Terraform

https://learn.microsoft.com/en-us/azure/devops/pipelines/process/phases?view=azure-devops&tabs=yaml

https://dev.azure.com/aldfrance/ALDFAPI/\_git/Aldfbp.TnC.Templates?path=/templates&\_a=contents&version=GBmaster

IdentityServer

SendGrid

https://aldfrance.visualstudio.com/ALDFAPI/\_wiki/wikis/ALDFAPI.wiki/1020/Introduction

**Terraform**

<https://blog.stephane-robert.info/docs/infra-as-code/provisionnement/terraform/introduction/>

<https://aldfrance.visualstudio.com/ALDFAPI/_wiki/wikis/ALDFAPI.wiki/1022/Terraform>

is an open-source infrastructure as code (IaC) tool developed by HashiCorp

-provider.tf : defines the used providers azurerm and azuread

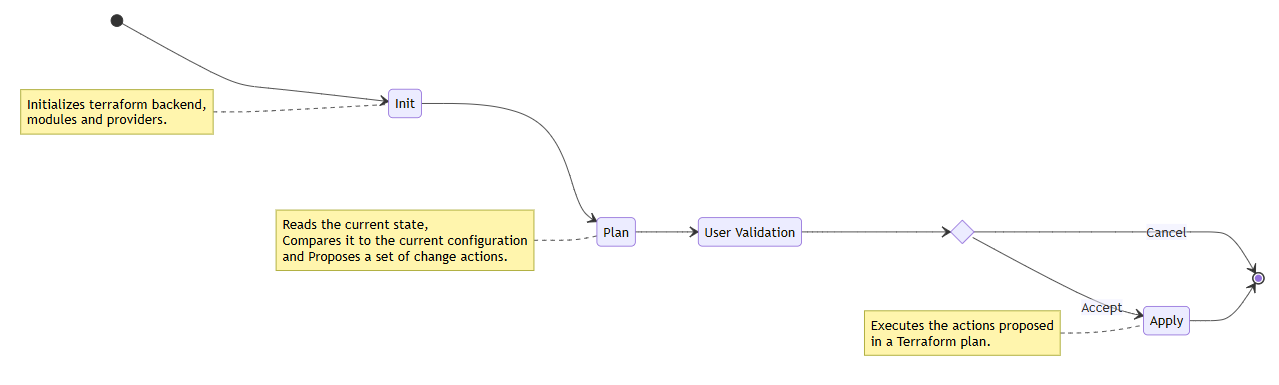
-variables.tf : defines the variables that will used in terraform script

-variables-int.tfvars, variables-stg.tfvars and variables-prd.tfvars : defines variables values for each environment

-main.tf : this is the main terraform script that will describe the infrastructure for provisioning

-modules : contains the definition of the modules that will be consumed in main.tf

The main terraform script is excecuted by a YAML pipeline (azure-pipeline-tf-resources.yml) following these steps :



**Pipelines**

<https://aldfrance.visualstudio.com/ALDFAPI/_wiki/wikis/ALDFAPI.wiki/1024/Azure-Pipelines>

variables.yml defines the default and shared variables that will be consumed by other YAML pipelines such as application name, pools names, Terraform,

.NET SDK, Entity Framework, Docker, AKS, MSSQL, APIM and NSwag variables.

These values will be overrided in variables-int.yml, variables-stg.yml and variables-prd.yml to adapt to each environment.

Terraform pipeline

azure-pipeline-tf-resources.yml executes the Terraform script over all environments to provision needed Azure resources on Azure subscription defined in

the environment according variables file.

AKS pipelines

These pipelines build, test and deploy code on Kubernetes. It may generate and execute SQL migration script if the API was generated with --ef arguments.

List of AKS pipelines:

azure-pipeline-int.yml

azure-pipeline-stg.yml

azure-pipeline-prd.yml

APIM pipelines

These pipelines publish the APIM package to the T&C APIM instance. The APIM package located in .apim-package solution folder contains an OpenAPI specification in swagger.json file. This specification defines API to be published.

The APIM pipelines use the aiInstrumentationKey defined in the variables YAML file to enable APIM monitoring through Azure Application Insights service.

List of APIM pipelines:

azure-pipeline-apim-dev.yml

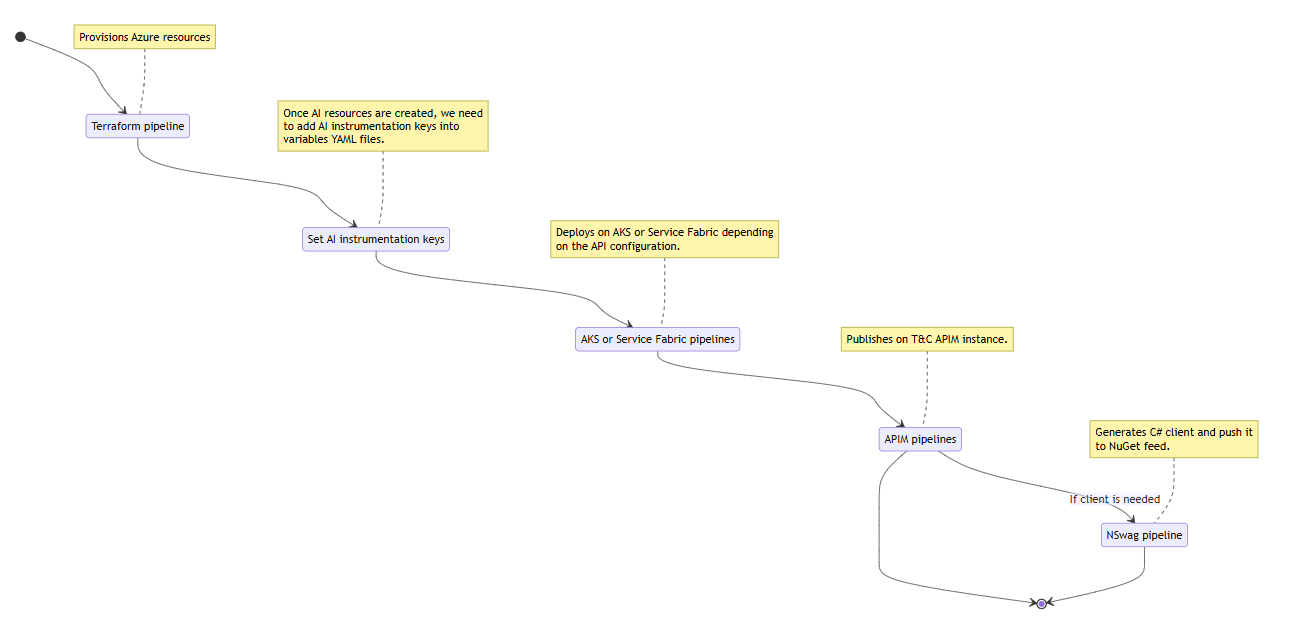
azure-pipeline-apim-master.yml

NSwag pipeline (https://github.com/RicoSuter/NSwag)

NSwag is a Swagger/OpenAPI 2.0 and 3.0 toolchain for .NET, .NET Core, Web API, ASP.NET Core, TypeScript (jQuery, AngularJS, Angular 2+, Aurelia, KnockoutJS and more) and other platforms, written in C#. The OpenAPI/Swagger specification uses JSON and JSON Schema to describe a RESTful web API.

The azure-pipeline-nswag-api-client.yml pipeline generates a C# clients from an OpenAPI specification based on swagger.json in .apim-package and publishes it on the NuGet feed aldfr.packages.

Pipelines execution workflow:



**APIM**

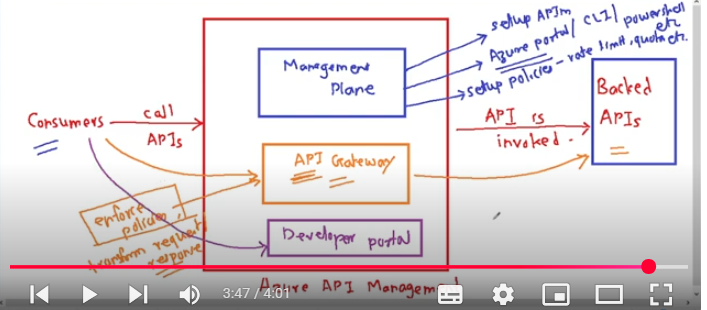
<https://aldfrance.visualstudio.com/ALDFAPI/_wiki/wikis/ALDFAPI.wiki/1026/APIM>

When generating a new API project using Aldfbp Api Template, the solution comes with the folder .apim-package. This folder contains the the API product, policies, Swagger file and the content file that defines the api, its variables and product.

Using the APIM pipelines, APIM package will be published on all environments based on the configuration specified in the pipeline and variables files and the OpenAPI specification of the API in the APIM package.

Azure API Management is a fully managed service that helps organizations publish, secure, monitor, and scale APIs across cloud and hybrid environments.

<https://www.youtube.com/watch?v=oQYhiN9bj3Y&list=PLU1w_BFZFd2oSHrHLdy0uhmqsYsUvgvby&index=8>



**DOCKER**

Docker est une technologie qui permet de créer et exécuter des **conteneurs**.  
Un **conteneur** = une boîte isolée qui contient une application + tout ce qu’il lui faut (librairies, dépendances, configuration), mais **sans embarquer un système d’exploitation complet** → contrairement à une machine virtuelle.

## 1. Les composants principaux

1. **Image Docker**
   * Une image = une **recette figée** de ton application (par ex. un serveur Node.js avec ton code).
   * Construite à partir d’un fichier Dockerfile.
   * Ex : nginx:latest, mysql:8.0.
2. **Conteneur Docker**
   * Instance **vivante** d’une image.
   * Un conteneur peut être démarré, stoppé, supprimé, recréé.
   * Plusieurs conteneurs peuvent tourner à partir de la même image.
3. **Docker Engine**
   * Le moteur qui exécute les conteneurs.
   * S’appuie sur des fonctionnalités du noyau Linux (namespaces, cgroups) pour isoler les processus.
4. **Docker Hub (ou registre)**
   * Une sorte de “GitHub pour les images Docker”.
   * Tu peux télécharger (docker pull) ou publier (docker push) des images.

## 2. Cycle typique

### a) Écrire un Dockerfile

Exemple simple pour une API Node.js :

# Utiliser une image de base

FROM node:18

# Créer un dossier dans le conteneur

WORKDIR /app

# Copier le code

COPY package\*.json ./

RUN npm install

COPY . .

# Démarrer l'application

CMD ["npm", "start"]

### b) Construire l’image

docker build -t mon-api .

### c) Lancer un conteneur

docker run -d -p 3000:3000 mon-api

👉 Cela lance ton API dans un conteneur et l’expose sur localhost:3000.

## 4. En résumé

* **Docker Images** = plans de construction (recettes).
* **Docker Containers** = instances vivantes de ces images.
* **Docker Engine** = le moteur qui fait tourner le tout.
* Docker rend tes applis **portables** : “ça marche chez moi” devient “ça marche partout”.

**Kubernetes (K8s)**

Kubernetes K8s agit comme un chef d’orchestre : il prend tes conteneurs et s’assure qu’ils tournent correctement, qu’ils sont répartis sur plusieurs machines, qu’ils redémarrent en cas de crash, et qu’ils peuvent monter en charge si nécessaire

Les concepts clés K8s

1-Cluster:

Ensemble de machines (physiques ou virtuelles) gérées par Kubernetes.

Deux rôles principaux :

Master (Control Plane) → le cerveau, qui décide.

Nodes (Workers) → les machines qui exécutent les conteneurs.

2-Pod:

L’unité de base de Kubernetes.

Un Pod = 1 ou plusieurs conteneurs qui partagent le même réseau et le même stockage.

Exemple : une API + un conteneur de "sidecar" pour les logs dans le même Pod.

3-Deployment:

Décrit comment déployer et gérer un ensemble de Pods.

Permet le scaling (augmenter ou réduire le nombre de Pods) et les mises à jour sans interruption (rolling updates).

4-Service:

Permet d’exposer un ensemble de Pods via une IP fixe ou un nom DNS.

Comme les Pods sont éphémères (ils peuvent mourir et renaître ailleurs), le Service agit comme un "load balancer" interne.

5-ConfigMap & Secret:

Pour injecter des configurations ou des informations sensibles (mots de passe, clés API).

6-Ingress:

Gère l’accès externe au cluster (reverse proxy, SSL, etc.).

7-Scheduler:

Décide sur quel nœud lancer un Pod, selon les ressources dispo (CPU, RAM, etc.).

Cycle typique:

Tu décris ton application dans un fichier YAML (ex. un Deployment).

Tu l’envoies à Kubernetes avec kubectl apply -f mon-deployment.yaml.

Le control plane enregistre l’état désiré (ex : "je veux 3 Pods de mon app").

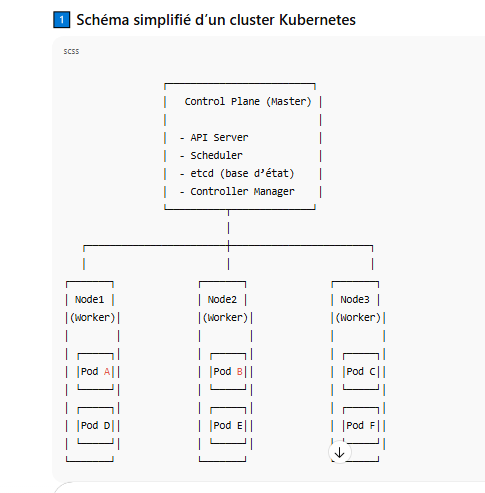
Le scheduler décide sur quels nœuds les lancer.

Les kubelets (agents sur chaque nœud) lancent les conteneurs via Docker/Containerd.

Kubernetes vérifie en permanence que l’état réel = état désiré.

Si un Pod meurt → il en recrée un.

Si la charge monte → il scale automatiquement.



## Exemple concret d’un déploiement Kubernetes (YAML)

Voici un fichier YAML qui déploie une petite API dans Kubernetes :

apiVersion: apps/v1

kind: Deployment

metadata:

name: mon-api

spec:

replicas: 3 # Nombre de Pods voulus

selector:

matchLabels:

app: mon-api

template:

metadata:

labels:

app: mon-api

spec:

containers:

- name: mon-api-container

image: monimage/api:1.0

ports:

- containerPort: 80

---

apiVersion: v1

kind: Service

metadata:

name: mon-api-service

spec:

selector:

app: mon-api

ports:

- port: 80

targetPort: 80

type: LoadBalancer # Expose le service vers l’extérieur

**Ce qui se passe :**

* **Deployment** : Kubernetes lance 3 Pods de ton API.
* **Service** : Fournit une IP/port stable pour accéder aux Pods (et load balance entre eux).
* Si un Pod tombe → Kubernetes en recrée un automatiquement.
* Si la charge augmente → tu peux changer replicas: 3 en 5 ou activer l’auto-scaling.

Kubernetes , also known as K8s, is an open-source system for automating deployment, scaling, and management of containerized applications.

A Kubernetes manifest file is your personal guide through a Kubernetes cluster: A configuration file written in a format called YAML or JSON, that describes the resources you want in your cluster. These resources can be a myriad of things: pods (that run your applications), services (that help your applications communicate), and deployments (that manage your applications).

Deployment

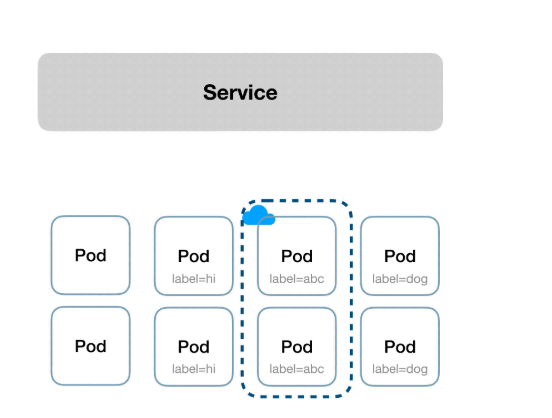
A Kubernetes Deployment YAML specifies the configuration for a Deployment object—this is a Kubernetes object that can create and update a set of identical pods.

The Deployment object not only creates the pods but also ensures the correct number of pods is always running in the cluster, handles scalability, and takes care of updates to the pods on an ongoing basis. All these activities can be configured through fields in the Deployment YAML.

Service

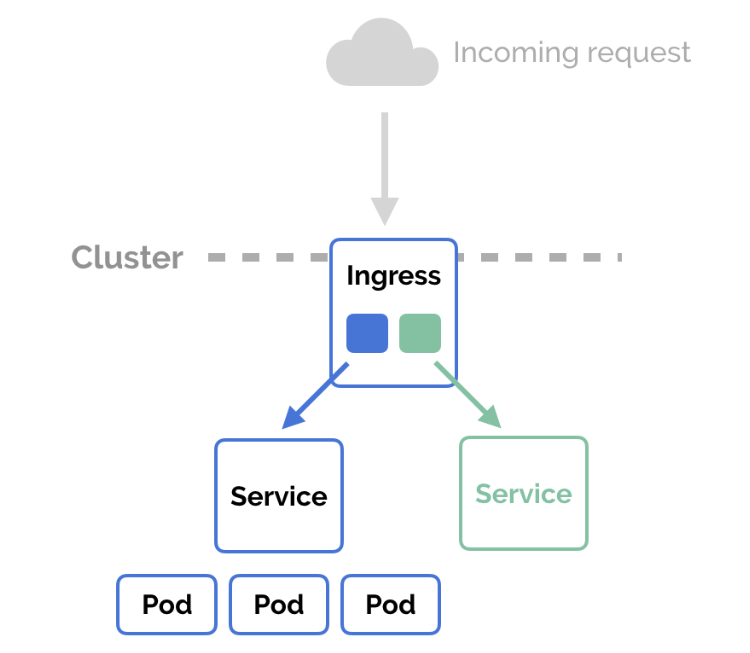
A Service enables network access to a set of Pods in Kubernetes.

Services select Pods based on their labels. When a network request is made to the service, it selects all Pods in the cluster matching the service's selector, chooses one of them, and forwards the network request to it.



**Ingress**

In Kubernetes, an Ingress is an object that allows access to your Kubernetes services from outside the Kubernetes cluster. You configure access by creating a collection of rules that define which inbound connections reach which services.



**SonarQube**

**AutoRest** is a powerful tool that facilitates the generation of client libraries for interacting with RESTful web services. It requires a specification that outlines the REST API using the OpenAPI format, enabling the streamlined production of client code

**NSwag** serves as a robust toolchain for working with Swagger/OpenAPI versions 2.0 and 3.0, designed for environments like .NET, .NET Core, ASP.NET Core, and TypeScript, and it is developed in C#. This tool empowers developers to create OpenAPI specifications directly from their existing API controllers and subsequently generate client code based on those specifications.

**Azure Service Bus**

[Azure Service Bus using Aldfbp.Boilerplate - Overview](https://aldfrance.visualstudio.com/ALDFAPI/_wiki/wikis/ALDFAPI.wiki/1187/Azure-Service-Bus-using-Aldfbp.Boilerplate)

Azure Service Bus is a fully-managed message broker with message queues and topics in a namespace. It is used to decouple applications and services from each other, providing the following benefits:

* Load-balancing work across competing workers
* Safely routing and transferring data and control across service and application boundaries
* Coordinating transactional work that requires a high-degree of reliability

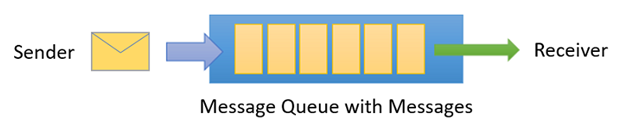
Data is transferred between different applications and services using messages. A message is a container decorated with metadata, and contains data. The data can be any kind of information, including structured data encoded with the common formats such as the following ones: JSON, XML, Apache Avro, Plain Text.

## Namespaces

A namespace is a container for all messaging components (queues and topics). Multiple queues and topics can be in a single namespace, and namespaces often serve as application containers.

## Queues

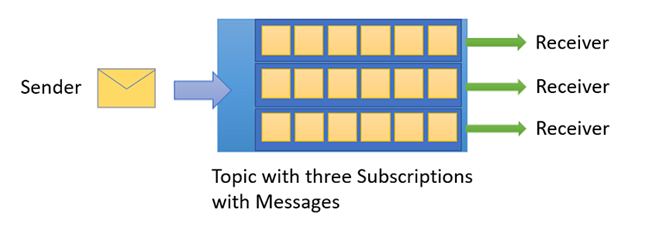
Messages are sent to and received from queues. Queues store messages until the receiving application is available to receive and process them.

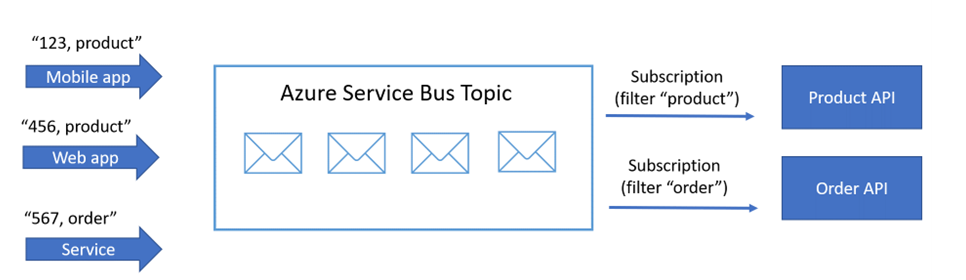


Messages in queues are ordered and timestamped on arrival.

## Topics

Topics can have multiple, independent subscriptions, which attach to the topic and otherwise work exactly like queues from the receiver side. A subscriber to a topic can receive a copy of each message sent to that topic.





**Queues and Topics:** Azure Service Bus offers both queues and topics as communication channels. Queues enable point-to-point communication, ensuring that each message is processed by only one receiver, making it ideal for workload distribution and load-leveling scenarios. On the other hand, topics support publish/subscribe messaging patterns, allowing multiple subscribers to receive relevant messages independently, and facilitating scalable event-driven architectures.

**Angular**

## 1. Structure d’une application Angular

Une app Angular est composée de plusieurs **briques** principales :

* **Modules** → regroupent des fonctionnalités (ex: AppModule).
* **Components (Composants)** → blocs de base qui affichent une partie de l’UI.
* **Templates** → HTML enrichi avec des expressions Angular.
* **Services** → logique métier, accès aux données (ex: appels HTTP).
* **Directives** → instructions dans le HTML pour manipuler le DOM.
* **Data Binding** → mécanisme pour lier les données du TS au HTML.
* **Routing** → navigation entre les pages de l’app.

## 2. Les Composants

👉 C’est **l’unité de base** d’Angular.  
Un composant = **TypeScript + Template HTML + CSS**.

Exemple :

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

@Component({

selector: 'app-hello', // balise HTML pour appeler le composant

template: `<h1>{{ title }}</h1>`,

styles: [`h1 { color: blue; }`]

})

export class HelloComponent {

title = 'Bienvenue dans Angular !';

}

👉 Utilisation dans un HTML :

<app-hello></app-hello>

## 3. Data Binding

C’est le lien entre **la logique (TS)** et **l’interface (HTML)**.  
Il y a 4 formes principales :

1. **Interpolation (affichage de données)**
2. <h1>{{ titre }}</h1>
3. **Property Binding (lier une valeur à un attribut HTML)**
4. <img [src]="imageUrl">
5. **Event Binding (écouter des événements)**
6. <button (click)="direBonjour()">Clique ici</button>
7. **Two-way Binding (liaison bidirectionnelle)**
8. <input [(ngModel)]="nom">

<p>Bonjour {{ nom }}</p>

## 4. Directives

Ce sont des instructions spéciales dans le HTML.

* **Structurelles** (modifient le DOM) :
  + \*ngIf → afficher/masquer
  + \*ngFor → boucle
* <p \*ngIf="isLoggedIn">Bienvenue !</p>
* <li \*ngFor="let user of users">{{ user.name }}</li>
* **Attribuées** (modifient l’apparence) :
  + [ngClass], [ngStyle], etc.

## 5. Services & Injection de dépendances

👉 La logique métier et les appels API se mettent dans des **services**.  
Ils sont injectés dans les composants avec l’**injection de dépendances**.

Exemple :

@Injectable({ providedIn: 'root' })

export class UserService {

getUsers() {

return ['Alice', 'Bob', 'Charlie'];

}

}

Utilisation dans un composant :

constructor(private userService: UserService) {}

## 6. Routing

👉 Permet de créer une SPA avec plusieurs pages.  
Définition des routes :

const routes: Routes = [

{ path: 'home', component: HomeComponent },

{ path: 'users', component: UserListComponent }

];

Utilisation dans un template :

<a routerLink="/home">Accueil</a>

<a routerLink="/users">Utilisateurs</a>

<router-outlet></router-outlet> <!-- Ici s’affiche la page -->

## En résumé

* **Modules** = regroupements.
* **Composants** = briques UI.
* **Data Binding** = relie données ↔ UI.
* **Directives** = manipulent le DOM.
* **Services** = logique + données (API).
* **Routing** = navigation entre pages.

# 🟢 NGRX dans Angular

## 1. Pourquoi NGRX ?

Dans une application Angular **simple**, on peut stocker l’état (les données) directement dans les composants ou services.  
👉 Mais dès que l’app devient **grande et complexe**, ça devient difficile de :

* Partager l’état entre plusieurs composants.
* Suivre les changements (qui modifie quoi ?).
* Déboguer et tester.

👉 NGRX résout ça en appliquant le **pattern Redux** dans Angular.  
C’est-à-dire :

* Un **store unique** qui contient l’état global de l’application.
* Des **actions** pour décrire les changements.
* Des **reducers** pour appliquer ces changements.
* Des **effects** pour gérer les appels asynchrones (ex: API).

## 2. Les concepts de base NGRX

### 🏪 Store

* Le **store** est une **source de vérité unique**.
* C’est un gros objet JavaScript qui contient l’état global.
* Les composants peuvent **lire** depuis le store et **dispatcher** des actions pour le modifier.

### 🎬 Actions

* Une **action** = un objet qui décrit ce qu’il s’est passé.
* Exemple :

import { createAction, props } from '@ngrx/store';

export const ajouterTache = createAction(

'[Todo] Ajouter',

props<{ titre: string }>()

);

**Reducers**

* Un **reducer** = une fonction pure qui prend :
  + l’état actuel
  + une action
  + et retourne **le nouvel état**
* Exemple :

import { createReducer, on } from '@ngrx/store';

import { ajouterTache } from './todo.actions';

export interface TodoState {

taches: string[];

}

export const initialState: TodoState = { taches: [] };

export const todoReducer = createReducer(

initialState,

on(ajouterTache, (state, { titre }) => ({

...state,

taches: [...state.taches, titre]

}))

);

**Selectors**

* Les **selectors** servent à lire des morceaux du store (plutôt que de lire tout l’état).

import { createSelector } from '@ngrx/store';

export const selectTaches = (state: { todo: TodoState }) => state.todo.taches;

**Effects**

* Les **effects** gèrent les **tâches asynchrones** (ex: appels HTTP).
* Exemple :

@Injectable()

export class TodoEffects {

chargerTaches$ = createEffect(() =>

this.actions$.pipe(

ofType('[Todo API] Charger'),

mergeMap(() =>

this.todoService.getTaches().pipe(

map(taches => ({ type: '[Todo API] Charger Success', taches }))

)

)

)

);

constructor(

private actions$: Actions,

private todoService: TodoService

) {}

}

## 3. Cycle de fonctionnement NGRX

1. Un **composant** déclenche une **action** (store.dispatch).
2. L’**action** passe dans les **reducers** (mise à jour de l’état synchronisé).
3. Les **effects** écoutent certaines actions et lancent des opérations asynchrones (API, etc.).
4. Quand c’est fini, un **nouvel événement (action)** est envoyé.
5. Les composants s’abonnent au store via des **selectors** pour afficher les données.