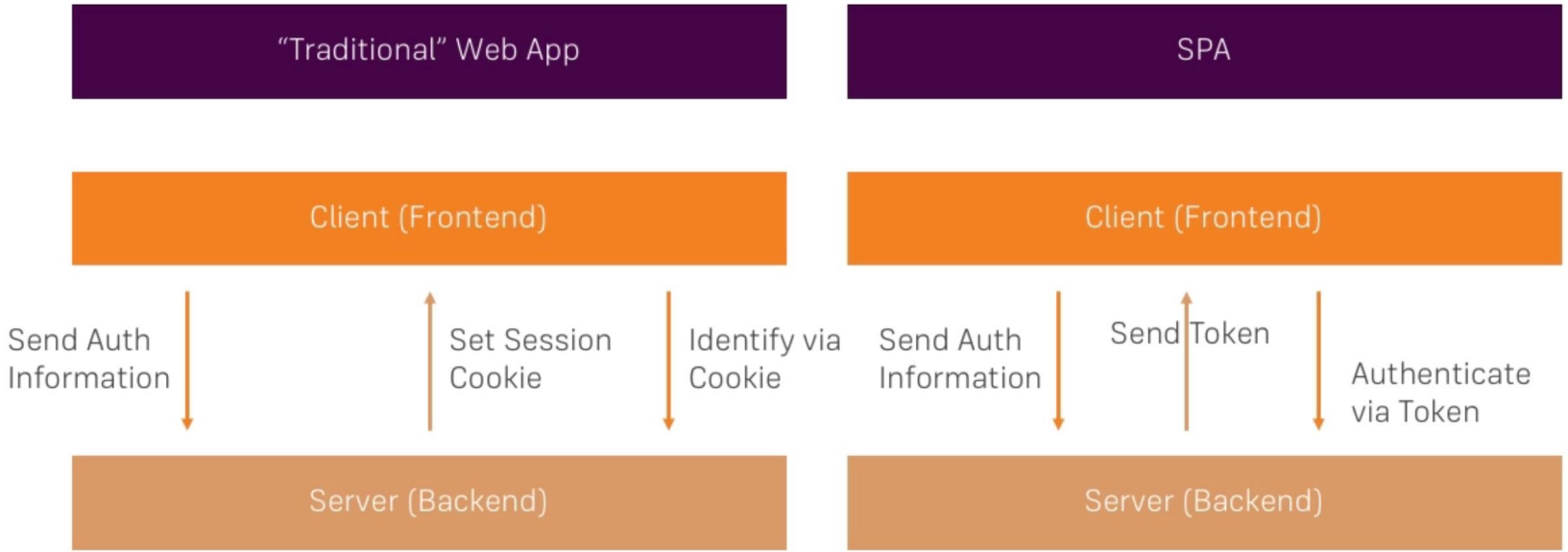
L’application a désormais deux pages: une vue d’admin d’ajout/listings d’utilisateurs et une nouvelle vue dashboard affichant le nombre d’utilisateurs.

Ajouter une navbar (header) pour basculer entre les différentes vues

La vue de listing est affichée par défaut

Authentification

Workflow d’authentification



Workshop 11

Authentification
1h

Créer un service AuthService **bouchonnant** les appels serveurs d'authentification sur l'application. Le service expose les méthodes suivantes:

- login(user, passwd): void
- isAuthenticated(): boolean
- logout(): void
- getToken(): string

Au chargement de l'application, par défaut nous arrivons sur la page d'authentification (Formulaire: login, mdp), une fois authentifié, l'utilisateur est redirigé vers la vue Admin

Contrôle de routes

Guards

Certaines routes de l’application ne devraient pas être accessibles à tous. L’utilisateur est-il authentifié ? A-t-il les permissions nécessaires ? Bien sûr, il faut désactiver ou masquer les liens pointant sur ces routes inaccessibles. Le backend doit aussi empêcher l’accès aux ressources interdites à l’utilisateur courant. Mais cela n’empêchera pas l’utilisateur d’accéder aux routes qui lui sont interdites, simplement en entrant leur URL dans la barre d’adresse du navigateur.

Guards

Le Guard doit être appliqué sur une route donnée. Ceci est indiqué en configuration de la route.

```
export const routes: Routes = [
 {
   path: 'admin',
   component: AdminComponent,
   canActivate: [LoggedInGuard]
  }
];
```

WinDecision.

Guards

C’est là que le guard CanActivate intervient, lorsqu’un tel guard est appliqué à une route, il peut empêcher l’activation de la route.

```
// imports..
import { CanActivate } from '@angular/router';
@Injectable()
export class LoggedInGuard implements CanActivate {
 constructor(private router: Router,
             private userService: UserService) {}
 canActivate(route: ActivatedRouteSnapshot,
             state: RouterStateSnapshot) {
   const loggedIn = this.userService.isLoggedIn();
   if (!loggedIn) {
     this.router.navigate(['/login']);
   }
   return loggedIn;
 }
}
```

WinDesign.

Resolvers

Pour éviter un effet de flickering, on peut souhaiter de ne charger une route que quand les données ont été chargées

```
export const AppRoutes: Routes = [
 ...
 {
   path: 'user/:id',
   component: UserDetailComponent,
   resolve: {
     user: UserResolve
   }
 }
];
```

WinDecision.

Resolvers

```
@NgModule({
 ...
 providers: [
   UsersService,
   UserResolve
 ]
})
export class AppModule {}
```

```
// imports...
import { ActivatedRouteSnapshot } from '@angular/router';

@Injectable()
export class UserResolve implements Resolve<Observable<User>> {

 constructor(private usersService: UsersService) {}

 resolve(route: ActivatedRouteSnapshot): Observable<User> {
   return this.usersService.getUser(route.paramMap.get('id'));
 }
}
```

WinDecision.

Resolvers

Afin d'accéder à la donnée qui a été récupérée via mon resolver, j'utilise le service route injecté depuis mon constructeur.

```
@Component()
export class UserDetailComponent implements OnInit {

 user;

 constructor(private route: ActivatedRoute) {}

 ngOnInit() {
   this.user = this.route.snapshot.data['user'];
 }
}
```

WinDecision

Workshop 12

Guards & Resolve
30mn

- Modifier la route admin pour que seuls les utilisateurs authentifiés puissent y accéder
- La liste des utilisateurs doit être affichée avant le chargement de la route afin d’éviter l’effet de flickering

Custom Directives

Un composant sans template

Une directive est très semblable à un composant, sauf qu'elle n'a pas de template. En fait, **la classe Component hérite de la classe Directive** dans le framework.

Comme pour un composant, la directive sera annotée d'un décorateur, mais au lieu de @Component, ce sera **@Directive**.

Les directives sont des briques minimalistes. On peut les concevoir comme des décorateurs pour notre HTML : elles **attacheront du comportement aux éléments du DOM**. On peut avoir **plusieurs directives appliquées à un même élément**.

Une directive **doit avoir un sélecteur CSS**, qui indique au framework où l'activer dans notre template.

WinDecision.

Directive - Définition

Voici une directive très simple qui ne fait rien mais est activée si un élément possède l'attribut doNothing

```
@Directive({
   selector: '[doNothing]'
})
export class DoNothingDirective {

   constructor() {
        console.log('Do nothing directive');
   }
}
```

```
<div doNothing> </div>
```

WinDecision.

Directive - Sélecteur

Voici une directive avec un sélecteur plus complexe. La directive s'appliquer sur toutes les divs de mon application qui répondent au sélecteur.

```
@Directive({
   selector: 'div.loggable[logText]'
})
export class ComplexSelectorDirective {

   constructor() {
       console.log('Complex selector directive');
   }
}
```

```
<div class="loggable"
     logText="text">Hello</div>
```

WinDecision.

Directive - inputs

A l'image des composants, les inputs d'une directives sont définis à travers le décorateur @Input()

```
@Directive({
   selector: '[loggable]'
})
export class InputDecoratorDirective {

   @Input('logText') text: string;
}
```

WinDecision.

Directive - inputs

Nous pouvons également utiliser la propriété inputs du décorateur.

Le guide de style officiel indique de **préférer plutôt @Input()**.

```
@Directive({
    selector: '[loggable]',
    inputs: ['text: logText']
})
export class SimpleTextWithSetterDirective {
    // Optionnel
    // Utile si l'on veut faire un traitement spécifique
    set text(value) {
        console.log(value);
    }
}
```

WinDecision.

Directives - ElementRef et Renderer2

ElementRef permet de récupérer l'élément du DOM dans notre classe.

Renderer2 permet de modifier le style du composant de façon **portable.**

```
import { Directive, Renderer2, ElementRef } from
'@angular/core';

@Directive({
   selector: '[appHighlight]'
})
export class HighlightDirective {

   constructor(el: ElementRef, renderer: Renderer2) {
       renderer.setStyle(el.nativeElement, 'color', 'red');
   }

}
```

WinDecision.

Directives - Événements JavaScript

Angular propose le décorateur @HostListener pour positionner des Listeners sur l'élément.

Il prend en paramètre l'événement que l'on veut écouter. Notre classe devient alors :

```
import { Directive, Renderer2, ElementRef, HostListener }
from '@angular/core';
@Directive({
   selector: '[appHighlight]'
})
export class HighlightDirective {
 constructor(el: ElementRef, renderer: Renderer2) {
   renderer.setStyle(el.nativeElement, 'color', 'red');
 }
 @HostListener('mouseenter', ['$event'])
 onMouseEnter(event: Event) {
     console.log('mouseenter');
     console.log(event);
 }
}
```

WinDesign.

Directives - Host binding

Angular propose le décorateur @HostBinding permettant d'agir sur le host par binding systématiquement quand l'attribut bindé est modifié.

```
import { Directive, HostBinding, HostListener } from
'@angular/core';
@Directive({
   selector: '[appButtonPress]'
})
export class ButtonPressDirective {
   @HostBinding('class.pressed') isPressed: boolean;

   @HostListener('mousedown') hasPressed() {
       this.isPressed = true;
   }
   @HostListener('mouseup') hasReleased() {
       this.isPressed = false;
   }
}
```

WinDesign.

Directives - ExportAs

Il est possible d'accéder à l'instance de la directive à partir du template appelant. C'est de cette façon que fonctionne le:

#form = "ngForm"

#input = "ngModel"

```
import { Directive, HostBinding, HostListener } from
'@angular/core';
@Directive({
   selector: '[appButtonPress]',
   exportAs: 'appButton'
})
export class ButtonPressDirective {
   public message = 'hello';
}

<div appButtonPress #app="appButton">
   {{app.message}}
</div>
```

WinDecision.

Directives de validation

Nous souhaitons définir une directive spécifique qui, ajoutée à un input, permet d'y appliquer la validation à l'image de: email, required..

```
<input id="login"
       name="login"
       required minlength="4"
       forbiddenLogin="admin"
       [(ngModel)]="user.login" #name="ngModel" />
```

WinDecision.

Directives de validation

```
@Directive({
   selector: '[forbiddenLogin]',
   providers: [{ provide: NG_VALIDATORS, useExisting: MyValidationDirective, multi: true }]
})
export class MyValidationDirective implements Validator {
   @Input() forbiddenLogin: string;

   validate(control: AbstractControl): { [key: string]: any } {
       return this.forbiddenLogin ? callValidationFunctionOn(control): null;
   }
}
```

WinDecision.

Directives structurelles

Les directives qu'on a définies actuellement sont des directives qui agissent sur les propriétés / aux événements d'un élément existant sur le DOM.

Si nous souhaitons contrôler l'ajout / suppression d'un élément du DOM, nous devons utiliser des directives structurelles de type *ngIf, *ngFor, *ngSwitch…

WinDecision.

Directives structurelles - Template

Nous devons passer par un élément template spécifique qui ne s'affiche pas mais dont on souhaite contrôler l'ajout/suppression depuis le code de notre directive

```
<ng-template>
   <div>
      Hello
   </div>
</ng-template>
```

WinDecision.

Directives structurelles - appIf = ngIf custom

```
<ng-template [appIf]="isAuth">
   <div>
       Hello
   </div>
</ng-template>
```

```
import { Directive, Input, TemplateRef, ViewContainerRef } from '@angular/core';
@Directive({
  selector: '[appIf]'
})
export class IfDirective {
   constructor(private templateRef: TemplateRef<any>,
       private viewContainer: ViewContainerRef) {
   }
   @Input()
   set appIf(shouldAdd: boolean) {
       if (shouldAdd) {
          this.viewContainer.createEmbeddedView(this.templateRef);
       } else {
           this.viewContainer.clear();
       }
   }
}
```

WinDecision.

Directives structurelles - *appIf

```
<ng-template [appIf]="isAuth">
   <div>
       Hello
   </div>
</ng-template>

// Equivalent à
<div *appIf="isAuth">
   Hello
</div>
```

```
import { Directive, Input, TemplateRef, ViewContainerRef } from '@angular/core';
@Directive({
  selector: '[appIf]'
})
export class IfDirective {
   constructor(private templateRef: TemplateRef<any>,
       private viewContainer: ViewContainerRef) {
   }
   @Input()
   set appIf(shouldAdd: boolean) {
       if (shouldAdd) {
           this.viewContainer.createEmbeddedView(this.templateRef);
       } else {
           this.viewContainer.clear();
       }
   }
}
```

WinDecision.

Directives - Création via Angular Cli

- ng generate directive highlight
- ng g d **highlight**

Angular CLI se chargera de faire

la création/insertion dans le module

```
import { Directive } from '@angular/core';

@Directive({
   selector: '[appHightlight]'
})
export class HightlightDirective {
   constructor() { }
}
```

WinDecision.

Workshop 13

Custom directives
30mn

Au choix:

- Créer une directive spécifique applicable sur tous les champs input text permettant de mettre à vide le placeholder quand le curseur passe sur l'input.
- Créer une nouvelle directive de validation email validant qu'un email a le format x@y.com

Les formulaires

Formulaires

La gestion des formulaires a toujours été super soignée en Angular. C'était une des fonctionnalités les plus mises en avant en 1.x, et, comme toute application inclut en général des formulaires, elle a gagné le cœur de beaucoup de développeurs.

Les formulaires, c'est compliqué : il faut **valider les saisies** de l'utilisateur, **afficher les erreurs** correspondantes, on peut avoir des **champs obligatoires ou non**, ou qui **dépendent d'un autre champ**, on peut pouvoir **réagir sur les changements** de certains, etc.

WinDecision.

Formulaires - Template

On peut écrire son formulaire en utilisant seulement des directives dans le template : c’est la façon **"pilotée par le template"**. C’est particulièrement utile pour des **formulaires simples**, sans trop de validation.

WinDecision.

Formulaires - Réactifs

L’autre façon de procéder est la façon **"pilotée par le code"**, où l’on écrit une description du formulaire dans le composant, **puis utilisons ensuite des directives** pour lier ce formulaire aux inputs/textareas/selects du template.

C’est plus verbeux, mais aussi plus puissant, notamment pour faire de la **validation custom**, ou pour générer des formulaires dynamiquement.

Formulaires - Template - ngForm

Dans cette méthode, **on va mettre en œuvre un paquet de directives dans notre formulaire, et laisser le framework construire les instances nécessaires**. Ces instances nous seront utiles pour contrôler notre formulaire.

Toutes ces directives sont incluses dans le module **FormsModule**, nous devons donc l'importer dans notre module racine.

```
…
import { FormsModule } from '@angular/forms';
@NgModule({
 ...
 imports: [..., FormsModule],
})
export class AppModule {}
------
@Component({
 selector: 'app-register',
 template: `
 <h2>Sign up</h2>
 <form></form>
 `
})
export class RegisterFormComponent {}
```

WinDecision.

Formulaires - Template - ngSubmit

J'ai ajouté un bouton, et défini un **handler d'événement ngSubmit** sur l'élément <form>. L'événement ngSubmit est émis lors de la soumission du formulaire. Cela invoquera la méthode register() de notre composant, qu'on implémentera plus tard.

```
import { Component } from '@angular/core';

@Component({
 selector: 'app-register',
 template: `
 <h2>Sign up</h2>
 <form (ngSubmit)="register()">
  <button type="submit">Register</button>
 </form>
 `
})
export class RegisterFormComponent {
 register() {
   // we will have to handle the submission
 }
}
```

Formulaires - NgModel

Nous avons déjà vu la directive NgModel.

Chaque fois qu'on tapera quelque chose dans le champ, le modèle sera mis à jour. Et si le modèle est mis à jour dans notre composant, le champ affichera automatiquement la nouvelle valeur

```
<h2>Sign up</h2>
<form (ngSubmit)="register()">
 <div>
   <label>Username</label>
   <input name="username" [(ngModel)]="user.username">
   <small>{{ user.username }}</small>
 </div>
 <div>
   <label>Password</label>
   <input type="password"
          name="password" [(ngModel)]="user.password">
 </div>
 <button type="submit">Register</button>
</form>
```

WinDesign.

Databinding - Two-way databinding

La directive NgModel met à jour la valeur de l'input à chaque changement du modèle lié user.username, d'où la partie **[ngModel]="user.name"**.

La directive génère un événement depuis un output nommé ngModelChange à chaque fois que l'input est modifié par l'utilisateur, où l'événement est la nouvelle valeur, d'où la partie **(ngModelChange)="user.name = $event"**, qui met donc à jour le modèle user.username avec la valeur saisie.

```
<input type="text" [(ngModel)]="user.name" />
{{ user.name }}

<input type="text"
    [ngModel]="user.name"
    (ngModelChange)="user.name = $event" />
```

WinDecision.

Formulaires - FormGroup et FormControl

La directive **ngModel crée le FormControl** pour nous, et **le <form> crée automatiquement le FormGroup.**

On doit systématiquement donner un name à l'input, qui sera utilisé par le framework pour construire le FormGroup composé de FormControl nommés.

```
<h2>Sign up</h2>
<form (ngSubmit)="register()">
 <div>
   <label>Username</label>
   <input name="username" [(ngModel)]="user.username">
   <small>{{ user.username }}</small>
 </div>
 <div>
   <label>Password</label>
   <input type="password"
          name="password" [(ngModel)]="user.password">
 </div>
 <button type="submit">Register</button>
</form>
```

Formulaires - Template - Validation

La validation de données est traditionnellement une partie importante de la construction de formulaire. Certains champs sont obligatoires, certains dépendent d'autres, certains doivent respecter un format spécifique….

Commençons par ajouter quelques règles basiques : **tous nos champs sont obligatoires**.

```
<h2>Sign up</h2>
<form (ngSubmit)="register()">
 <div>
   <label>Username</label>
   <input name="username"
          required minlength="3"
          [(ngModel)]="user.username">
   <small>{{ user.username }}</small>
 </div>
 <div>
   <label>Password</label>
   <input type="password" required
          name="password" [(ngModel)]="user.password">
 </div>
 <button type="submit">Register</button>
</form>
```

Formulaires - Template - FormControl

Un FormControl a plusieurs attributs, dont :

- **valid** : si le champ est valide
- invalid : si le champ est invalide
- **pristine** : l'opposé de dirty.
- dirty : false jusqu'à ce que l'utilisateur modifie la valeur du champ.
- **untouched** : l'opposé de touched.
- touched : false jusqu'à ce que l'utilisateur soit entré dans le champ.
- value : la valeur du champ.
- errors : un objet contenant les erreurs du champ.

```
<h2>Sign up</h2>
<form (ngSubmit)="register()">
 <div>
   <label>Username</label>
   <input name="username" [(ngModel)]="user.username">
   <small>{{ user.username }}</small>
 </div>
 <div>
   <label>Password</label>
   <input type="password"
          name="password" [(ngModel)]="user.password">
 </div>
 <button type="submit">Register</button>
</form>
```

Formulaires - Template - FormControl (required)

pristine	untouched	valid
dirty	touched	invalid

Formulaires - Template - FormControl (required)

pristine	untouched	valid
dirty	touched	invalid

Formulaires - Template - FormControl (required)

pristine	untouched	valid
dirty	touched	invalid

Formulaires - Template - FormControl (required)

pristine	untouched	valid
dirty	touched	invalid

Formulaires - Template - FormControl (required)

pristine	untouched	valid
dirty	touched	invalid

Formulaires - Template - FormControl (required)

pristine	untouched	valid
dirty	touched	invalid

Formulaires - Template - Erreurs de soumission

Maintenant il nous faut afficher les erreurs sur chaque champ.

Comme le formulaire, chaque contrôle expose son objet FormControl, on peut donc créer une variable locale pour accéder aux erreurs :

```
<form (ngSubmit)="register()">
   <div class="form-group">
       <label for="firstname">First Name</label>
       <input required name="firstname"
         type="text" id="firstname"
          [(ngModel)]="user.firstname"
         #firstname="ngModel">
       <div *ngIf="firstname.dirty
            && firstname.hasError('required')">
          First name is required
       </div>
   </div>
   <button type="submit">Register
   </button>
</form>
```

Formulaires - Template - FormGroup

Un FormGroup a les mêmes propriétés qu'un FormControl, le binding de ses propriétés se fait cependant sur des vérifications sur l'ensemble des ses FormControl

```
<h2>Sign up</h2>
<form (ngSubmit)="register()">
 <div>
   <label>Username</label>
   <input name="username" [(ngModel)]="user.username">
   <small>{{ user.username }}</small>
 </div>
 <div>
   <label>Password</label>
   <input type="password"
          name="password" [(ngModel)]="user.password">
 </div>
 <button type="submit">Register</button>
</form>
```

WinDesign.

Formulaires - Template - Erreurs de soumission

Evidemment, on veut que l'utilisateur ne puisse pas soumettre le formulaire tant qu'il reste des erreurs, et ces erreurs doivent être parfaitement affichées.

```
<form (ngSubmit)="register()" #userForm="ngForm">
 <div>
   <label>Username</label>
   <input name="username"
          required minlength="3"
          [(ngModel)]="user.username">
   <small>{{ user.username }}</small>
 </div>
 <div>
   <label>Password</label>
   <input type="password" required
          name="password" [(ngModel)]="user.password">
 </div>
 <button type="submit"
    [disabled]="userForm.invalid">Register
 </button>
</form>
```

Formulaires - Template - ngModelGroup

La directive ngModelGroup permet de grouper la validation de champs à l'intérieur du formulaire.

Ici nous affichons "Name is invalid" tant que le nom ou le prenom n'ont pas été saisis.

```
<form>
   <p *ngIf="nameCtrl.invalid">Name is invalid.</p>

   <div ngModelGroup="name" #nameCtrl="ngModelGroup">
      <input name="first" [ngModel]="name.first" required>
      <input name="last" [ngModel]="name.last" required>
   </div>

   <button>Submit</button>
</form>
```

WinDecision

Formulaires - Template - Style

Angular réalise une autre tâche bien pratique pour nous : il **ajoute et enlève automatiquement certaines classes CSS sur chaque champ** (et le formulaire) pour nous permettre d'affiner le style visuel.

Par exemple, **un champ aura la classe ng-invalid si un de ses validateurs échoue, ou ng-valid** si tous ses validateurs passent.

```
<style>
 input.ng-invalid {
   border: 3px red solid;
 }
</style>
<form (ngSubmit)="register()" #userForm="ngForm">
   <div class="form-group">
       <label for="firstname">First Name</label>
       <input name="username"
              required minlength="3"
              [(ngModel)]="user.username">
   </div>
   <button type="submit"
           [disabled]="userForm.invalid">Register
   </button>
</form>
```

Formulaires - Template - Style

Quand un formulaire est affiché pour la première fois, un champ portera généralement les classes CSS ng-pristine ng-untouched ng-invalid. Ensuite, quand l'utilisateur sera rentré puis sorti du champ, elle basculeront à ng-pristine ng-touched ng-invalid. Quand l'utilisateur modifiera la valeur, toujours invalide, on aura ng-dirty ng-touched ng-invalid. Et enfin, quand la valeur deviendra valide : ng-dirty ng-touched ng-valid.

```
<style>
 input.ng-invalid {
   border: 3px red solid;
 }
 input.ng-touched {
   border: 3px blue solid;
 }
</style>
<form (ngSubmit)="register()" #userForm="ngForm">
   <div class="form-group">
       <label for="firstname">First Name</label>
       ....
   </div>
   <button type="submit"
           [disabled]="userForm.invalid">Register
   </button>
</form>
```

Workshop 14

Formulaires
1h

La soumission de la création d’un utilisateur ne doit désormais être possible que si:

- Tous les champs sont saisis
- Les noms et prénoms ne comportent que des caractères alphabétiques
- L’email a un format valide

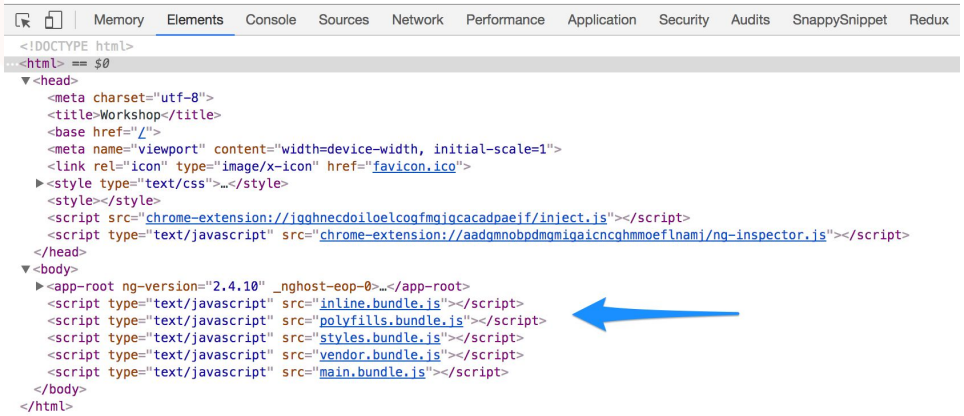
Si un champ est mal saisi, Une erreur doit être indiquée en rouge bas du champ puis disparaître quand le champ a été corrigé.

Déploiement

Application Bootstrap - index.html

```
<!doctype html>
<html>
<head>
 <meta charset="utf-8">
 <title>Angular</title>
 <base href="/">

 <meta name="viewport" ... >
 <link rel="icon" ....>
</head>
<body>
 <app-root>Loading...</app-root>
</body>
</html>
```

app works!



Compilation Angular

Un application Angular est composée majoritairement de composants incluant leur template HTML. Avant que le navigateur puisse afficher (render) l'application, les composants ainsi que leur templates doivent être compilés / convertis en code JavaScript via le compilateur Angular.

JIT and AOT

Angular propose deux façons de compiler l’application

- Just-in-Time (JIT), qui compile l’application dans le browser durant l’exécution
- Ahead-Of-Time (AOT), qui compile l’application durant l’étape build avant exécution dans le browser (ou autre).

WinDecision.

Pourquoi AOT ?

- Faster rendering: JavaScript exécutable
- Fewer asynchronous requests: Inline de l'HTML et de la CSS
- Smaller Angular framework download size: Le compilateur n'est pas fourni au browser
- Detect template errors earlier: Les erreurs de compilation sont détectés avant exécution
- Better security: Moins de risque d'injection de code malicieux car l'html est préalablement compilé

WinDecision

JIT par défaut

Par défaut quand nous exécutons les commandes de build ou serve via CLI sans arguments, la compilation JIT est celle utilisée.

```
$ ng build

$ ng serve
```

WinDecision.

AOT

Pour activer la compilation AOT, il faut passer l'argument --aot.

```
$ ng build --aot

$ ng serve --aot
```

WinDecision.

Build

```
$ ng build
```

Si sous-répertoire / contexte

```
$ ng build --base-href /myContext/
```

=> <base href="/myContext"> ajouté à index.html

```
dist/
0.ceb53d0e1107e4418bc2.js
1.36e35a9d0e1ea5c9bba6.js
3rdpartylicenses.txt
favicon.ico
...woff2
index.html
main.8b5beff96921e1fba499.js
polyfills.2c0f5aebae395e43bc64.js
runtime.ed5a604bb379807fa2ac.js
styles.2cfdf3b5c0df44f2527d.css
```

Optimisation pour production

`ng build --prod` permet d'optimiser le bundle produit.

- Ahead-of-Time (AOT) Compilation
- Production mode (Configuration de l'environnement de prod)
- Bundling: Concaténation des fichiers dans quelques bundles.
- Minification: Suppression des espaces, commentaires.
- Uglification: réectiture des noms de variables et fonctions.
- Dead code elimination: Suppression de code mort (non utilisé / atteignable)

WinDecision.

Apache - Déploiement

Le contenu du répertoire dist représente le livrable à déposer dans le répertoire apache.

dist/
0.ceb53d0e1107e4418bc2.js
1.36e35a9d0e1ea5c9bba6.js
3rdpartylicenses.txt
favicon.ico
...woff2
index.html
main.8b5beff96921e1fba499.js
polyfills.2c0f5aebae395e43bc64.js
runtime.ed5a604bb379807fa2ac.js
styles.2cfdf3b5c0df44f2527d.css

WinDesign.

Apache - Configuration

Si l’asset existe le servir
sinon rediriger vers index.html

```
<Directory "/var/www/html/webapp">
      Require all granted
      RewriteEngine On
      RewriteCond %{DOCUMENT_ROOT}%{REQUEST_URI} -f [OR]
      RewriteCond %{DOCUMENT_ROOT}%{REQUEST_URI} -d
      RewriteRule ^ - [L]
      RewriteRule ^ /en/index.html
</Directory>
```

WinDecision.

Spring Boot - Déploiement

Il existe deux approches de déploiement dans Spring Boot

- Copier manuellement le répertoire dist folder dans /src/main/resources/static du projet.
- Utiliser Maven plugin maven-resources-plugin afin de copier tous les fichiers depuis le répertoire dist dans /src/main/resources/static du projet.
- Utiliser maven-war-plugin en indiquant le répertoire dist comme valeur de la propriété warSourceDirectory.

WinDecision.

Spring Boot - Configuration

```
@SpringBootApplication
public class Application extends SpringBootServletInitializer  {
   @Bean
   public EmbeddedServletContainerCustomizer
     containerCustomizer (){
        return new MyWebApp ();
   }

 private static class MyWebApp implements
        EmbeddedServletContainerCustomizer  {
  @Override
  public void customize (ConfigurableEmbeddedServletContainer
container){
   container.addErrorPages (new ErrorPage (HttpStatus.NOT_FOUND, "/"));
    }
   }
}
```

```
@Override
public void configure(WebSecurity web) {
    web.ignoring()
        .antMatchers(HttpMethod.OPTIONS, "/**")
        .antMatchers("/app/**/*.{js,html}")
        .antMatchers("/i18n/**")
        .antMatchers("/content/**")
        .antMatchers("/redoc/index.html")
        .antMatchers("/swagger-ui/index.html")
        .antMatchers("/test/**")
        .antMatchers("/h2-console/**");
}
```

WinDecision.

Spring Boot - Valve Tomcat

Servir tous les assets: css, img, js….

Tout le reste est redirigé vers index.html

```
RewriteCond %{REQUEST_URI}
^/(css|img|js|partials|api|favicon).*$
RewriteRule ^.*$ - [L]

RewriteRule ^.*$ /index.html [L,QSA]
```

WinDecision.

Workshop 15

Déploiement
1h

Déployer votre application sur votre serveur web/d'application favori:

- Apache / Nginx
- Tomcat / Netty

Ressources

Ressources - Liens

- Framework
  - http://angular.io
- Blogs
  - https://blog.thoughtram.io/
  - http://blogs.msmvps.com/deborahk/
  - https://toddmotto.com/
  - http://www.bennadel.com/
- Vidéos
  - http://egghead.io
  - https://angularclass.com/
- Modules
  - https://material.angular.io/
  - https://ng-bootstrap.github.io/
  - https://www.nativescript.org
- Autres
  - https://www.madewithangular.com
  - https://github.com/jmcunningham/AngularJS2-Learning
  - https://stackblitz.com/

WinDecision.