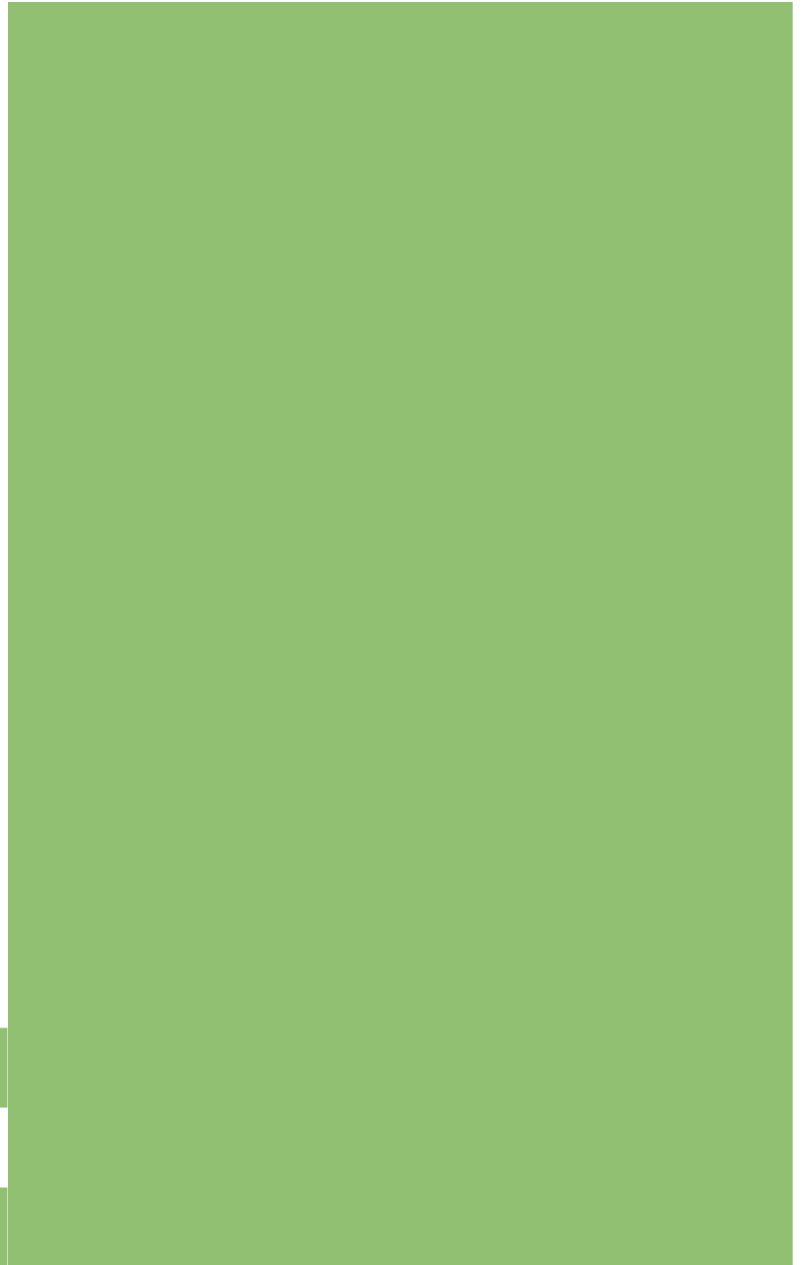


Formation Angular, développement avancé



- Votre nom, prénom
- Votre expérience et vécu Angular
- Vos attentes
- Des projets ?





Plan

01

Angular, mise en œuvre des bonnes pratiques

02

Fonctionnement interne d'Angular

03

Création de composants distribuables

04

Composants riches et librairies externes

05

Formulaire dynamique : le FormBuilder

06

Tests unitaires. Bonnes pratiques et outils.

Organisation & suivi du cours

Déroulement du cours

Demi journée	Contenu
lundi matin	Introduction Angular, mise en œuvre des bonnes pratiques Travaux pratiques
lundi après-midi	Fonctionnement interne d'Angular Travaux pratiques
mardi matin	Création de composants distribuables Travaux pratiques
mardi après-midi	Composants riches et librairies externes Travaux pratiques
mercredi matin	Formulaire dynamique : Le FormBuilder Travaux pratiques
mercredi après-midi	Tests unitaires. Bonnes pratiques et outils Travaux pratiques

Introduction

Angular est un Framework JavaScript/TypeScript open source sous la licence MIT.

Il assure la création des applications web dynamiques monopages, permettant de développer ses propres balises et attributs HTML.

Historique

- Angular a été créé par **Misko Hevery** et **Adam Abrons** en 2009 dans les locaux de Google par **Brat Tech LLC**.
- Initialement appelé **GetAngular**.
- Le framework a ensuite été rendu disponible en Open Source.
- Google a par la suite repris le code source pour le développer.
- Sept 2014. Annonce de la sortie d'Angular 2 (un peu prématurée).
- Mai 2016. Sortie de la première release candidate Angular 2.



Date	Version	Date	Version
Mars 2017	Angular 4	Février 2020	Angular 9
Novembre 2017	Angular 5	Juin 2020	Angular 10
Mai 2018	Angular 6	Novembre 2020	Angular 11
Octobre 2018	Angular 7	Mai 2021	Angular 12
Mai 2019	Angular 8		

Qu'est-ce qu'Angular ?

Angular est un framework JavaScript/Typescript pour créer des applications monopages (SPA), web et mobiles.

Quels types d'applications peut-on développer ?

- Site web complet. Exemple : <https://www.24ur.com/>
- Même un **pokedex** : <https://ng-pokedex.firebaseio.com/>
- De petits widgets interactifs pour un site web existant (moteur de recherche, module de réservation).



Plus de références : <http://builtwithangular2.com/> <https://www.madewithangular.com/>

La communauté

La communauté d'Angular est très active.

- Programmeurs
- Présente sur Github et Stackoverflow <https://github.com/angular/>
- Développeurs web (professionnels ou non)
- Conférences organisées partout dans le monde
 - ❖ AngularU à San Francisco
 - ❖ AngularConnect à Londres au Royaume-Uni
 - ❖ Ng-conf à Salt Lake City
 - ❖ Ng-Vegas à Las Vegas <https://ngeurope.org/>
 - ❖ Ng-Europe à Paris
 - ❖ ngMorocco au Maroc
 - ❖ Et plusieurs autres



Les applications monopages (SPA)

- **Angular** permet de développer des applications Web de type SPA.
- Une **SPA** (Single Page Application) est une application web accessible via une page web unique.
- Le but est d'éviter le chargement d'une nouvelle page à chaque action demandée et d'améliorer ainsi l'expérience utilisateur (meilleure fluidité).

Les applications monopages (SPA)

- La différence entre une **SPA** et un site web classique réside dans leur structure et dans la relation qu'ils établissent entre le navigateur et le serveur :
 - Une SPA est donc composée d'une seule page.
 - Le rôle du browser (front-end) est beaucoup plus important : Toute la logique applicative y est déportée.
 - Le serveur (back-end) est "seulement" responsable de la fourniture des ressources à l'application et surtout de l'exposition des données.

SPA : Pourquoi on en parle ?

- Les frameworks JS/TS comme **Angular** participent à la popularité des SPA.
- Les SPA en s'appuyant sur de tels frameworks ont l'avantage d'être :
 - Testables (unitairement et fonctionnellement)
 - Fluides (pas de rechargement d'url etc.)
 - Bien organisées
 - Maintenables et évolutives
 - ...

Pourquoi Angular ?

- Angular est plus facile à apprendre que AngularJS, enfin si on connaît TypeScript
- Angular a été réécrit en TypeScript
- Performance et mobile : Angular a été conçu initialement pour le mobile
- Maintenabilité
- Testabilité
- Google ...

Principales caractéristiques d'Angular

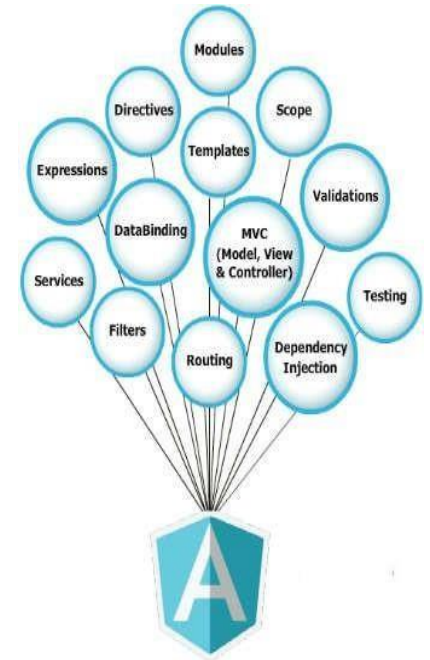
Plusieurs langages supportés(1). ES5, ES6, TypeScript...

Complet. Inclut toutes les briques nécessaires à la création d'une application professionnelle. Routeur, requêtage HTTP, gestion des formulaires, internationalisation...

Modulaire. Le framework lui-même est découpé en sous-paquets correspondant aux grandes aires fonctionnelles (core, router, http...). Vos applis doivent être organisées en composants et en modules (1 module = 1 fichier).

Rapide. D'après les benchmarks, Angular est aujourd'hui 5 fois plus rapide que la version 1.

Tout est composant. Composant = brique de base de toute appli Angular.



Angular, mise en œuvre des bonnes pratiques

Injection de dépendances

- L'injection de dépendances est un pattern de conception permettant de faciliter la gestion des dépendances, améliorer l'extensibilité d'une application et faciliter les tests-unitaires.
- Sans injection de dépendances :

```
class UserStore {  
    getUser(userId: string): Observable<User> {  
        let restApi = new RestApi(new ConnectionBackend(), new RequestOptions({headers: ...}));  
        return restApi.users.get(userId);  
    }  
}
```

- ❖ Il faut savoir comment instancier `RestApi` ?
- ❖ Comment factoriser ?
- ❖ Comment contrôler l'implémentation de la classe `RestApi` ?

Injection de dépendances

- Angular dispose d'un "injector" qui implémente une factory permettant d'instancier des classes et maintenir les instances.

```
const injector = new Injector([RestApi]);  
const restApi1 = injector.get(RestApi);  
const restApi2 = injector.get(RestApi); // restApi2 is the same instance as restApi1.
```

- Lors du "bootstrap", Angular crée le "root injector" qui sera chargé d'injecter les dépendances de l'application.
- On indique qu'une dépendance est injectable à l'aide du "decorator" `@Injectable`.
- Angular créera alors une instance unique de la dépendance disponible dans toute l'application.

Injection de dépendances

- Grâce à TypeScript, Angular arrive à retrouver la dépendance à partir de son type.

```
class UserStore {  
  
    constructor(restApi: RestApi) {  
  
        ...  
    }  
}
```

```
@Injectable()  
class RestApi {  
  
    ...  
}  
  
@NgModule({  
    bootstrap: [ UserNameEditorComponent ],  
    declarations: [  
        UserNameEditorComponent  
    ],  
    imports: [  
        BrowserModule,  
        FormsModule,  
        HttpClientModule  
    ],  
    providers: [  
        RestApi  
    ]  
})  
export class AppModule {  
}
```

Service

Classe qui contiennent une logique ou un traitement

Code partagé et utilisé par d'autres modules ou composants

Pattern d'injection de dépendance

Service

CustomerService.ts

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

```
@Injectable()
```

```
export class CustService {
```

```
  customers: Customer[] = [
```

```
    { "id": 1, "firstname": "David", "lastname": "Giard" },
```

```
    { "id": 2, "firstname": "Bill", "lastname": "Gates" },
```

```
    { "id": 3, "firstname": "Steve", "lastname": "Ballmer" },
```

```
    { "id": 4, "firstname": "Satya", "lastname": "Nadella" }
```

```
  ];
```

```
  getCustomers() {
```

```
    return this.customers;
```

```
  }
```

```
}
```

Injection de dépendances

CustomerService.ts

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

```
@Injectable()
```

```
export class CustService {
```

```
  ..
```

```
  getCustomers() {
```

```
    return customers;
```

```
  }
```

```
}
```

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

```
import { CustService } from CustomerService
```

```
@Component({
```

```
  selector: 'xxx', template: 'yyy',
```

```
  ...
```

```
  providers: [CustService]
```

```
})
```

```
  export class DemoComponent implements OnInit {
```

```
    constructor(private customerService:CustService) { }
```

```
    ngOnInit() {
```

```
      this.customers = this.customerService.getCustomers();
```

```
    }
```

```
}
```

Injection de dépendances

Avec Angular, une dépendance est généralement l'instance d'une classe permettant de factoriser certaines fonctionnalités ou d'accéder à un état permettant ainsi aux composants de communiquer entre eux.

Dans le vocabulaire Angular, ces classes sont appelées "services".

Les services sont le plus souvent des singletons

```
@Component({  
  ...  
})
```

```
export class BookPreviewComponent {  
  constructor(private _httpClient:  
    HttpClient) {  
  }  
}
```

Injection de dépendances

Injector Tree

Angular ne dispose pas que d'un seul "injector" mais d'un arbre d'"injectors".

Root Injector

Tous les "providers" définis par les modules importés directement ou indirectement par l'**AppModule** sont injectés par le "root injector" et sont donc accessibles dans toute l'application.

Injection de dépendances

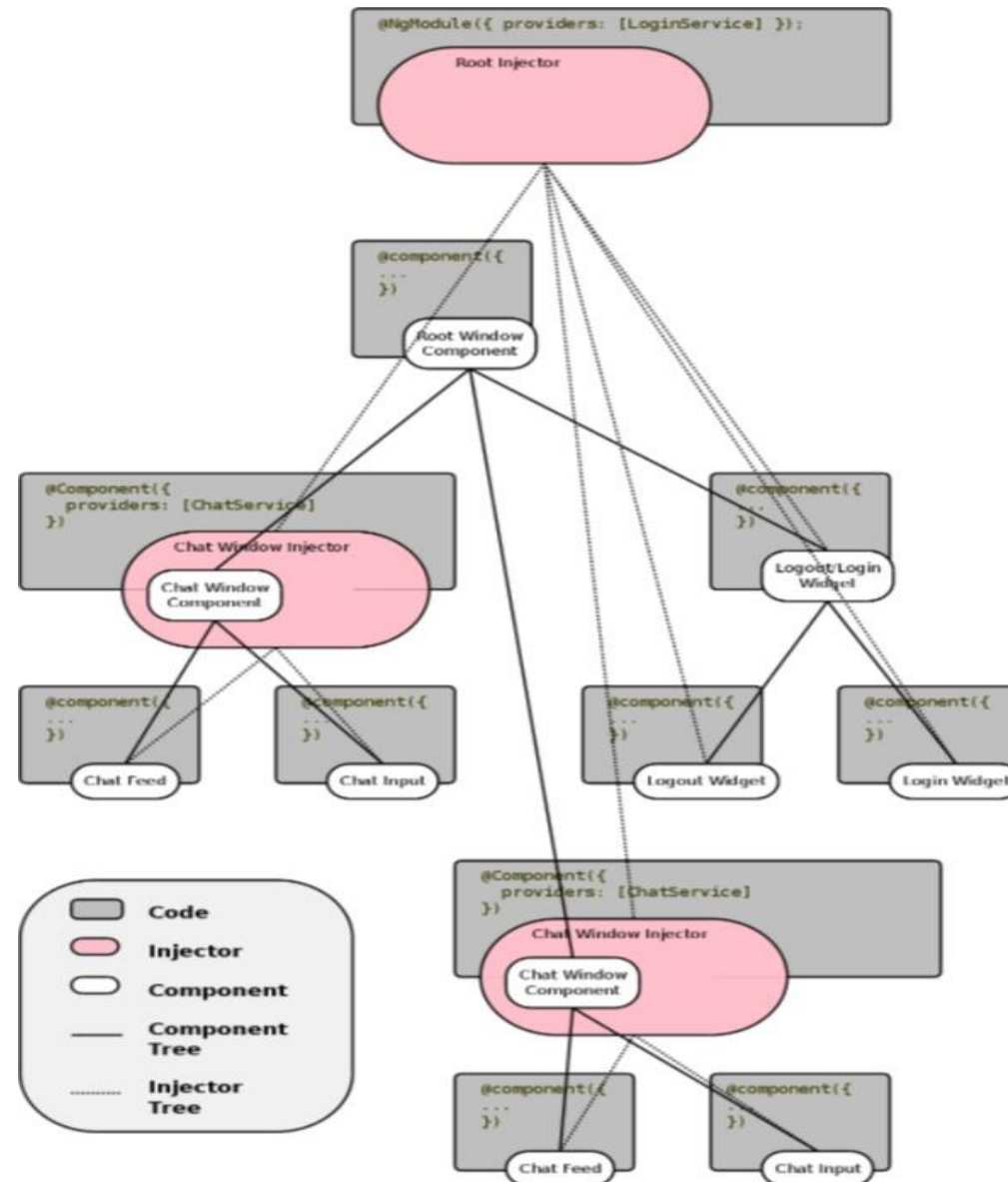
Chaque composant dispose d'un "injector" et peut définir des "providers" via la propriété providers de sa configuration.

Ces "providers" vont écraser les "providers" parents (*ceux des composants parents ou du "root injector"*) et définir **de nouvelles instances de services associés pour chaque instance du composant**

book-preview.component.ts

```
1  @Component({  
2    providers: [  
3      BookRepository  
4    ]  
5  })  
6  export class BookPreviewComponent {  
7  }
```

Injection de dépendances



Types applicatifs partagés : SharedModule

Un **SharedModule** est un simple [module Angular](#), dont le rôle est de *déclarer* et *exporter* tous les composants, directives et pipes susceptibles d'être réutilisés partout dans le projet.

Le nom “**SharedModule**” est une simple convention, et vous n'êtes pas obligé de créer un tel module dans votre projet.

Le **SharedModule** contient généralement des éléments d'interface réutilisables (barre de navigation, HTML pour afficher un champ de formulaire ou un tableau...) ou des directives et pipes très génériques.

```
@NgModule({  
  imports: [CommonModule],  
  declarations: [NavbarComponent, TableComponent, HighlightDirective, MarkdownPipe...],  
  exports: [NavbarComponent, TableComponent, HighlightDirective, MarkdownPipe...]  
})  
export class SharedModule {}
```

Types applicatifs partagés : SharedModule

Certains modules sont utilisés par quasiment tous les composants de l'application (*e.g.*: *CommonModule*, *FlexLayoutModule*, *RouterModule*).

Les importer dans chaque module peut s'avérer pénible.

Il est courant de factoriser ces imports en rassemblant toutes ces dépendances dans un module SharedModule importé par quasiment tous les autres modules de l'application.



Attention à ne pas trop surcharger ce module et en faire un "God Module".

N'y importez que les modules nécessaires pour quasiment tous les composants de l'application.

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { HeaderComponent } from '../header/header.component';
import { SidebarComponent } from '../sidebar/sidebar.component';
import { FooterComponent } from '../footer/footer.component';

@NgModule({
  declarations: [HeaderComponent, SidebarComponent, FooterComponent],
  imports: [
    CommonModule
  ],
  exports: [HeaderComponent, SidebarComponent, FooterComponent]
})
export class SiteFrameworkModule { }
```

PWA : Les services workers

Un Service Worker est un script chargé parallèlement aux scripts de la page et qui va s'exécuter en dehors du contexte de la page web.

Bien que le Service Worker n'ait pas accès au DOM ou aux interactions avec l'utilisateur, il va pouvoir communiquer avec vos scripts via l'[API postMessage](#).

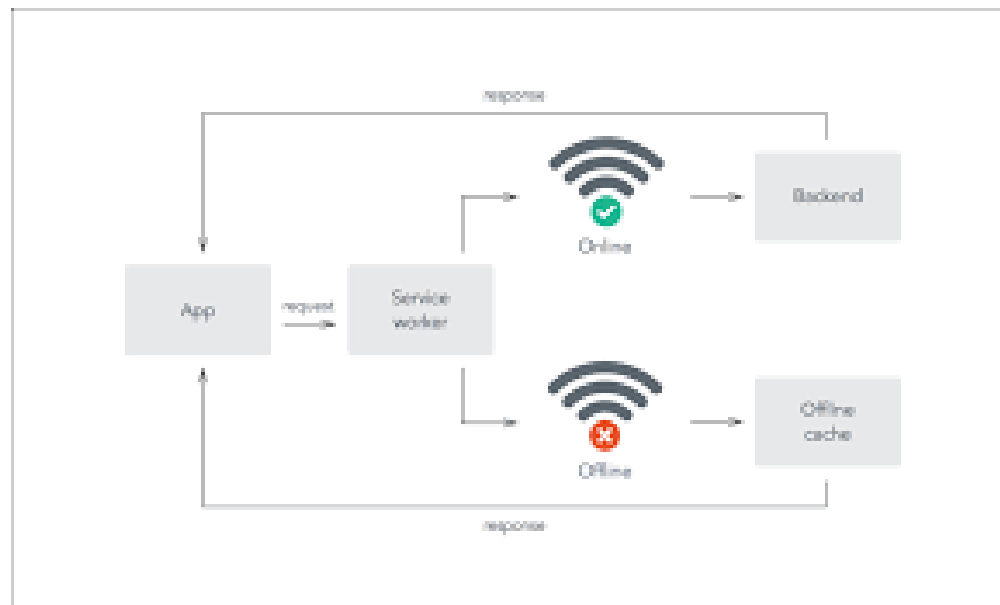
Il se place en proxy de votre Web App, interceptant toutes les requêtes serveur et propose par exemple d'y répondre avec un cache ou en récupérant des données du [LocalStorage](#) ou d'[IndexedDB](#).

Il rend donc votre application [disponible offline](#).

PWA : Les services workers

Que nous soyons dans un contexte d'application mobile hybride ou de webapp responsive, la mobilité des terminaux d'accès traîne souvent : la perte de connectivité (ou la faible connectivité)

Les Service Workers nous permettent de pallier à ces problèmes en utilisant des ressources en cache si le réseau n'est pas disponible, permettant ainsi à notre application de fournir une expérience peu dégradée avant de retrouver un réseau disponible.



PWA : Les services workers

Le script d'un Service Worker tourne dans le navigateur, mais en arrière-plan, sans accès au DOM ni aux interactions avec les utilisateurs.

Il se place entre votre webapp et le réseau, permettant de jouer le rôle de proxy pour nos ressources.

Il a une durée de vie indépendante du site web, il s'arrête lorsqu'il n'est pas utilisé et redémarre si besoin.

Il n'a pas besoin de l'application pour tourner et peut donc permettre l'envoi de notifications

PWA : Les services workers

Le cycle de vie d'un Service Worker

Pour installer un Service Worker pour votre site, nous devons l'enregistrer (**register**) dans le script de notre webapp.

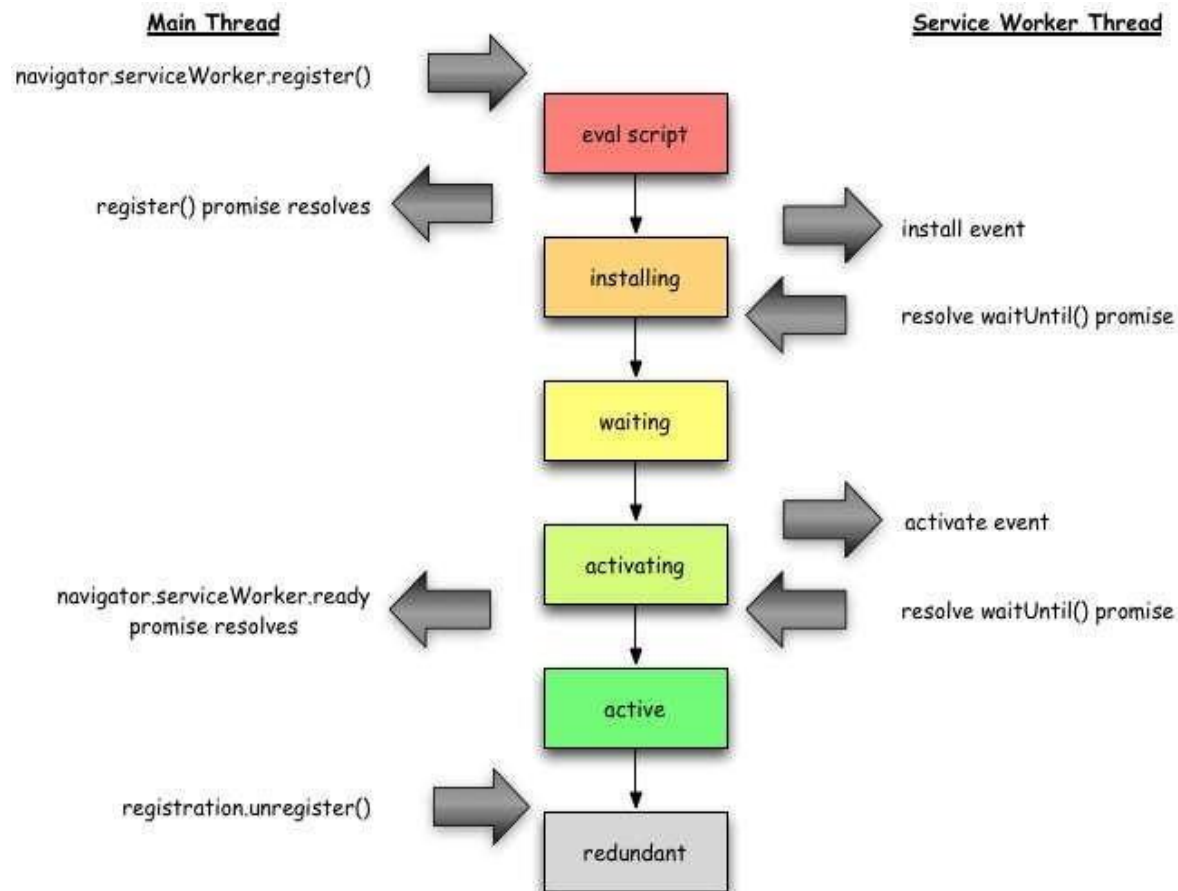
Une fois installé, notre Service Worker va s'activer (**activate**), c'est à ce moment que nous allons gérer la mise à jour du cache ou du Service Worker lui-même.

Après l'activation, le Service Worker est prêt à intercepter les événements **fetch** et **message** émis respectivement par une requête serveur ou un appel via l'API *postMessage*.

PWA : Les services workers

Le cycle de vie d'un Service Worker

Le schéma suivant décrit la séquence d'exécution des instructions dans chaque thread : le navigateur (main) et le worker



Requêtes HTTP avancées

Requêtes HTTP avancées

Angular dispose d'un client pour consommer les web services de type REST

Il s'agit de l'objet de la classe `Http`

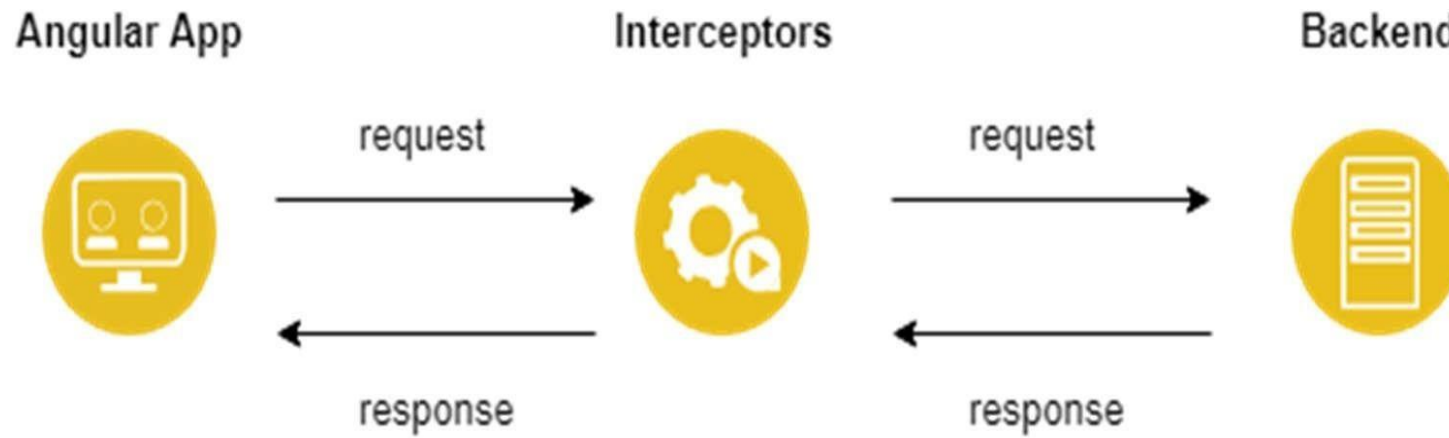
```
Http http = ...  
  
http.get(url).subscribe(  
    response => console.log(response.json()),  
    error => console.error(error)  
);
```

<https://angular.io/docs/ts/latest/guide/server-communication.html>

<https://angular.io/docs/ts/latest/api/http/Http-class.html>

Requêtes HTTP avancées

HTTP Interceptor



Requêtes HTTP avancées

HTTP Interceptor

Un **intercepteur HTTP** vous permet d'intercepter les demandes et les réponses **HTTP** afin de pouvoir les modifier.

C'est très utile, en particulier lorsque vous souhaitez ajouter des en-têtes supplémentaires à toutes les demandes sortantes ou intercepter des réponses d'erreur du serveur.

Un scénario d'utilisation d'intercepteur consiste à ajouter un en-tête d'autorisation pour l'authentification à toutes les demandes.

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

import { Observable } from 'rxjs';

@Injectable()
export class EnsureHttpsInterceptor implements HttpInterceptor {
  intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {
    // clone request and replace 'http://' with 'https://' at the same time
    const secureReq = req.clone({
      url: req.url.replace('http://', 'https://')
    });
    // send the cloned, "secure" request to the next handler.
    return next.handle(secureReq);
  }
}
```

Lazy loading

Lazy Loading

Une application volumineuse doit être chargée dans le navigateur pour fonctionner

Pour éviter un chargement long, il est possible de demander certains modules « au bon moment » (lazy loading)

En configurant l'intégralité du Routing de l'application dans le module AppRoutingModuleModule, on serait amené à **importer tous les modules de l'application avant son démarrage**. A titre d'exemple, plus l'application sera riche, plus la page d'accueil sera lente à charger par effet de bord.

Pour éviter ces problèmes de "scalability", **Angular permet de charger les modules à la demande** (*i.e.* : "*Lazy Loading*") afin de ne pas gêner le chargement initial de l'application.

Lazy loading

Configuration du Lazy Loading

La configuration du "Lazy Loading" se fait au niveau du "Routing".

Le module de "Routing" `AppRoutingModule` peut **déléguer la gestion du "Routing" d'une partie de l'application à un autre module**. Ce module "Lazy Loaded" sera donc chargé de façon asynchrone à la visite des "routes" dont il est en charge

```
1  @NgModule({  
2    declarations: [  
3      AppComponent,  
4      LandingComponent  
5    ],  
6    imports: [  
7      AppRoutingModule,  
8      BrowserModule  
9    ],  
10   bootstrap: [AppComponent]  
11 })  
12 export class AppModule {  
13 }
```

Lazy loading

Configuration du Lazy Loading

src/app/app-routing.module.ts

```
1 export const appRouteList: Routes = [  
2   {  
3     path: 'landing',  
4     component: LandingComponent  
5   },  
6   {  
7     path: 'book',  
8     loadChildren: './views/book/book-routing.module#BookRoutingModule',  
9   },  
10  {  
11    path: '**',  
12    redirectTo: 'landing'  
13  }  
14 ];  
15  
16 @NgModule({  
17   exports: [  
18     RouterModule  
19   ],  
20   imports: [  
21     HttpClientModule,  
22     RouterModule.forRoot(appRouteList)  
23   ]  
24 })
```

Lazy loading

Configuration du Lazy Loading

src/app/views/book/book-routing.module.ts

```
1 export const bookRouteList: Routes = [
2   {
3     path: 'search',
4     component: BookSearchComponent
5   },
6   {
7     path: '**',
8     redirectTo: 'search'
9   }
10 ];
11
12 @NgModule({
13   imports: [
14     BookModule,
15     RouterModule.forChild(bookRouteList)
16   ]
17 })
18 export class BookRoutingModule {
19 }
```

Cette configuration **délègue** le "Routing" de toute la partie /book/... de l'application au module **BookRoutingModule**.

Lazy loading

Configuration du Lazy Loading



Pour profiter du "Lazy Loading", assurez-vous que les modules "Lazy Loaded" ne sont jamais chargé explicitement (*"Eagerly Loaded"*)!

Il faut donc épurer au maximum les imports d' AppModule .



La syntaxe

```
loadChildren: './views/book/book-routing.module#BookRoutingModule'
```

est un raccourci pour le chargement asynchrone de la classe

```
BookRoutingModule ;
```

```
1 loadChildren: () => import('./views/book/book-routing.module')
2   .then(module => module.BookRoutingModule);
```

L'internalisation

L'Approche Native

<https://angular.io/guide/i18n>

Angular

Angular is a platform for building mobile and desktop web applications. Join the community of millions of developers who build compelling user interfaces with Angular.

angular.io



Cette approche est actuellement limitée par le problème d'utilisation de la traduction impérativement dans le code TypeScript : <https://github.com/angular/angular/issues/11405>

L'internalisation

Angular Translate

Une alternative au problème décrit précédemment est d'utiliser la librairie indépendante Angular Translate développée également par Olivier Combe de la Core Team Angular.

<http://www.ngx-translate.com/>

```
import { TranslateLoader, TranslateModule, TranslateService } from '@ngx-translate/core';
```

NGX-Translate: The i18n library for Angular 2+

NGX-Translate: The internationalization (i18n) library for Angular 2+

www.ngx-translate.com

```
    }),  
    TranslateModule.forRoot({  
      loader: {  
        provide: TranslateLoader,  
        useFactory: HttpLoaderFactory,  
        deps: [HttpClient]  
      }  
    })  
  ],  
  exports: [TranslateModule],  
});
```

Travaux pratiques

Angular, mise en œuvre des bonnes pratiques

TP

Fonctionnement interne d'Angular



ZoneJS : le concept

Angular a introduit Zone.js pour gérer la détection des changements.

Cela permet à Angular de décider quand l'interface utilisateur doit être actualisée.

En général, vous n'avez à vous soucier de rien de tout cela, car Zone.js fonctionne nativement.

Zone.js patch toutes les API asynchrones courantes telles que **setTimeout**, **setInterval**, **Promise**, etc.

La zone est un mécanisme d'interception et de suivi du travail asynchrone.

Pour chaque opération asynchrone, Zone.js crée une tâche.

Une tâche est exécutée dans une zone.

```
private : true,  
"dependencies": {  
  "@angular/animations": "^6.0.3",  
  "@angular/common": "^6.0.3",  
  "@angular/compiler": "^6.0.3",  
  "@angular/core": "^6.0.3",  
  "@angular/forms": "^6.0.3",  
  "@angular/http": "^6.0.3",  
  "@angular/platform-browser": "^6.0.3",  
  "@angular/platform-browser-dynamic": "^6.0.3",  
  "@angular/router": "^6.0.3",  
  "bootstrap": "^4.1.3",  
  "core-js": "^2.5.4",  
  "jquery": "^1.9.1",  
  "popper.js": "^1.14.3",  
  "rxjs": "^6.0.0",  
  "zone.js": "^0.8.26"  
},
```

Choisir RxJS

RxJS (pour Reactive Extensions for JavaScript) est l'implémentation javascript de ReactiveX.

ReactiveX est une API basée sur le pattern Observer et la programmation fonctionnelle pour gérer des événements asynchrones. ReactiveX est activement développé par Microsoft.

Aujourd'hui, RxJS est largement utilisée dans Angular, en particulier dans **HTTP** et **EventEmitter**



Choisir RxJS

Observable vs Promise :

Promise gère un événement unique lorsqu'une opération asynchrone se termine ou échoue.

Il résout la promesse. Auquel cas, il fournit à l'objet Promise une et une seule valeur. Dans ce cas, il fait appel à la fonction **resolve()**.

Il rejette la promesse. Dans ce cas, il fournit la raison du rejet. Il fait appel à la fonction **reject()**.

Un Observable est comme unStream (dans de nombreuses langues) et permet de passer zéro ou plusieurs événements où le rappel est appelé pour chaque événement.

Observable est souvent préférable à Promise car il fournit les fonctionnalités de Promise et plus.

Avec Observable, peu importe si vous voulez gérer 0, 1 ou plusieurs événements.

Vous pouvez utiliser la même API dans chaque cas.

Choisir RxJS

Observable a aussi l'avantage sur Promise d'être annulable.

Si le résultat d'une requête HTTP adressée à un serveur ou d'une opération asynchrone coûteuse n'est plus nécessaire, la variable **Subscription** d'une variable Observable permet d'annuler l'abonnement, alors qu'une variable Promise appellera éventuellement le rappel réussi ou échoué, même si vous ne le faites pas.

**Observable fournit des opérateurs : like map, forEach, reduce, ...
semblable à un tableau.**

Utilisation des observables

Observable : Producteur d'une sequence de valeurs.

Observer : Consommateur des valeurs observées.

Subscriber : Connecte les observers au observables.

Operator: Route et transforme les valeurs.

Utilisation des observables

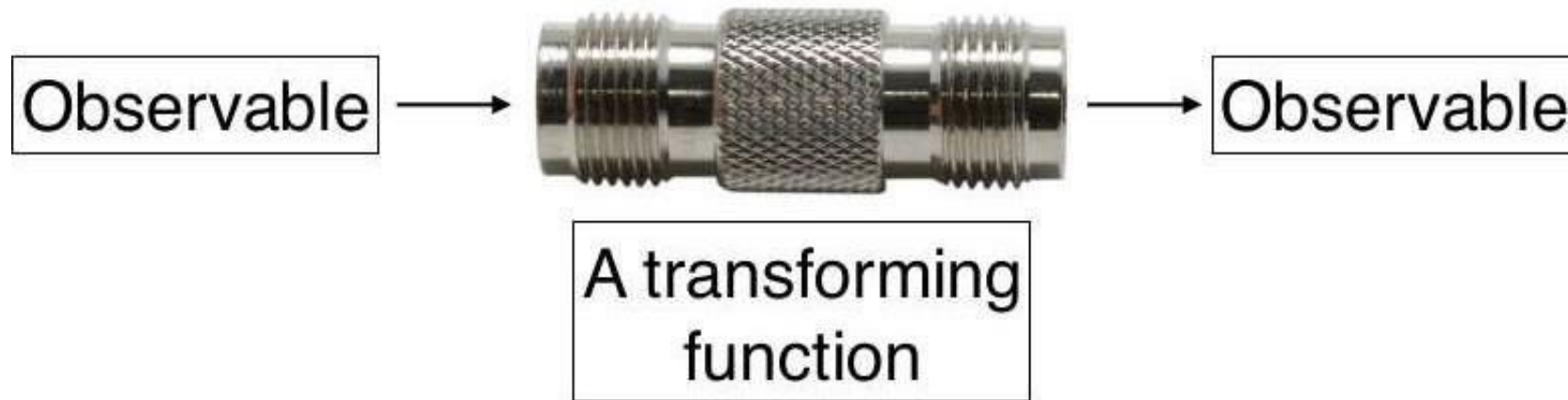
Utilisation des Observables



Utilisation des observables

```
let beers = [  
  {name: "Stella", country: "Belgium", price: 9.50},  
  {name: "Sam Adams", country: "USA", price: 8.50},  
  {name: "Bud Light", country: "USA", price: 6.50},  
  {name: "Brooklyn Lager", country: "USA", price: 8.00},  
  {name: "Sapporo", country: "Japan", price: 7.50}  
];
```

Utilisation des observables



```
observableBeers  
  .filter(beer => beer.price < 8))
```

Création, combinaison, opérateurs clés

Creating the
Observable

```
observableBeers = Rx.Observable.from(beers)  
  .filter(beer => beer.price < 8)  
  .map(beer => beer.name + ": $" + beer.price);  
  
observableBeers  
  .subscribe(  
    beer => console.log(beer),  
    err  => console.error(err),  
    ()   => console.log("Streaming is over")  
  );
```

Création, combinaison, opérateurs clés

Operators

```
observableBeers = Rx.Observable.from(beers)  
    .filter(beer => beer.price < 8)  
    .map(beer => beer.name + ": $" + beer.price);
```

Observer

```
observableBeers  
    .subscribe(  
        beer => console.log(beer),  
        err => console.error(err),  
        () => console.log("Stream is over")  
    );
```

Création, combinaison, opérateurs clés

No
streaming yet

```
→ observableBeers = Rx.Observable.from(beers)  
  .filter(beer => beer.price < 8)  
  .map(beer => beer.name + ": $" + beer.price);
```

Streaming
begins

```
observableBeers  
→ .subscribe(  
  beer => console.log(beer),  
  err  => console.error(err),  
  ()   => console.log("Streaming is over")  
);
```

Création, combinaison, opérateurs clés

```
function getDrinks() {  
  let beers = Rx.Observable.from([  
    {name: "Stella", country: "Belgium", price: 9.50},  
    {name: "Sam Adams", country: "USA", price: 8.50},  
    {name: "Bud Light", country: "USA", price: 6.50}  
  ]);  
  
  let softDrinks = Rx.Observable.from([  
    {name: "Coca Cola", country: "USA", price: 1.50},  
    {name: "Fanta", country: "USA", price: 1.50},  
    {name: "Lemonade", country: "France", price: 2.50}  
  ]);  
  
  return Rx.Observable.create( observer => {  
    observer.next(beers);      // pushing the beer pallet (observable)  
    observer.next(softDrinks); // pushing the soft drinks pallet (observable)  
  }  
);  
}  
  
getDrinks()  
→ .flatMap(drinks => drinks) // unloading drinks from pallets  
  .subscribe(  
    drink => console.log("Subscriber got " + drink.name + ": " + drink.price )  
  );
```

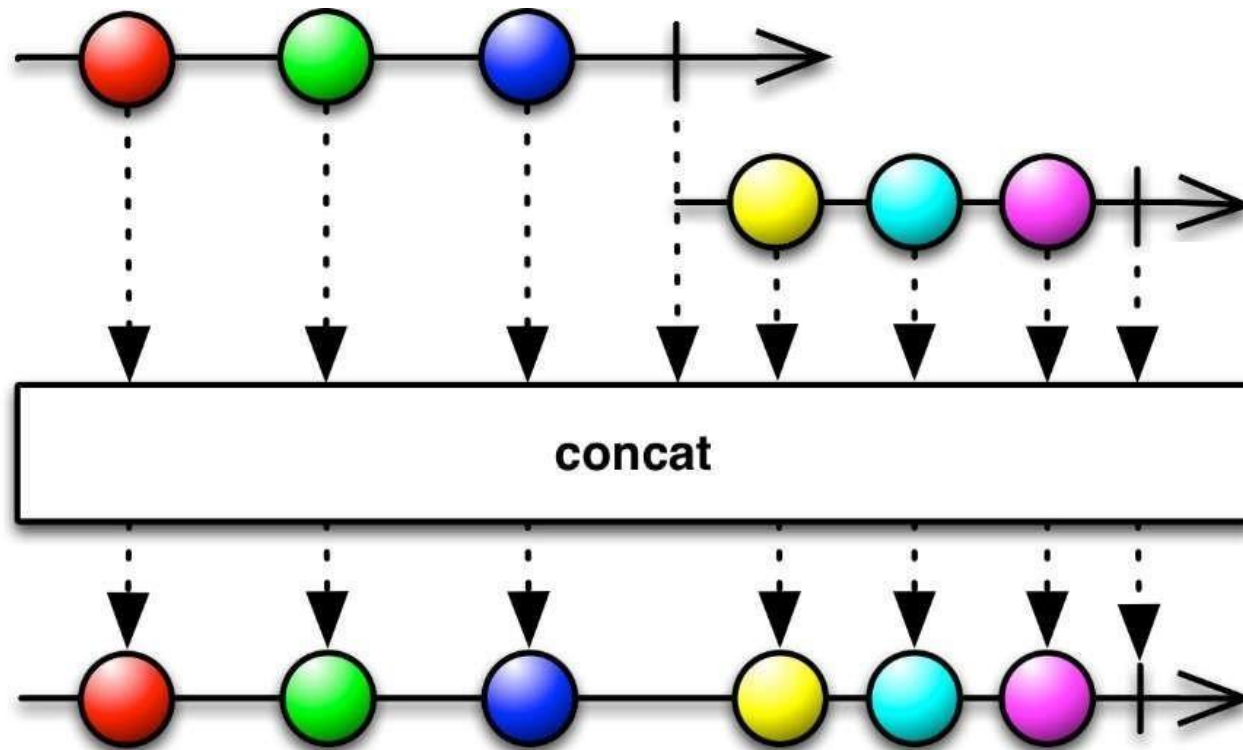

Création, combinaison, opérateurs clés

```
// Emulate HTTP requests
let fourSecHTTPRequest = Rx.Observable.timer(4000).mapTo('First response');

let oneSecHTTPRequest = Rx.Observable.timer(1000).mapTo('Second response');

Rx.Observable
    .concat(fourSecHTTPRequest, oneSecHTTPRequest)
    .subscribe(res => console.log(res));
```

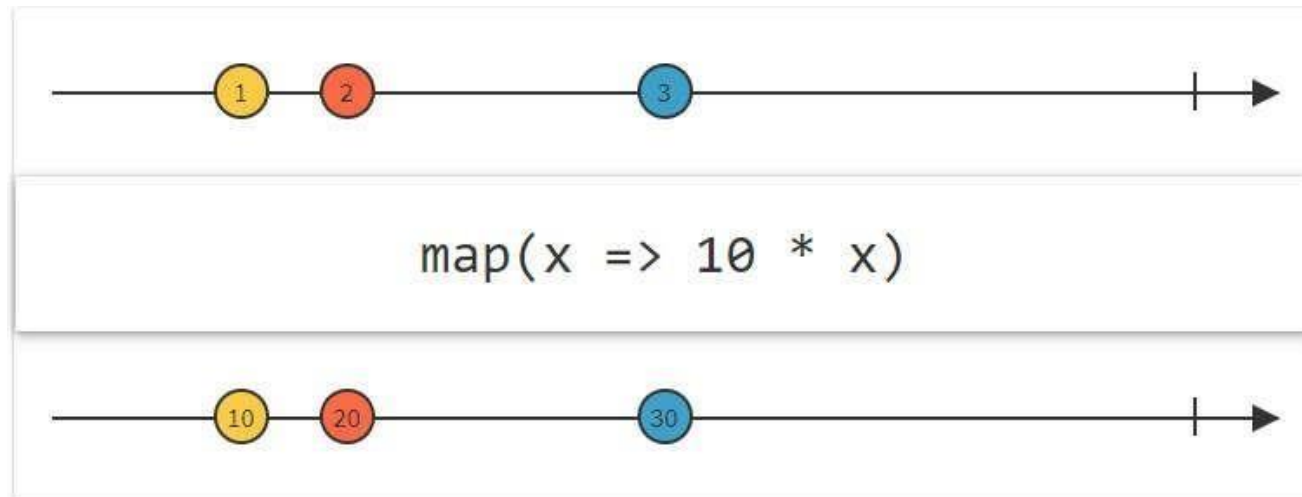

Création, combinaison, opérateurs clés



Création, combinaison, opérateurs clés

Map

Emettre uniquement les éléments d'un observable qui réussissent un test de prédicat



Compilation ahead of time

Angular propose deux types de compilation de templates :

JiT (Just-in-Time) et AoT (Ahead-of-Time).

Pourquoi ?

Quelles sont les différences ? Lequel dois-on utiliser ?

Compilation ahead of time

JiT : Just-in-Time

Par défaut, la compilation des templates d'une application va être effectuée pendant l'exécution de l'application, c'est à dire que Angular va compiler à la volée les templates. C'est ce qu'on appelle la compilation JiT qui est utilisée via la commande ***ng build*** ou ***ng build --prod --no-aot***.



main.bundle.js généré par la build

```
1  /**/ }),  
2  
3  /**/ "../..../src/app/components/home/home.component.html":  
4  /**/ (function(module, exports) {  
5  
6  module.exports = "{{user|json}}\n\n<button (click)='logoff()>logoff</button>"  
7  
8  /**/ }),
```

Compilation ahead of time

JiT : Just-in-Time

Ce mécanisme a deux effets négatifs :

-  Le bundle Javascript va être plus gros puisque les sources de l'application devront intégrer le compilateur de templates (dans le fichier **vendor.bundle.js**).
-  L'application va devoir compiler les templates lors de son exécution ce qui aura nécessairement un impact négatif sur le temps d'affichage

Compilation ahead of time

La compilation des templates est invariante, c'est-à-dire qu'un même template engendrera toujours la même fonction (de la même manière que la compilation TypeScript par exemple).

Cela veut dire que la compilation des templates peut faire partie du process de build, puisqu'elle n'est pas dépendante de son contexte d'utilisation.

C'est sur cette constatation que se base le principe de la compilation AoT, utilisable via les commandes ***ng build --aot*** ou ***ng build --prod***.

Plutôt que de compiler les templates au lancement de l'application, cette compilation va être effectuée lors de la phase de build, directement par webpack.

Compilation ahead of time

Le bundle généré ne contiendra plus les templates HTML mais directement les templates compilés.

Si vous inspectez le fichier *main.bundle.js* généré par la build, vous trouverez des portions de code contenant vos templates compilés, comme ci-dessous :

```
1  function View_HomeComponent_0(_l) {
2      return __WEBPACK_IMPORTED_MODULE_1__angular_core__["_30" /* euid */](0, [(_l()
3          var ad = true;
4          var _co = _v.component;
5          if (('click' === en)) {
6              var pd_0 = (_co.logoff() !== false);
7              ad = (pd_0 && ad);
8          }
9          return ad;
10     }, null, null)), (_l()), __WEBPACK_IMPORTED_MODULE_1__angular_core__["_28
11     var _co = _v.component;
12     var currVal_0 = __WEBPACK_IMPORTED_MODULE_1__angular_core__["_29" /* eunv
13     _ck(_v, 0, 0, currVal_0);
14     });
15 }
```

Compilation ahead of time

Sur la même application, la compilation **AoT** donne les bundles suivants :

main.bundle.js : 159k

vendor.bundle.js : 2281k

On gagne énormément sur la taille du fichier **vendor.bundle.js**, puisque le compiler Angular n'est plus embarqué.

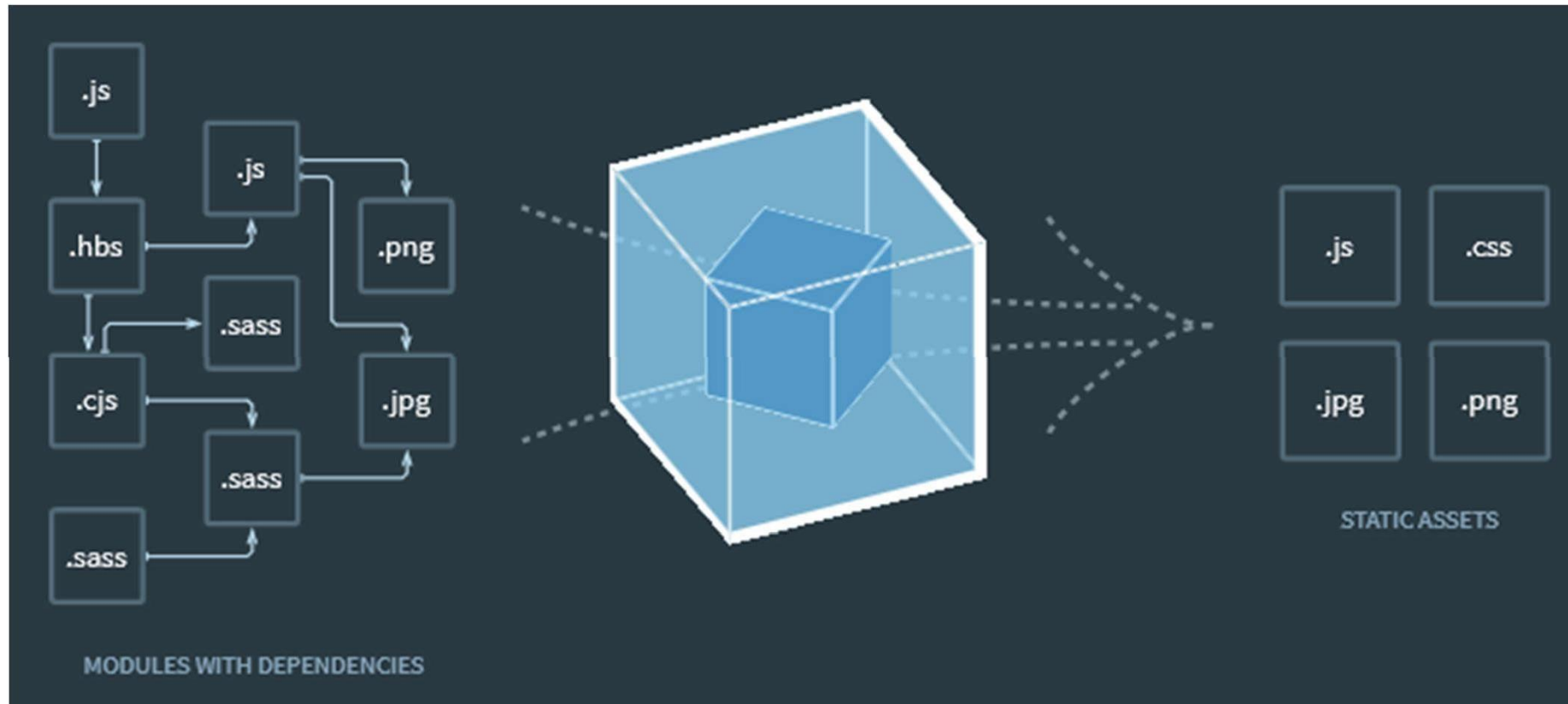
Webpack Bundle Analyzer

Webpack est ce qu'on appelle un module bundler. Technologie principalement utilisée en front-end, elle va permettre de rassembler tous les fichiers de votre code source et les rassembler en un seul fichier.

Cela va des fichiers javascript jusqu'aux images en passant par les fichiers css.

Webpack va permettre également d'optimiser le poids de notre code source et donc contribuer aux bonnes performances de votre application.

Webpack Bundle Analyzer



Webpack Bundle Analyzer

Pour l'installation de webpack analyser bundler il suffit de modifier le fichier **package.json** comme suit:

```
"build-report": "ng build --stats-json",  
"bundle-report": "webpack-bundle-analyzer dist/reactiveForms/stats.json",
```

Travaux pratiques

Fonctionnement interne d'Angular

TP

Création de composants distribuables



Les web components



Ensembles de spécifications (W3C) définissant des API permettant la création de balises réutilisables

`<CustomComponent>...</ CustomComponent >`

Documentation officielle :

https://developer.mozilla.org/fr/docs/Web/Web_Components

Les web components



Les composants web permettent aux développeurs de créer des balises HTML personnalisées et réutilisables.

"Les composants web permettent de combiner plusieurs éléments pour créer des composants d'interface graphique (**widgets**) réutilisables avec un niveau de richesse et d'interactivité sans commune mesure avec ce qui est aujourd'hui possible en se limitant aux CSS".

Les web components

Les composants web s'appuient sur plusieurs interfaces de programmation (API) du W3C :

Templates : squelettes pour des éléments HTML instanciables

Shadow DOM : ce qui sera public ou privé dans les éléments

Custom elements : pour créer et enregistrer de nouveaux éléments HTML et les faire reconnaître par le navigateur



Les web components

Template

La spécification des Templates, décrit la création de fragment HTML.
Ces fragments ne sont pas chargés au démarrage de la page.
Ces fragments sont instanciés plus tard.



Les web components

Template

```
<template id="pbTemplate">
  <style>
    .container { height: 10px; border: solid 1px #ddd }
    .progress { height: 10px; background-color: orange; }
  </style>
  <div class="container">
    <div class="progress"></div>
  </div>
</template>
```

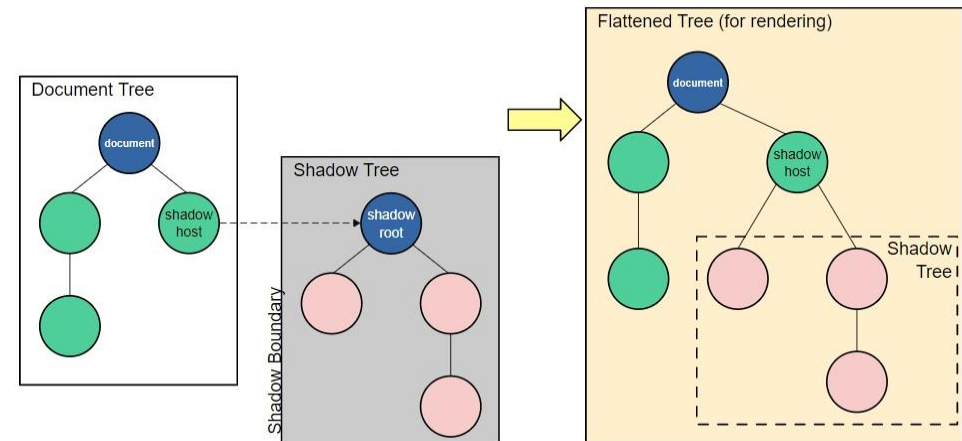
Les web components

Shadow DOM

Un Shadow DOM (DOM Fantôme) doit toujours être lié à un élément existant. L'élément peut être soit un élément au sein d'un fichier HTML, soit un élément du DOM produit par un script. Il peut s'agir d'un élément natif comme

<div> ou <p>, ou un élément personnalisé tel que <mon-element>. Pour attacher le Shadow DOM à un élément il suffit de faire **Element.attachShadow()** comme dans cet exemple :

```
<html>
<head></head>
<body>
  <p id="element-hote"></p>
  <script>
    // Créer le Shadow DOM sur l'élément <p>
    var shadow =
      document.querySelector('#element-hote')
        .attachShadow({mode: "open"});
  </script>
</body>
</html>
```



Les web components

Custom elements

La spécification Custom elements définit comment concevoir et utiliser des nouveaux types d'élément DOM

HTML 5 supporte le fait que l'on puisse inventer et utiliser nos propres éléments. C'est ce que l'on appelle un custom element.

Imaginez maintenant qu'à la place du *div* et de son *id*, on avait directement un élément `<CustomComponent></ CustomComponent >`

Les web components

Custom elements

Il suffirait alors de créer notre élément et de lui créer un shadow DOM.

Mais ce n'est pas qu'une histoire de balise.

Quand on parle de HTML 5, il faut directement penser HTML ET Javascript.

Car la plupart de ce que l'on trouve en HTML, on peut le créer et le manipuler en javascript

Pour que notre balise soit reconnue, qu'elle ait des propriétés custom de base ou que l'on puisse la créer aussi simplement qu'un autre élément HTML natif, il faut "l'enregistrer".

Pour ça, on a la fonction `document.registerElement(nom-de-l-element, options)`.

Les web components

Custom elements

Cette fonction va nous permettre de dire au navigateur qu'il va devoir prendre en compte notre nouvel élément et agir avec comme si c'était un élément de base du HTML.

Il va donc le surveiller pendant toute sa durée de vie et exécuter les différents callbacks qu'il faut, au bon moment, sur l'élément.

Par exemple, pour notre élément **CustomComponent**, on va créer une "classe" **CustomComponent** en utilisant la fonction *Object.create(proto[, propriétés])*

Les web components

Custom elements

```
<html>
  <body>
    <hello-world></hello-world>
    <script>
      var HelloWorld = document.registerElement('hello-world', {
        prototype: Object.create(HTMLElement.prototype, { createdCallback: {
          // exécuté à chaque création d'un élément <hello-world>
          value: function() {
            var root = this.createShadowRoot();
            var content = document.createElement("h1");
            content.innerText = "Hello world!";
            root.appendChild(content);
          }
        }
      });
    </script>
  </body>
</html>
```

Les web components

HTML 5 avec la notion de web component a aidé la création de Angular

C'est un composant essentiel pour la création des directives Angular



Les décorateurs

Les **décorateurs** permettent par simple annotation (i.e.: `@magic() prop: string;`) de modifier le comportement d'une classe, d'une propriété ou d'une fonction.

Le rôle des décorateurs est de factoriser certains "patterns" en surchargeant le comportement natif.

Cette fonctionnalité pythonique est prévue dans les versions futures d'ECMAScript (<https://tc39.github.io/proposal-decorators/>).

Mais étant donné qu'Angular en avait besoin (d'où l'idée de l'**AtScript**), **TypeScript** l'a implémentée en avance.

Les décorateurs

- Les décorateurs représentent une nouvelle fonctionnalité ajoutée seulement en TypeScript 1.5 pour supporter Angular.
- Un décorateur est une façon de faire de la méta-programmation et ressemblent beaucoup aux annotations qui sont principalement utilisées en java, c#, et python.
- Les décorateurs peuvent modifier leur cible (classes, méthodes, etc...) et par exemple modifier les paramètres ou le résultat retourné, appeler d'autres méthodes quand la cible est appelée, ou ajouter des métadonnées.
- En TypeScript, les annotations sont préfixées par @, et peuvent être appliquées sur une classe, une propriété de classe, une fonction, ou un paramètre de fonction.
- Pas sur un constructeur en revanche, mais sur ses paramètres oui.

Les décorateurs

❖ Décorateurs de fonctions :

- Pour mieux comprendre ces décorateurs, on illustre un exemple de décorateur de fonction **@Log()**, qui va écrire le nom de la méthode à chaque fois qu'elle sera appelée.

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

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

 - **target** : la méthode ciblée par notre décorateur
 - **name** : le nom de la méthode ciblée
 - **descriptor** : le descripteur de la méthode ciblée, par exemple est-ce que la méthode est énumérable, etc...

```
class RaceService {  
  
  @Log()  
  getRaces() {  
    // call API  
  }  
  
  @Log()  
  getRace(raceId) {  
    // call API  
  }  
}
```

Les décorateurs

❖ Décorateurs de propriétés :

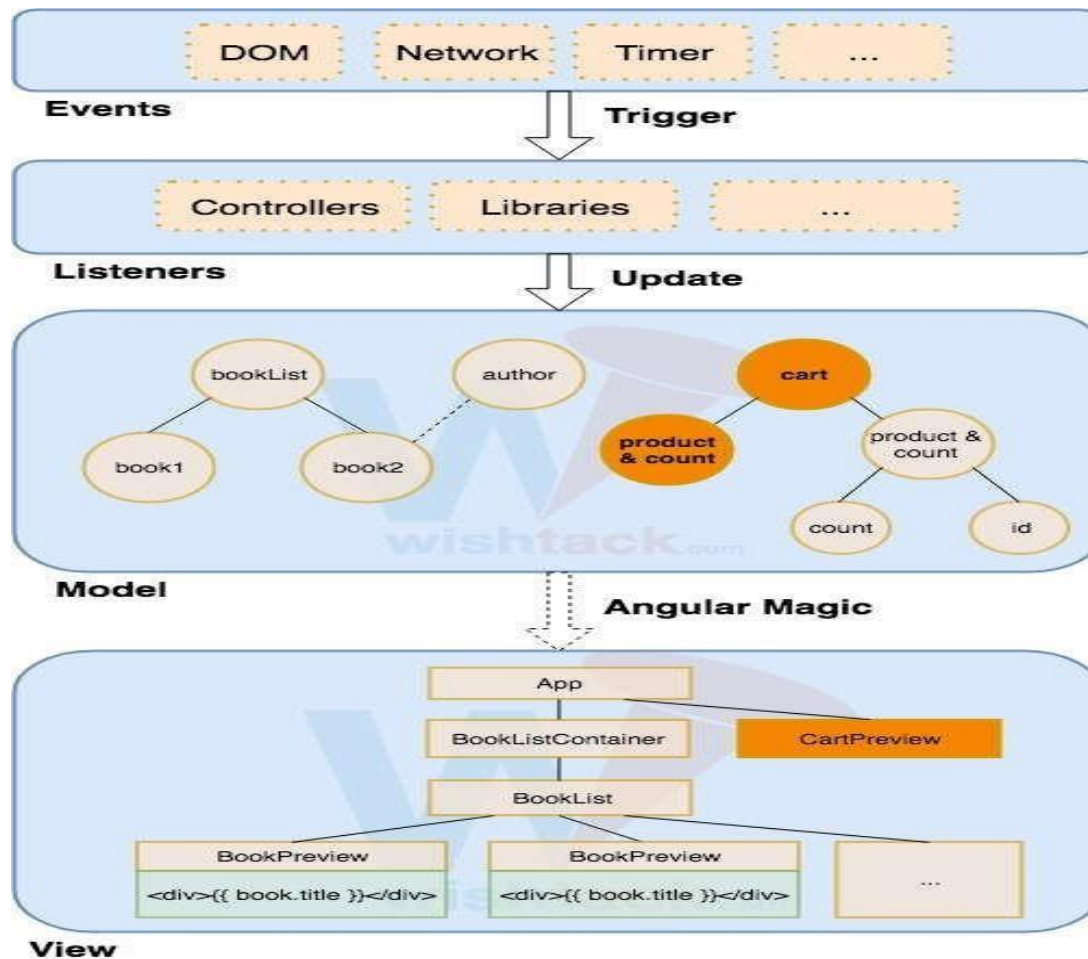
- Dans cet exemple on retourne la chaîne de valeur "test" au lieu de la valeur de la propriété
- La valeur d'une propriété ne peut être modifiée. On utilise donc un accesseur (get)

```
function Override(label: string) {  
  return function (target: any, key: string) {  
    Object.defineProperty(target, key, {  
      configurable: false,  
      get: () => label  
    });  
  }  
}  
  
class Test {  
  @Override('test')    // invokes Override, which returns the decorator  
  name: string = 'pat';  
}  
  
let t = new Test();  
console.log(t.name);  // 'test'
```

Le change detection mode

Le Change Detection Mode

Peu importe la librairie ou le framework utilisés, il est nécessaire de synchroniser le modèle avec la vue



Le change detection mode

Le Change Detection Mode

Chaque composant dispose d'un "change detector" qui comme son nom l'indique se charge de détecter les changements de modèle concernant la vue et de mettre à jour la vue en conséquence.

Le "**Change Detector**" est fortement couplé à la vue du composant associé.

Cela lui permet d'analyser la liste des expressions utilisées dans la vue (**i.e. Template Interpolation ou Property Binding**) en les comparant à la dernière valeur retournée par chacune d'elles.

Si la valeur retournée par l'expression change, la vue est mise à jour en conséquence.

THINGS THAT CAUSE CHANGE

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

Le change detection mode

Le Change Detection Mode

Pour détecter les changements, Angular utilise la librairie Zone.js dont le rôle est d'encapsuler et d'intercepter tous les appels asynchrones (e.g. **setTimeout**, **event listeners** etc...).

Avant le chargement d'Angular, **Zone.js** procède au Monkey patching des fonctions natives permettant d'inscrire des "callbacks" associés à des traitements asynchrones (e.g. **setTimeout**) afin de pouvoir détecter chaque "tick" et notifier Angular

Le change detection mode

1. Déclenchement de la "change detection"

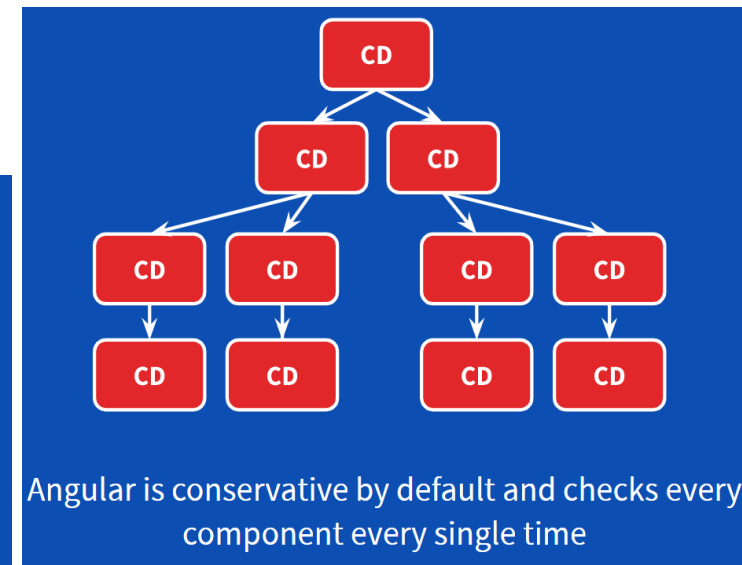
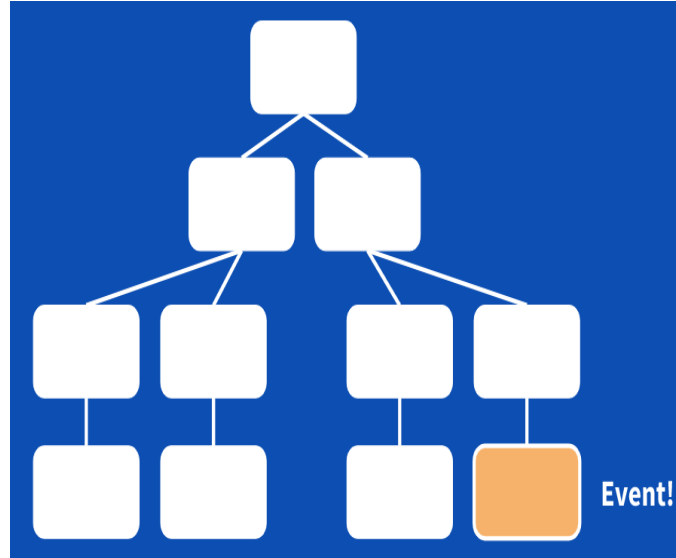
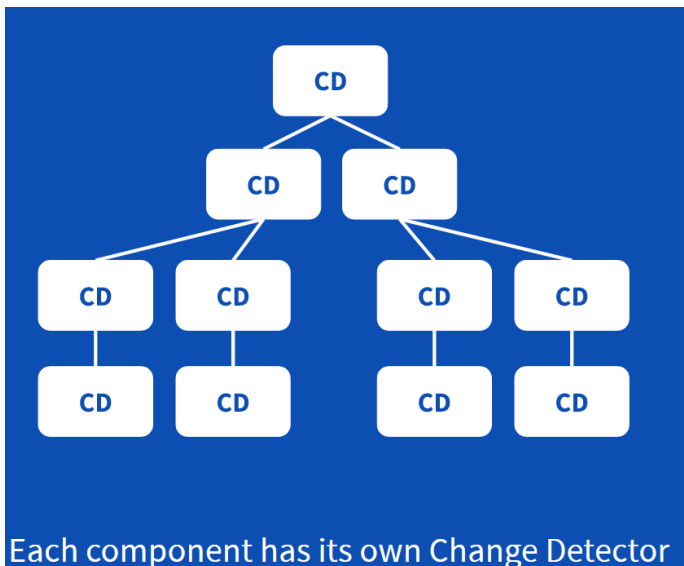
A la fin de chaque "tick" (détecté grâce à *Zone.JS*), Angular déclenche la "Change Detection" du "Root Component".

2. "Change Detection" de chaque composant

Le "Change Detector" du composant compare les anciennes et les nouvelles valeurs de chaque expression utilisée dans les bindings (Template Interpolation ou Property Binding).

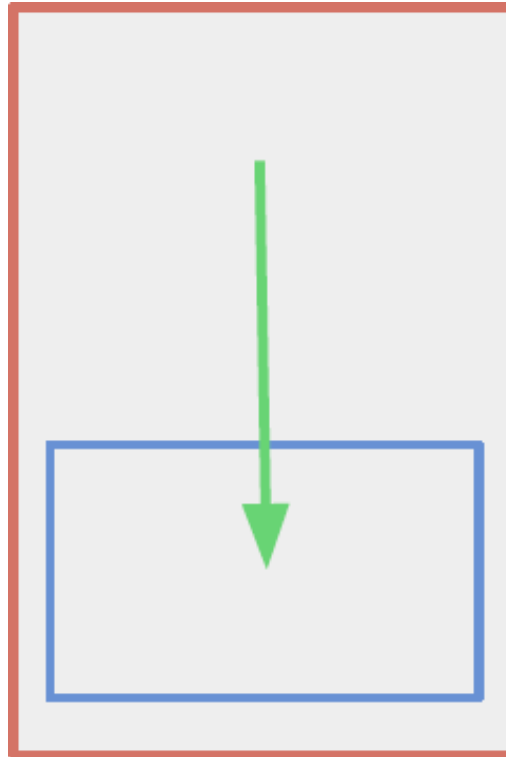
3. Mise à jour de la vue si nécessaire

En cas de changement, l'élément concerné est mis à jour dans la vue



Communication entre composants

Descendante Parent => Enfant



Communication entre composants

```
@Component({
  selector: 'parent',
  template: `
<div>
  Parent content
  <child [param]="myVar"></child>
  Parent content
</div>
`
})
class ParentComponent {
  myVar = 'hello';
}
```

```
@Component({
  selector: 'child',
  template: '<div>Child: {{param}}</div>'
})
class ChildComponent {
  @Input() param: string;
}
```

Input

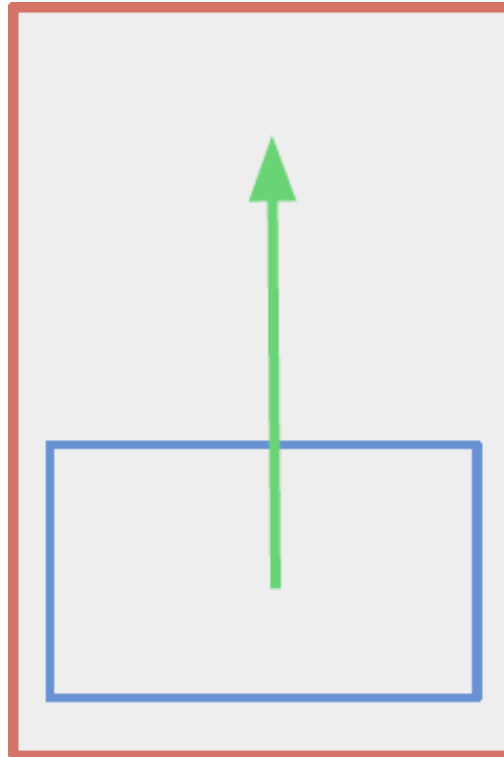
```
<html>
  <parent></parent>
</html>
```

Output

```
<html>
  <parent>
    <div>
      Parent content
      <child>
        Child hello
      </child>
      Parent content
    </div>
  </parent>
</html>
```

Communication entre composants

Ascendante Enfant => Parent



Communication entre composants

EventEmitter

Pour pouvoir utiliser notre Output, nous avons besoin d'importer et de binder une nouvelle instance d'**EventEmitter** :

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

@Component({...})
export class CounterComponent {

  // ...

  @Output()
  change = new EventEmitter();

  // ...

}
```

Communication entre composants

EventEmitter

En utilisant TypeScript, nous pouvons dire que notre **change output** est de type **EventEmitter**. Dans notre cas nous allons émettre un type **number** :

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

@Component({...})
export class CounterComponent {

  // ...

  @Output()
  change: EventEmitter = new EventEmitter();

  // ...

}
```

Communication entre composants

EventEmitter

En résumé, nous avons créé une propriété **change**, et lié une nouvelle instance de **EventEmitter**.

Nous pouvons maintenant simplement appelé la méthode **this.change**.

Nous pouvons appeler **.emit()** pour émettre notre événement à notre composant parent.

```
@Component({...})
export class CounterComponent {

  @Input()
  count: number = 0;

  @Output()
  change: EventEmitter = new EventEmitter();

  increment() {
    this.count++;
    this.change.emit(this.count);
  }

  decrement() {
    this.count--;
    this.change.emit(this.count);
  }
}
```

Communication entre composants

EventEmitter

Ceci va émettre un changement pour notre **(change)** listener que nous avons configuré dans le parent.

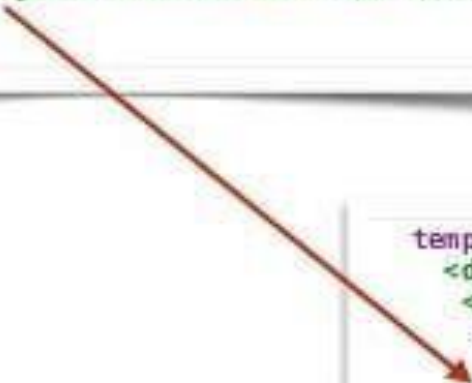
Il va appeler la fonction callback **countChange(\$event)**, et la donnée associée à notre événement va être transmise par la propriété **\$event**.

Projection de contenu, pilotage de composants enfants

Les composants avancés : Les projections

Parent

```
template:
  <div class="wrapper">
    <h2>Parent</h2>
    <div>This div is defined in the Parent's template</div>
    <child>
      <div>Parent projects this div onto the child </div>
    </child>
  </div>
```



Child

```
template:
  <div class="wrapper">
    <h2>Child</h2>
    <div>This is child's content</div>
    <ng-content></ng-content>
  </div>
```

----- insertion point

Un composant ne peut avoir qu'une seule template.

Le parent peut transmettre du HTML à un composant fils au runtime

Build et déploiement

```
scripts: {  
  "build": "ng build -prod",  
  "postbuild": "npm run deploy",  
  "predeploy": "rimraf ../server/src/main/resources/static &&  
    mkdirp ../server/src/main/resources/static",  
  "deploy": "copyfiles -f dist/** ../server/src/main/  
resources/static"  
}
```

Spring Boot app

static resources

Bundled Angular app

Documentation : génération dynamique

Compodoc

Compodoc est un outil open-source permettant de générer rapidement une documentation de votre application Angular.

Installation

La commande suivante installe le module `@compodoc/compodoc`.

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

Documentation : génération dynamique

Add *compodoc* script in package.json

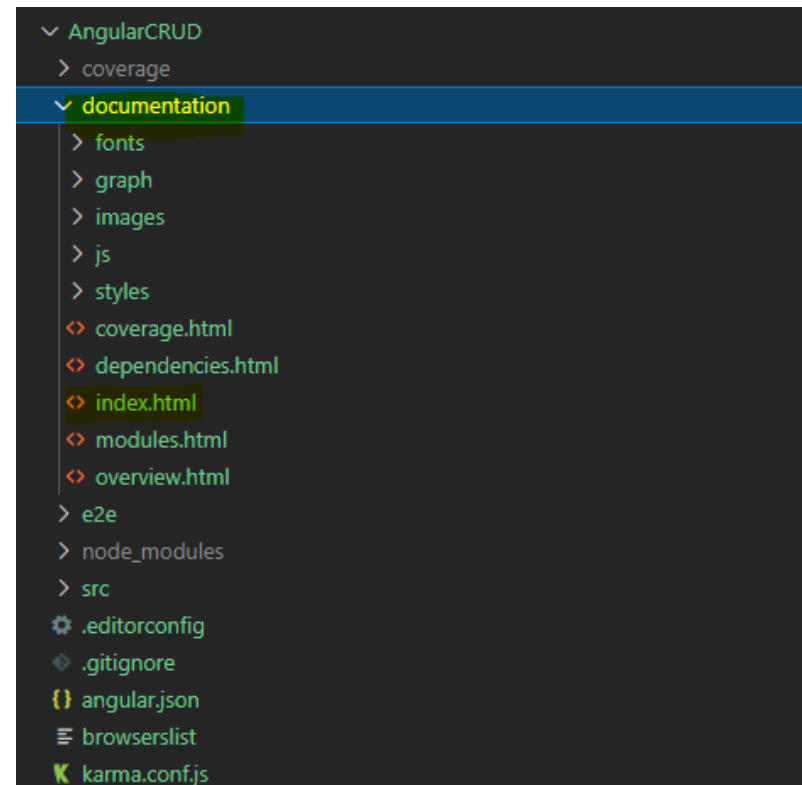
Now open package.json file and add the following command under "**scripts**" in JSON object:

```
"scripts": {  
  "ng": "ng",  
  "start": "ng serve",  
  "build": "ng build",  
  "test": "ng test",  
  "lint": "ng lint",  
  "e2e": "ng e2e",  
  "compodoc": "npx compodoc -p tsconfig.app.json"
```

Create Documentation

Now run following command to create documentation:

```
$ npm run compodoc
```



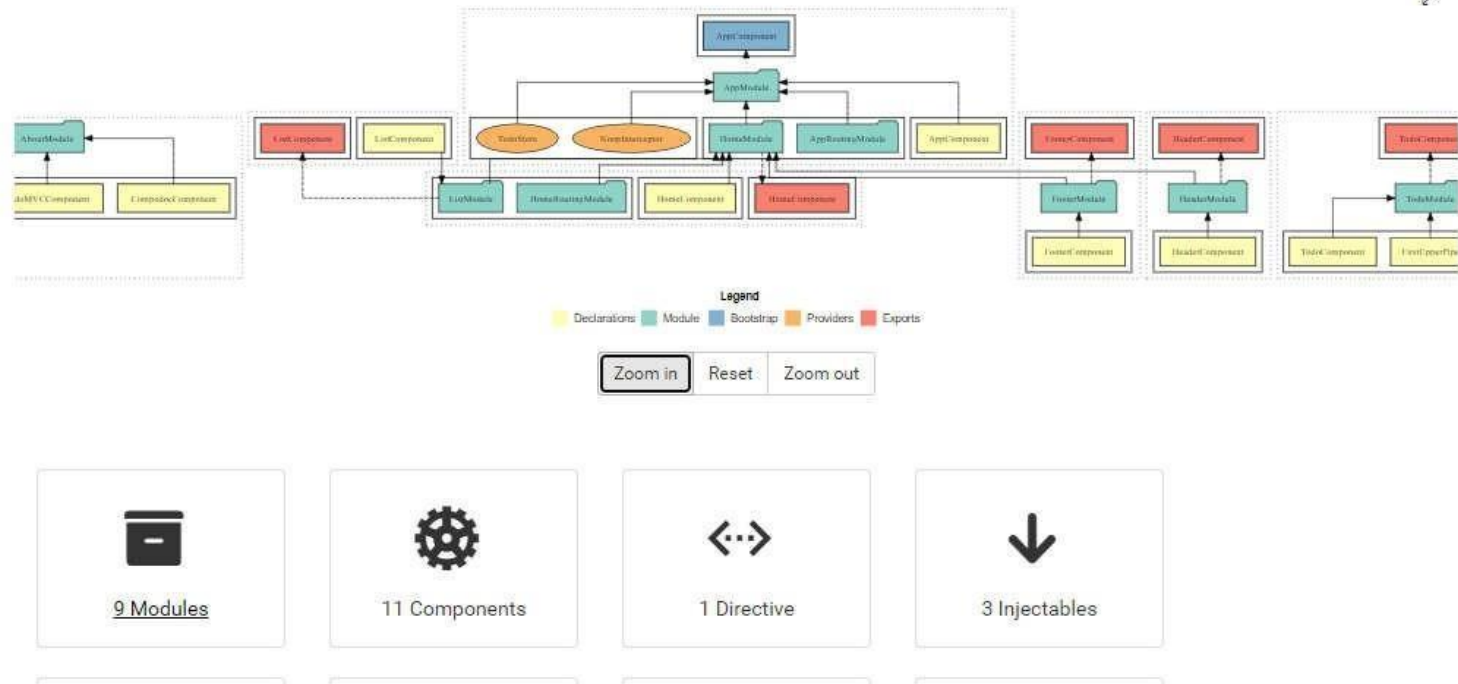
Documentation : génération dynamique

TodoMVC Angular documentation

Type to search

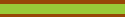
- Getting started
- Overview
- README
- LICENSE
- Dependencies
- Additional documentation
- Modules
- Components
- Classes
- Injectables
- Interceptors

Overview



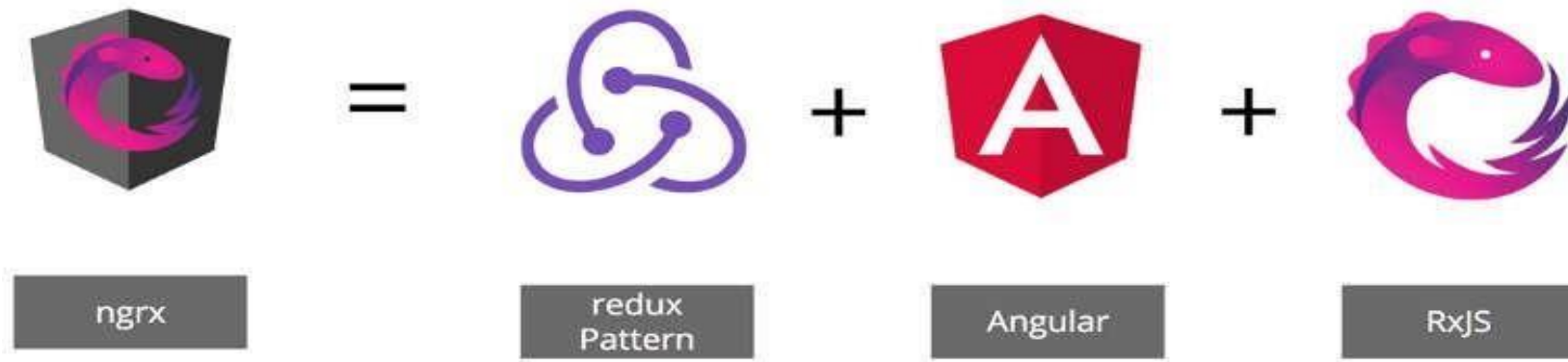
Communication entre composants

Composants riches et bibliothèques externes



Découverte de l'écosystème Angular

NGRX

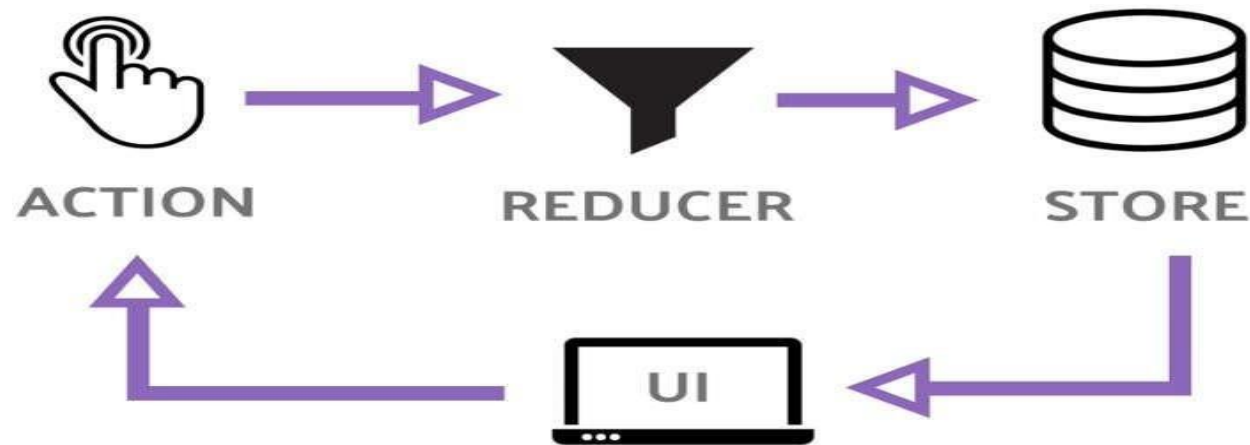


NGRX propose une conception du développement d'applications qui tourne autour **des actions utilisateurs et serveur**.

L'objectif est de supprimer la **manipulation de données** entre les composants et service **ANGULAR** pour les **centraliser** dans un **objet global**, qu'on pourra modifier seulement via des actions typées.

Découverte de l'écosystème Angular

REDUX



REDUX est un pattern né de FLUX, une architecture créée par Facebook. Il apporte un workflow de données unidirectionnelles grâce à un dispatcher, qui recueille des actions distribuées par le serveur ou par l'utilisateur. Il conserve la nouvelle instance d'une donnée dans un ou plusieurs stores qui mettent à jour la vue.

Découverte de l'écosystème Angular

Le Store, la base de tout

Le *store* est une fonction qui contient l'état des *reducers*, un *getter*, une fonction de *dispatch* et des *subscribers*.

Elle utilise les **Observables** pour communiquer la mise à jour des états dans les composants Angular.

Installation

```
npm install @ngrx/effects @ngrx/entity @ngrx/store @ngrx/store-devtools  
rxjs-compat—save
```

Découverte de l'écosystème Angular

AngularFire c'est la librairie officielle d'Angular pour Firebase

AngularFire = Angular + Firebase

Observable based - Use the power of RxJS, Angular, and Firebase.

Realtime bindings - Synchronize data in realtime.

Authentication - Log users in with a variety of providers and monitor authentication state.

Offline Data - Store data offline automatically with AngularFirestore.

Server-side Render - Generate static HTML to boost perceived performance or create static sites.

ngrx friendly - Integrate with ngrx using AngularFire's action based APIs.

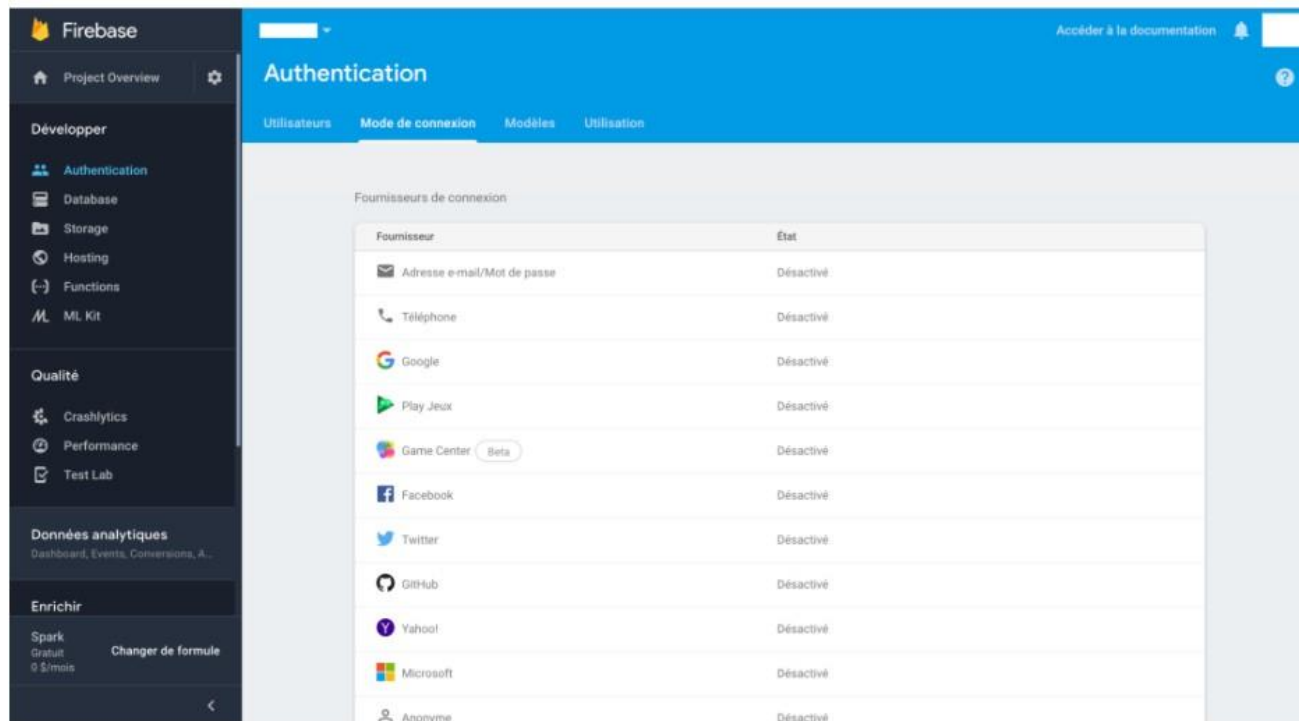
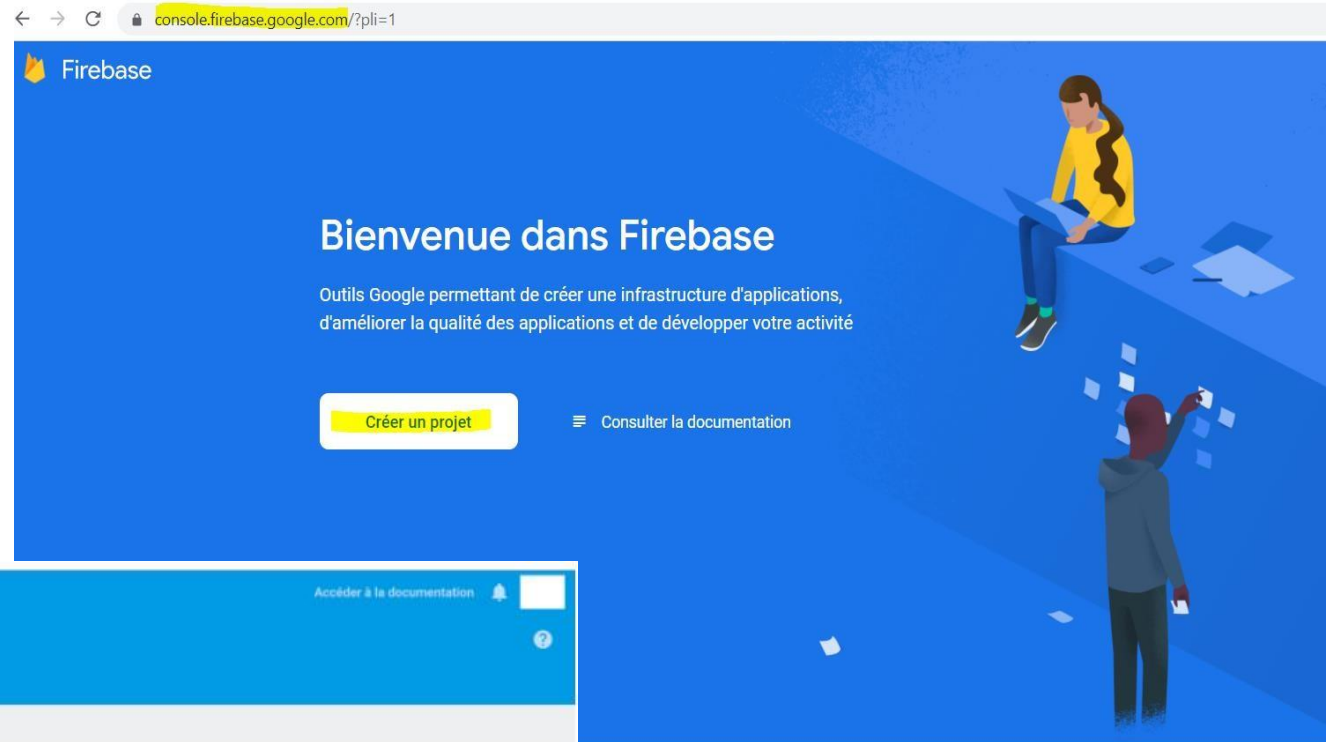
Manage binary data - Upload, download, and delete binary files like images, videos, and other blobs.

Call server code - Directly call serverless Cloud Functions with user context automatically passed.

Push notifications - Register and listen for push notifications

Découverte de l'écosystème Angular

Firestore



Découverte de l'écosystème Angular

AngularFire

Install

npm install firebase @angular/fire -save

```
import { Component } from '@angular/core';
import { AngularFirestore } from '@angular/fire/firestore';
import { Observable } from 'rxjs';

@Component({
  selector: 'app-root',
  template: `
    <ul>
      <li *ngFor="let item of items | async">
        {{ item.name }}
      </li>
    </ul>
  `
})
export class MyApp {
  items: Observable<any[]>;
  constructor(db: AngularFirestore) {
    this.items = db.collection('items').valueChanges();
  }
}
```

Découverte de l'écosystème Angular

AngularFire

```
export const firebaseConfig = {
  apiKey: 'AIzaSyDsIf2X_MIBekqzygg8hFpaB-ZTSZ3n7q0',
  authDomain: 'portfolio-2efcf.firebaseio.com',
  databaseURL: 'https://portfolio-2efcf.firebaseio.com',
  projectId: 'portfolio-2efcf',
  storageBucket: 'portfolio-2efcf.appspot.com',
  messagingSenderId: '395738324852'
};

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    FormsModule,
    AngularFireModule.initializeApp(firebaseConfig.firebase),
    AngularFirestoreModule, // imports firebase/firestore, only needed for database features
    AngularFireAuthModule, // imports firebase/auth, only needed for auth features
    AngularFireDatabaseModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Angular matériel

Angular Material est un module d'Angular, c'est un module d'intégration qui permet d'obtenir facilement une interface responsive harmonieuse et unie dans le style 'flat' de Google.

La documentation de Material se trouve ici material.angular.io et est très bien faite

Cette petite introduction de Material porte sur la mise en page d'un login/register que l'on pourrait retrouver dans n'importe quelle application Angular en material design.

Angular matériel

Form Controls

Controls that collect and validate user input.

Navigation

Menus, sidenavs and toolbars that organise your content.

Layout

Essential building blocks for presenting your content.

Buttons & Indicators

Buttons, toggles, status and progress indicators.

Popups & Modals

Floating components that can be dynamically shown or hidden.

Data table

Tools for displaying and interacting with tabular data.

Quelle est la différence entre un AngularMaterial et Angular ?

Angular est un framework JavaScript/Type Script tandis que Angular material est un framework d'interface utilisateur qui implémente Google material design pour le Web.

Angular material ne peut être utilisé qu'avec angular, mais l'inverse n'est pas vrai.

Vous pouvez utiliser angular avec n'importe quel autre framework web comme **bootstrap**, **PrimeNG** etc.

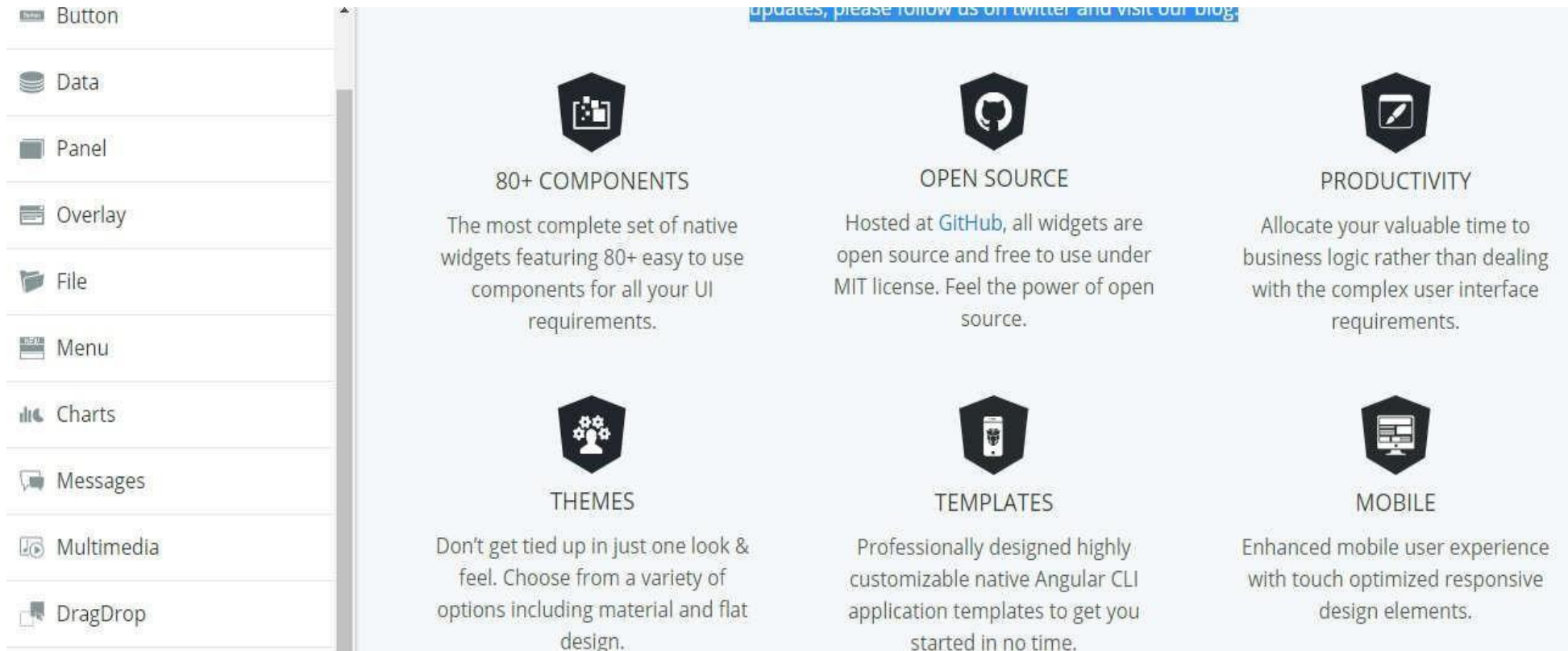
PrimeNG

PrimeNG est une collection de composants d'interface utilisateur riches pour Angular. Tous les widgets sont open source et gratuits sous licence MIT.

PrimeNG est développé par PrimeTek Informatics, un fournisseur avec des années d'expertise dans le développement de solutions d'interface utilisateur open source.



PrimeNG



The image displays the PrimeNG UI library features. On the left is a vertical sidebar with icons and labels for various UI components: Button, Data, Panel, Overlay, File, Menu, Charts, Messages, Multimedia, and DragDrop. The main content area is a light blue grid with six feature cards, each with a hexagonal icon, a title, and a description. The cards are: 1. 80+ COMPONENTS (icon: grid of squares) with text 'The most complete set of native widgets featuring 80+ easy to use components for all your UI requirements.' 2. OPEN SOURCE (icon: GitHub logo) with text 'Hosted at [GitHub](#), all widgets are open source and free to use under MIT license. Feel the power of open source.' 3. PRODUCTIVITY (icon: notepad and pencil) with text 'Allocate your valuable time to business logic rather than dealing with the complex user interface requirements.' 4. THEMES (icon: person with gears) with text 'Don't get tied up in just one look & feel. Choose from a variety of options including material and flat design.' 5. TEMPLATES (icon: smartphone) with text 'Professionally designed highly customizable native Angular CLI application templates to get you started in no time.' 6. MOBILE (icon: tablet) with text 'Enhanced mobile user experience with touch optimized responsive design elements.' A blue link at the top right of the main area reads 'updates, please follow us on twitter and visit our blog.'

Button

Data

Panel

Overlay

File

Menu

Charts

Messages

Multimedia

DragDrop

updates, please follow us on twitter and visit our blog.

80+ COMPONENTS
The most complete set of native widgets featuring 80+ easy to use components for all your UI requirements.

OPEN SOURCE
Hosted at [GitHub](#), all widgets are open source and free to use under MIT license. Feel the power of open source.

PRODUCTIVITY
Allocate your valuable time to business logic rather than dealing with the complex user interface requirements.

THEMES
Don't get tied up in just one look & feel. Choose from a variety of options including material and flat design.

TEMPLATES
Professionally designed highly customizable native Angular CLI application templates to get you started in no time.

MOBILE
Enhanced mobile user experience with touch optimized responsive design elements.

<https://www.primefaces.org/primeng/#/>

Les bibliothèques UI

PrimeNG

```
npm install primeng --save
npm install primeicons --save
```

Styles

The css dependencies are as follows, Prime Icons, theme of your choice and structural css of components.

```
node_modules/primeicons/primeicons.css
node_modules/primeng/resources/themes/saga-blue/theme.css
node_modules/primeng/resources/primeng.min.css
```

```
import { AppComponent } from './app.component';
import { ButtonModule } from 'primeng/button';
import { InputTextModule } from 'primeng/inputtext';

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModule,
    ButtonModule,
    InputTextModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```



Mise en œuvre NgRx et Angular Matériel

Formulaire dynamique : le FormBuilder



Les formulaires

- Les formulaires sont la première forme d'interaction avec l'utilisateur pour mettre à jour des données.
- Angular propose deux types de formulaire :

FormsModule
(Template-driven)

Logique de validation dans le template

ReactiveFormsModule
(Reactive)

Logique de validation dans le composant

Formulaire : Template-driven

Template-driven

```
<form (ngSubmit)="onSubmit()" #heroForm="ngForm"> => template reference variable (#) to NgForm directive !!!  
<label for="name">Name</label> => onSubmit() {save data in currentHero object 😊}  
<input type="text" class="form-control" id="name"  
  required  
  [(ngModel)]="currentHero.name" name="name"  
  #name="ngModel"> (exports NgModel into a template local variable called name)  
<div [hidden]="name.valid || name.pristine"  
  class="alert alert-danger">  
  Name is required  
</div>  
<button type="submit" class="btn btn-success" [disabled]="!heroForm.form.valid">Submit</button>
```

```
import { FormsModule } from '@angular/forms';  
  
import { AppComponent } from './app.component';  
import { HeroFormComponent } from './hero-form/hero-form.component';  
  
@NgModule({  
  imports: [  
    BrowserModule,  
    FormsModule  
  ],  
  declarations: [  
    AppComponent,  
    HeroFormComponent  
  ],  
  providers: [],  
  bootstrap: [ AppComponent ]  
})  
export class AppModule { }
```


Formulaire : Reactive

- Arbre d'éléments de formulaire créé directement dans le composant
- Eléments de l'arbre liés (bind) aux composants du formulaire contenu dans le template
- Formulaire manipulé entièrement par le composant, valeurs toujours synchronisées

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

import { AppComponent } from './app.component';
import { DynamicFormComponent } from './dynamic-form.component';
import { DynamicFormQuestionComponent } from './dynamic-form-question.component';

@NgModule({
  imports: [ BrowserModule, ReactiveFormsModule ],
  declarations: [ AppComponent, DynamicFormComponent, DynamicFormQuestionComponent ],
  bootstrap: [ AppComponent ]
})
export class AppModule {
  constructor() {
  }
}
```

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

import { QuestionBase } from './question-base';
import { QuestionControlService } from './question-control.service';

@Component({
  selector: 'app-dynamic-form',
  templateUrl: './dynamic-form.component.html',
  providers: [ QuestionControlService ]
})
export class DynamicFormComponent implements OnInit {

  @Input() questions: QuestionBase<any>[] = [];
  form: FormGroup;
  payload = '';

  constructor(private qcs: QuestionControlService) { }

  ngOnInit() {
    this.form = this.qcs.toFormGroup(this.questions);
  }
}
```

```
<div [formGroup]="form">
  <label [attr.for]="question.key">{{question.label}}</label>

  <div [ngSwitch]="question.controlType">

    <input *ngSwitchCase="'textbox'" [formControlName]="question.key"
      [id]="question.key" [type]="question.type">

    <select [id]="question.key" *ngSwitchCase="'dropdown'" [formControlName]="question.key">
      <option *ngFor="let opt of question.options" [value]="opt.key">{{opt.value}}</option>
    </select>

  </div>

  <div class="errorMessage" *ngIf="!isValid">{{question.label}} is required</div>
</div>
```


FormControl, FormGroup et FormArray

- 3 composants principaux à mémoriser :

FormControl

FormGroup

FormArray

FormControl, FormGroup et FormArray

First name *

FormControl

Value

Validation status

User interactions

Events

FormControl, FormGroup et FormArray

```
const control = new FormControl();  
  
control.value           // null  
  
control.status          // VALID  
  
control.valid           // true  
  
control.pristine        // true  
  
control.untouched       // true
```


FormControl, FormGroup et FormArray

```
const control = new FormControl();  
  
control.setValue('Nancy');  
  
control.value           // 'Nancy'  
  
control.reset();  
  
control.value           // null  
  
control.disable();  
  
control.status           // DISABLED
```



FormControl, FormGroup et FormArray

Street _____

City _____ Select state  Zip _____

FormGroup

FormControl name: street	FormControl name: city
FormControl name: state	FormControl name: zip

FormControl, FormGroup et FormArray

Street

City

Select state

 Zip

Street

City

Select state

 Zip

Street

City

Select state

 Zip

Street

City

Select state

 Zip

FormArray

FormControl

0

FormControl

1

FormControl

2

FormControl

3

FormControl, FormGroup et FormArray

```
const arr = new FormArray([
  new FormControl('SF'),
  new FormControl('NY')
]);

arr.value // ['SF', 'NY']

arr.status // VALID

arr.setValue(['LA', 'LDN']); // ['LA', 'LDN']

arr.push(new FormControl('MTV')); // ['LA', 'LDN', 'MTV']
```

Validation et gestion d'erreur personnalisée

➤ Les validateurs natifs

- Validators.required
- Validators.minLength
- Validators.maxLength
- Validators.pattern

```
ngOnInit(): void {  
  this.heroForm = new FormGroup({  
    'name': new FormControl(this.hero.name, [  
      Validators.required,  
      Validators.minLength(4),  
      forbiddenNameValidator(/bob/i)  
    ]),  
    'alterEgo': new FormControl(this.hero.alterEgo, {  
      asyncValidators: [this.alterEgoValidator.validate.bind(this.alterEgoValidator)],  
      updateOn: 'blur'  
    }),  
    'power': new FormControl(this.hero.power, Validators.required)  
  }, { validators: identityRevealedValidator }); // <-- add custom validator at the FormGroup level  
}
```


Validation et gestion d'erreur personnalisée

```
<div *ngIf="name.invalid && (name.dirty || name.touched)"
  class="alert alert-danger">
  <div *ngIf="name.errors.required">
    Name is required.
  </div>
  <div *ngIf="name.errors.minlength">
    Name must be at least 4 characters long.
  </div>
  <div *ngIf="name.errors.forbiddenName">
    Name cannot be Bob.
  </div>
</div>
```

Utilisation du FormBuilder

➤ FormBuilder

Facilite la création de formulaire. C'est un service :

Qu'il faut injecter dans le constructeur qui

permet la création de FormGroup

Le FormGroup permet de stocker les éléments du formulaire sous la forme d'un objet :

name => definition

```
import {FormBuilder, FormGroup, Validators} from '@angular/forms';
import {ProductsService} from '../services/products.service';

@Component({
  selector: 'app-product-add',
  templateUrl: './product-add.component.html',
  styleUrls: ['./product-add.component.css']
})
export class ProductAddComponent implements OnInit {

  productFormGroup?:FormGroup;
  submitted:boolean=false;

  constructor(private fb:FormBuilder, private productsService:ProductsService) { }

  ngOnInit(): void {
    this.productFormGroup=this.fb.group({
      name:["",Validators.required],
      price:[0,Validators.required],
      quantity:[0,Validators.required],
      selected:[true,Validators.required],
      available:[true,Validators.required],
    });
  }
}
```

Mise en œuvre des formulaires dynamiques ReactiveFormsModule

Tests unitaires. Bonnes pratiques et outils



Les tests dans Angular

Angular a été conçu et créé en utilisant une architecture modulaire et testable.

On dit aussi que Angular a été fait pour être testé.

Type des tests

Unit tests

Tester unitairement
des fonctions du code.



Jasmine



e2e test

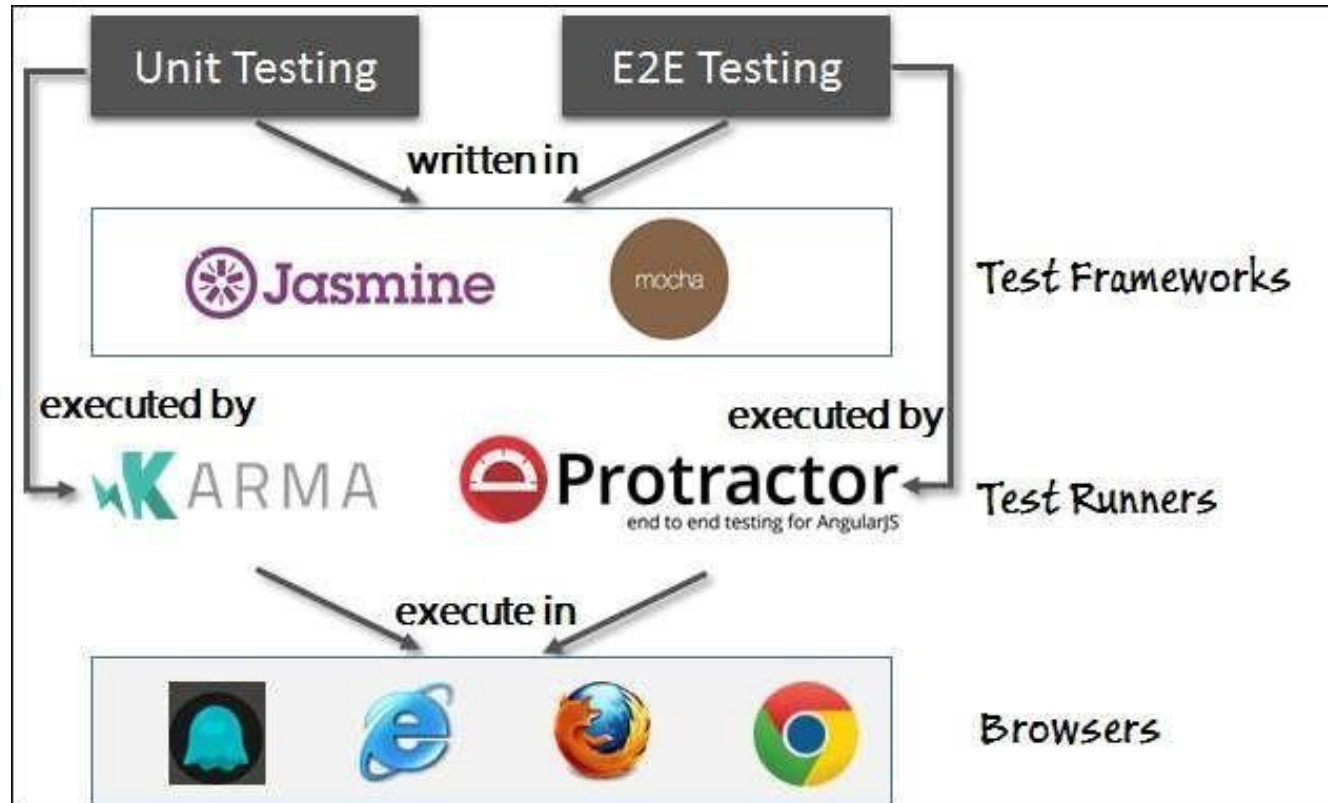
Tester le comportement
réel d'une application.

Simulation réel d'une
interaction utilisateur.



Protractor
end to end testing for AngularJS

Architecture et outils de tests



KARMA

Le but de [Karma](#) est de fournir un environnement d'exécution de vos tests Javascript. Les navigateurs Internet ne sont pas conçus pour charger nativement des fichiers de tests en javascript, les exécuter et afficher un rapport d'exécution.

Karma a été créé pour répondre à cette problématique. Voici les principales fonctionnalités de cette librairie :

Serveur web : il démarre un serveur web allégé pour lancer vos fichiers de tests. Ces fichiers peuvent être écrits à l'aide de différents framework (Jasmine, Qunit, Nunit, Mocha,...)

Runner : il fournit une page custom qui lancera les tests. Cette page est différente selon le framework de test utilisé.

Manager : il démarre un navigateur Internet (Client) pour charger cette page. Vous pouvez utiliser différents navigateurs comme Firefox, Chrome, IE, PhantomJS...

Reporter : il génère des rapports d'exécution soit sous forme de fichiers, dans la console d'exécution...

Watcher : Karma fournit des plugins pour analyser les changements sur le filesystem pour relancer automatiquement les tests...

Karma s'interface facilement avec vos serveurs d'intégration continue comme [Jenkins](#) ou [Travis](#).

KARMA

Installation :

```
npm install -g karma karma-jasmine karma-chrome-launcher
```

Configuration :

```
karma init
```

Lancement:

```
karma start
```



KARMA

karma et typescript :

<https://www.npmjs.com/package/karma-typescript>

```
npm init  
npm install --save-dev karma-typescript
```



KARMA

```
TERMINAL 1: node
68% building modules 869/887 modules 18 active ...ar/common/src/pipes/uppercase_pipe.j
68% building modules 870/887 modules 17 active ...ar/common/src/pipes/uppercase_pipe.j
68% building modules 871/887 modules 16 active ...ar/common/src/pipes/uppercase_pipe.j
68% building modules 872/887 modules 15 active ...ar/common/src/pipes/uppercase_pipe.j
69% building modules 873/887 modules 14 active ...ar/common/src/pipes/uppercase_pipe.j
69% building modules 874/887 modules 13 active ...ar/common/src/pipes/uppercase_pipe.j
69% building modules 875/887 modules 12 active ...ar/common/src/pipes/uppercase_pipe.j
69% building modules 876/887 modules 11 active ...ar/common/src/pipes/uppercase_pipe.j
69% building modules 877/887 modules 10 active ...ar/common/src/pipes/uppercase_pipe.j
12 01 2017 12:46:04.202:WARN [karma]: No captured browser, open http://localhost:9876/
12 01 2017 12:46:04.214:INFO [karma]: Karma v1.2.0 server started at http://localhost:9876/
12 01 2017 12:46:04.214:INFO [launcher]: Launching browser Chrome with unlimited concurrency
12 01 2017 12:46:04.224:INFO [launcher]: Starting browser Chrome
12 01 2017 12:46:05.077:INFO [Chrome 55.0.2883 (Mac OS X 10.12.2)]: Connected on socket /#fU7Q4c3qyRaadrnuAAAA with id 720766
Chrome 55.0.2883 (Mac OS X 10.12.2): Executed 6 of 6 SUCCESS (0.649 secs / 0.639 secs)
```



Jasmine : simple test

```
describe("A spec using beforeEach and afterEach", function() {  
  var foo = 0;  
  
  beforeEach(function() {  
    foo += 1;  
  });  
  
  afterEach(function() {  
    foo = 0;  
  });  
  
  it("is just a function, so it can contain any code", function() {  
    expect(foo).toEqual(1);  
  });  
  
  it("can have more than one expectation", function() {  
    expect(foo).toEqual(1);  
    expect(true).toEqual(true);  
  });  
});
```

Jasmine : Test d'un composant

```
import { async, ComponentFixture, TestBed } from '@angular/core/testing';
import { CreateProductComponent } from './create-product.component';

describe('CreateProductComponent', () => {
  let component: CreateProductComponent;
  let fixture: ComponentFixture<CreateProductComponent>;

  beforeEach(async(() => {
    TestBed.configureTestingModule({
      declarations: [ CreateProductComponent ],
    })
    .compileComponents();
  }));

  beforeEach(() => {
    fixture = TestBed.createComponent(CreateProductComponent);
    component = fixture.componentInstance;
    fixture.detectChanges();
  });

  it('should create', () => {
    expect(component).toBeTruthy();
  });
});
```



Jasmine : TestBed API

```
Testbed.configureTestingModule({  
  imports: [HttpModule],  
  declarations: [JokeComponent],  
  providers: [JokeService]  
});
```



Jasmine : TestBed API

```
fixture = TestBed.createComponent(JokeComponent);  
  
component = fixture.componentInstance;  
  
debugElement = fixture.debugElement;  
  
fixture.detectChanges();
```



Jasmine : TestBed API

```
jokeText = debugElement.query(By.css(`p`)).nativeElement;
```

```
it('should find the <p> with fixture.debugElement.query(By.css)', () => {  
  const bannerDe: DebugElement = fixture.debugElement;  
  const paragraphDe = bannerDe.query(By.css('p'));  
  const p: HTMLElement = paragraphDe.nativeElement;  
  expect(p.textContent).toEqual('banner works!');  
});
```

```
@Component({  
  selector: 'app-banner',  
  template: `<p>banner works!</p>`,  
  styles: []  
})  
export class BannerComponent { }
```

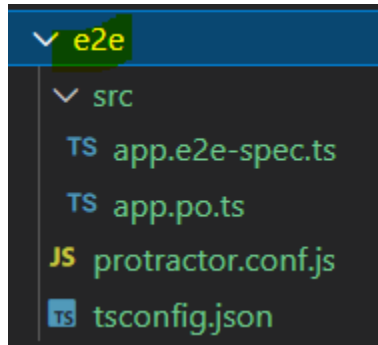


Jasmine : TestBed API

```
describe(`Component: JokeComponent`, () => {  
  let component: JokeComponent;  
  let jokeService: JokeService;  
  let fixture: ComponentFixture<JokeComponent>;  
  let de: DebugElement;  
  
  beforeEach(() => {  
    TestBed.configureTestingModule({  
      imports: [HttpModule],  
      declarations: [JokeComponent],  
      providers: [JokeService],  
    });  
  
    fixture = TestBed.createComponent(JokeComponent);  
    component = fixture.componentInstance;  
    jokeService = TestBed.get(JokeService);  
    de = Fixture.debugElement;  
  });  
});
```

Tests d'intégration avec Protractor

Les tests E2E



```
import { browser, by, element } from 'protractor';

export class AppPage {
  navigateTo(): Promise<unknown> {
    return browser.get(browser.baseUrl) as Promise<unknown>;
  }

  getTitleText(): Promise<string> {
    return element(by.css('app-root .content span')).getText() as Promise<string>;
  }
}
```

```
import { AppPage } from './app.po';
import { browser, logging } from 'protractor';

describe('workspace-project App', () => {
  let page: AppPage;

  beforeEach(() => {
    page = new AppPage();
  });

  it('should display welcome message', () => {
    page.navigateTo();
    expect(page.getTitleText()).toEqual('shop4less app is running!');
  });

  afterEach(async () => {
    // Assert that there are no errors emitted from the browser
    const logs = await browser.manage().logs().get(logging.Type.BROWSER);
    expect(logs).not.toContain(jasmine.objectContaining({
      level: logging.Level.SEVERE,
    } as logging.Entry));
  });
});
```

Le Code-Coverage

"Code Coverage"

L'option `--code-coverage` permet de produire un rapport indiquant quelles parties de code sont couvertes ou non par les tests.

Les rapports sont produits dans le dossier `coverage` et contiennent des fichiers HTML "Human Readable" mais aussi d'autres formats tels que "Icov" pour une intégration plus facile avec les outils d'intégration continue.

 Cette option n'est pas activé par défaut pour éviter de ralentir les tests lancés avec l'option `--watch`.

 Le "code coverage" n'est indiqué que pour les fichiers qui sont importés par les tests unitaires.

Le code source qui n'est jamais importé par les tests unitaires n'est donc pas comptabilisé.

Documentation

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

Mise en œuvre des tests unitaires