ORSYS formation

Angular, développement avancé.

Intervenant :

Renaud Dubuis

View profile

LinkedIn

Approche pédagogique.

Participants. (Tour de table)

Le tour de table initial favorise la création du groupe et permet la contextualisation des réponses aux questions individuelles.

Evaluation des pré-requis.

Le tour de table initial et les premiers exercices ont pour but l'évaluation effective des participants au regard des pré-requis.

Récapitulatif (matinal).

Chaque journée commence par un exercice collectif de restitution des concepts abordés la veille. L'objectif étant de renforcer (répétition) les connaissances acquises pour les utiliser comme socle lors de la journée à venir.

Concertation personnelle.

Le formateur passera assister les participants individuellement aussi souvent que possible.
Les participants sont invités à le solliciter pendant la journée de formation ou pour revenir sur un point particulier en fin de journée.

Travaux Pratiques.

L'acquisition des concepts abordés est découpée propotionnellement:
- **60%** de manipulation pratique.
- **40%** d'apports théoriques.

La mise en oeuvre des travaux pratiques se déroule en 3 phases, permettant la reproduction en autonomie des exercices.

1. Ennoncé > 2. Démonstration > 3. Manipulation

Version numérique du support de formation.

Pour renforcer le confort de lecture et de manipulation **(copier/coller, liens cliquables, coloration du code)** le support est également distribué en version numérique. **au format PDF.** **(sous license Creative Commons Attribution-ShareAlike 4.0 International (CC BY-SA 4.0)).**

Les références des sources utilisée pour la création de ce document pédagogique sont disponibles sur demande .

Présentation des participants.

Afin d'animer ensemble la formation et de constituer notre groupe il est utile de nous présenter en fournissant les informations lés pour le déroulement de ce stage.

> - Identité.
> - Rôle au sein du groupe.
> - Positionnement sur les pré requis.
> - Attente de cette formation.
> - Précision.

Exemple

(identité) Bonjour je suis Renaud Dubuis.

(rôle) Je suis l'animateur de notre formation. (pré requis) En tant qu'architecte Font-End je suis amené à en maîtriser les différents outils. Je connais trés bien HTML, CSS et JavaScript.

(attente) Je souhaite partager avec vous ces compétences par cette formation.

(précision) Etant dyslexique, j'ai parfois des soucis d'orthographe, je vous prie de m'en excuser.

Organisation pratique : repas, pause, temps de questions/réponses.

Lors d'une formation INTRA (dans les locaux de l'entreprise) les horaires sont adaptés en fonction de votre rythme habituels.

> **9:00 :** Début de formation.
>
> **10:30:** pause de la matinée (15 à 20 minutes)
>
> **Pause déjeuner:** (choisir un horaire et une durée)
>
> **15:30:** pause de l'après midi. (15 à 20 minutes)
>
> **17:00 :** Temps des questions spécifiques.
>
> **17:30 :** fin de journée.

Le récapitulatif matinal est noté et tenu à jour par le formateur dans un fichier nommé RECAPITULATIF.md. Ce fichier vous sera remis à la fin de la formation.

Les questions posées font soit:

1. L'objet d'une réponse immédiate.
2. L'objet d'une prise de note dans un fichier nommé QUESTIONS.md pour une réponse ultérieure avant les pauses ou en fin de journée, ou plus tard dans la formation.

Introduction du modèle de formation :

L’acquisition des concepts abordés est découpée proportionnellement :

- 60% de manipulation pratique.
- 40% d’apports théoriques.

La mise en oeuvre des travaux pratiques se déroule en 3 phases, permettant la reproduction en autonomie des exercices.

> 1. Ennoncé > 2. Démonstration > 3.Manipulation.

1. Ennoncé Verbal, avec l'appui du support de cours ou d'une documentation extérieure, telle qu'une documentation en ligne ou croquis.

2. Démonstration Le formateur illustre pratiquement le point expliqué.

> Durant ces deux premières phases il est recommandé de :
>
> Se concentrer sur le concept abordé. Prendre des notes.
>
> **NE PAS** essayer reproduire les manipulations en même temps que le formateur.

3. Manipulation Vous reproduisez les manipulations en vous appuyant sur vos prises de notes. **N'hésitez pas à solliciter le formateur pour vous assister.**

😉

Présentation de la structure du support de formation :

Dans la mesure du possible le support se décompose en trois points.

> 1. Théorie > 2. Exemple > 3. Validation.

1. Théorie Explications théoriques internes au support ou basées sur une documentation en ligne.

> **Pourquoi utilisée une documentation en ligne :**
>
> Les solutions de développement telles que **ionic** évoluent trés rapidement pour répondre au besoins techniques du marché et aux attentes des développeurs.
>
> Les traductions peuvent souvent en être approximative et très rapidement obsolètes. Beaucoup de solutions et de langages sont créés en langues anglaises **les concepts y sont souvent plus aisément présentés**.
>
> **Votre formation s'appuiera sur l'utilisation, la compréhension et l'usage de la documentation en ligne**

2. Exemple Exemple local ou illustré en ligne.

3. Validation Espace de validation basé sur des question écrites ou orales.

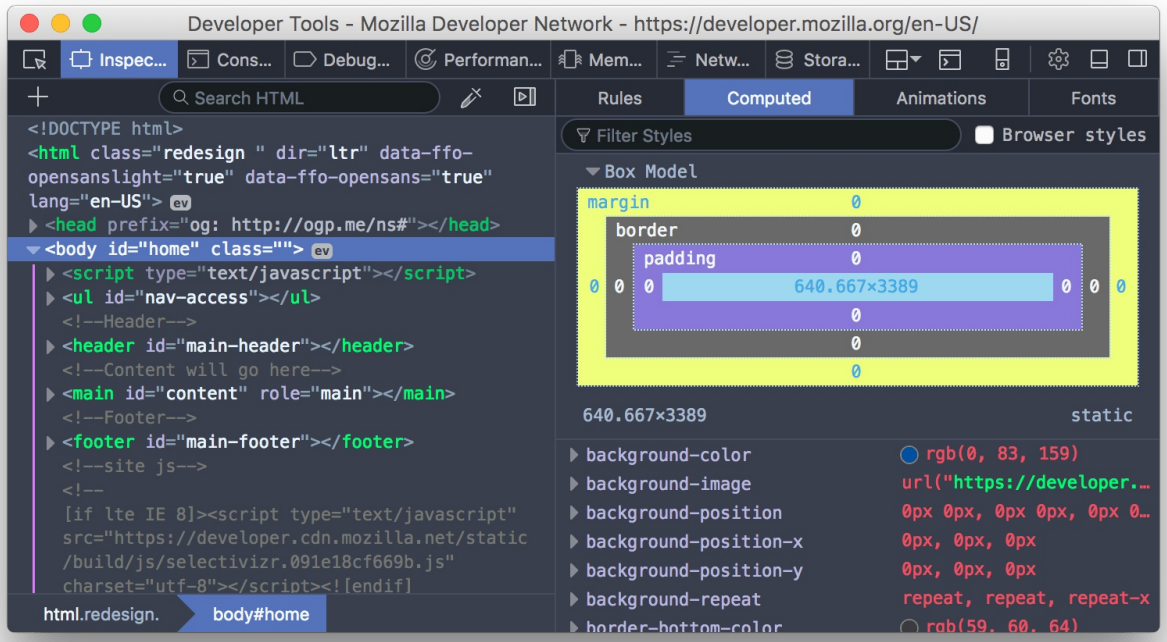
Configurer un environnement de développement moderne.

Pour programmer en JavaScript, un éditeur de texte suffit, il est intéressant de connaître l'offre en outillage autour de Node.

> **Un bon environnement tel aue Chrome** Examinez, modifiez, et déboguez du HTML, du CSS, et du JavaScript sur ordinateur, et sur mobile.

- Inspecteur
- Console web
- Débogueur JavaScript
- Moniteur réseau
- Performances

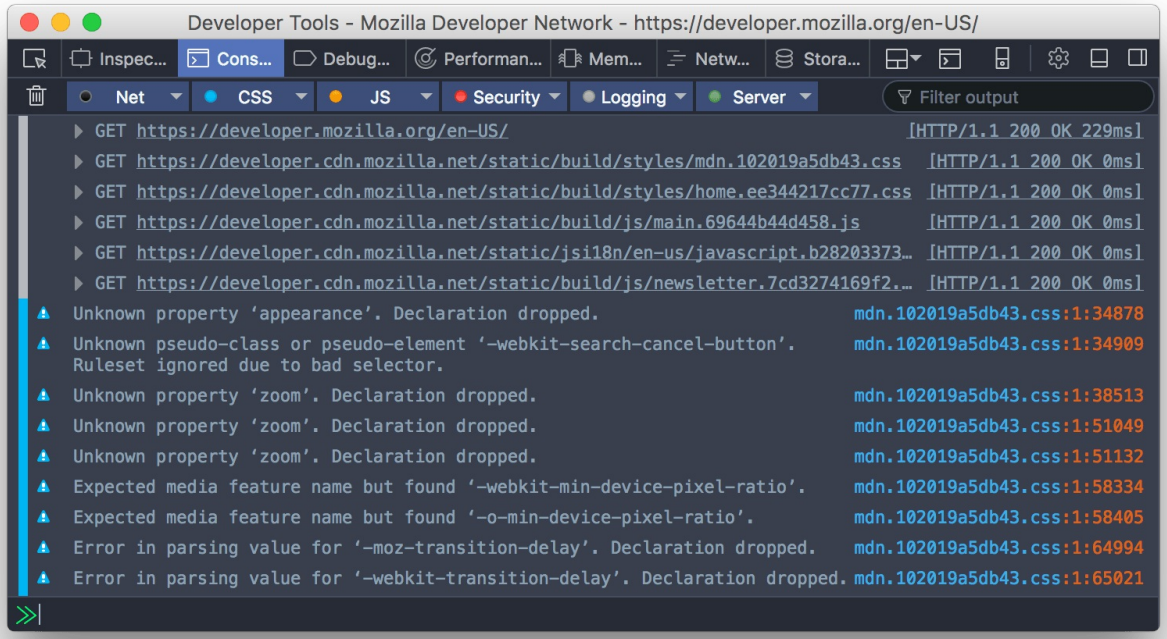
Inspecteur



Permet de voir et modifier une page en HTML et en CSS. Permet de

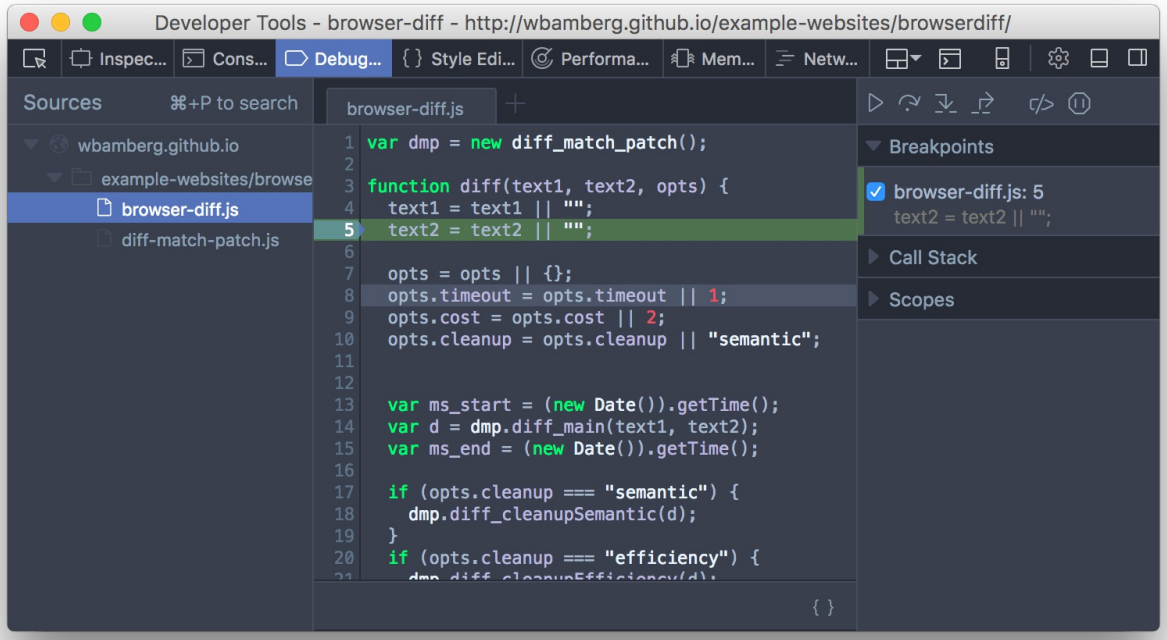
visualiser différents aspects de la page y compris les animations, l'agencement de la grille.

Console Web



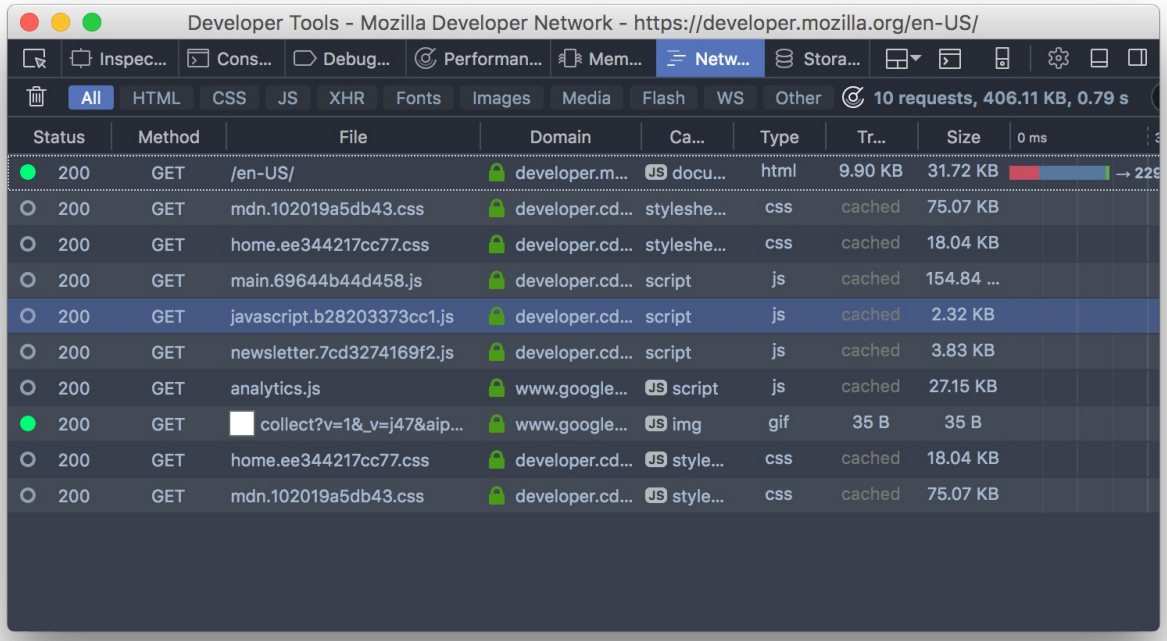
Affiche les messages émis par la page web. Permet également d'interagir avec la page via JavaScript.

Débogueur JavaScript



Permet de parcourir, stopper, examiner et modifier le code JavaScript s'exécutant dans une page.

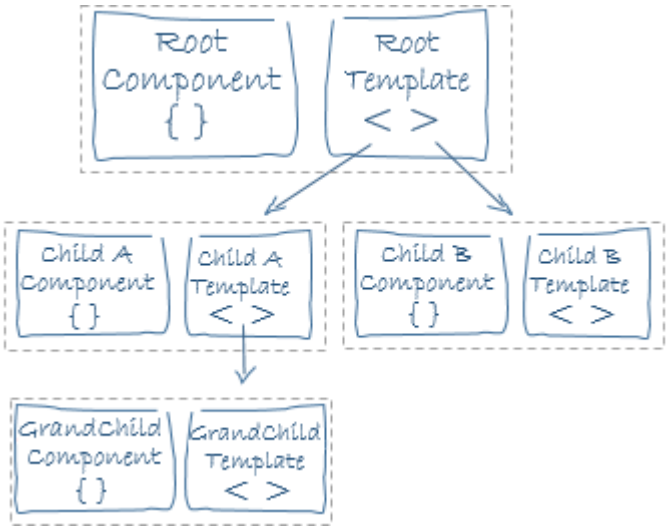
Réseau



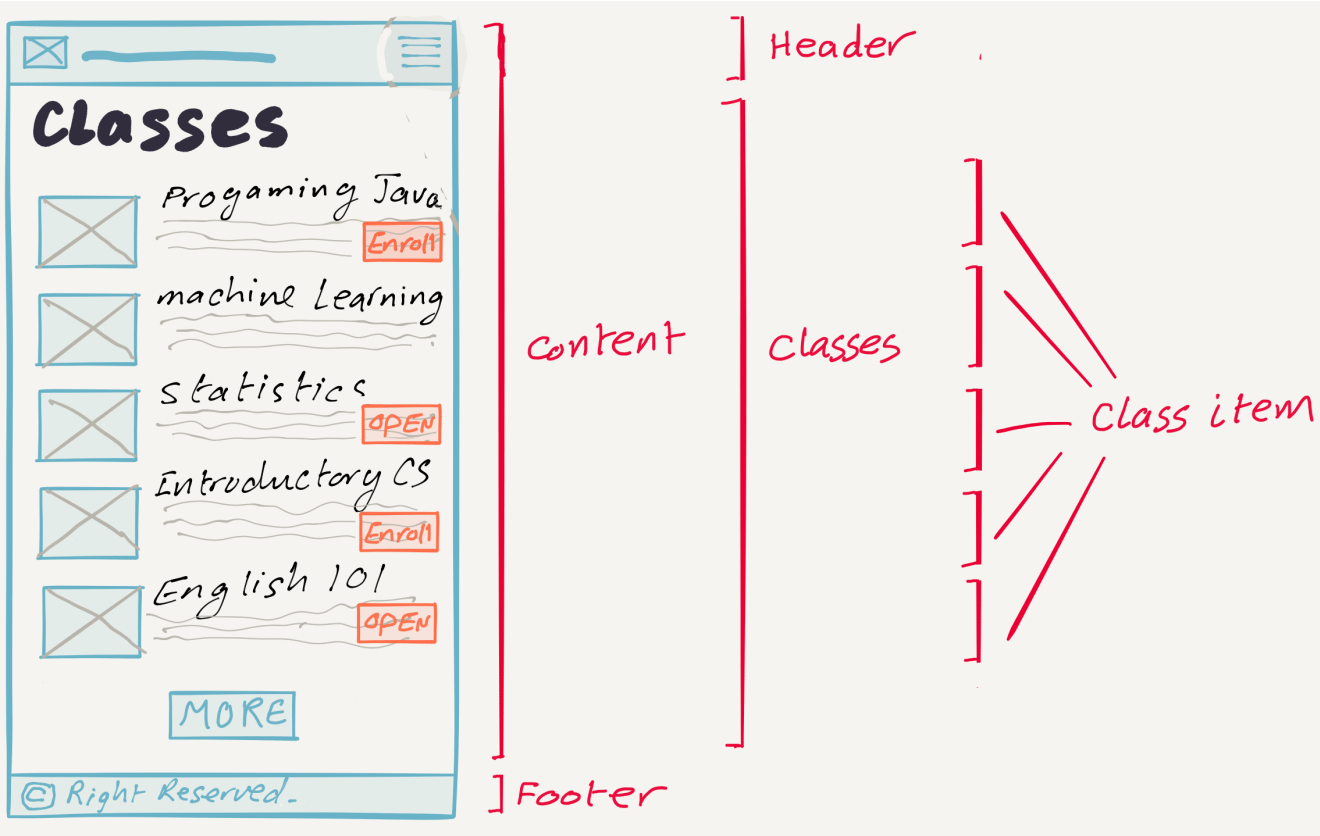
Permet d'inspecter les requêtes réseau lors du chargement de la page.

Angular, mise en œuvre des bonnes pratiques.

Le framework angular permet de concevoir une application comme une arborescence de composants.



Les composants définissent des unités fonctionnelles plus ou moins réutilisables au sein de l'interface utilisateur.



Références à l'ouvrage et autres références

La formation est illustrée par la projection d'une version numérique (pdf) de votre support et l'utilisation d'autres ressources pertinentes (site internet, démonstration).

Un livre complet (**voir ./documentation**) au format PDF édité par **rangle.io** vous est également délivrer sous licence **Creative Commons Attribution-ShareAlike 4.0 International (CC BY-SA 4.0)** LICENSE

Autres références

- angular.io
- learnangular2
- angular2
- angular-2-training-book.rangle.io
- node.js
- npmjs

Galaxie ng2

- cli.angular.io
- universal.angular.io
- mobile.angular.io
- augury.angular.io

Sanitization, Cross Site Scripting.

> Maintenir Angular up-to-date `ng update`

> Ne pas modifier Angular.

> Éviter lest Api marquéea dans la documentation comme “Risque pour la Sécurité.”

Angular’s cross-site scripting

Pour bloquer les attaques de type XSS, Angular traite toutes les valeurs comme non approuvées par défaut lorsqu'une valeur est insérée dans les DOM à partir d'un `temaplte`.

Cela signifie que les applications doivent empêcher que des valeurs qui peut permettre à un pirate de contrôle de jamais de le faire dans le code source d'un modèle.

> Ne jamais produire de modèle de code source par la concaténation de la saisie de l'utilisateur et des modèles.

Pour éviter ces problèmes, utilisez le mode hors connexion de modèle compilateur, également connu en tant que modèle d'injection.

Sanitization et les contextes de sécurité

> Nettoyage de valeurs, en transformant en une valeur sûre pour les insérer dans le DOM.

Angular définit les contextes de sécurité:

- HTML est utilisé lors de l'interprétation de valeur HTML, par exemple, lors de la liaison à innerHtml.
- Style est utilisé lors de la liaison de CSS.
- URL est utilisé pour les propriétés de l'URL, comme `<a href>`.
- Ressource URL est une URL qui sera chargé et exécuté en tant que code, par exemple, dans `<script src>`.

> Angular sanitizes les valeurs non approuvés pour HTML, styles, et URLs;

Internationalisation. Coding Guide Style.

Application internationalization is a many-faceted area of development, focused on making applications available and user-friendly to a worldwide audience.

Angular and i18n

Internationalization is the process of designing and preparing your app to be usable in different languages. Localization is the process of translating your internationalized app into specific languages for particular locales.

Angular simplifies the following aspects of internationalization:

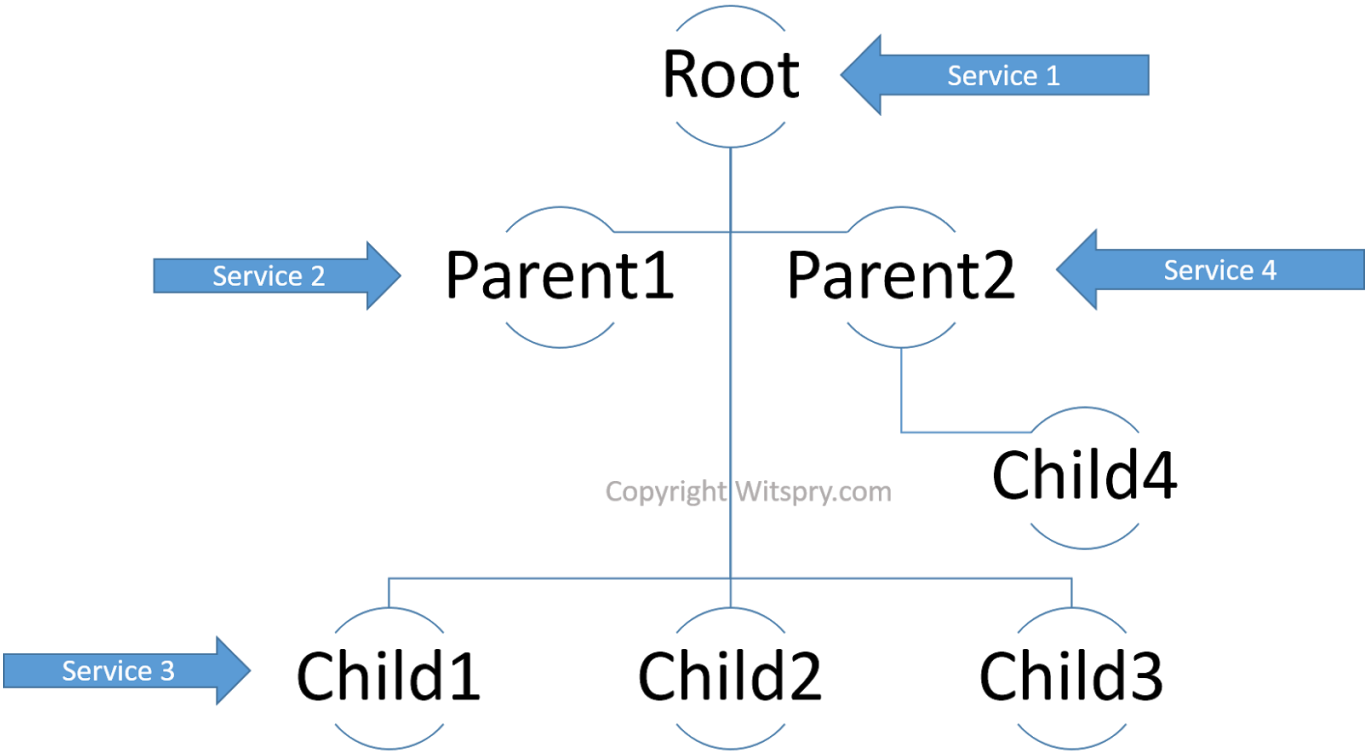
- L'affichage des dates, du nombre, des pourcentages et des devises dans un format local.
- La préparation du texte dans les modèles de composants pour la traduction.
- Manipulation des formes du pluriel de mots.
- La manipulation d'un autre texte.

> Pour la localisation, vous pouvez utiliser le CLI pour générer des fichiers a traduire, et publier l'application en plusieurs langues. Après avoir configuré votre application pour utiliser i18n, la CLI peut vous aider avec les étapes suivantes:

- L'extraction de texte localisables dans un fichier que vous pouvez envoyer à traduire.
- *Build* spécifiaue de l'application en utilisant le texte traduit.
- Création de plusieurs versions de langue de votre application.
- Configurer les paramètres régionaux de votre application
- Un jeu de paramètres régionaux est un identificateur (id) qui fait référence à un ensemble de préférences de l'utilisateur qui a tendance à être partagé au sein d'une région du monde.

> **Unicode locale identifier** est composé d'un langage Unicode identifiant et (éventuellement) le caractère suivi d'une extension locale. Par exemple, dans l'id de paramètres régionaux fr-CA le fr se réfère à la langue française, l'identificateur, et la CA fait référence à la localisation de l'extension Canada.

Injection de dépendances.



@Self()

3.

```
@Component({providers: [ToysService]})
```

ParentComponent

```
@Component({providers: [ToysService]})
```

KidComponent

```
constructor(
  @Self() toys: ToysService
)
```

@Optional()

4.

```
@Component({providers: [ToysService]})
ParentComponent
@Component({providers: []})
KidComponent
constructor(
  @Self() toys: ToysService
)
[ERROR] No provider for ToysService
```

5.

```
@Component({providers: [ToysService]})
ParentComponent
@Component({providers: []})
KidComponent
constructor(
  @Self() @Optional() toys: ToysService
) {
  toys === null; // no error though
}
```

@SkipSelf()

6.

```
@Component({providers: [ToysService]})
ParentComponent
@Component({providers: [ToysService]})
KidComponent
constructor(
  @SkipSelf() toys: ToysService
)
```

@Host()

7.

```
@Component({providers: [ToysService]})
ParentComponent
@Component({providers: [ToysService]})
KidComponent
constructor(
  @Host() toys: ToysService
)
```

8.

```
@Component({providers: [ToysService]})
ParentComponent
@Component({providers: []})
KidComponent
constructor(
  @Host() toys: ToysService
)
```

Types applicatifs partagés.

Il y a cinq catégories générales des modules de fonctionnalités qui ont tendance à tomber dans les catégories suivantes:

- Domain feature modules.
- Routed feature modules.
- Routing modules.
- Service feature modules.
- Widget feature modules.

Feature Module	Declarations	Providers	Exports	Imported by
Domain	Yes	Rare	Top component	Feature, AppModule
Routed	Yes	Rare	No	None
Routing	No	Yes (Guards)	RouterModule	Feature (for routing)
Service	No	Yes	No	AppModule
Widget	Yes	Rare	Yes	Feature

> *Domain feature modules* offrir une expérience utilisateur dédié à un domaine d'application, comme la modification d'un client ou d'une commande. Ils ont généralement un composant de huat niveau qui agit comme la fonction racine et privés, de sous-composants.

> *Domain feature modules* sont constitués principalement de déclarations. Seule la partie supérieure est exporté.

Domain feature modules sont généralement importés exactement une fois par une plus grande fonctionnalité.

> *Routed feature modules* sont des modules dont les composantes sont les objectifs de routeur routes de navigation. Tous les lazy-loaded modules sont *routed feature modules*.

Routed feature modules n'ont pas d'exports parce que leurs composants n'apparaissent jamais dans le modèle d'un composant externe.

> **Un lazy-loaded routed feature module ne doivent pas être importées par tout module.**

> Un *routing module* fournit la configuration de routage pour un autre module et se sépare de routage des préoccupations de son compagnon module.

Service modules fournit des services publics tels que l'accès aux données et de messagerie. Idéalement, ils sont entièrement constitués de *providers* et n'ont pas de déclarations.

> **La racine AppModule est le seul module qui devrait importer des modules de service.**

Un *widget module* expose les components, directives, et pipes aux autres modules.

- Un widget module doit être entièrement composé de déclarations, la plupart d'entre eux exportés.
- Un widget module doit rarement fournisseurs.

PWA : les Services Workers.

> Dans sa plus simple expression, un ***Service Worker*** est un script qui s'exécute dans le navigateur web et gère la mise en cache d'une application.

Les ***Service Worker*** assurent la fonction deproxy réseau. Ils interceptent toutes les requêtes HTTP effectuées par l'application et vous pouvez choisir la façon d'y répondre. Par exemple, ils peuvent interroger un cache local et d'offrir une réponse en cache si l'on est disponible.

Service workers in Angular

Les applications Angular, monopage, sont dans une position privilégiée pour profiter des avantages du service des travailleurs.

Angular's service worker est conçu pour optimiser l'expérience des utilisateurs finaux sur une connexion réseau lente ou instable, tout en réduisant au minimum les risques de servir du contenu obsolète :

- La mise en cache d'une application, comme l'installation d'une application native. L'application est mise en cache, comme une seule unité, et tous les fichiers de mise à jour ensemble.
- Une application en cours d'exécution continue de fonctionner avec la même version de tous les fichiers.
- Lorsque les utilisateurs actualisent l'application, ils voient les dernières version mise en cache.
- Les mises à jour se produisent dans le fond, assez rapidement après que les modifications sont publiées. La précédente version de l'application est servi jusqu'à ce qu'une mise à jour soit installée et prête.
- Les ressources sont téléchargées uniquement si elles ont changées.

> Le ***Service Worker*** charge un fichier de manifeste à partir du serveur.

Le manifeste décrit les ressources de cache et comprend des tables de hachage de tous les contenus du fichier. Lorsqu'une mise à jour de l'application est déployée, le contenu du manifeste de changement, doit être téléchargé et mis en cache.

Ce manifeste est généré à partir d'une interface de ligne de commande généré un fichier de configuration appelé `ngsw-config.json`.

Requêtes HTTP avancées.

> Détecter qu'une erreur s'est produite puis interprétater de l'erreur.

```
private handleError(error: HttpErrorResponse) {
  if (error.error instanceof ErrorEvent) {
    // A client-side or network error occurred. Handle it accordingly.
    console.error('An error occurred:', error.error.message);
  } else {
    // The backend returned an unsuccessful response code.
    // The response body may contain clues as to what went wrong,
    console.error(
      `Backend returned code ${error.status}, ` +
      `body was: ${error.error}`);
  }
  // return an observable with a user-facing error message
  return throwError(
    'Something bad happened; please try again later.');
};
```

> Parfois, l'erreur est temporaire et disparaîtra automatiquement si vous essayez à nouveau.

```
getConfig() {
  return this.http.get<Config>(this.configUrl)
    .pipe(
      retry(3), // retry a failed request up to 3 times
      catchError(this.handleError) // then handle the error
    );
}
```

Requesting non-JSON data

Toutes les Api ne retourne pas des données JSON.

```
getTextFile(filename: string) {
  // The Observable returned by get() is of type Observable<string>
  // because a text response was specified.
  // There's no need to pass a <string> type parameter to get().
  return this.http.get(filename, {responseType: 'text'})
    .pipe(
      tap( // Log the result or error
        data => this.log(filename, data),
        error => this.logError(filename, error)
      )
```

```
    );
}
```

Debouncing

> `switchMap()` a trois caractéristiques importantes.

1. Elle prend un argument de fonction qui renvoie un Observable.
2. Si une demande de recherche est encore en vol (comme lorsque la connexion est mauvaise), il annule la demande et envoie une nouvelle.
3. Il retourne des réponses à leur demande initiale de l'ordre.

```
this.packages$ = this.searchText$.pipe(
    debounceTime(500),
    distinctUntilChanged(),
    switchMap(packageName =>
      this.searchService.search(packageName, this.withRefresh))
  );
```

Intercepting requests and responses

`HTTP Interception` est une caractéristique majeure de `@angulaire/common/http`.

Avec l'interception, vous déclarez d'intercepteurs qui inspectent et/ou de transforment les requêtes HTTP à partir de votre application vers le serveur.

La même intercepteurs peuvent également inspecter et de transformer le serveur de réponses sur leur chemin de retour à la demande.

> Plusieurs intercepteurs forme une chaîne des gestionnaires de requête/réponse.

```
import { Injectable } from '@angular/core';
import {
  HttpEvent, HttpInterceptor, HttpHandler, HttpRequest
} from '@angular/common/http';

import { Observable } from 'rxjs';

/** Pass untouched request through to the next request handler. */
@Injectable()
export class NoopInterceptor implements HttpInterceptor {

  intercept(req: HttpRequest<any>, next: HttpHandler):
    Observable<HttpEvent<any>> {
```

```
    return next.handle(req);
  }
}
```

Providers

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

import { NoopInterceptor } from './noop-interceptor';

/** Http interceptor providers in outside-in order */
export const httpInterceptorProviders = [
  { provide: HTTP_INTERCEPTORS, useClass: NoopInterceptor, multi: true },
];
```

Lazy Loading.

1. Créer un `feature module`.
2. Créer le module de routage du `feature module`.
3. Configurer les routes.

```
const routes: Routes = [
  {
    path: 'customers',
    loadChildren: './customers/customers.module#CustomersModule'
  },
  {
    path: 'orders',
    loadChildren: './orders/orders.module#OrdersModule'
  },
  {
    path: '',
    redirectTo: '',
    pathMatch: 'full'
  }
];
```

Fonctionnement interne d'Angular.

Fonctionnement interne d'Angular.

> Angular utilise zone.js pour créer sa propre zone et l'écouter, cela signifie que le code exécutè à l'intérieur d'une application angular est automatiquement écouté.

La raison pour laquelle cela fonctionne est, à l'intérieur de ces zones, zone.js remplacements et augmente le méthodes natives– `Promises, timeouts...`

Chaque fois que vous appelez setTimeout, par exemple, vous appeler une version augmentée par la zone qui l'utilise pour garder un œil sur les choses.

https://angular.io/api/core/ChangeDetectionStrategy

> Change Detection mise à jour le DOM chaque fois que les données sont modifiées. Angular dispose de deux stratégies pour la Détection de Changement.

Dans sa stratégie par défaut, chaque fois que des données sont mutés ou modifiés, Angular va exécuter le changement de mettre à jour le DOM.

Dans sa stratégie onPush, Angular le changement sera exécuté uniquement lorsqu'une nouvelle référence est passée à @Input() de données.

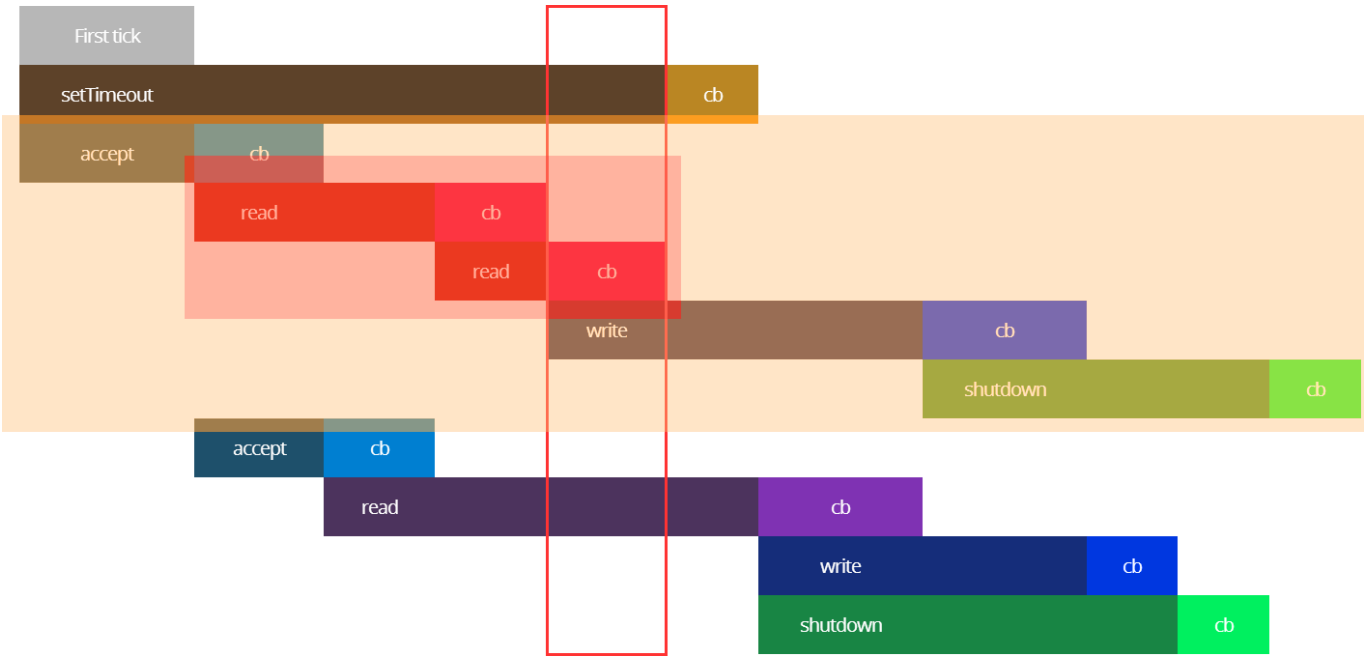
ZoneJS : le concept.

> https://angular.io/api/core/NgZone

Zones est un nouveau mécanisme qui permet aux développeurs de travailler avec plusieurs logiques connectées à des opérations asynchrones.

> ZoneJS de travaille en associant à chaque opération asynchrone avec une zone.

- Associer des données avec la zone,qui est accessible à toute opération asynchrone à l'intérieur de la zone.
- Suivre automatiquement l'encours des opérations asynchrones au sein d'une zone donnée pour effectuer un nettoyage ou le rendu ou l'assertion de test étapes
- Le temps le temps total passé dans une zone, à des fins d'analyse ou dans le domaine de profilage
- Gérer toutes les exceptions non traitées ou non traitées promesse de rejets à l'intérieur d'une zone, au lieu de les laisser se propager vers le haut niveau.



Optimisation des cycles de rendu, exécution hors ZoneJS.

L'utilisation la plus courante de ce service est d'optimiser les performances lors du démarrage d'une œuvre composée d'une ou de plusieurs tâches asynchrones qui ne nécessitent pas de mises à jour de l'INTERFACE utilisateur ou de la gestion d'erreur.

```
processOutsideOfAngularZone() {
    this.label = 'outside';
    this.progress = 0;
    this._ngZone.runOutsideAngular(() => {
      this._increaseProgress(() => {
        // reenter the Angular zone and display done
        this._ngZone.run(() => { console.log('Outside Done!'); });
      });
    });
  }
```

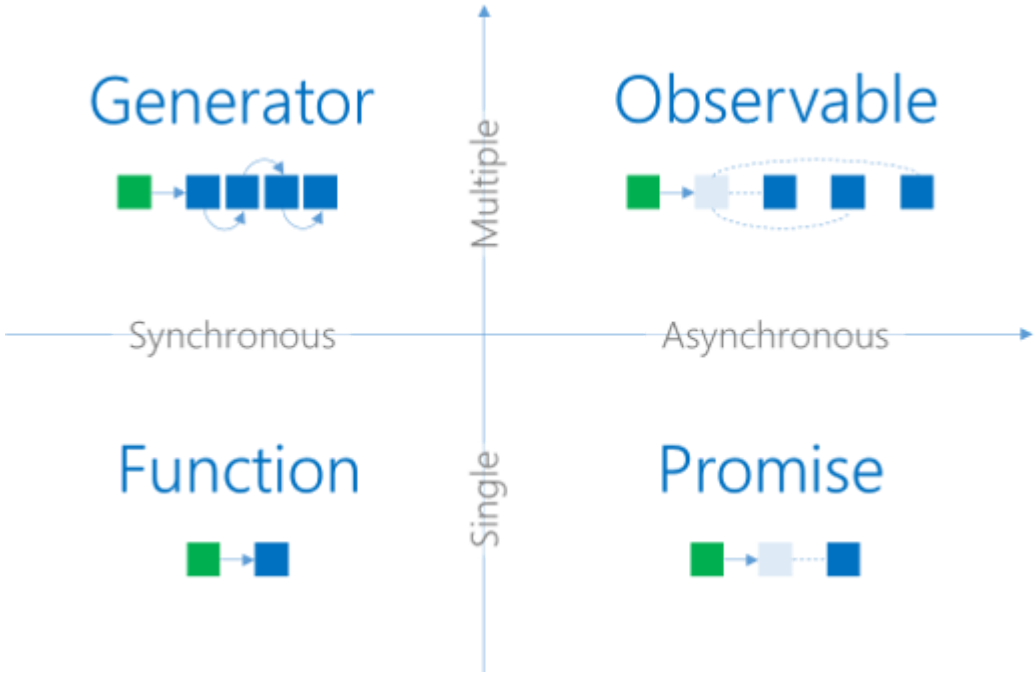
Choisir RXJS.

> **Définition** : Flux Evennementiels. La programmation réactive est un paradigme visant à conserver une cohérence d'ensemble en propageant les modifications d'une source réactive (modification d'une variable, entrée utilisateur, etc.) aux éléments dépendants.

Des librairies telles que **reactivex.io** posent de nouveau paradigmes tels que la représentation d'évennement sous la forme de flux itérables.

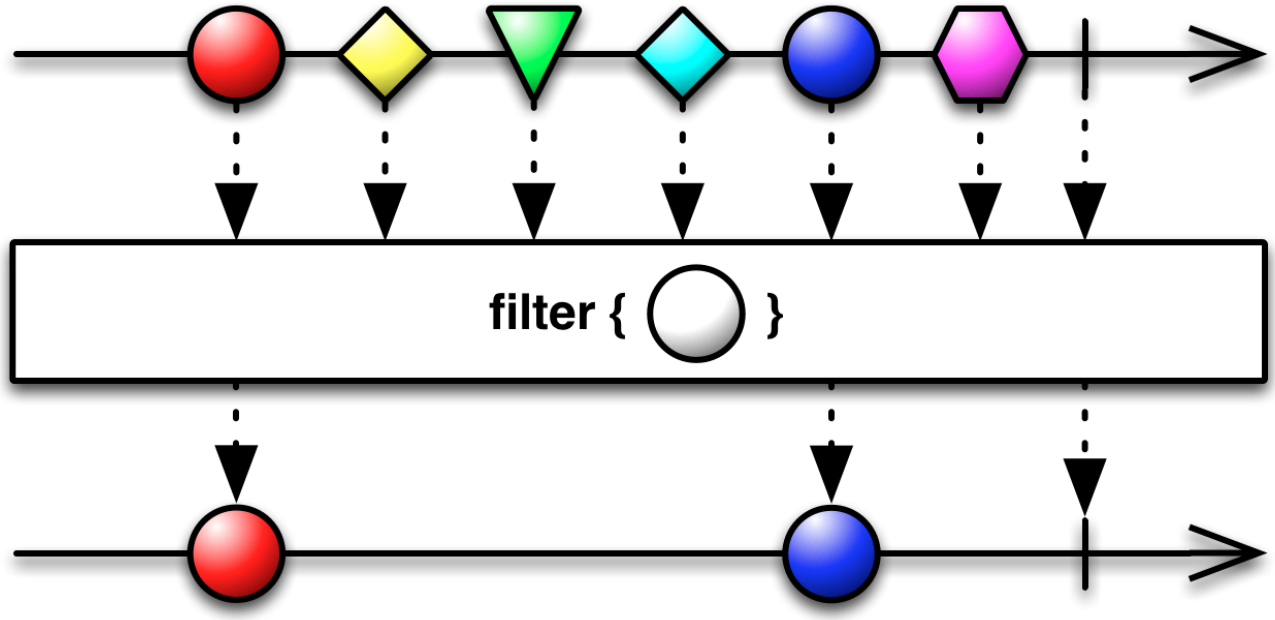
Ressources

- https://www.learnrxjs.io/
- http://reactivex.io/rxjs/



La librairie JavaScript RxJS permet de créer des flux de données qui arrivent au fil du temps, de les filtrer, de les modifier, de les combiner.
Les données d'un flux peuvent être des valeurs (string, number…) mais également des événements du DOM, des tableaux, des promises etc.

> **Concepts Clés :**
>
> - **Observable:** collection de valeurs futures.
> - **Observer:** collection de callbacks surveillant un `Observable`.
> - **Subscription:** association vers un `Observable`.
> - **Operators:** `pure functions` utilisées dans les operations telles `map, filter, concat, flatMap`, etc.
> - **Subject:** permet le `multicasting` vers de multiples `Observers`.
> - **Schedulers:** gestion centralisée des mise à jour.




```
//Exemple écouter un click
var button = document.querySelector('button');
Rx.Observable.fromEvent(button, 'click')
  .throttleTime(1000) // Un seul click par secondes
  .scan(count => count + 1, 0) // Création d'un compteur par la méthode scan
  .subscribe(() => console.log('Clicked!'));
```

Créer un Observable

Il existe plusieur méthode permettant la création d'un `Observable` selon la nature de la source originale

- **create** - Création personnalisée.
- **empty** - Collection terminée.
- **from** - A partir de `Array Iterable, Promise`
- **fromEvent** - A partir d' `Event`
- **fromPromise** - A partir de `Promise`
- **interval** - A partir d'un interval donné pour la fréquence d'émmision de valeur.
- **of** - A partir de valeurs de natures différentes.
- **range** - Pour une plage donnée.
- **throw** - Retourne un erreur.
- **timer** - Selon une fréquence de temps programmée.

Manipuler les séquences

RxJS permet la manipulation fine des traitement asynchrones à travers de trés nombreuses méthodes :

<table>
<tr><td colspan="3">Manipuler chaque valeur</td><td>map/select</td></tr>
<tr><td colspan="3">Accéder à une propriété de chacune des valeurs</td><td>pluck</td></tr>
<tr><td colspan="3">Notification de l'évolution des valeurs sans modification</td><td>do/tap
doOnNext/tapOnNext
doOnError/tapOnError
doOnCompleted/tapOnCompleted</td></tr>
<tr><td rowspan="5">Inclure des valeurs additionnelles</td><td colspan="2">Logique personnalisée</td><td>filter/where</td></tr>
<tr><td rowspan="2">Depuis le début de la séquence</td><td></td><td>take</td></tr>
<tr><td>Logique personnalisée</td><td>takeWhile</td></tr>
<tr><td colspan="2">A partir de la fin de la séquence</td><td>takeLast</td></tr>
<tr><td colspan="2">Jusqu'à la complétion d'une séquence différente</td><td>takeUntil</td></tr>
<tr><td rowspan="3"></td><td colspan="2">Toutes</td><td>ignoreElements</td></tr>
<tr><td rowspan="2">Depuis le</td><td></td><td>skip</td></tr>
<tr><td></td><td></td></tr>
</table>

Ignorer des valeurs	début de la séquence	Logique personnalisée	skipWhile
	A partir de la fin de la séquence		skipLast
	Jusqu'à la complétion d'une séquence différente		skipUntil
	de valeur identique		distinctUntilChanged
	trop fréquentes		throttle
Calculer	Somme	des valeurs	sum
	Moyenne		average
	Logique personnalisée	Retourner seulement le résultat final	aggregate reduce
		Retourner toutes les valeurs calculées	scan
Compte du nombre de valeurs			count
Verification de la présence d'une valeur			contains
Ajouter des `meta-data`	descriptifs		materialize
	contenant le temps écoulé depuis le dernier message		timeInterval
	avec timestamp		timestamp
Gérer l'inactivité	Lever une Erreur		timeout
	Changer de séquence		timeout
Vérifier l'unicité de valeur	Et lever une erreur		single
	et utiliser une valeur par défaut		singleOrDefault
Accéder seulement à	Et lever une erreur à défaut		first
	ou utiliser une valeur par défaut		firstOrDefault

la première valeur	Dans une période de temsp données		sample
Accéder seulement à la dernière valeur	Et lever une erreur à défaut		last
	et utiliser une valeur par défaut		lastOrDefault
I want to know if a condition is satisfied	by any of its values		some
	by all of its values		all/every
Retarder les messages			delay
		Logique personnalisée	delayWithSelector
Grouper les valeurs	Jusqu'a la complétion		toArray toMap toSet
	Logique personnalisée	as arrays	buffer
		as sequences	window
	Par taille	as arrays	bufferWithCount
		as sequences	windowWithCount
	Selon le temps	as arrays	bufferWithTime
		as sequences	windowWithTime
	Alternance Temps / Taille	as arrays	bufferWithTimeOrCount
		as sequences	windowWithTimeOrCount
	Selon une clé	jus'quà la completion	groupBy
		avec control du temps	groupByUntil

Compilation Ahead Of Time.

L'application se compose principalement des composants et de leurs ***template*** HTML. Parce que les composants et les ***template*** fournis ne sont pas compris par le navigateur directement, les applications nécessitent un processus de compilation avant de pouvoir les exécuter dans un navigateur.

Angular Ahead-of-Time (AOT) est un compilateur qui convertit le HTML et texte dans du code JavaScript au cours de la phase de construction avant que le navigateur télécharge et exécute le code. La compilation de votre application pendant le processus de génération permet d'accélérer le rendu dans le navigateur.

> Angular propose deux façons de compiler votre application:

> - Just-in-Time (JIT), qui est une compilation de votre application dans le navigateur lors de l'exécution.
> - Ahead-of-Time (AOT), qui est une compilation de votre application au moment de la construction.

```
ng build --aot
ng serve --aot
```

> https://angular.io/guide/aot-compiler

Tree Shaking.

Terme généralement utilisé dans le contexte JavaScript pour l'élimination du code *mort*. Il s'appuie sur la structure statique des module ES2015 , c'est à dire à l'importation et à l'exportation.

Le nom et le concept a été popularisé par le ES2015 module bundler **rollup**.

```
ng build --prod --build-optimizer
```

> Angular: la version 6 a introduit une nouvelle fonctionnalité, *Tree Shakeable Providers*.

Un moyen de définir les services et d'autres choses pour être utilisé par l'injection de dépendance d'une manière qui peut améliorer les performances de l'application.

Angular Tree Shaking Providers

With Tree Shaking Providers (TSP) on peut utiliser un mécanisme différent pour inscrire les services.

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

@Injectable({
  providedIn: 'root'
})
export class SharedService {
  constructor() { }
}
```

Webpack Bundle Analyzer.

Le fichier `stats.json` est une caractéristique de Webpack.

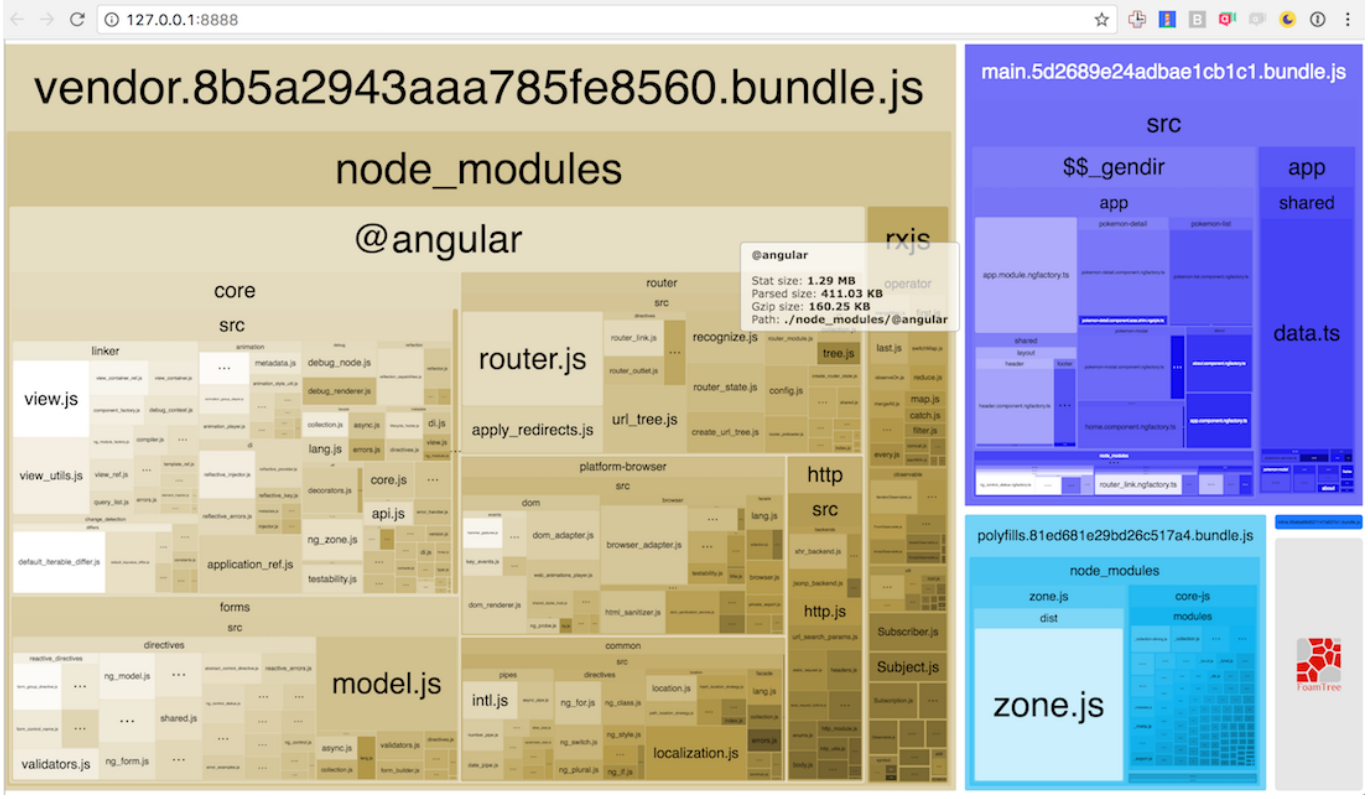
Webpack est le JavaScript package/bundler de l'outil que le CLI utilise sous le capot.

Avec ce fichier spécial Webpack généré pour nous, nous pouvons utiliser quelques outils pour comprendre notre application.

```
ng build --prod --stats-json.
npm install --save-dev webpack-bundle-analyzer
```

Une fois installé, ajoutez l'entrée suivante à la mnp scripts dans le `package.json`: "bundle-report": "webpack-bundle-analyzer dist/stats.json"

> Une fois ajouté, exécutez la commande suivante: `npm run bundle-report`



Création de composants distribuables.

Création de composants distribuables.

Cette commande va créer un projet de bibliothèque au sein de votre CLI espace de travail, et le configurer pour le test et le *build*.

```
ng g library <name> [-p PREFIX]
```

Building

```
ng build <library-name>
```

Once built yoou can run `npm pack` from the library `dist` folder.

A propos des Web Components:

Composants d'interface graphique réutilisables, qui ont été créés en utilisant des technologies Web libres.

Les Composants Web sont constitués de plusieurs technologies distinctes. Ils font partie du navigateur, et donc ils ne nécessitent pas de bibliothèques externes comme jQuery ou Dojo.

Un composant Web existant peut être utilisé sans l'écriture de code, en ajoutant simplement une déclaration d'importation à une page HTML. Les Composants Web utilisent les nouvelles capacités standards de navigateur, ou celles en cours de développement.

Les Composants Web sont constitués de ces quatre technologies (bien que chacun peut être utilisé séparément):

- Custom Elements: pour créer et enregistrer de nouveaux éléments HTML et les faire reconnaître par le navigateur,
- HTML Templates: squelettes pour des éléments HTML instanciables,
- Shadow DOM: ce qui sera public ou privé dans vos éléments,
- HTML Imports: pour packager ses composants (CSS, JavaScript, etc.)

Une démonstration minimaliste dans chrome :

```
<body>
    <simple-increment data-step="5"></simple-increment>
    <script>
    class SimpleIncrement extends HTMLElement {
        constructor(args) {
            super();
        }
        //Cycle de vie
        createdCallback() {
            this.count = 0;
            this.step = this.dataset.step;
            this.max = 100;
            this.innerHTML = `<button><i>${this.count}</i> of ${this.max}</button>`;
        }

        increment(){
            return (this.count < this.max) ?(this.count += Number(this.step)):this.cou
        }

        attachedCallback() {
            this.querySelector('button').addEventListener('click', (evt) => evt.target
        }
    }
    document.registerElement('simple-increment', SimpleIncrement)
    </script>
</body>
```

> **En résumé, un Web Component** est un fragment fonctionnel d'interface encapsulé dans :
> *Un modèle générique* `HTMLElement`.
> Un bloc logique `class` .
> *Un sytème de rendu* `document.registerElement` *ici le DOM.*
> Un cycle de vie exposé par le système de rendu.

Méthodologie : Interactive Component Sheet.

> Angular Elements package it's extremely easy to create native custom elements.

Afin d'avoir les `custom elements` disponibles nous avons besoin du `polyfill` installable depuis le CLI:

```
ng add @angular/elements
```

Create a component

Nous allons en créer pour voir comment ils se traduisent par des éléments personnalisés qui sont compris par les navigateurs:

```
ng g component button --inline-style --inline-template -v Native
```

```
@Component({
  selector: 'custom-button',
  template: `<button (click)="handleClick()">{{label}}</button>`,
  styles: [`
    button {
      border: solid 3px;
      padding: 8px 10px;
      background: #bada55;
      font-size: 20px;
    }
  `],
  encapsulation: ViewEncapsulation.Native
})
export class ButtonComponent {
  @Input() label = 'default label';
  @Output() action = new EventEmitter<number>();
  private clicksCt = 0;

  handleClick() {
    this.clicksCt++;
    this.action.emit(this.clicksCt);
  }
}
```

C'est la partie essentielle: nous utilisons la fonction `createCustomElement` pour créer une classe qui peut être utilisé avec les navigateurs nativement.

```
@NgModule({
  declarations: [ButtonComponent],
  imports: [BrowserModule],
  entryComponents: [ButtonComponent]
})
export class AppModule {
  constructor(private injector: Injector) {
    const customButton = createCustomElement(ButtonComponent, { injector });
    customElements.define('custom-button', customButton);
  }

  ngDoBootstrap() {}
}
```

Les décorateurs.

> Il existe quatre types principaux:

https://angular.io/api?type=decorator

- Class decorators, e.g. `@Component` and `@NgModule`
- Property decorators , e.g. `@Input` and `@Output`
- Method decorators, e.g. `@HostListener`
- Parameter decorators , e.g. `@Inject`

La création d'un décorateur

Cela rend les choses beaucoup plus facile si nous comprenons ce qu'est un décorateur fait. Pour ce faire, nous pouvons créer un exemple rapide de décorateur.

Decorator functions Les décorateurs sont en fait que des fonctions, c'est aussi simple que cela, et sont appelés a**vec pour valeur ce qu'ils la décorent**.

Une méthode décorateur sera appelée avec la valeur de la méthode, et une classe décorateur va être appelé avec la classe à décorée.

```
function Console(message) {
  // access the "metadata" message
  console.log(message);
  // return a function closure, which
  // is passed the class as `target`
  return function(target) {
    console.log('Our decorated class', target);
  };
}
```

Le Change Detection Mode.

> La tâche de base de la détection de changement est de prendre l'état interne d'un programme et de le rendre en quelque sorte visible à l'interface utilisateur.

Cet état peut être tout type d'objets, des tableaux, des primitives de

![](illustrations/cd.svg

> Fondamentalement une demande de modification de l'état peut être provoqué par trois choses:

- Events - click, submit, …
- XHR - Fetching data from a remote server
- Timers - setTimeout(), setInterval()

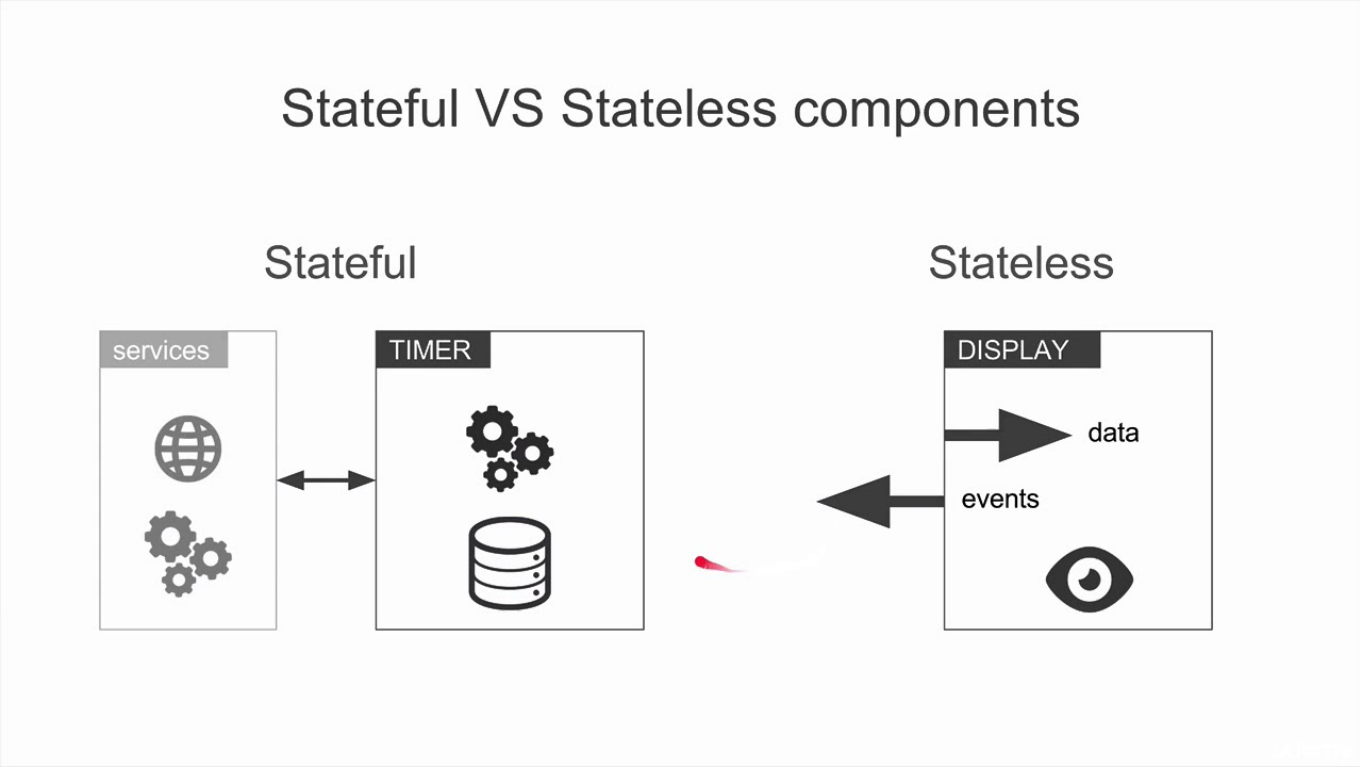
Control the change detection.

```
@Component({
  selector: 'my-app',
  template: `Number of ticks: {{numberOfTicks}}`,
  changeDetection: ChangeDetectionStrategy.OnPush,
})

class AppComponent {
  numberOfTicks = 0;

  constructor(private ref: ChangeDetectorRef) {
    setInterval(() => {
      this.numberOfTicks++;
      // require view to be updated
      this.ref.markForCheck();
    }, 1000);
  }
}
```

Composants neutres versus à état.



Stateful

Quand quelque chose est “Stateful”, c'est un point central qui stocke des informations dans la mémoire à propos de l'application/état du composant.

> Il a également la possibilité de le changer.

Il s'agit essentiellement d'une “vie” de la chose qui a connaissance du passé, actuels et futurs des changements d'état.

Stateless

Quand quelque chose est “Stateless”, il calcule uniquement son état interne.

Cela permet de compléter la transparence référentielle, étant donné que pour les mêmes entrées, elle produit toujours le même résultat.

Ils ne sont pas essentiellement “vivant”, car ils sont simplement l'informé.

Components

Lorsque nous parlons de “Stateless” et “Stateful” dans le développement d'applications web, nous pouvons commencer à appliquer ces concepts aux composants.

> Un composant est une pièce isolée du comportement ou de la fonctionnalité qui nous permet de diviser le comportement dans les rôles, un peu comme on le ferait avec des fonctions JavaScript.

Stateful components

- Modifications de l'état par le biais de fonctions
- Fournit des données (c'est à dire à partir de l'adresse http couches)

- Peut recevoir des données initiales, en passant par la route résout au lieu de la couche de service d'appels
- A connaissance de l'état actuel.
- Est informé par les composants “Stateless” lorsque quelque chose doit changer.
- Peut communiquer avec dépendances externes (comme une couche http)
- Rend “Stateless” (ou même stateful) les composants enfant.

Stateless components

- Ne pas demander/récupérer les données
- Données reçues via la propriété de liaison
- Émettent des données via les d'événement
- Rend plus “Stateless” d'autres composants
- Peut contenir des valeurs locales de l'INTERFACE
- Sont un petit morceau d'une image plus grande

10/ Définition de composants

Les composants applicatifs **Angular2** sont des `Directives` activant un template dans cycle de compilation.

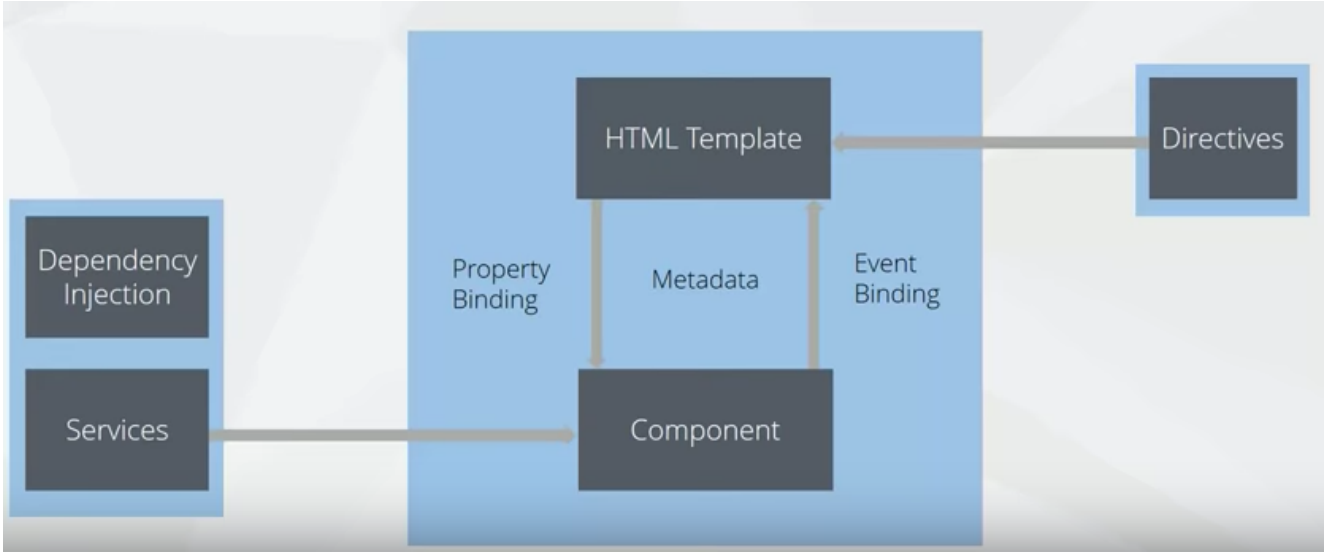
Il utilise :
Une `class` de définition.
Des données issues de `Services`.
Une syntaxe de template.
Des directives de strcuture (`template Directives`).
Des filtres (`Pipe`).
Des `Directives` personnalisées.
* D'autres composants.

Définition d'un composant

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

@Component({
    selector: 'hello',
    template: '<p>Hello, {{name}}</p>'
})
export class Hello {
  name: string;

  constructor() {
    this.name = 'World';
  }
}
```

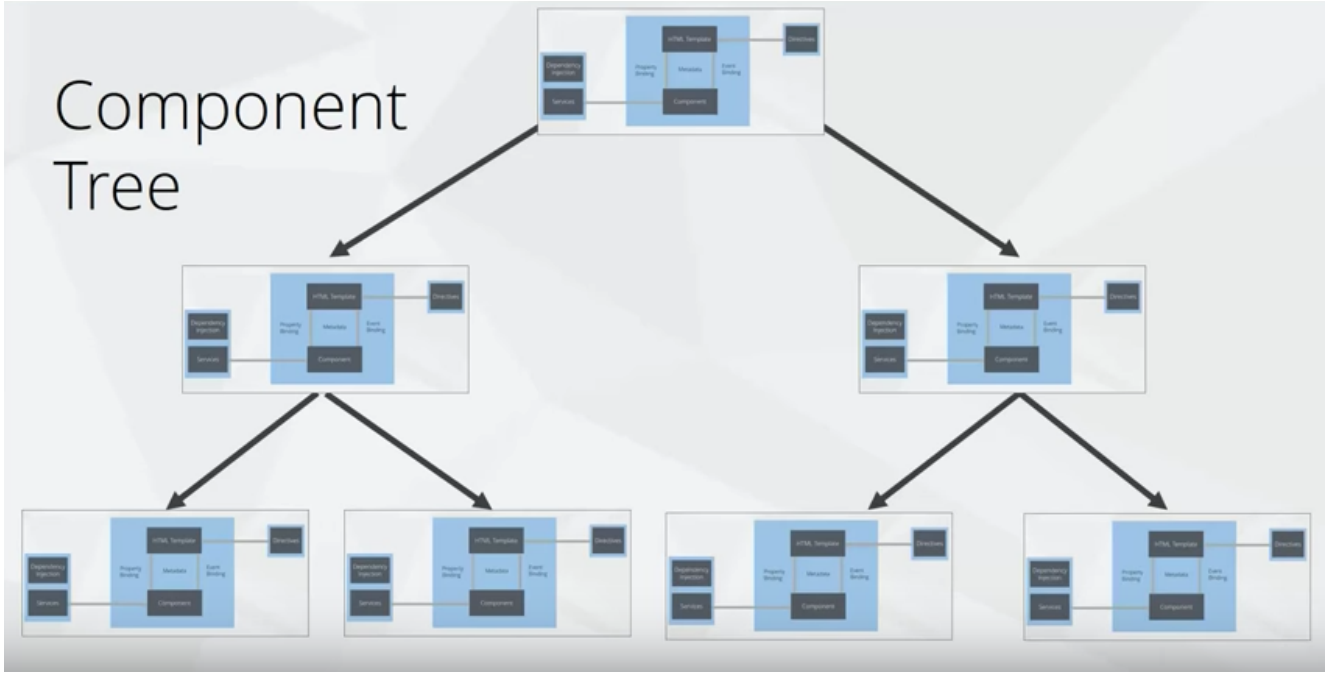


Une application est constituée d'un abre de composant. `Component Tree`

```
<AppComponent>

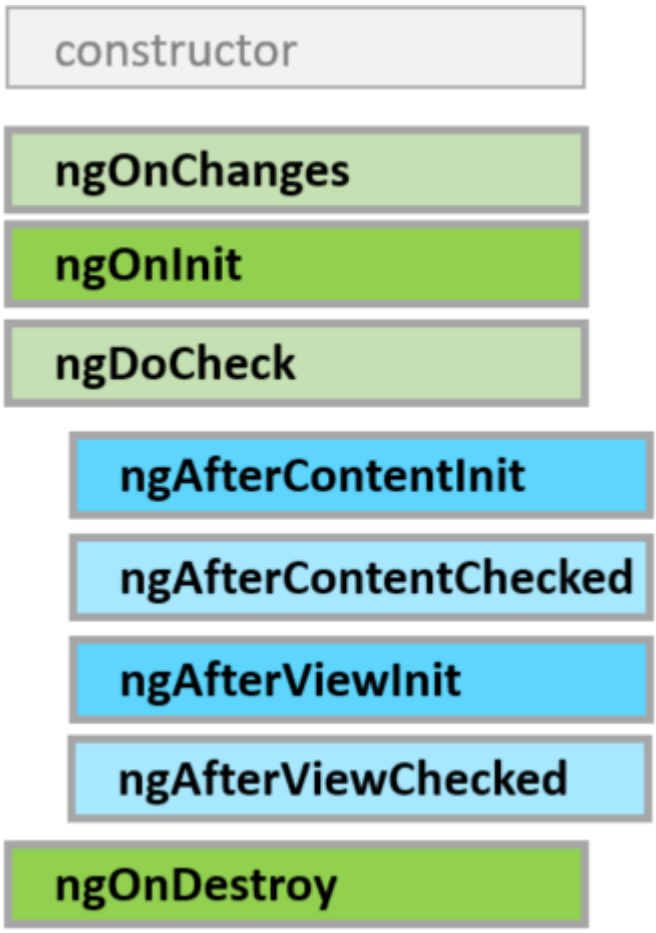
  <ListComponent>
    <ListItem></ListItem>
    <ListItem></ListItem>
    <ListItem></ListItem>
  </ListComponent>

  <ListForm></ListForm>
</AppComponent>
```



Cycle de vie.

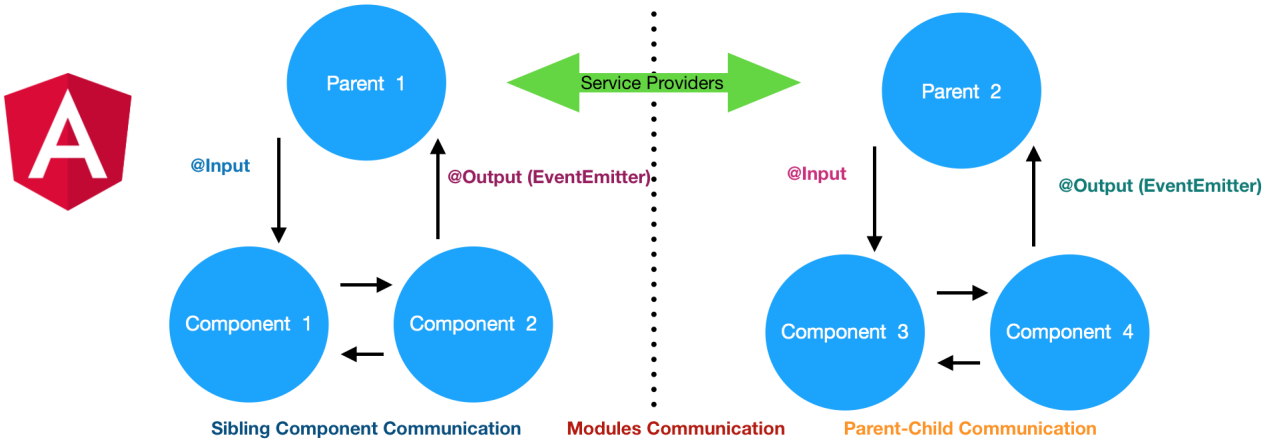
Les composants suivent le cycle de gestion d'angular (création, rendu, insertion de données…) il est possble de réimplémenter (surcharge) les methodes correspondant à ce cycle (`LifeCycle Hooks`)



- `ngOnChanges` - un `@Input` change.
- `ngOnInit` - Début du cycle.
- `ngDoCheck` - Aprés chaque détection de changement.
- `ngAfterContentInit` - Après initialisation du contenu.
- `ngAfterContentChecked` - Après vérification du contenu.
- `ngAfterViewInit` - Aprés initialisation des `views`.
- `ngAfterViewChecked` - Aprés vérification des `views`.
- `ngOnDestroy` - Aprés la destruction du composant.

Cycle de vie par l'exemple ou en détail

Communication entre composants, optimisation ES6.



> Services : Les Services sont accessibles dans les modules de l'application.

C'est Classe Singleton, de sorte que c'est avoir qu'une seule instance de l'application. Tout ce qui est déclarée dans la classe du service n'est pas persistent, lorsque l'utilisateur final actualiser le navigateur.

10.1/ Cycle de vie dans l'application.

Transmission de donnés.

En considérant le `Component Tree` les données peuvent être transmises à composant via une déclaration d' `input` **unidirectionnel**.

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

@Component({
  selector: 'hello',
  template: '<p>Hello, {{name}}</p>'
})
export class Hello {
  //Décorateur
  @Input() name: string;
}
```

Pour retourner des données le composant peut `émettre` un événements.

```
import {Component, EventEmitter, Input, Output} from '@angular/core';

@Component({
  selector: 'counter',
  template: `
    <div>
      <p>Count: {{ count }}</p>
      <button (click)="increment()">Increment</button>
    </div>
  `
})
export class Counter {
  @Input() count: number = 0;
  @Output() result: EventEmitter = new EventEmitter();

  increment() {
    this.count++;
    //Décorateur
    this.result.emit(this.count);
  }
}
```

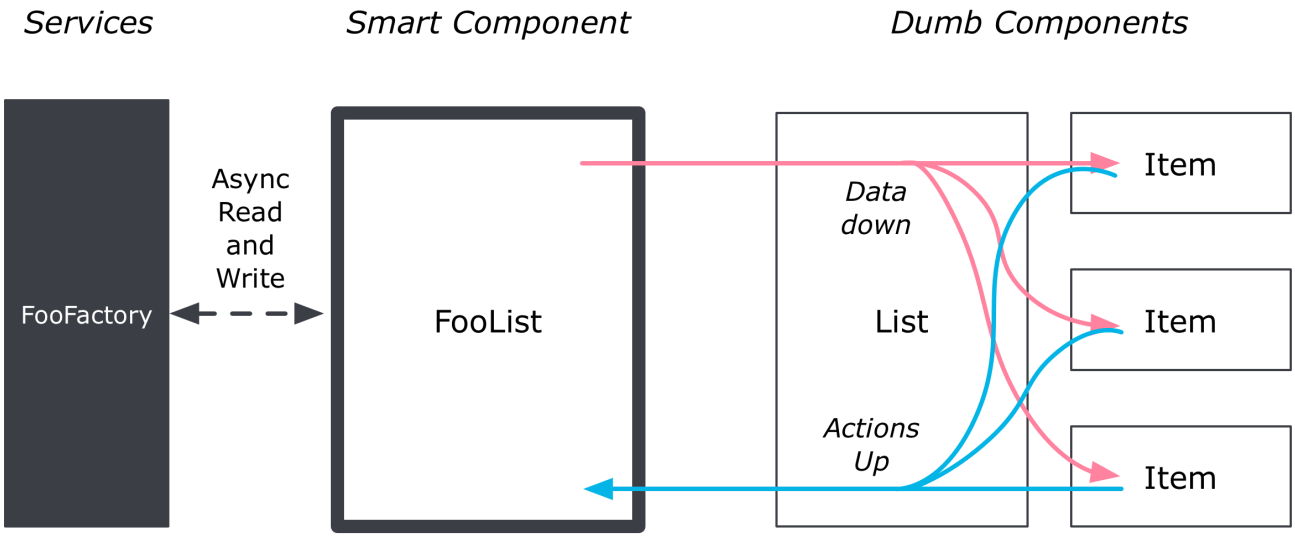
> **A noter** les événements se propagent selon le modèle du DOM le paramètre `$event` permet leur capture.

```
<button (click)="clicked($event)"></button>
```

```
@Component(...)
class MyComponent {
  clicked(event) {
    event.preventDefault();
  }
}
```

Accès à un composant enfant.

Il est souvent nécessaire de créer des composant **ignorant la logique** exploitée ou `Dummy Component`. Ces composnant sont des briques fonctionelles réutilisables.



Un composant parent peut [accèder au propriété]() d'un enfant (`Component Tree`) en le référençant par `DI ou variable local`

Dependency Injection par Inférence :

```
@Component({
  selector: 'my-example',
  templateUrl: 'app/my-example.component.html'
})
export class MyExampleComponent {
  submitForm (form: NgForm) {
    console.log(form.value);
  }
}
```

Variable locale :

```
<red-ball #myBall></red-ball>
The ball is {{ myBall.color }}.
```

Un accès plus explicite est possible grâce au décorateurs `@ViewChild & @ViewChildren`

@ViewChild sélectionne la `class` d'un composant enfant par inférence de type.
@ViewChildren sélectionne un ensmble ou `QueryList`.

Les deux décorateurs supporte un `selector` en paramètres.

```
import {Component, QueryList, ViewChildren} from '@angular/core';
import {Hello} from './hello.component';

@Component({
    selector: 'app',
    template: `
      <div>
        <title></title>
        <hello></hello>
        <hello></hello>
        <hello></hello>
      </div>
      <button (click)="onClick()">Call Child function</button>`
})
export class App {
  @ViewChild(Title) child: Title;
  @ViewChildren(Hello) helloChildren: QueryList<Hello>;

  constructor() {}

  onClick() {
    this.helloChildren.forEach((child) => child.exampleFunction());
    this.child.exampleFunction();
  }
}
```

Les décorateurs `@ContentChild & @ContentChildren` fonctionnent sur le même principe mais sur le `contenu projeté` c-a-d inséré depuis un composant parent.
(Ce mécanisme est similaire à la transclusion ng1)

Projection de contenu, pilotage de composants enfants.

```
@Component({
  selector: 'fa-input',
  template: `
    <i class="fa" [ngClass]="classes"></i>
    <ng-content></ng-content>
  `,
  styleUrls: ['./fa-input.component.css']
})
export class FaInputComponent {

  @Input() icon: string;

  @ContentChild(InputRefDirective)
  input: InputRefDirective;

  @HostBinding("class.focus")
  get focus() {
    return this.input ? this.input.focus : false;
  }

  get  classes() {
    const cssClasses = {
      fa: true
    };
    cssClasses['fa-' + this.icon] = true;
    return cssClasses;
  }
}
```

On peut requêter quoi que ce soit dans le contenu de la partie de l'élément HTML et l'utiliser en interne API, à l'aide de la `@ContentChild` et `@ContentChildren` les décorateurs.

Styling projected content

Pour le style, nous avons besoin de changer les styles comme ceci:

```
:host ::ng-deep input {
  border: none;
  outline: none;
}
```

Manipulation du DOM

> **ElementRef** expose un accès au DOM natif.

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

@Component({
  selector: 'app',
  template: `
  <h1>My App</h1>
  <pre style="background: #eee; padding: 1rem; border-radius: 3px; overflow: auto;">
    <code>{{ node }}</code>
  </pre>
`
})
export class App implements AfterContentInit {
  node: string;

  constructor(private elementRef: ElementRef) {}

  ngAfterContentInit() {
    const tmp = document.createElement('div');
    const el = this.elementRef.nativeElement.cloneNode(true);

    tmp.appendChild(el);
    this.node = tmp.innerHTML;
  }

}
```

Définition syntaxique, le symbole (*). Variables locales.

> **Attention** les direcitive ayant un impact sur la structure même du `Component Tree` sont préfixées de *.
> Il est important de ne pas omettre les symbole *

Les **Variable locale** ou (Template reference variables) sont des références au DOM ou à une directive exprimées dans le template.

```
<!-- phone refers to the input element; pass its `value` to an event handler -->
<input #phone placeholder="phone number">
<button (click)="callPhone(phone.value)">Call</button>

<!-- fax refers to the input element; pass its `value` to an event handler -->
<input ref-fax placeholder="fax number">
<button (click)="callFax(fax.value)">Fax</button>
```

Créer une directive de structure

La démarche est la même que pour une directive d'attibut mais il faut indiquer à Angular l'impact sur la structure.

```
@Directive({
  selector: '[delay]'
})
export class DelayDirective {
  constructor(
    private templateRef: TemplateRef<any>,
    private viewContainerRef: ViewContainerRef
  ) { }

  @Input('delay')
  set delayTime(time: number): void {
    setTimeout(()=>{
      this.viewContainerRef.createEmbeddedView(this.templateRef);
    }, time);
  }
}
```

Utilisation

Préparer les composants pour la distribution.

- Essayez de ne pas accéder directement au DOM (c'est à dire suivre le principe d'inversion des dépendances).
- Fournir de l' de la version esm de votre bibliothèque afin de permettre le *Tree Shaking*.
- Référencer de l'esm version sous le module et `jsnext:main` du `package.json`.
- Fournir de l'ES5 UMD ensemble de votre bibliothèque.
- Fournir les définitions de type de votre bibliothèque en les créant avec `tsc` les déclarations.
- Compiler votre bibliothèque avec `ngc`.

Documentation : génération dynamique.

> Compodoc est un outil de documentation pour les applicationsAngular . Il génère une la documentation statique de votre application.

Installer à partir de `npm` :

```
npm install -g @compodoc/compodoc
```

Définir une tâche de script dans votre `package.json` (avec npm 6.x) :

```
"scripts": {
  "compodoc": "npx compodoc -p src/tsconfig.app.json"
}
```

```
npm run compodoc
```

Composants riches et librairies externes.

Composants riches et librairies externes.

L'utilisation des librairies communautaire peut représenter un gain conséquent de producivité.

> https://angular.io/resources

UI Components

ag-Grid
A datagrid for Angular with enterprise style features such as sorting, filtering, custom rendering, editing, grouping, aggregation and pivoting.

Alyle UI
Minimal Design, a set of components for Angular.

Amexio - Angular Extensions
Amexio (Angular MetaMagic EXtensions for Inputs and Outputs) is a rich set of Angular components powered by Bootstrap for Responsive Design. UI Components include Standard Form Components, Data Grids, Tree Grids, Tabs etc. Open Source (Apache 2 License) & Free and backed by MetaMagic Global Inc

Angular Material
Material Design components for Angular

Ant Design Mobile of Angular (ng-zorro-antd-mobile)
A set of enterprise-class mobile UI components based on Ant Design Mobile and Angular

Ant Design of Angular (ng-zorro-antd)
A set of enterprise-class UI components based on Ant Design and Angular

Blox Material
A lightweight Material Design library for Angular, based upon Google's Material Components for the Web

Création de formulaire dynamique : ReactiveFormsModule.

Reactive forms permet l'utilisation explicite et immuable de la gestion de l'état d'un formulaire à un moment donné dans le temps.

Chaque modification de la forme de l'état renvoie un nouvel état, qui maintient l'intégrité du modèle entre les changements.

Les *Reactive forms* sont construits autour d'observables,où les entrées de formulaire et les valeurs sont fournies en tant que flux de valeurs d'entrée, qui peut être consulté de manière synchrone.

```
import { ReactiveFormsModule } from '@angular/forms';

@NgModule({
  imports: [
    // other imports ...
    ReactiveFormsModule
  ],
})
export class AppModule { }
```

Component

```
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']
})
export class NameEditorComponent {
  name = new FormControl('');
}
```<!-- CONTENT ## FormControl et FormGroup, AbstractContrl, FormArray.-->

<break></break>

FormControl et FormGroup, AbstractContrl, FormArray.

Reactive forms vous donne accès à l'écran de contrôle de l'état et de la valeur à un point dans le temps. Vous pouvez manipuler l'état actuel et de la valeur grâce à la classe de composant ou le modèle de composant.

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

@Component({
  selector: 'app-profile-editor',
```

```
  templateUrl: './profile-editor.component.html',
  styleUrls: ['./profile-editor.component.css']
})
export class ProfileEditorComponent {
  profileForm = new FormGroup({
    firstName: new FormControl(''),
    lastName: new FormControl('')
  });

  updateName() {
    this.profileForm.lastName.setValue('Nancy');
  }

  updateProfile() {
    this.profileForm.patchValue({
      firstName: 'Nancy',
      address: {
        street: '123 Drew Street'
      }
    });
  }
}
```

Usage

```
<form [formGroup]="profileForm">
  <label> First Name: <input type="text" formControlName="firstName" />
</label>

  <label> Last Name: <input type="text" formControlName="lastName" />
</label>
</form>
```

Règle de validation personnalisée

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

@Directive({
  selector: '[validateEmail][ngModel]'
})
export class EmailValidator {}
```

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

@Directive({
  ...
})
export class EmailValidator {

  validator: Function;

  constructor(emailBlackList: EmailBlackList) {
    this.validator = validateEmailFactory(emailBlackList);
  }

  validate(c: FormControl) {
    return this.validator(c);
  }
}
```

Validation et gestion d'erreur personnalisée.

Les validateurs *async* personnalisés sont similaires aux validateurs, mais ils doivent plutôt retourner une Promesse ou un Observable qui émettra la valeur null ou une erreur de validation de l'objet.

> La fonction de validation est en fait une `factory` qui prend une expression régulière pour détecter un cas interdit et renvoie une fonction de validation.

```
export function forbiddenNameValidator(nameRe: RegExp): ValidatorFn {
  return (control: AbstractControl): {[key: string]: any} | null => {
    const forbidden = nameRe.test(control.value);
    return forbidden ? {'forbiddenName': {value: control.value}} : null;
  };
}
```

Utilisation du FormBuilder.

La création de formulaire et des instances de contrôle manuellement peut devenir répétitif lorsque vous traitez avec de multiples formulaires.

> Le FormBuilder service fournit des méthodes pratiques pour générer des contrôles.

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

@Component({
  selector: 'app-profile-editor',
  templateUrl: './profile-editor.component.html',
  styleUrls: ['./profile-editor.component.css']
})
export class ProfileEditorComponent {
  profileForm = this.fb.group({
    firstName: [''],
    lastName: [''],
    address: this.fb.group({
      street: [''],
      city: [''],
      state: [''],
      zip: ['']
    }),
  });

  constructor(private fb: FormBuilder) { }
}
```

FormArray

FormArray is an alternative to FormGroup for managing any number of unnamed controls. As with form group instances, you can dynamically insert and remove controls from form array instances, and the form array instance value and validation status is calculated from its child controls. However, you don't need to define a key for each control by name, so this is a great option if you don't know the number of child values in advance. The following example shows you how to manage an array of aliases in ProfileEditor.

Création dynamique de template.

`<ng-template>` permet le chargement dynamique de composants.

```
export class AdBannerComponent implements OnInit, OnDestroy {
  @Input() ads: AdItem[];
  currentAdIndex = -1;
  @ViewChild(AdDirective) adHost: AdDirective;
  interval: any;

  constructor(private componentFactoryResolver: ComponentFactoryResolver)
{ }

  ngOnInit() {
    this.loadComponent();
    this.getAds();
  }

  ngOnDestroy() {
    clearInterval(this.interval);
  }

  loadComponent() {
    this.currentAdIndex = (this.currentAdIndex + 1) % this.ads.length;
    let adItem = this.ads[this.currentAdIndex];

    let componentFactory =
this.componentFactoryResolver.resolveComponentFactory(adItem.component);

    let viewContainerRef = this.adHost.viewContainerRef;
    viewContainerRef.clear();

    let componentRef = viewContainerRef.createComponent(componentFactory);
    (<AdComponent>componentRef.instance).data = adItem.data;
  }

  getAds() {
    this.interval = setInterval(() => {
      this.loadComponent();
    }, 3000);
  }
}
```

Tests unitaires. Bonnes pratiques et outils.

Tests unitaires. Bonnes pratiques et outils.

Les test Angular (recommandés avec Jasmine) peuvent s'appuyer sur l'utilitaire `karma`.

Anciennement connu sous le nom de Testacular. `karma` un ogestionnaire de suites de tests.

Installation

```
npm i -g karma-cli
npm i karma
karma init
karma start
```

12.1/ Configurer l'environnement de test.

La mise en oevre de `karma` avec angular s'appuie sur un ensemble de fichier de définition.

- **karma.conf.js**
- **karma-test-shim.js**
- **systemjs.config.js**
- **systemjs.config.extras.js**

Bien que l'utilitaire `angular-cli` prépare l'environnement pour le développeur, il est utile d'en comprendre la structure.

Par convention les fichiers de test ou `specs` sont nommés sur le modèle `*.spec.ts`.

Le framework inclut des utilitaires `@angular/core/testing` et particulièrement une `class` nommée `TestBed` facilitant lisolation d'un module.

On appelle `fixture` le composant fonctionnel extrait du module pour être testé.

```
beforeEach(() => {

  // refine the test module by declaring the test component
  TestBed.configureTestingModule({
    declarations: [ BannerComponent ],
  });

  // create component and test fixture
  fixture = TestBed.createComponent(BannerComponent);

  // get test component from the fixture
  comp = fixture.componentInstance;
});

it('should display original title', () => {

  // trigger change detection to update the view
  fixture.detectChanges();

  // query for the title <h1> by CSS element selector
  de = fixture.debugElement.query(By.css('h1'));

  // confirm the element's content
  expect(de.nativeElement.textContent).toContain(comp.title);
});
```

Ecrire les tests avec Jasmine. Couverture.

Jasmine n'est pas le seul Framework qui permet de créer des tests en JavaScript, il en existe d'autres tel que Mocha ou Tape.

Jasmine offre deux **ensembles syntaxique** à travers différentes fonction.

La structuration de la suite de test :
describe *: famille de test.*
it : pas de test.

```
describe('Multiplication', function(){
    it('should work', function(){
        expect(value).toBe(valueTest)
    });
});
```

La documentation de jasmine est complète et simple.

jasmine.done

Le `callback done()` fournit par jasmine permet de retarder l'exéctution d'un test.

```
 beforeEach(function(done) {
    setTimeout(function() {
      value = 0;
      done();
    }, 1);
  });

This spec will not start until the done function is called in the call to beforeEach a

  it("should support async execution of test preparation and expectations", function(d
    value++;
    expect(value).toBeGreaterThan(0);
    done();
  });
```

Cas de test : pipe, composant, application. Il existe

différentes **stratégies de test**

Composant

```
let comp:    BannerComponent;
let fixture: ComponentFixture<BannerComponent>;
let de:      DebugElement;
let el:      HTMLElement;

describe('BannerComponent', () => {
  beforeEach(() => {
    TestBed.configureTestingModule({
      declarations: [ BannerComponent ],
      /* pour ne pas activer manuellement le cycle de détection il est possible d'ajou
        providers: [
            { provide: ComponentFixtureAutoDetect, useValue: true }
        ]
        */
    });

    fixture = TestBed.createComponent(BannerComponent);

    comp = fixture.componentInstance; // BannerComponent Instance

    // query for the title <h1> by CSS element selector
    de = fixture.debugElement.query(By.css('h1'));
    el = de.nativeElement;

  });
});

it('should display original title', () => {
  fixture.detectChanges();
  /* pour ne pas activer manuellement le cycle de détection il est possible d'ajouter
  providers: [
    { provide: ComponentFixtureAutoDetect, useValue: true }
  ]
  */
  expect(el.textContent).toContain(comp.title);
});

it('should display a different test title', () => {
  comp.title = 'Test Title';
  fixture.detectChanges();
  expect(el.textContent).toContain('Test Title');
});
```

Composant avec dépendance

Stratégie simuler la dépendance.

```
beforeEach(() => {
    // Mock de la dépendance
    userServiceStub = {
      isLoggedIn: true,
      user: { name: 'Test User'}
    };

    TestBed.configureTestingModule({
       declarations: [ WelcomeComponent ],
       //Redéfinition de la dépendance
       providers:    [ {provide: UserService, useValue: userServiceStub } ]
    });

    fixture = TestBed.createComponent(WelcomeComponent);
    comp    = fixture.componentInstance;

    // UserService from the root injector
    userService = TestBed.get(UserService);

    //  get the "welcome" element by CSS selector (e.g., by class name)
    de = fixture.debugElement.query(By.css('.welcome'));
    el = de.nativeElement;
  });

  it('should welcome the user', () => {
    fixture.detectChanges();
    const content = el.textContent;
    expect(content).toContain('Welcome', '"Welcome ..."');
    expect(content).toContain('Test User', 'expected name');
    });

    it('should welcome "Bubba"', () => {
    userService.user.name = 'Bubba'; // welcome message hasn't been shown yet
    fixture.detectChanges();
    expect(el.textContent).toContain('Bubba');
    });

    it('should request login if not logged in', () => {
    userService.isLoggedIn = false; // welcome message hasn't been shown yet
    fixture.detectChanges();
    const content = el.textContent;
    expect(content).not.toContain('Welcome', 'not welcomed');
    expect(content).toMatch(/log in/i, '"log in"');
    });
```

Composant avec service asynchrone

Stratégie intercepter le service.

```
beforeEach(() => {
  TestBed.configureTestingModule({
     declarations: [ TwainComponent ],
     providers:    [ TwainService ],
  });

  fixture = TestBed.createComponent(TwainComponent);
  comp    = fixture.componentInstance;

  // TwainService actually injected into the component
  twainService = fixture.debugElement.injector.get(TwainService);

  // Les SPY de Jasmine permetteent d'intercepter la méthode du service.
  spy = spyOn(twainService, 'getQuote')
        .and.returnValue(Promise.resolve(testQuote));

  // Get the Twain quote element by CSS selector (e.g., by class name)
  de = fixture.debugElement.query(By.css('.twain'));
  el = de.nativeElement;
});
```

Composant avec `templateUrl`

Stratégie utiliser la méthode `.compileComponents()` .

La fonction `async` est un utilitaire simplifiant la gestion asynchrone du code.

```
beforeEach( async(() => {
  TestBed.configureTestingModule({
    declarations: [ DashboardHeroComponent ],
  })
  .compileComponents(); // compile template and css
}));
```

service

```
it('#getTimeoutValue should return timeout value',  done => {
  service = new FancyService();
  service.getTimeoutValue().then(value => {
    expect(value).toBe('timeout value');
    done();
  });
});
```

Pipe

```
describe('TitleCasePipe', () => {
  // This pipe is a pure, stateless function so no need for BeforeEach
  let pipe = new TitleCasePipe();
  it('transforms "abc" to "Abc"', () => {
    expect(pipe.transform('abc')).toBe('Abc');
  });
  it('transforms "abc def" to "Abc Def"', () => {
    expect(pipe.transform('abc def')).toBe('Abc Def');
  });
  // ... more tests ...
});
```