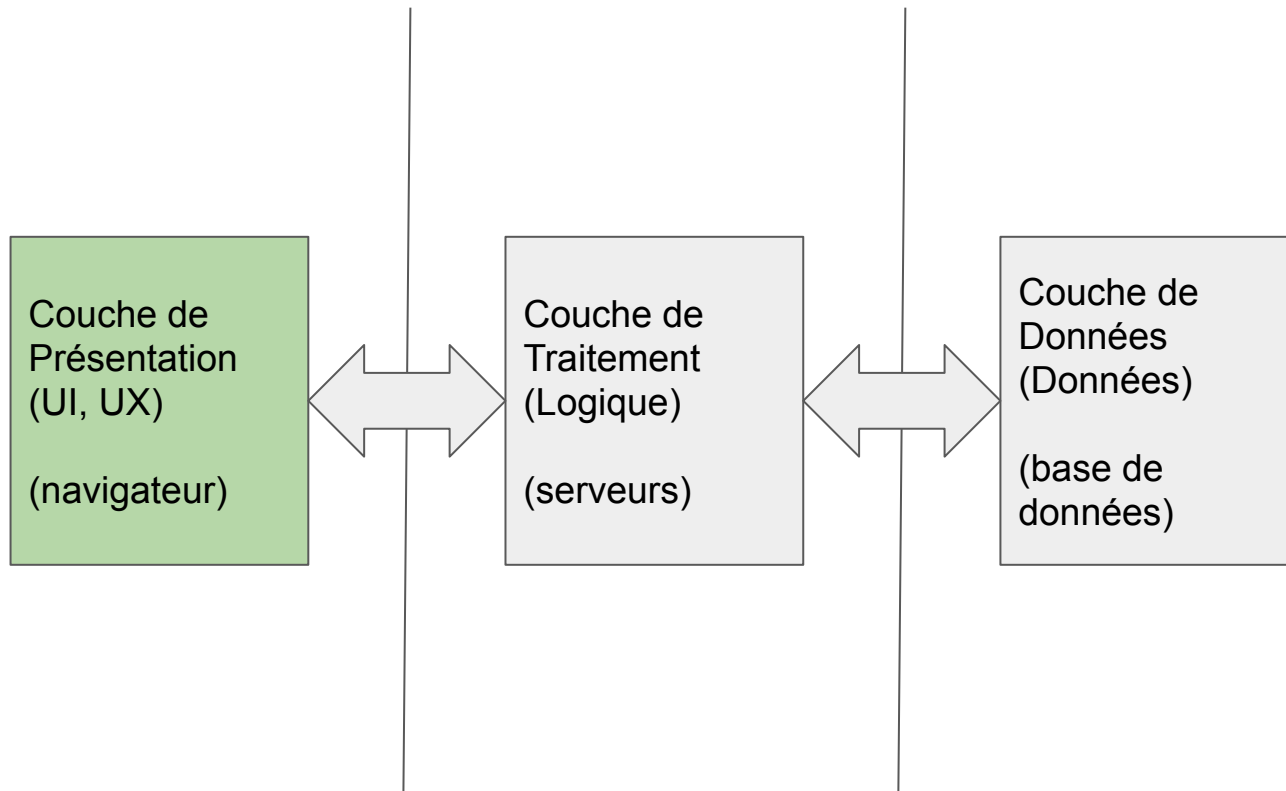


Couche présentation

# Plan et objectifs

- Identifier les éléments de la couche de présentation
- Différencier UX et UI
- Exemple de norme d'interface utilisateur
- Conception d'application monopage et cadriciel
- Composants et Modules
- Services et Modèle
- Navigation
- Options de stockage web local



# Les éléments de la couche de présentation

La couche de présentation est le **point de contact entre les utilisateurs et l'application** (UI).

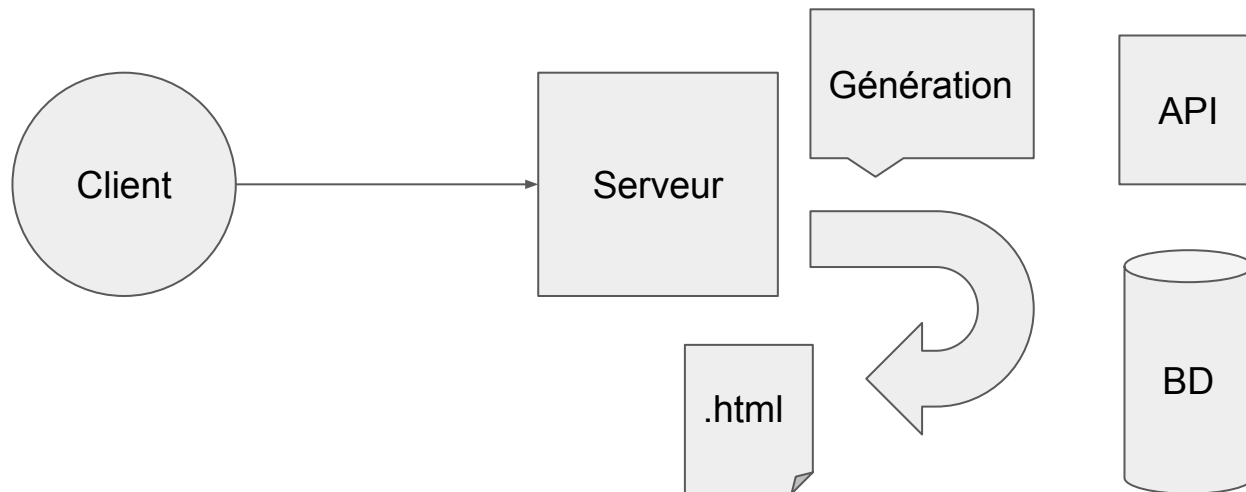
Cette couche s'occupe de l'**affichage** et de l'**interaction** avec les utilisateurs.

Elle **communique avec la couche de traitement** pour lire, traiter et mettre à jour les données.

Exécutée dans le navigateur exclusivement (dans le cas d'une application monopage) ou générée par le serveur (pages dynamiques avec gabarits)

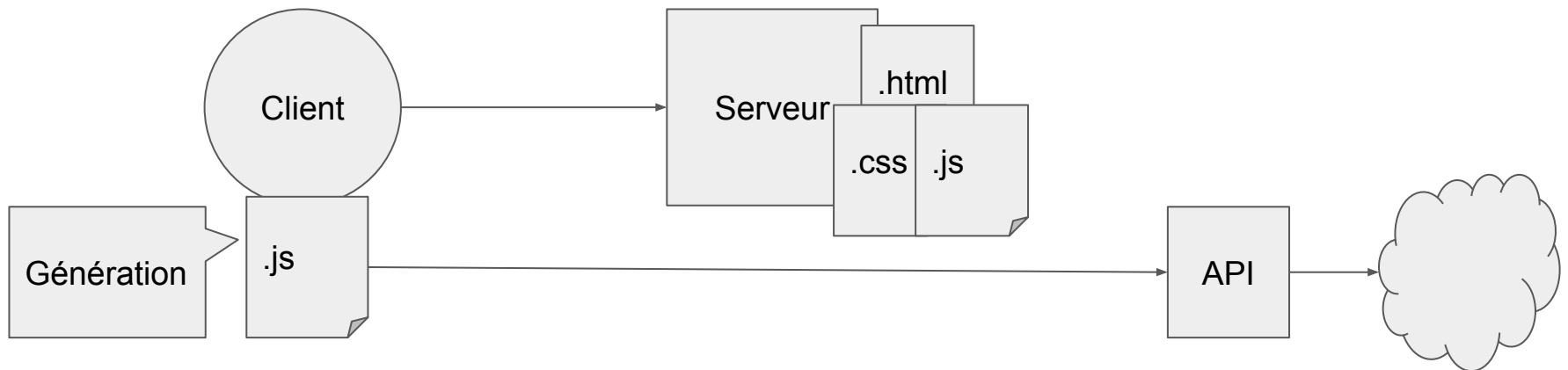
# Rappel: Sites dynamiques

Le **serveur** génère les pages web dynamiquement en réponse aux interactions de l'utilisateur.



# Rappel: Applications Monopage

Le client télécharge un ensemble de fichiers statiques depuis le serveur, puis le code javascript modifie l'apparence de la page dynamiquement et effectue des appels aux APIs pour charger (et modifier) les données.



# UI et UX - la différence

*L'**interface utilisateur** (UI) est un dispositif matériel ou logiciel qui permet à un usager d'interagir avec un produit informatique. C'est une interface informatique qui coordonne les interactions homme-machine, en permettant à l'utilisateur humain de contrôler le produit et d'échanger des informations avec le produit.*

[https://fr.wikipedia.org/wiki/Interface\\_utilisateur](https://fr.wikipedia.org/wiki/Interface_utilisateur)

*L'**expérience utilisateur** (UX) est la qualité du vécu de l'utilisateur dans des environnements numériques ou physiques. C'est une notion de plus en plus courante là où l'on utilisait, encore récemment, les notions d'ergonomie des logiciels et d'utilisabilité.*

[https://fr.wikipedia.org/wiki/Exp%C3%A9rience\\_utilisateur](https://fr.wikipedia.org/wiki/Exp%C3%A9rience_utilisateur)

# Normes pour les interfaces utilisateur

*Le **Material Design** est un ensemble de règles de design proposées par Google et qui s'appliquent à l'interface graphique des logiciels et applications.*

<https://material.io/>

Des implémentations spécifiques pour divers cadres sont disponibles.

Angular - <https://material.angular.io/>

React - <https://v4.mui.com/>

[https://fr.wikipedia.org/wiki/Material\\_Design](https://fr.wikipedia.org/wiki/Material_Design)



# Exploration des éléments Material Design

<https://material.angular.io/components/>

Pour chaque composant, il est possible de voir des exemples de code pour les intégrer facilement dans vos applications.

Avantages:

- Éléments de UI fonctionnels réutilisables
- Assure une cohérence entre les éléments visuels

Désavantages:

- Éléments complexes et parfois lourd
- Pas toujours simple à modifier

**CADRICIELS**

**PARTOUT DES CADRICIELS!**

# Pourquoi utiliser un cadriciel frontend?

Une application monopage utilise le code javascript pour manipuler le DOM et créer une interface utilisateur dynamique.

Il est possible de n'utiliser que le HTML, CSS et javascript pour concevoir une application monopage, mais il devient rapidement fastidieux de programmer de manière déclarative les opérations sur le DOM.

Les cadriciels ajoutent une couche d'abstraction qui permet aux programmeurs d'utiliser des concepts de plus haut niveau, ce qui améliore la productivité.

Aussi, les cadriciels (ou leur écosystème) offrent des solutions pour les fonctionnalités les plus fréquentes (e.g. navigation), ce qui diminue le temps de développement.

Certains cadriciels structurent la base de code et favorisent un ensemble de pratiques de développement, ce qui est un avantage considérable pour des grandes équipes de développement.

# Pourquoi ne pas utiliser un cadriciel frontend?

Par contre, l'apprentissage d'un cadriciel et de ses bonnes pratiques prend du temps (courbe d'apprentissage).

De plus, le cadriciel est souvent conçu pour supporter un ensemble de scénarios d'utilisation et il peut être difficile de déborder du 'cadre'.

Comme le cadriciel cache une partie de la complexité sous son API, il est parfois ardu de déboguer quand le problème provient d'un mauvais usage de l'API ou, pire, d'un bogue dans le cadriciel.

Il faut donc s'assurer que les bénéfices sont supérieurs aux limitations avant de sélectionner un cadriciel.

En général, c'est le cas.

*We're not designing pages, we're designing  
systems of components.*

*— Stephen Hay*

# Conception par composants

La majorité des cadres de développement frontend utilise une approche de conception par composants.

Un composant est:

- réutilisable (peut être utilisé dans différents scénarios)
- combinable (peut être combiné avec d'autres composants)
- remplaçable (peut être échangé contre un autre composant offrant la même interface)
- encapsulé (une fonctionnalité bien définie)
- indépendant (minimum de dépendances)

# Cycle de vie des composants


Le cycle de vie des composants offre plusieurs accroches pour que les développeurs puissent insérer du code en fonction du stade dans le cycle de vie.

Chaque cadriciel offre une ensemble différents d'accroches, mais en général nous retrouvons les suivants:

- création
- initialisation
- mise à jour
- destruction

# Composants en développement frontend

Pour le développement frontend, les composants servent à définir des éléments UI et combinent le HTML, CSS et javascript nécessaires à cet élément.


 Material


Components

CDK

Guides

14.2.1



 GitHub

Components

Autocomplete

Badge

Bottom Sheet

Button

Button toggle


Card

Checkbox

Chips


Core

Angular Material offers a wide variety of UI components based on the [Material Design specification](#)



**Autocomplete**

Suggests relevant options as the user types.



**Badge**

A small value indicator that can be overlaid on another object.



# Sans état vs Avec état (stateless, stateful)

Un composant sans état ne conserve aucune donnée. Son comportement est toujours le même et complètement défini par ses entrées (inputs). Donc, pour les mêmes entrées, les sorties (outputs) seront toujours les mêmes.

Au contraire, le comportement un composant avec état varie selon l'état du composant. Donc, pour les mêmes entrées, les sorties peuvent être différentes.

Un composant sans état a plusieurs avantages:

- plus simple à comprendre
- plus simple à tester
- parallélisable
- remplaçable

# Immutabilité

Un objet immuable est un objet dont l'état ne peut pas être modifié après sa création. Par définition, un objet immuable est sans état.

L'utilisation d'objets immuables simplifie la compréhension d'un programme, car ils réduisent les cas possibles lors de l'exécution du code.

Un objet immuable a aussi la propriété de pouvoir être utilisé concurremment par plusieurs processus (threads) sans risque.

Lorsqu'on partage des objets entre plusieurs composants, il est préférable d'utiliser des objets immuables ou de faire des copies pour éviter les effets de bord et autres bogues difficiles à reproduire.

# Gérer l'état de l'application

En développement front-end, et plus particulièrement pour les applications monopage, la complexité vient de la gestion de l'état de l'application.

L'état de l'application est composé de plusieurs éléments liés au domaine de l'application (est-ce que l'utilisateur est connecté? est-ce que les produits sont chargés?), mais aussi lié au UI lui-même (le bouton est activé ou non? est-ce que la barre de navigation est visible ou non?).

# Composants de présentation / simple

Pour simplifier la programmation par composants au niveau de l'application frontend, il est souvent utile d'identifier deux types de composants qui ont des caractéristiques différentes.

Les composants de présentation (ou simple):

- responsables de l'**affichage** de l'élément UI
- contiennent un **minimum d'état** (variables) nécessaire à l'affichage du composant (e.g. interrupteur à bascule)
- peuvent être **paramétrable** via des inputs (e.g. texte, taille, style, etc)
- **délèguent la logique d'affaire** des événements (e.g. clic sur un bouton) via des outputs

Ces composants ne devraient pas:

- charger des données
- connaître la logique d'affaire

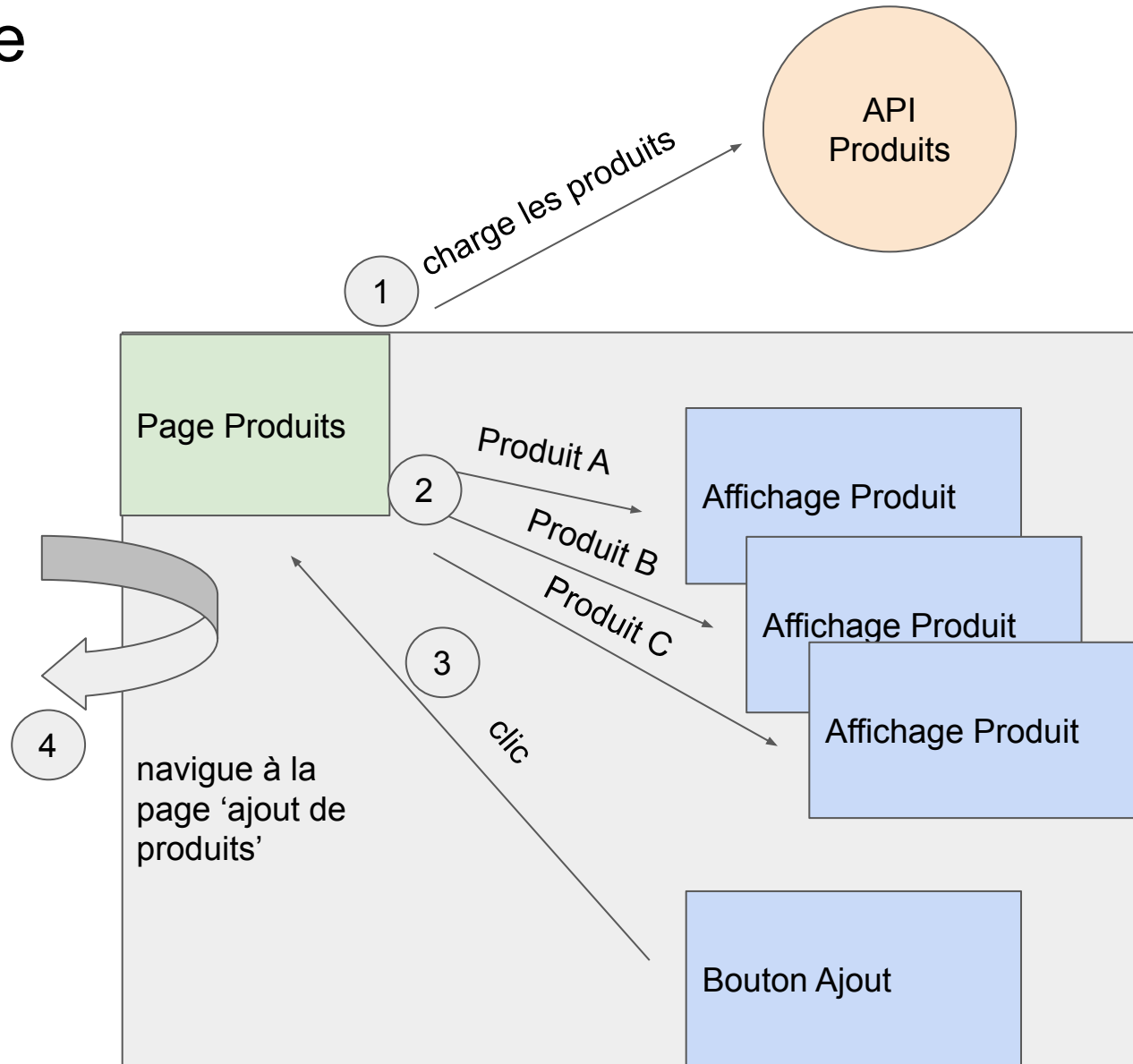
# Composants conteneurs / intelligents

À l'inverse, les composants conteneurs ou intelligents sont responsables d'orchestrer l'agencement de plusieurs composants.

Ces composants, souvent à la racine de la structure de composants,

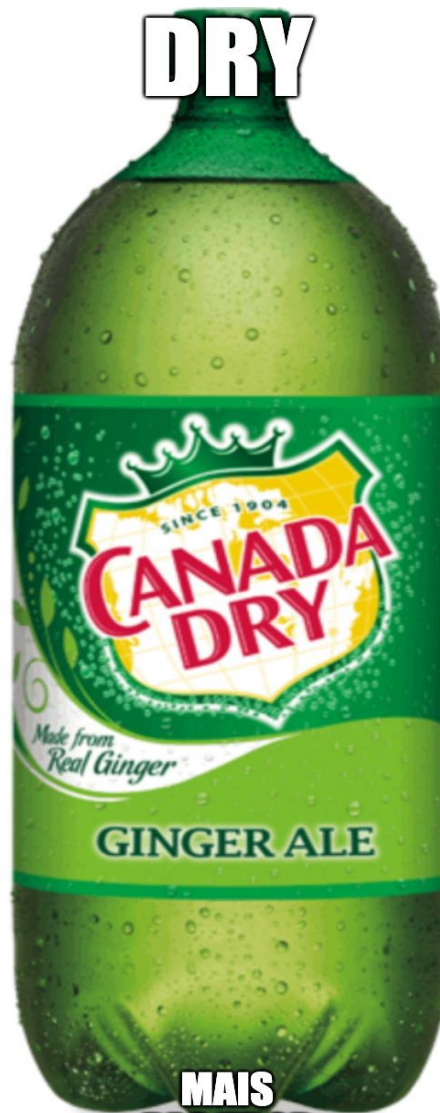
- **fournissent l'état** de l'application aux composants de présentation (via inputs)
- appellent les **APIs**
- appliquent la **logique d'affaire** en réponse aux différents événements du cycle de vie ou des interactions de l'utilisateur

# Exemple



Gardez vos composants **simples!**

**DRY**



**MAIS  
PAS TROP**

# Structure!

Il est important de bien structurer la base de code pour s'y retrouver lorsque le nombre de composants grandit, mais aussi pour l'intégration de nouveaux membres dans l'équipe de développement.

Structure par type ou par fonctionnalité? L'important c'est d'être consistant.

```
app/  
  components/  
    login.comp  
    display-product.comp  
    cool-button.comp  
  helpers/  
    constants  
  pages/  
    dashboard.page  
    products.page  
  services/  
    login.service  
    products.service
```

```
app/  
  dashboard/  
    dashboard.page  
  common/  
    constants  
    cool-button  
  login/  
    login.comp  
    login.service  
  products/  
    display-product.comp  
    products.page  
    products.service
```



# Structure!

La structure par fonctionnalité évolue plus facilement que la structure par type lorsque le projet prend de l'ampleur.

En général, une tâche de développement se concentre sur une fonctionnalité précise, donc si on utilise la structure par fonctionnalité l'ensemble de modifications se retrouve dans une ensemble de fichiers groupés.

<https://angular.io/guide/styleguide>

<https://fr.reactjs.org/docs/faq-structure.html>

# Services en développement frontend

Les services servent à encapsuler des fonctionnalités qui n'ont pas d'impact direct sur l'affichage. Ils sont souvent utilisés pour accéder aux données via des appels API ou pour encapsuler une logique commune à plusieurs composants.

Les services sont de simples classes (pas de HTML ni de CSS).

Selon le cadriciel utilisé, les services peuvent être injectés à la construction d'un composant et avoir un cycle de vie différent selon s'ils sont globaux ou limités à un module ou un composant.

# Détection de changements et mise à jour du DOM

Les différents cadres utilisent des approches différentes pour détecter les changements et déclencher la mise à jour du DOM.

Angular parcourt l'arbre des composants lors de certains événements (e.g. clique, minuteur, résolution d'une promesse) et évalue si leur état a changé. Si oui, *Angular* met à jour le DOM. Il est possible d'utiliser une stratégie différente (*OnPush*) qui limite les cas où *Angular* va réévaluer l'état d'un composant (changement d'Input, événements) et ainsi améliorer les performances.

*Vue* et *React* utilisent un mécanisme qui intercepte les changements d'état. Ils utilisent un DOM virtuel avant de le synchroniser avec le 'vrai' DOM.

Démo - <https://github.com/coderunner/ng-change-detect>

# Angular

Angular est un cadre de développement pour construire des applications monopages conçu par Google (1<sup>ère</sup> version 2016).

- Typescript
- Construction par composants
- Offre une multitude de bibliothèques
- Offre une suite d'outils de développement puissante (CLI)
  - automatise la création de projets, services et composants
  - automatise la construction du projet pour la production

Différent d'AngularJS (2010).

# Pourquoi Angular?

Les raisons pour lesquelles Angular est proposé comme cadre frontend pour ce cours:

- familiarité (INF3190)
- très structuré
- illustre bien la conception par composants
- grande communauté (toutes vos réponses se trouvent sur le web)

# Typescript

<https://www.typescriptlang.org/play>

<https://gist.github.com/coderunner/372f2a4af8c64a00a126648976831192>

<https://www.typescriptlang.org/docs/handbook/typescript-in-5-minutes.html>

# Démarrer avec Angular

Suivre les instructions d'installation sur <https://angular.io/guide/setup-local>.

- Requiert **nodejs** et **npm**.

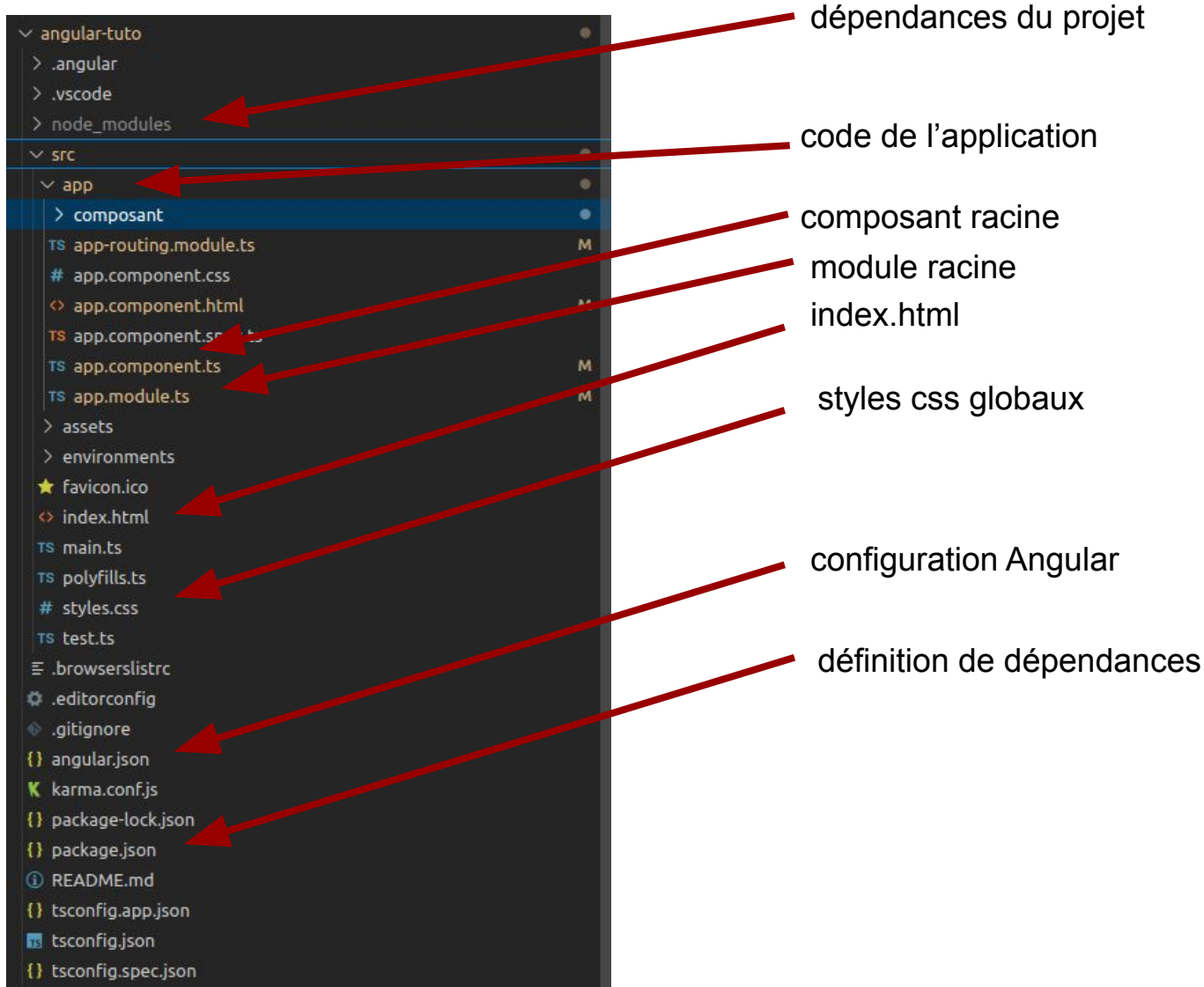
**nodejs** est un environnement d'exécution pour javascript en dehors d'un navigateur web.

**npm** est un gestionnaire de dépendance pour l'écosystème javascript.

**Angular** vient avec un outil de ligne de commande (CLI) qui facilite le développement et qui permet de simplifier, entre autres, la création de projets et l'ajout de composants et de services (<https://angular.io/cli>).

Pour le développement, il est pratique d'utiliser **ng serve**, qui exécute le projet angular et qui recharge automatiquement les changements effectués dans le code.

# Anatomie du projet Angular

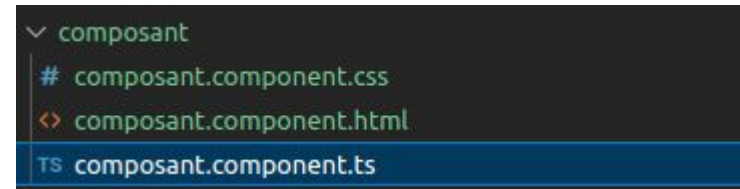




# Composant

Un composant Angular est composé de

- HTML (x.component.html)
- CSS (x.component.css)
- Typescript (x.component.ts)



Identifié par l'annotation `@Component` sur la *class* dans le fichier *ts*.

Les attributs et méthodes publiques du composant sont accessibles dans le HTML du composant.

Les styles définis dans le fichier *css* ne s'appliquent qu'au composant.

# Module

Un module Angular contient plusieurs éléments et constitue un groupement logique.

Le module expose certains composants.

Pour utiliser les composants d'un module, on doit l'inclure.

```
@NgModule({
  declarations: [
    AppComponent,
    ComposantComponent,
    PersonsComponent,
    PersonComponent,
  ],
  imports: [BrowserModule, AppRoutingModule, FormsModule, ReactiveFormsModule],
  providers: [],
  bootstrap: [AppComponent],
})
export class AppModule {}
```

# Démo - Partie 1

<https://github.com/coderunner/angular-tuto>

Si vous clonez le code, n'oubliez pas d'exécuter npm install avant ng serve.

1. définition d'un composant
2. interpolation
3. *pipe*
4. [(ngModel)]
5. gestion d'événements
6. \*ngIf
7. \*ngFor
8. ng-template et #
9. cycle de vie

# Démo - Partie 2

- 10. model
- 11. service
- 12. observable
- 13. composant enfant : Input() et Output()
- 14. composant parent
- 15. injection de dépendances
- 16. formulaire
- 17. subscription
- 18. async

# Démo - Partie 3

- 19. #
- 20. ViewChild
- 21. Projection de contenu

# Navigation

Angular fournit un service de navigation: Router.

Ceci permet d'associer des URLs différents aux différentes pages sans effectuer de requête au serveur.

Le Router intercepte les changements d'URLs et affiche le composant correspondant dans l'élément `<router-outlet>`.

```
const routes: Routes = [  
  { path: 'composant', component: ComposantComponent },  
  { path: 'persons', component: PersonsComponent },  
  { path: '**', component: ComposantComponent },  
];  
@NgModule({  
  imports: [RouterModule.forRoot(routes)],  
  exports: [RouterModule],  
})  
export class AppRoutingModule {}  
|
```

# Gestion de l'état de l'application

La gestion de l'état d'une application frontend peut devenir compliquée lorsque la taille et la complexité de l'application augmente.

Dans la mesure du possible, il est préférable de conserver l'état dans le composant qui l'utilise.

Pour les variables d'état qui doivent être partagées entre plusieurs composants, une solution simple consiste à utiliser un **service** qui est accédé par ces composants.

L'utilisation d'`Observable` facilite la transmission des changements d'état aux composants intéressés.

Il est aussi possible d'utiliser des bibliothèques spécialisées (type *redux*) pour centraliser et gérer l'état d'une application complexe, mais attention à la complexité ajoutée.

# Stockage web local

Deux options pour stocker des données dans le navigateur (local storage et session storage).

- API très simple
- Peut stocker jusqu'à 5MB de données (aucun transfert au serveur) par origine (principe d'origine identique [protocol, domain, port])
- Les données dans le LocalStorage n'ont pas d'expiration
- Les données dans le SessionStorage expirent lorsque la session se termine (quand le navigateur est fermé)



# IndexedDB

*IndexedDB est une API de bas niveau qui permet le **stockage côté client** de **quantités importantes de données structurées**, incluant des fichiers/blobs. Cette API utilise des index afin de permettre des **recherches performantes** sur ces données.*

*IndexedDB vous permet de **stocker et de récupérer des objets qui sont indexés** avec une **clef**.[...] Vous devez spécifier le schéma de la base de données, ouvrir une connexion à votre base de données, puis récupérer et mettre à jour des données dans une série de **transactions**.*

# Aide-mémoires et références pour Angular/RxJS

<https://angular.io/guide/cheatsheet>

<https://rxjs.dev/guide/overview>

<https://rxmarbles.com/>