Angular

Framework Javascript

Ressources utiles

Documentations et liens

Documentations officielles

Peu digestes, mais existantes

HTML

https://html.spec.whatwg.org/

JS/Ecma/JSON

https://ecma-international.org/technical-committees/tc39/?tab=published-standards

CSS

https://www.w3.org/Style/CSS/Overview.en.html

Typescript

https://www.typescriptlang.org/docs/

Angular pre-19

https://v17.angular.io/docs

Angular 19+

https://angular.dev/

ATTENTION AUX FAUX AMIS

Ressources utiles

- MDN
  - HTML : https://developer.mozilla.org/en-US/docs/Web/HTML
  - CSS : https://developer.mozilla.org/fr/docs/Web/CSS
  - Javascript : https://developer.mozilla.org/en-US/docs/Web/JavaScript
- Zestes de savoir
  - https://zestedesavoir.com/tutoriels/3577/apprendre-le-javascript-moderne-en-creant-une-to-do-list-de-a-a-z/

- Attention aux faux amis comme W3Schools
  - Aucun rapport avec le W3C
  - Ressources souvent dépassées
  - « Fonctionne » plus ou moins avec des erreurs et mauvaises conceptions
- Si usage de Copilot ou consorts
  - Vérifiez toujours vos résultats
  - Si vous ne comprenez pas, n'utilisez pas
    - *Comprenez d'abord*
    - *Recherchez*
    - *Puis utilisez*

Rappels

HTML, CSS, Javascript

HTML

Hypertext Markup Language

Langage à portée sémantique

« sens »

N’a pas pour objectif de:

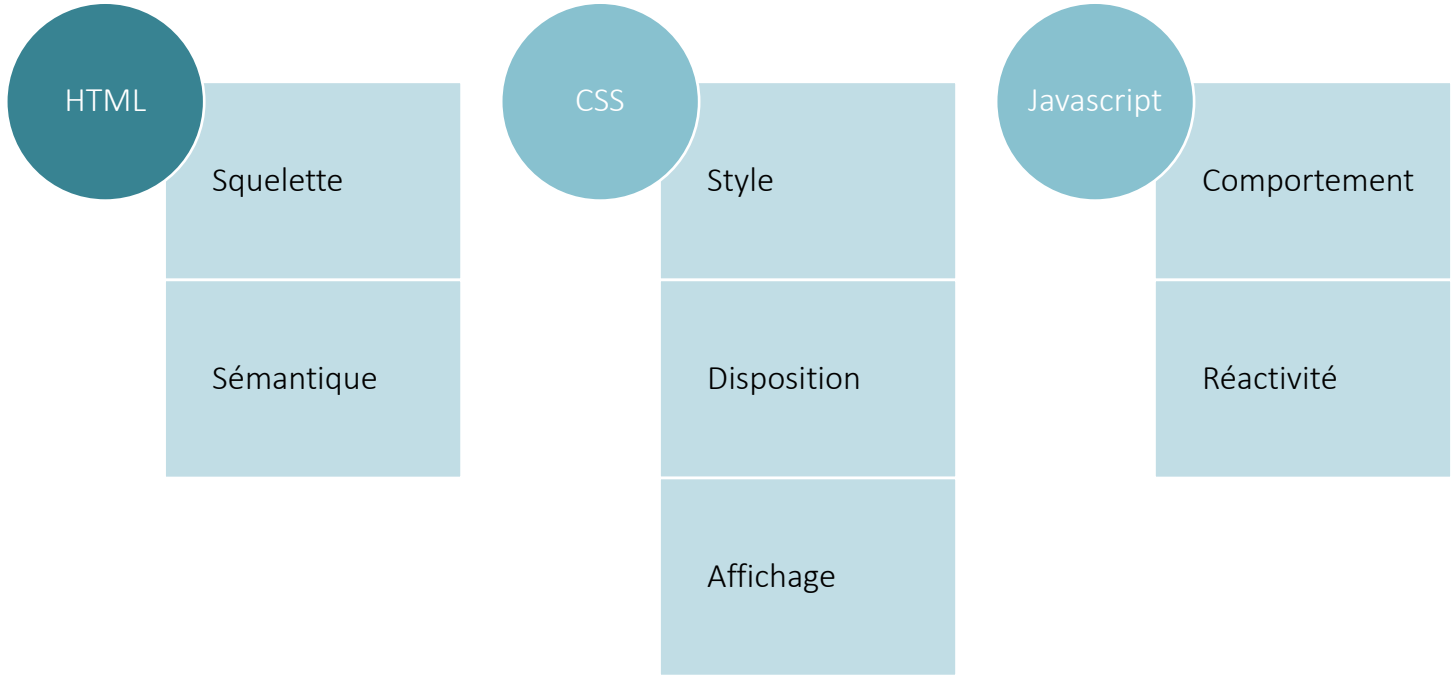
Style

CSS

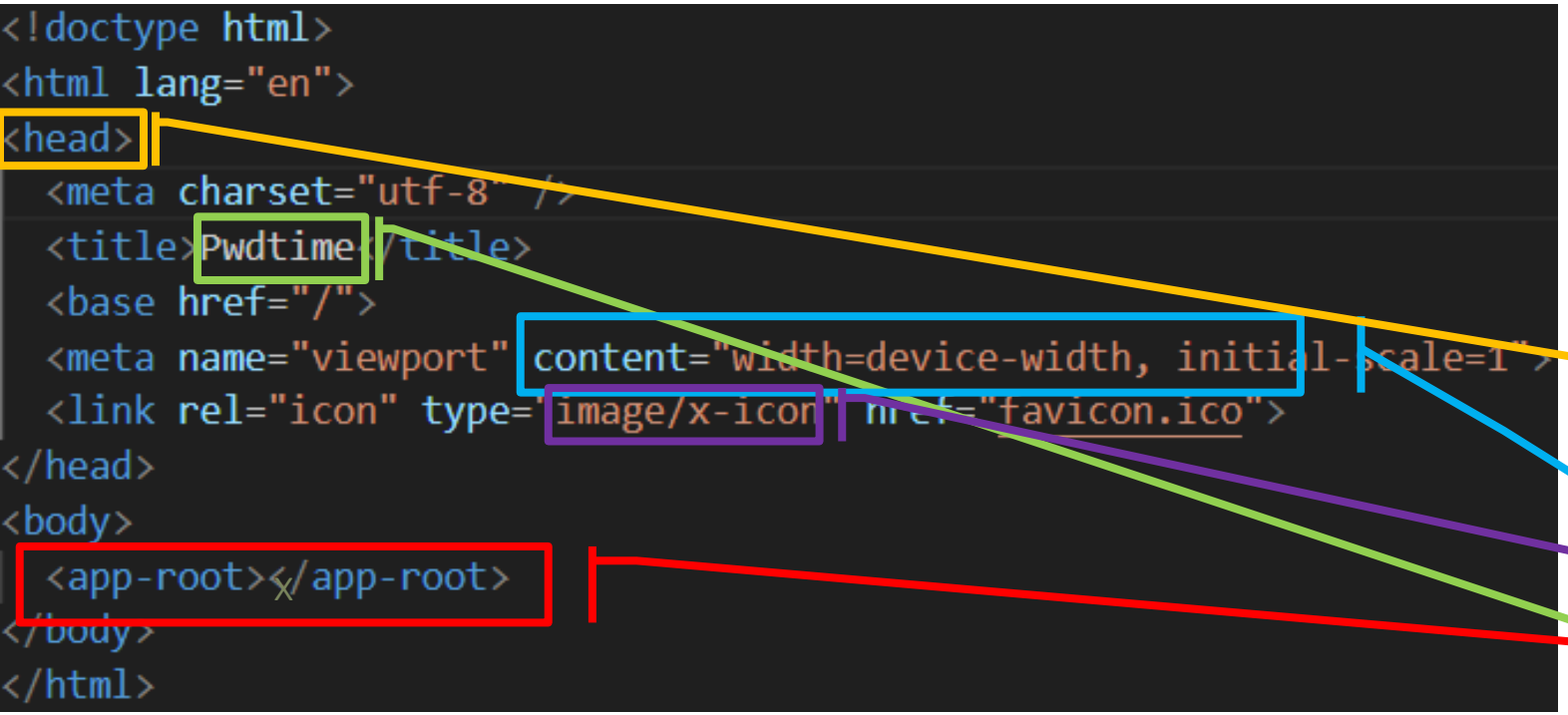
Comportement

Javascript

Structure, squelette



HTML : balisage



Langage à balises

<maBalise></maBalise>

Forme classique

<maBalise />

Forme sans contenu

« Auto-fermante »

Attention

Structure invalide

= comportement indéfini !

Vocabulaire

Balise

Nom de la balise

Attribut

Valeur de l'attribut

Contenu

HTML : balises

Balises de base

<html>, <head>, <body> …

Documentation (ex : MDN)

Attributs « universels »

title, id, class..

Balises et attributs codifiés

Mais possible d'ajouter ses éléments « custom »

```
<!doctype html>
<html lang="en">
    <head>
        <meta charset="utf-8" />
        <title>Pwdtime</title>
        <base href="/">
        <meta name="viewport"
              content="width=device-width, initial-scale=1">
        <link rel="icon"
              type="image/x-icon"
              href="favicon.ico">
    </head>
    <body>
        <app-root></app-root>
    </body>
</html>
```

CSS

```css
:root {
    --bg-color: #141218;
    --text-color: #e6e0e9;
    --prim-color: #d0bcff;
    --on-prim-color: #381e72;
    --prco-color: #4f378b;
    --on-prco-color: #eaddff;
}

html, body {
    background: var(--bg-color);
    color: var(--text-color);
    font-family: Roboto, Verdana, serif;
}

a, .link {
    color: var(--prim-color);
    text-decoration: none;
}

a:hover, a:focus, .link:hover, .link:focus {
    color: var(--on-prim-color);
    text-decoration: underline;
}

a[href^="https://"]:after {
    content: " ↪";
}
```

- Cascading Stylesheets
  - « feuilles de styles en cascade »
- Syntaxe simple
  - Fonctionnement bien moins simple
- Principe de sélecteurs, règles, valeurs
  - Sélecteur : sélectionne sur quoi appliquer des règles
  - Règle : directive accompagnée de valeurs

CSS : application

- Cascade
  - Au fil des directives
  - Mais selon un système de précision
    - *Un élément plus précis « surcharge » un moins précis*
- Modèle en boîtes
  - Block, inline, flex, grid…
- Quatre « entrées »
  - Style séparé (fichiers CSS) via <link>
    - *Import possible dans une feuille existante*
  - Style dans document via <style>
  - Style en ligne via attribut style

```
    <title>Document</title>
    <style type="text/css">
        em#rd {
            color: red;
        }

        em.tst {
            color: blue;
        }
    </style>
</head>

<body>
    <p id="myId">
        <em id="rd" class="tst">TEST STYLE PLEASE IGNORE</em>
    </p>
</body>
```

TEST STYLE PLEASE IGNORE

```
em {
    color: red;
}

em {
    color: blue;
}
```

TEST STYLE PLEASE IGNORE

```
  <title>Document</title>
  <link rel="stylesheet" type="text/css" href="./mystyle.css" />
  <style type="text/css">
      p {
          font-weight: bold;
          font-style: italic;
      }
  </style>
head>
ody>
  <p style="font-weight: bold; font-style: italic;"></p>
body>
```

Javascript

- 1995 • Proposition
- 1997 • Normalisation ECMA
- 1998 • ECMA262-2
- 2002 • ECMA 3 + ISO
- 2011 • ECMA 5
- 2015 • ES6
- 2016 • ES7
- 2022 • ES8

Langage dédié au comportement
- Réaction aux événements
- Lecture du DOM
- Manipulation du DOM
- Interactions utilisateurs

Date de 1996
- Evolution
- Mais certaines documentations pas à jour
- Certains développeurs pas à jour

ATTENTION obsolescence

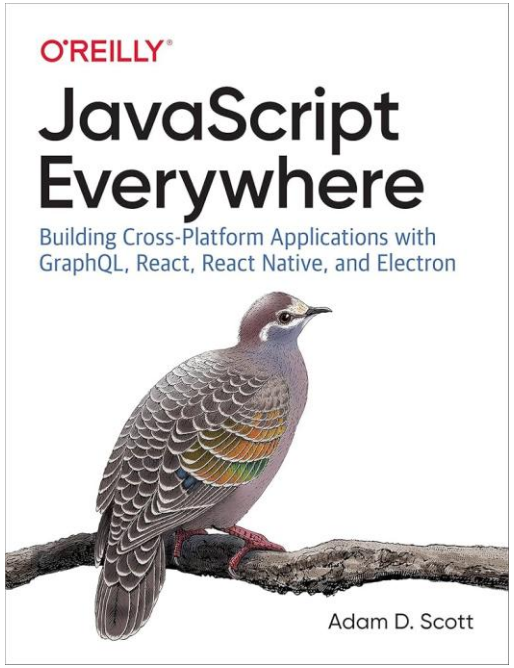
Javascript : au-delà

Evolution forte vers 2015/2016
- Et continue depuis
- Même si ralentissements récents

Javascript s'est imposé à tous les niveaux
- Interfaces Web
- Ou Web-like (smartphones…)

Packaging d'applications
- Electron

Serveur Web
- Nodejs



Javascript

« avancé »

(ou juste moderne en fait)

Variables : old

- Déclarer une variable
  - Si usage d'une variable non-déclarée
  - Problèmes
    - *Ou erreur si strict mode*
- Ancienne façon : var
  - Portée selon là où on est
  - Dans une fonction : locale
  - Hors de fonction : « globale » au fichier
- Eventuellement initialiser une valeur
- Possible de déclarer plusieurs variables en une fois

```
var myVariable = 10; // déclaration classique
var myOther = 8;
var myVar; // sans initialisation

console.log(myVar); // undefined

function myFunc(a) {
    if(a > 0) {
        var myVariable = 5 + a; // variable différente !
        myOther = 2 + a; // même variable !
    }
}

console.log(myVariable); // pas de changement
myFunc(1);
console.log(myVariable); // pas de changement
console.log(myOther); // bien changée
```

```
var a = 0, b, c = 0;
console.log(a); // 0
console.log(b); // undefined
console.log(c); // 0
```

Variables : gold

```
let x = 1;
if(x > 0) {
    let x = 2; // pas d'erreur, autre scope
}
console.log(x);

let x = 3; // erreur
```

```
const y = 10;
const z; // doit être initialisé !
y = 5; // const ! va jeter une erreur

const a = {'b': 10};
a.b = 5; // on ne modifie pas, on mute !
```

Nouvelle façon : ça dépend

Valeur pouvant varier : let

- Déclare une variable dont la portée est limitée au bloc courant
- Bloc = { } !
- TDZ
  - *Temporal dead zone*
  - *Contrairement à var, si accès avant déclaration*
  - *Réelle erreur*

Valeur ne variant pas : const

- ATTENTION pas vraiment une constante
- Peut toujours être « mutée »
- Donc voir ses propriétés internes changer
- Mais ne peut être réaffectée
- Doit donc forcément être initialisée !
- Même type de portée que let

Fonctions

Mot-clef function

Permet de créer

Une fonction

Rappel : nom, arguments, sortie (ou retour)

- Pas forcément de nom en vrai
- Fonction anonyme

Sortie : mot-clef return

Une fonction est un type de structure comme Number…

- Donc une variable peut contenir (faire référence) à une fonction !

```
function abc(e1, e2) {
    console.log(this); // contexte d'appel (window)
    e1++; // incrémentation de l'argument
    return e1 + e2; // retour
}

console.log('Test');
let s = abc(2, 4); // appel
console.log(s);
console.log(e1); // n'existe pas !
```

```
Test
▶ Window file:///C:/Users
7
▶ Uncaught ReferenceError
    <anonymous> debugger
  [En savoir plus]
```

```
console.log(abc);
▶ function abc(e1, e2)
```

Arrow function

```
let cb = (ar) => { return ar**3; };
let cube = (ar) => ar**3;
console.log(cb(10));
console.log(cube(3));
```

```
document.querySelector('#a').addEventListener('click', function(){
    console.log(this);
    return false;
});

document.querySelector('#b').addEventListener('click', () => {
    console.log(this);
    return false;
});
```

- Raccourci d'écriture
  - Avec quelques subtilités
- Fonction anonyme
- Importée dans le contexte courant
  - Rappel
  - this est une référence à l'objet en cours
  - Dépend du contexte d'exécution
  - Arrow function = pas de nouveau contexte !

```
<a id="a">
Window file:///C:/
```

Classes

```
class Polygon {
  constructor(height, width) {
    this.area = height * width;
  }
}

console.log(new Polygon(4, 3).area);
```

ECMAscript 2015

Provient de l'historique fonctionnement par prototype

- Qui rendait la prog objet en JS infâme

Mot-clef : class

Reste un simple « sucre syntaxique »

- Raccourci d'écriture qui sera « retransformé » en interne

Constructeur : constructor

Méthodes statiques : static

Propriétés publiques

- Dans la classe, sans symbole

Propriété privées

- Dans la classe, précédé d'un #

Héritage : extends

Référence à l'objet : this

Référence à la classe parente : super

Classes : exemple

```
class SubTest extends Test {
    constructor() {
        super();
    }

    doIt(x) {
        return 1 + super.doIt(x);
    }
}
```

```
class Test {
    static instance = null;
    publicValue = 0;
    #privateValue = 10;

    static getInstance() {
        if(null === Test.instance) {
            Test.instance = new Test();
        }
        return Test.instance;
    }

    constructor() {
        console.log('Constructing !');
    }

    get pv() {
        return this.#privateValue;
    }

    set pv(i) {
        this.#privateValue = i;
    }

    doIt(x) {
        return this.publicValue * (x + this.#privateValue);
    }
}
```

Asynchronisme

Date pas d'hier

Auparavant : callbacks

- Fonctions de rappels appelées à la fin d'une action
- C'est donc l'action qui doit lancer la callback
- Pas de renvoi de valeur

Maintenant : Promesses

- Promise

```
function faireUnTruc(arg1, callbackFn) {
    let res = 1+arg1;
    callbackFn(res);
    return res;
}

let myFunc = () => { console.log('Vraiment fini !'); };

console.log('A');
console.log(faireUnTruc(1, myFunc));
console.log('B');
```

Async : Promise

```
function resolveAfter2Seconds() {
  return new Promise((resolve) => {
    setTimeout(() => {
      resolve("resolved");
    }, 2000);
  });
}

async function asyncCall() {
  console.log("calling");
  const result = await resolveAfter2Seconds();
  console.log(result);
  // Expected output: "resolved"
}

asyncCall();
```

On définit une promesse

Et on décrit ce qu'on souhaite une fois réalisée

On peut conjuguer des promesses

- « une fois que TOUTES les actions sont faites… »
- « dès qu'au moins UNE action est réalisée… »

Et gérer les cas d'erreur ou de « rejet »

En ECMAscript en 2025 les fonctions asynchrones renvoient des promesses

Promise en effet

- Méthode then
  - Décrit ce qu'on souhaite une fois la promesse réalisée
  - Second argument facultatif : rejet
- Méthode catch
  - Décrit ce qu'on souhaite au rejet
- On utilise généralement des arrow functions pour then/catch
  - Evite les soucis de contexte
- Then et consorts peuvent être chaînés puisque renvoient… des promesses !

```
const myPromise = new Promise((resolve, reject) => {
    setTimeout(() => {
        resolve("foo");
    }, 300);
});

myPromise.then((r) => { console.log('Fini '+r+' !'); });
```

Modularité

```
export const name = "square";

export function draw(ctx, length, x, y, color) {
  ctx.fillStyle = color;
  ctx.fillRect(x, y, length, length);

  return { length, x, y, color };
}
```

- Les modules sont des regroupements de fonctionnalités
- On peut exporter / importer des modules
  - A voir comme des espaces de noms liés à des librairies
  - Mais pas totalement
- Export revient à « exposer »
  - Mot-clef : export
- Importer revient à « charger »
  - Mot-clef : import

```
import { name, draw, reportArea, reportPerimeter } from "./modules/square.js";
```

Modularité : plus

Deux mots-clefs identiques

import
- Effectue un import « live »
- Lecture seule du coté de celui qui importe
- Peut définir un ou plusieurs éléments à importer
- Peut renommer un élément

Import()
- Import dynamique
- Evalué à l'exécution lorsque nécessaire

Résolution du chemin : / ./ et ../
- Dépend des plateformes !

Importmap possible

```
import { name as squareName, draw } from "./shapes/square.js";
import { name as circleName } from "https://example.com/shapes/circle.js";
```

```
import("./modules/myModule.js").then((module) => {
  // Do something with the module.
});
```

Autres éléments utiles

```
console.log('A B ' + c + ' D ' + e);
console.log(`A B ${c} D ${e}`);
```

```
['A', '1', 3, 4, 'G'].filter((v) => !isNaN(v));
```

```
let polys = [
    [1, 2, 3],
    [1, 2, 4],
    [2, 3, 5]
];
function ply(a, b, c) { console.log(`${a}x² + ${b}x + ${c}`); }
ply(...polys[0]); // 1x² + 2x + 3
polys.forEach((entry) => ply(...entry));
```

Template literals

- En JS on utilise fortement la concaténation via ` plutôt que +
- Syntaxe : `maChaine ${v} avec var`

Opérations de tableaux

- Map, filter, reduce…
- Sont des opérations sur les tableaux plus efficaces et pratiques que réimplémenter soi-même certaines boucles

Spread operator : …

- Découpe une liste d'éléments en autant d'éléments
- Un tableau de 3 éléments devient trois éléments séparés
- Manipulation de tableaux

Typescript

Javascriptypé

Javascript et le laxisme

Javascript est un langage « lâche »

Permet de faire des opérations à l'issue incertaine

- Modifier des variables déclarées ailleurs
  - *Portée fluctuante, inconstance*
- Modifier des objets supposés cloisonnés
  - *Fonctionnement par prototype*
- Accepter n'importe quel argument
  - *Pas de typage d'entrée*
- Incertitude du retour
  - *Pas de typage de sortie*

2010 11 15 i am myself but also not myself
2010 11 10 false advertising
2010 10 15 ie 754
2010 09 16 eval changes
2010 07 23 im not a number really
2010 07 22 magic increasing number
2010 07 15 typeof number is not number
2010 07 12 fail
2010 07 11 length of what now
2010 06 09 function in ur string
2010 06 08 void is a black hole
2010 06 02 instances and default values
2010 04 31 syntax highlighting serverside with google prettyfy
2010 04 31 isNaN
2010 04 30 operators and regexp fun
2010 04 17 global scope mindtricks
2010 04 16 syntax highlighting wtf
2010 04 16 express js gotcha
2010 04 16 contributing to wtfjs
2010 04 16 build your own wtfjs
2010 04 15 hello world
2010 04 12 call in ur call
2010 04 02 object to primitive coerce
2010 03 04 max vs the infinite
2010 03 02 ie cursed recursion
2010 02 26 implicit tostring fun
2010 02 26 array crazy
2010 02 25 jsftw: google closure compiler
2010 02 25 ie scope
2010 02 25 ie and webkit agree
2010 02 24 messing with number prototype

Javascript et le présent

```
function someClass() {
    this.v = 42;
}

var o = new someClass();
someClass.prototype.getV = function() {
    return this.v;
};
```

Parce que langage à prototype

- Et non Objet en vrai

« Trous » dans l'OO

- Qui tentent de se combler au fil du temps
- Mais encore pas là

Encapsulation

Interfaces

Enumérations

Annotations (décorateurs)

Typage

Un avantage
- Le « any » c'est moche

Permet de détecter les erreurs AVANT de lancer le code

Garant de la robustesse et de la logique des entrées / sorties

Déclaration de variable

Déclaration d'argument

Déclaration de sortie

Y compris generics
- Ex : Array<number>

```
pwd = input<string>('');
time:Timo = {};
currentValue:number = 0;
```

```
getUnitFromNumber(t:number): Unitn {
  const x = [
    {'v': 1e9, 't': 'Md'},
    {'v': 1e6, 't': 'Mn'},
    {'v': 1e3, 't': 'k'}
  ];
  const r = {'unit': '', 'val': t};
  for(let v of x) {
    if(t >= v.v) {
      r.unit = v.t;
      r.val = Math.floor(t / v.v);
      if(r.val > 1) { r.unit += 's'; }
      break;
    }
  }
  return r;
}

unitnToString(u:Unitn): string {
  return ('' + u.val + ' ' + u.unit).trim();
}
```

Classes

```
abstract class Base {
  abstract getName(): string;

  printName() {
    console.log("Hello, " + this.getName());
  }
}
```

```
class Params {
  constructor(
    public readonly x: number,
    protected y: number,
    private z: number
  ) {
    // No body necessary
  }
}
```

- Encapsulation
  - private / protected / public
    - *ATTENTION ne change rien au runtime*
    - *# pour les propriétés : préféré*
  - Getters et setters via get/set !
- Staticité
  - Mot-clef static
- Abstraction
  - Mot-clef abstract
- Construction par paramètres
  - Comme en C++ !
- Gestion des generics

```
class Box<Type> {
  contents: Type;
  constructor(value: Type) {
    this.contents = value;
  }
}

const b = new Box("hello!");
```

```
const b: Box<string>
```

Interfaces

RAPPEL

- Une interface est la description d'une capacité
- Une classe implémentant une interface se déclare « capable de … »
- En OO, on se pose d'abord la question de la capacité

Mot-clef : interface

- ATTENTION
- Autrefois utilisé pour les déclarations de type personnalisés
- Interface vs type
  - *Cf. Cheatsheets TS*

Common Syntax

```
interface JSONResponse extends Response, HTTPAble {
  version: number;

  /** In bytes */
  payloadSize: number;

  outOfStock?: boolean;

  update: (retryTimes: number) => void;
  update(retryTimes: number): void;

  (): JSONResponse

  new(s: string): JSONResponse;

  [key: string]: number;

  readonly body: string;
}
```

Optionally take properties from existing interface or type

JSDoc comment attached to show in editors

This property might not be on the object

These are two ways to describe a property which is a function

You can call this object via () - (functions in JS are objects which can be called)

You can use **new** on the object this interface describes

Any property not described already is assumed to exist, and all properties must be numbers

Tells TypeScript that a property can not be changed

Object Literal Syntax

```
type JSONResponse = {
  version: number;                      // Fi
  /** In bytes */                       // At
  payloadSize: number;                  //
  outOfStock?: boolean;                 // Op
  update: (retryTimes: number) => void; // Ar
  update(retryTimes: number): void;     // Fu
  (): JSONResponse                      // Ty
  [key: string]: number;                // Ac
  new (s: string): JSONResponse;        // Ne
  readonly body: string;                // Re
}
```

Terser for saving space, see Interface Cheat Sheet for more info, everything but 'static' matches.

Enumerations

```
enum Direction {
  Up,
  Down,
  Left,
  Right,
}
```

```
enum UserResponse {
  No = 0,
  Yes = 1,
}

function respond(recipient: string, message: UserResponse): void {
  // ...
}

respond("Princess Caroline", UserResponse.Yes);
```

- Mot-clef : enum
- Utilisable comme un type
- Peuvent être mappés en string
- Possible de « mixer » les deux types
  - Peu recommandé
- Attention
  - Rappel que c'est du transpilage
  - Certains comportements peuvent être erratiques si on joue trop (keyof, typeof…)

Décorateurs

Ou « annotations » dans d'autres langages

Instructions spéciales ajoutées sur une classe (ou autre !)

Permet de spécifier des comportements paramétrés

Grand nombre de possibilités

Permet d'ajouter de la modularité et des aspects pratiques au code

Fortement utilisé en Angular

```
class ExampleClass {
  @first()
  @second()
  method() {}
}
```

```
@sealed
class BugReport {
  type = "report";
  title: string;

  constructor(t: string) {
    this.title = t;
  }
}
```

Angular - Intro

Nodejs et environnement

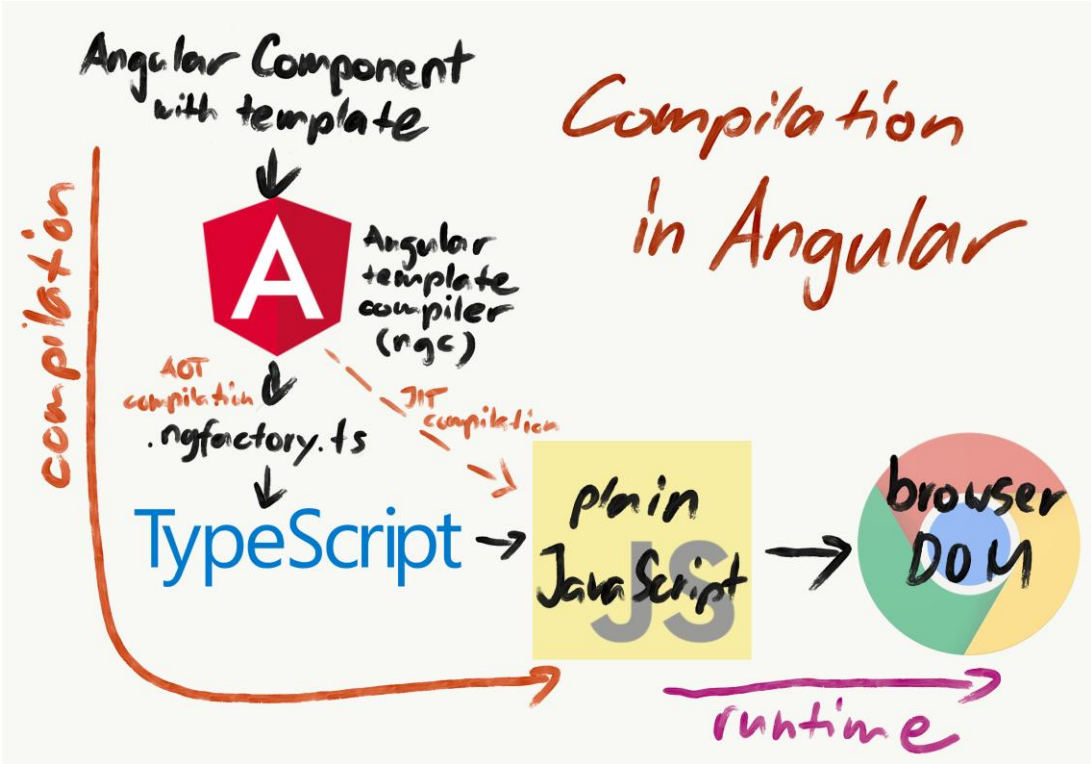
Framework front

Focus sur le coté « client » d'une application web
- Interface
- UI/UX
- Réactivité
- Points d'accès / entrées
- Validation saisie
- Traditionnellement appel à un back
  - *Appels API REST*

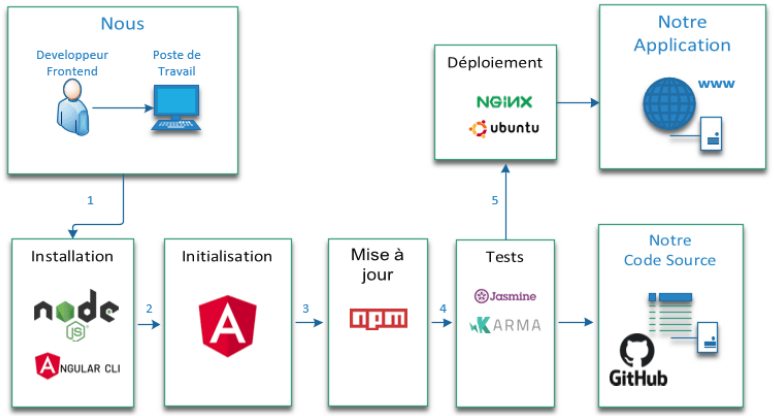
Basé sur Javascript

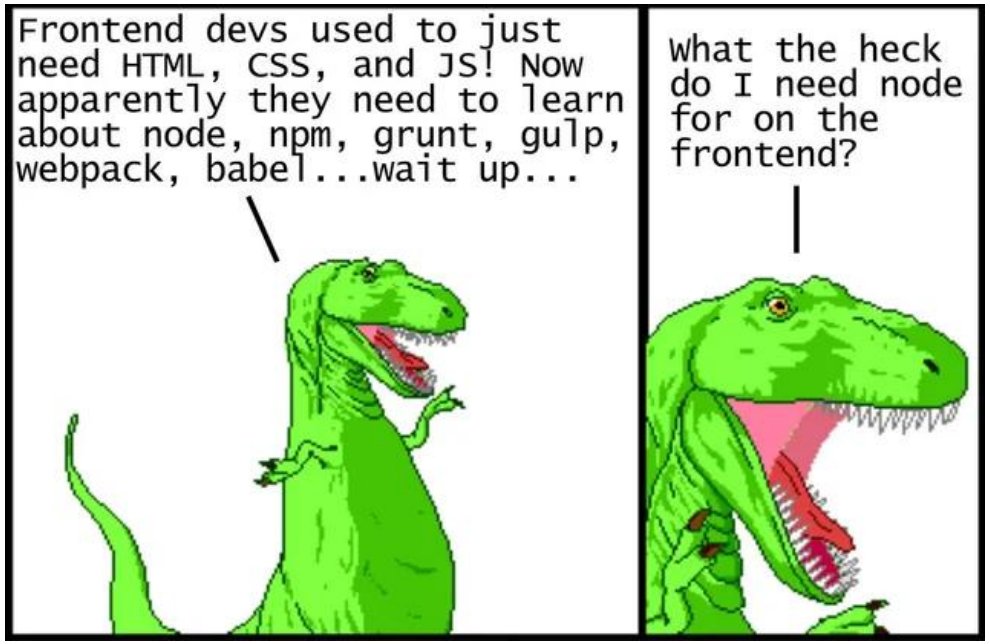
Compatible Typescript

Stack NPM



Stack





Projets JS en 2025 : pile d'éléments

- Stack

Node packet manager

- Gestionnaire de paquets basés sur Nodejs

Nodejs

- Environnement d'exécution JS
- Historiquement serveur web

Angular

- Composants installables via NPM
- Y compris Angular CLI

Installation du nécessaire

La stack logicielle, évidemment

Node + NPM

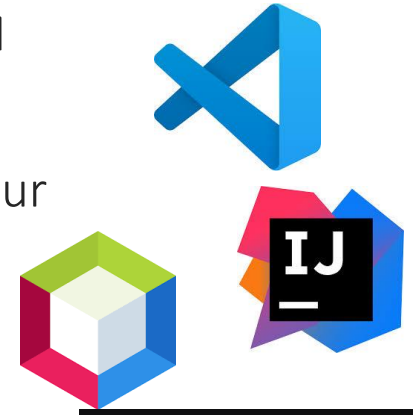
- NPM vient avec Node !
- D'autres installeurs alternatifs de NPM

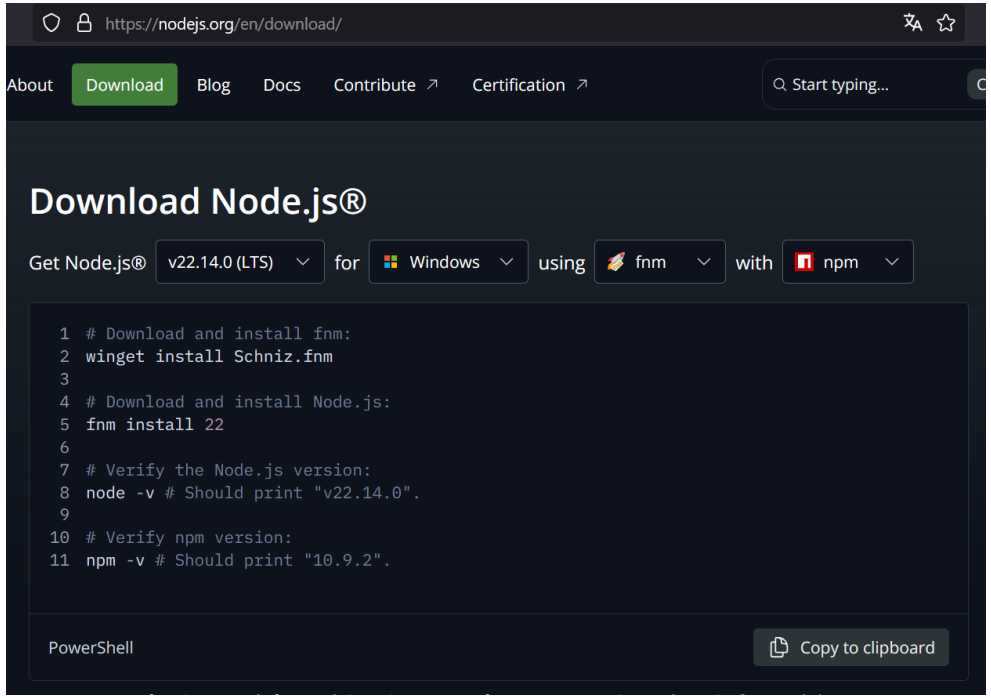
IDE

- Visual Studio Code est très efficace pour JS
- Mais d'autres existent !

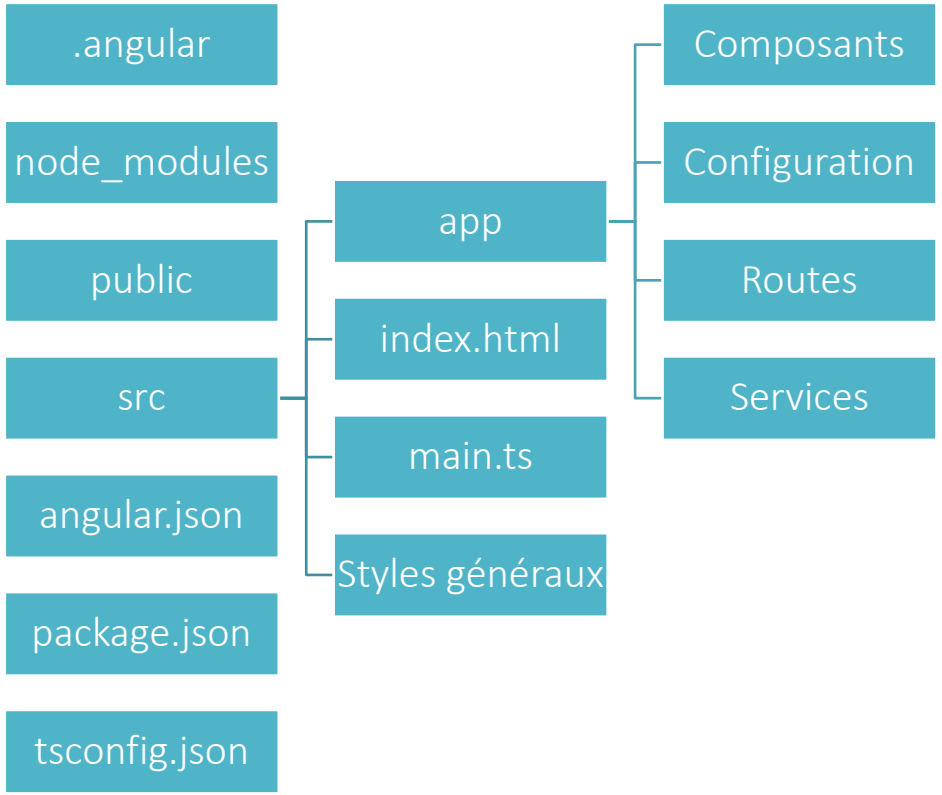
Angular CLI

- Ensemble de commandes pour Angular





Structure



src
: Fichiers sources de l'application

public
: Assets et fichiers accessibles (images)

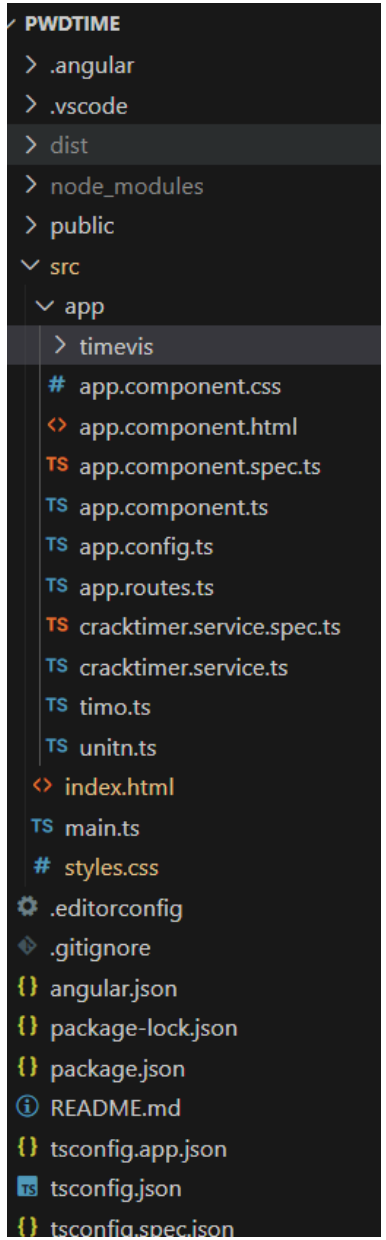
node_modules
: Dossier de dépendances

dist
: Dossier de génération des éléments

package.json
: Configuration NPM

angular.json
: Configuration Angular

tsconfig.json
: Configuration du transpiler



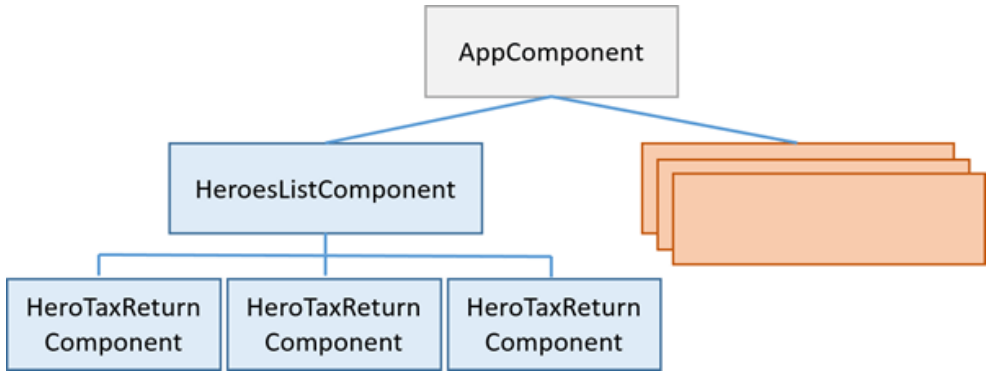
Structure - encore

src/app
- Contient les éléments d'entrée de base de votre application
- Toute la logique applicative

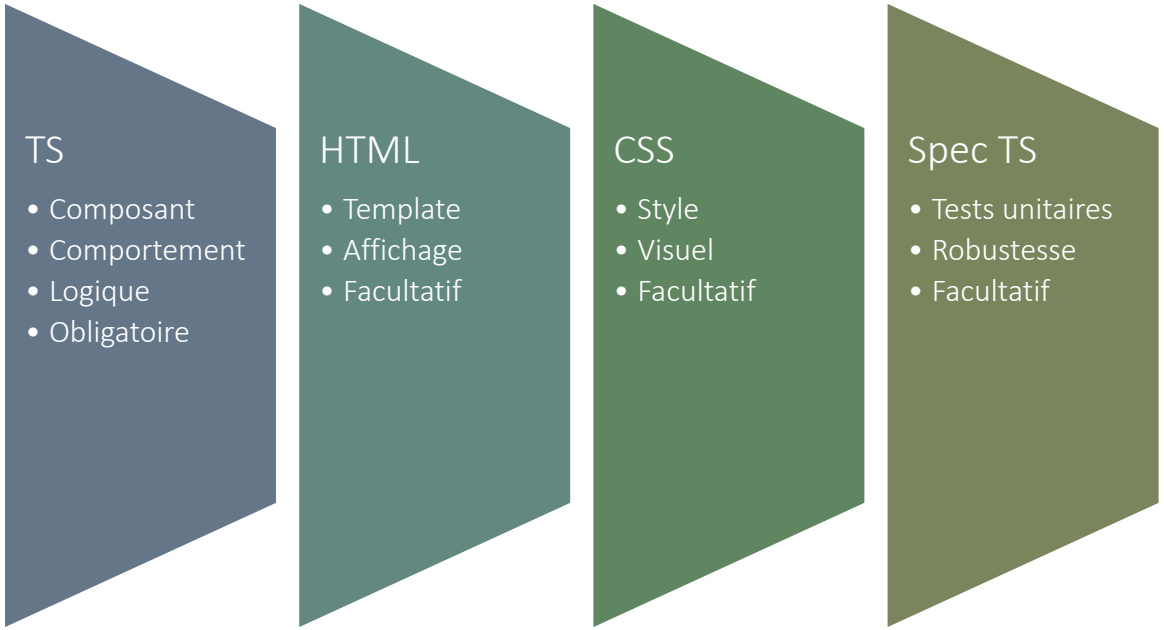
src/index.html
- L'index racine – généralement à laisser tel quel

src/main.ts
- Le fichier d'entrée TS qui « boostrap » l'application

src/styles.css
- Un fichier de style global pour toute l'application – généralement à éviter



Structure… encore ?



src/app/app.config.ts
Définit la façon dont angular va assembler les éléments de l'application

src/app/app.routes.ts
Définit la configuration de routage de l'application

*.component.css
Style du composant

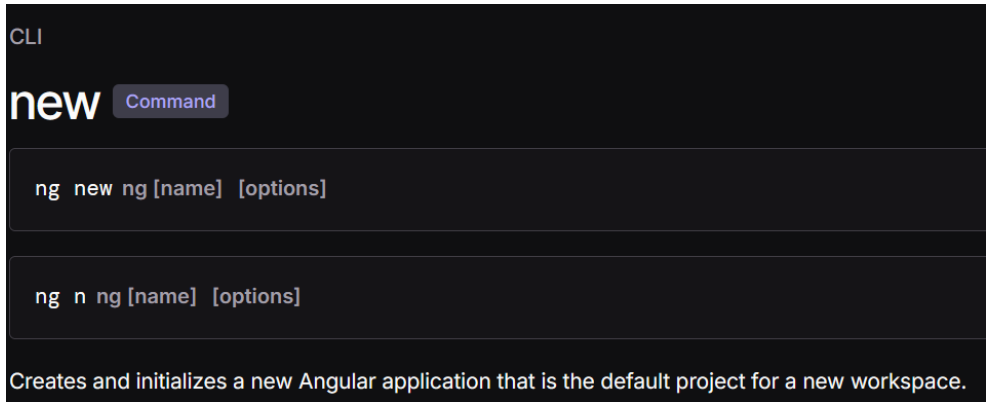
*.component.html
Squelette du composant

*.component.ts
Comportement du composant

*.component.spec.ts
Tests unitaires du composant

Un simple projet

- Génération du projet
  - ng new <nom projet>
  - Sans les <> évidemment
  - Génère un projet (dossier)
  - Avec tout ce qu'il faut dedans
- Lancer le projet
  - npm start
  - DANS le dossier
  - Lance le transpilage et démarre un serveur nodejs
    - *Lance un ng serve en arrière-plan*
  - Changements en temps réel

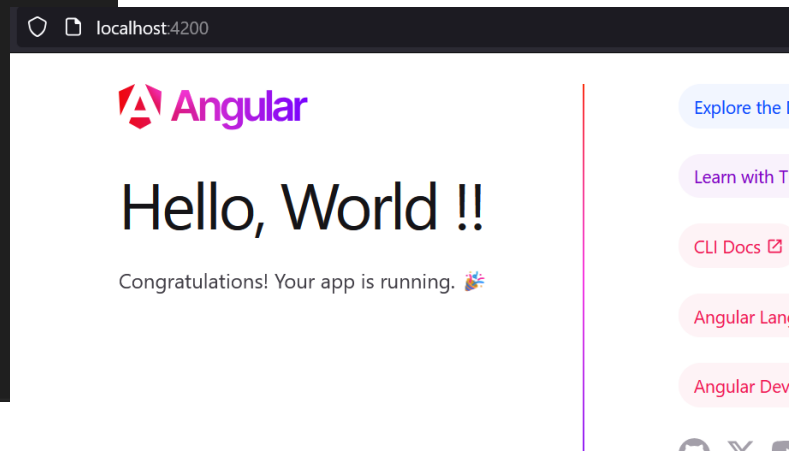



```
$ npm start

> pwdtime@0.0.0 start
> ng serve

Initial chunk files | Names         |  Raw size
polyfills.js        | polyfills     |  90.20 kB |
main.js             | main          |  11.16 kB |
styles.css          | styles        | 610 bytes |

                    | Initial total | 101.97 kB

Application bundle generation complete. [3.795 seconds]

Watch mode enabled. Watching for file changes...
NOTE: Raw file sizes do not reflect development server per-request transformations.
  → Local:   http://localhost:4200/
  → press h + enter to show help
```

Modification basique

```
      </linearGradient>
      <clipPath id="a"><path fill="#fff" d="M0 0h982v239H0z" /></cli
    </defs>
  </svg>
  <h1>Hello, {{ title }} !!</h1>
  <p>Congratulations! Your app is running. 🎉</p>
</div>
<div class="divider" role="separator" aria-label="Divider"></div>
<div class="right-side">
  <div class="pill-group">
```

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

@Component({
  selector: 'app-root',
  imports: [RouterOutlet],
  templateUrl: './app.component.html',
  styleUrl: './app.component.css'
})
export class AppComponent {
  title = 'World';
}
```



Dans app.component.html

- Modification de la structure
- Ajout de !! par exemple
- Fichier complexe visuellement, mais tout peut être détruit
- Remarquez l'usage de {title} (et gardez-le)

Dans app.component.ts

- Changement du titre title

Angular - Composants

@Component

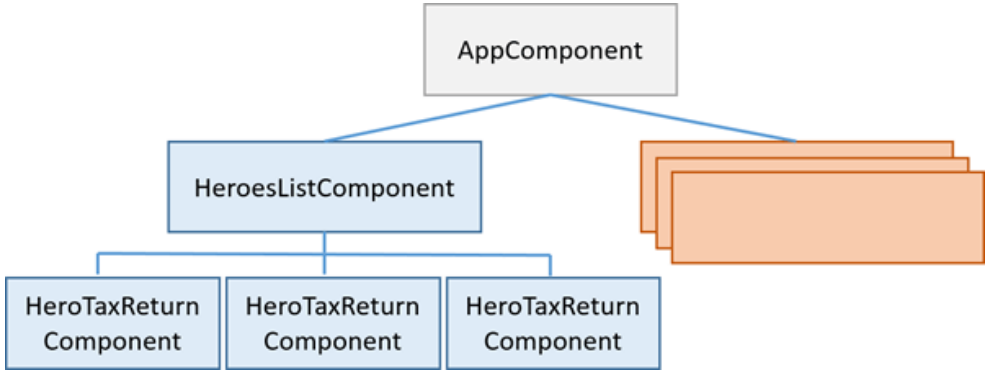
Composants

Briques principales

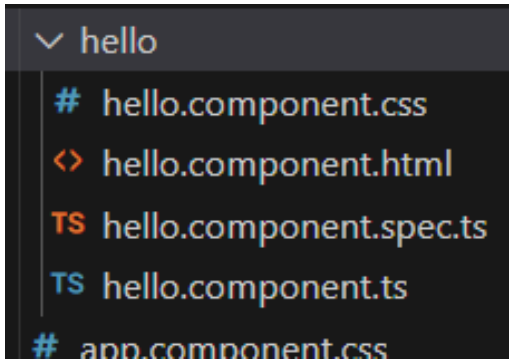
Chaque composant a un rôle

- Un affichage
- Un style
- Un comportement

Chaque composant peut inclure d'autres composants



Créer un composant




```
$ ng generate component hello
CREATE src/app/hello/hello.component.html (21 bytes)
CREATE src/app/hello/hello.component.spec.ts (608 bytes)
CREATE src/app/hello/hello.component.ts (221 bytes)
CREATE src/app/hello/hello.component.css (0 bytes)
```

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

@Component({
  selector: 'app-hello',
  imports: [],
  templateUrl: './hello.component.html',
  styleUrl: './hello.component.css'
})
export class HelloComponent {

}
```

Soit créer manuellement les fichiers nécessaires

- HTML, CSS, TS
- En vrai, juste le TS est nécessaire

Soit passer par ng generate

- ng generate component
- ng generate c

Ce qui « déclare » le composant est le décorateur @Component

Attention à bien importer le composant là où il est utilisé

@Component

La majorité passe par le décorateur @Component

selector
- Balise pour ce composant (en vérité sélecteur CSS)
- Indique sur quel(s) élément(s) s'applique le code

imports
- Liste d'imports nécessaires au fonctionnement du composant
- Transmis au template !

template ou templateUrl
- Le HTML ou le chemin vers le HTML

styles ou styleUrl
- Le CSS ou le chemin vers le CSS

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

@Component({
  selector: 'app-hello',
  imports: [],
  templateUrl: './hello.component.html',
  styleUrl: './hello.component.css'
})
export class HelloComponent {

}
```

```
@Component({
  selector: 'profile-photo',
  template: `<img src="profile-photo.jpg" alt="Your profile photo">`,
  styles: `img { border-radius: 50%; }`,
})
export class ProfilePhoto { }
```

Données

```
export class AppComponent {
  title = 'World';

  changeTitle() {
    this.title = 'NOBODY !';
  }
}
```

Nombreuses possibilités d'interaction TS/HTML

Réactivité

Variable simple

- Définition dans la classe
- Usage dans le HTML
- Réactif
  - *Tout changement dans le TS*
  - *Va être impacté en direct*

Cycle de vie d'un composant

Appel de fonctions prédéfinies

Permettent une customisation au fil du cycle de vie du composant

Exemples

- ngOnInit
  - *Exécutée une fois à l'initialisation*
- ngOnChanges
  - *Exécutée à chaque changement d'un paramètre d'entrée*

Angular - Signaux

signal()

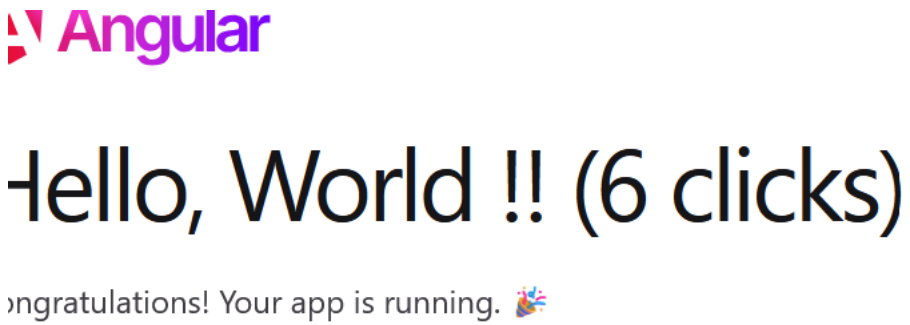
Signals

```
const count: WritableSignal<number> = signal(0);
const doubleCount: Signal<number> = computed(() => count() * 2);
```

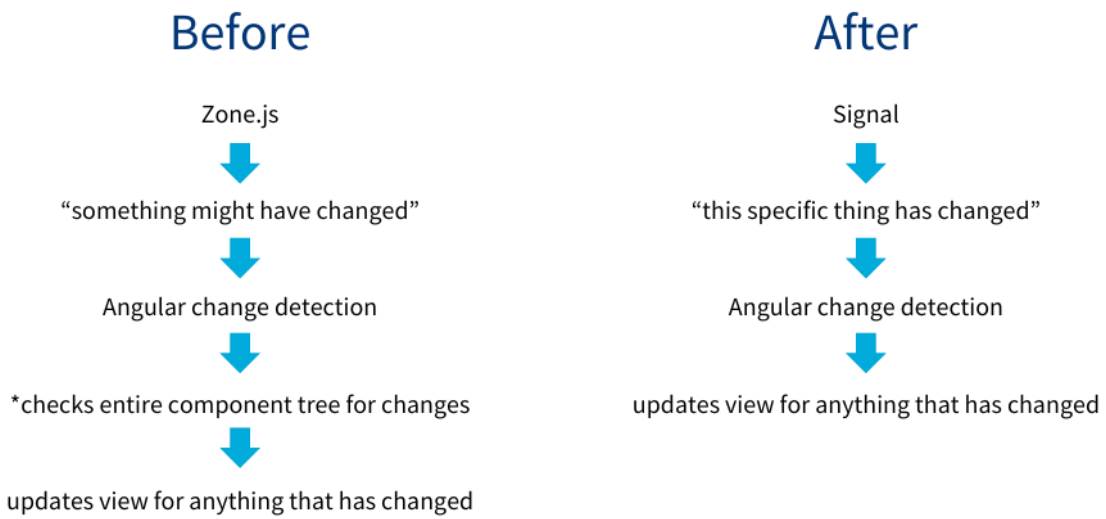
```
<h1 (click)="clicketis()">Hello, {{ title }} !! ({{ clicked() }} clicks)</h1>
<p>Congratulations! Your app is running. 🎉</p>
```

```
title = 'World';
clicked = signal(0);

clicketis() {
  this.clicked.update((v) => v+1);
}
```

- Type spécial gérant une valeur (état)
- Création via signal()
- Fournit des méthodes utilitaires
  - set, update…
- Usage dans un template
  - Accesseur intégré
- Computed
  - Permet de baser un signal sur d'autres signaux



Signaux vs Variables



- Réactivité « classique » avec variables
  - Détection automatisée des changements
  - Se base sur les API de navigateurs disponibles
  - Avec un *certain* nombre de rectifications pour les cas non-gérés (via zone.js par exemple)
- Réactivité via signaux
  - Tout est compris dans la librairie en question

Angular - Input

input() et @Input

@Input

```
<p>
    Hello
    {{ title }} !
</p>
<p (click)="clicketis()"
    Clicked :
    {{ clicked() }}
</p>
```

```
export class HelloComponent {
  @Input() title = 'World';
  clicked = signal(0);

  clicketis() {
    this.clicked.update((v) => v+1);
  }
}
```

```
<app-hello title="World"></app-hello>
<p>Congratulations! Your app is running. 🎉</p>
/div>
```

- Manière principale de passer des données à un composant
- Depuis d'autres composants
  - Parents
- Ou depuis le template
  - « cette donnée provient du template »
- Décorateur @Input
  - Sur un champ/propriété
- OG
  - Toujours supporté mais signal.input conseillé

Input

Surcouche/alternative à signal

Fonction input

- Avec valeur par défaut

Utilisable dans les templates

- Comme d'autres variables classiques

Un input se comporte comme un signal

```
export class HelloComponent {
  title = input('World');
  clicked = signal(0);

  clicketis() {
```

```
  {{ title() }} !
```

Input : plus

```
class HelloComponent {
e = input.required<string>({'alias': 't'});
ked = signal(0);
```

```
<app-hello t="World"></app-hello>
```

- Possibilité de définir le type
  - Permet de restreindre les erreurs de typage
- Un input peut être obligatoire
  - Input.required
- Un input peut être aliasé
  - Argument facultatif alias
  - Change uniquement dans le template

Input et model

Au lieu d'input

On peut utiliser model

S'il s'agit d'un élément qu'on souhaite pouvoir échanger entre le composant, les composants qui l'utilisent, et le template

Syntaxe similaire

Considéré comme input et output

Nécessite un two-way binding pour écrire depuis le template

```
class HelloComponent {
 = model.required<string>({'alias': 't'});
ed = signal(0);
```

Angular - Templates

Du HTML en Typescript ??

Template et templateurl

- Le template peut soit être saisi dans le TS
  - A la manière de react
- Ou dans un fichier séparé
  - Conseillé pour la majorité des composants
- Distinction via propriété utilisée
  - template vs templateurl
  - Attention URI relative

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

@Component({
  selector: 'app-hello',
  imports: [],
  templateUrl: './hello.component.html',
  styleUrl: './hello.component.css'
})
export class HelloComponent {

}
```

```ts
@Component({
  selector: 'profile-photo',
  template: `<img src="profile-photo.jpg" alt="Your profile photo">`,
  styles: `img { border-radius: 50%; }`,
})
export class ProfilePhoto { }
```

Templates

```
<meter value="{{ clicked() }}" min="0" max="100"></meter>
```

```
<meter [value]="clicked()" min="0" max="100"></meter>
```

Clicked : 25

```
p (click)="clicketis()">
   Clicked :
```

```
<meter value="{{ clicked() }}"
       min="0" max="100"
       [test]="{{ title }}"></meter>
```

- Basé sur les éléments DOM
- Propriété statique
  - disabled
  - Valeur statique
- Propriété dynamique
  - [disabled]
  - Valeur dynamique (variable)
- Gestionnaire d'événements
  - (event)
  - Valeur : fonction à exécuter
- ATTENTION
  - Le DOM doit rester valide et cohérent

Contrôle de flux

Templates

Possible de boucles, conditions…

@if / @else

- Condition classique

@for

- Boucle sur une liste d'éléments
- track nécessaire
  - *Obligation liée au DOM*

```
</p>
@if (clicked() > 99) {
    <h3>Félicitations !</h3>
}
```

```
<ul>
@for (prime of primes; track $index) {
    <li>{{ prime }}</li>
}
</ul>
```

Pipes

```ts
import { UpperCasePipe } from '@angular/common';
import { Component, model, signal } from '@angular/core';

@Component({
  selector: 'app-hello',
  imports: [UpperCasePipe],
```

```html
<p>
    Hello
    {{ title()|uppercase }} !
</p>
```

Caractère |

Modificateurs dans les templates

Permet d'effectuer des transformations sur des données

- Ex : « date » génère un affichage au format date
- « lowercase » pour basculer en minuscule
- Et d'autres

Créer un pipe ?

- Décorateur @Pipe
- Et implémentation de l'interface PipeTransform

Two-way binding

Simple way binding
- Affichage réactif d'une valeur
- Input, signal…
- Depuis le TS vers le HTML

Mais on peut vouloir modifier directement une valeur depuis le HTML
- Tout en « obtenant » sa nouvelle valeur et en la réfléchissant en HTML
- Dans les deux sens

[(propriété)]=model
- Modèle étant le modèle à modifier
- Entre composants

Alternative via ngModel
- Fait partie de Forms (on en reparle !)
- Dans un même composant

```ts
import { FormsModule } from '@angular/forms';

@Component({
  selector: 'app-hello',
  imports: [UpperCasePipe, FormsModule],
  templateUrl: './hello.component.html',
  styleUrl: './hello.component.css'
})
export class HelloComponent {
  title = model.required<string>({'alias': 't'});
  clicked = signal(0);
  multiplier:number = 1.;

  clicketis() {
    this.clicked.update((v) => v + this.multiplier);
```

```html
<input type="number"
       min="0"
       max="5"
       step="0.1"
       [(ngModel)]="multiplier" />
```

Angular - Services

L’OO c’est du service par le service

Service, injection et OO

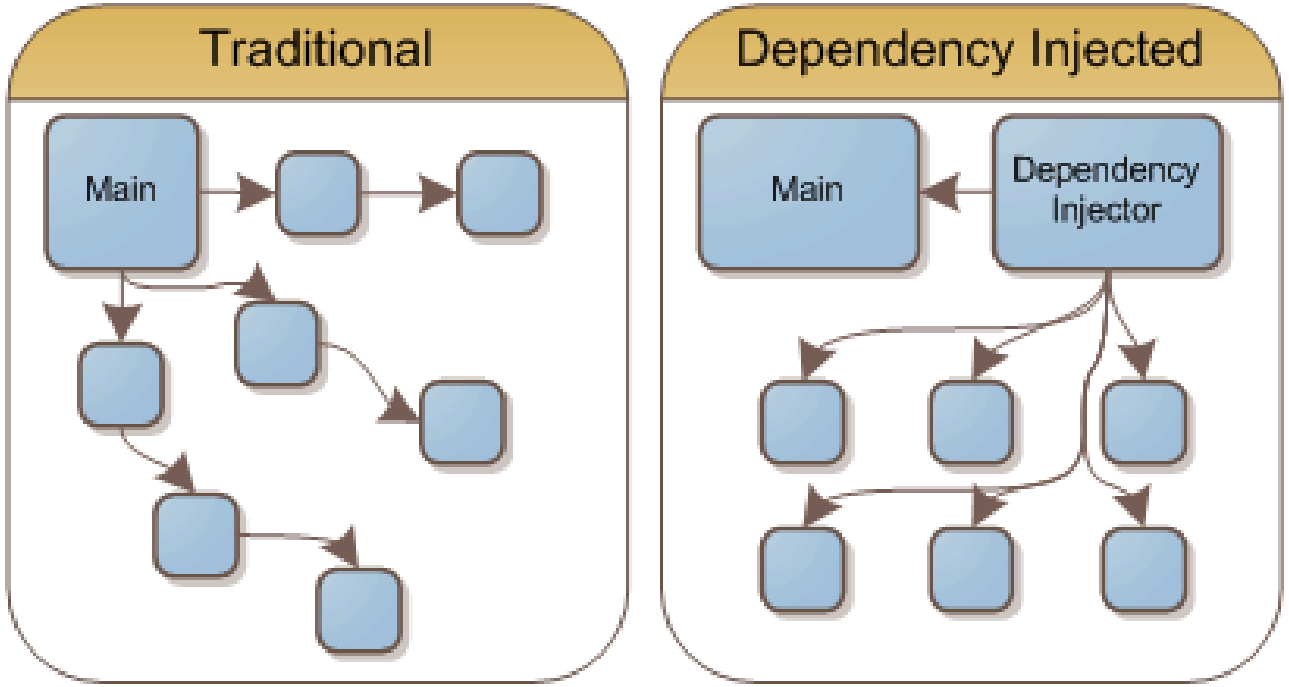
Orienté Objet

Parler en services rendus

Chaque élément rend un service précis

Besoin d'un service ?

On injecte le service dans le composant/service dans lequel on est



Définition

```
@Injectable({
  providedIn: 'root'
})
class HeroService {}
```

Toute classe peut être un service

Tant qu'elle peut être injectée

Décorateur : @Injectable

- Argument providedIn
- Décrit le contexte dans lequel on peut effectuer l'injection
- 'root' : racine (donc partout)

Injection du service

- Dans un composant ou service
  - Appel à inject()
  - Chargé lorsque nécessaire !
- Dans l'application complète
  - Dans la liste des providers de la config
  - Signifie qu'il est toujours chargé !
- Attention : reste nécessaire d'importer

```
xport class TimevisComponent {
 cracktimer:CracktimerService = inject(CracktimerService);
 oldPwd = '';
```

```
export const appConfig: ApplicationConfig = {
    providers: [
      { provide: HeroService },
    ]
};
```

Angular - Observable

Observable et RxJS

- Historiquement, RxJS était nécessaire dans Angular
  - Distinction marquée
  - Module rxjs
- Librairie gérant des Observable
- Observable
  - Objet réactif, déclenchant des événements asynchrones et réagissant de manière asynchrone aux événements
  - Pattern Observer
    - *Le sujet maintient une liste d'inscrits à ses événements*

```
import { Observable } from 'rxjs';

const foo = new Observable((subscriber) => {
  console.log('Hello');
  subscriber.next(42);
  subscriber.next(100); // "return" another value
  subscriber.next(200); // "return" yet another
});

console.log('before');
foo.subscribe((x) => {
  console.log(x);
});
console.log('after');
```

Exemple : HTTP Requests

```
constructor(private http: HttpClient) {
  // This service can now make HTTP requests via `this.http`.
}
```

```
http.get<Config>('/api/config').subscribe(config => {
  // process the configuration.
});
```

```
this.http.post('/api/upload', myData, {
  reportProgress: true,
  observe: 'events',
}).subscribe(event => {
  switch (event.type) {
    case HttpEventType.UploadProgress:
      console.log('Uploaded ' + event.loaded + ' out of ' + event.total + ' bytes');
      break;
    case HttpEventType.Response:
      console.log('Finished uploading!');
      break;
  }
});
```

Un usage régulier
- Appels HTTPs
- REST API
- Back-end

Requêtes asynchrones
- Réactivité aux événements de réception d'en-têtes, de contenu…

Création d'une requête
- Génère un « cold » Observable
- Un « prêt à l'emploi »
- Se lance réellement lorsqu'exécuté (donc quand nécessaire)

Exemple : HTTP Requests

Sur un Observable

On subscribe pour décrire l'action à effectuer au lancement / à la finalisation

En fait à chaque changement d'état

Les Observable sont entièrement compatibles avec les Promise

Même comportement async !

```ts
@Injectable({providedIn: 'root'})
export class UserService {
  constructor(private http: HttpClient) {}

  getUser(id: string): Observable<User> {
    return this.http.get<User>(`/api/user/${id}`);
  }
```

```ts
import { AsyncPipe } from '@angular/
@Component({
  imports: [AsyncPipe],
  template: `
    @if (user$ | async; as user) {
      <p>Name: {{ user.name }}</p>
      <p>Biography: {{ user.biography }}</p>
    }
  `,
})
export class UserProfileComponent {
  @Input() userId!: string;
  user$!: Observable<User>;

  constructor(private userService: UserService) {}

  ngOnInit(): void {
    this.user$ = this.userService.getUser(this.userId);
  }
}
```

Angular - Routing

Route

Une adresse et une méthode HTTP

- Route

L’adresse peut être un motif

/home

/issues/36

Etc.

Définir une route

```
const routes: Routes = [
  { path: 'first-component', component: FirstComponent },
  { path: 'second-component', component: SecondComponent },
  { path: '',   redirectTo: '/first-component', pathMatch: 'full' }, // redi
  { path: '**', component: PageNotFoundComponent },  // Wildcard route for a
];
```

```
path: 'first-component',
component: FirstComponent, // t
children: [
  {
    path: 'child-a', // child r
    component: ChildAComponent,
```

Typiquement dans app.routes.ts

Une route = un objet
- Path : le motif du chemin
- Component : le composant à utiliser
- Children : sous-routes, s'il y en a

Route par défaut
- Placeholder ou joker
- Motif : **

Redirections
- Argument redirectTo

Paramétrage de routes

- Dans le path
  - Arguments avec deux points
  - Ex : /issues/:id
- Dans le composant
  - Injection du service de routage
  - ActivatedRoute
  - Accès au tableau de paramètres via paramMap
- Pourquoi aussi compliqué ?
  - C'est… un observable !
  - Va donc être informé en cas de changement !
  - Pas le cas de snapshot !

```
{ path: 'hero/:id', component: HeroDetailComponent }
```

```
this.heroes$ = this.route.paramMap.pipe(
  switchMap(params => {
    this.selectedId = Number(params.get('id'));
    return this.service.getHeroes();
  })
);
```

```
const heroId = this.route.snapshot.paramMap.get('id');
```

Utiliser les routes

```
i><a routerLink="/first-component" routerLinkActive="active" ari
i><a routerLink="/second-component" routerLinkActive="active" ai
```

```
<!-- The routed views render in the <router-outlet>-->
<router-outlet />
```

Afficher une route

- Pour des liens
- [routerLink]

Ajouter le contenu routé

- En somme, la partie « dynamique » de votre application qui va donc changer selon la route
- <router-outlet>

RAPPEL

- Besoin d'importer dans votre composant
- Et dans le tableau imports de votre composant !

Angular - Forms

Deux approches

Reactive Forms

- Gestion des formulaires via éléments dédiés
- Objet FormControl

Template-driven Forms

- Gestion des formulaires comme n'importe quelle autre donnée dynamique

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

@Component({
  selector: 'app-reactive-favorite-color',
  template: `
    Favorite Color: <input type="text" [formControl]="favoriteColorControl">
  `,
  standalone: false,
})
export class FavoriteColorComponent {
  favoriteColorControl = new FormControl('');
}
```

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

@Component({
  selector: 'app-template-favorite-color',
  template: `
    Favorite Color: <input type="text" [(ngModel)]="favoriteColor">
  `,
  standalone: false,
})
export class FavoriteColorComponent {
  favoriteColor = '';
}
```

Reactive Forms

```
const login = new FormGroup({
    email: new FormControl('', {nonNullable: true}),
    password: new FormControl('', {nonNullable: true}),
});
```

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

@Component({
  selector: 'app-name-editor',
  templateUrl: './name-editor.component.html',
  styleUrls: ['./name-editor.component.css'],
  standalone: false,
})
export class NameEditorComponent {
  name = new FormControl('');
...
}
```

```
<label for="name">Name: </label>
<input id="name" type="text" [formControl]="name">
```

- Objet FormControl initialisé en TS
- Usage de l'attribut dynamique [formControl]
- Regroupement via FormGroup
- Validation via FormControl
- Tous les contrôles via les objets TS
- On dit que « la source de vérité » est dans l'objet FormControl

Template-driven

- Bien plus « artisanal »
- Pas d'objet dédié
- Binding classique
  - Two-way binding le plus souvent [(ngModel)] également
- Gestion des événements à faire
- Validation via HTML natif
- On dit que « la source de vérité » est dans le template
  - Dans l'exemple, #actorForma est une variable de template

```
export class ActorFormComponent {

  skills = ['Method Acting', 'Singing', 'Dancing', 'Swordfighting'];


  model = new Actor(18, 'Tom Cruise', this.skills[3], 'CW Productions');


  submitted = false;


  onSubmit() {

    this.submitted = true;

  }
```

```
<div class="form-group">
  <label for="skill">Skill</label>
  <select class="form-control" id="skill" required>
    <option *ngFor="let skill of skills" [value]="skill">{{ skill }}</option>
  </select>
</div>
```

```
<form #actorForm="ngForm">
```