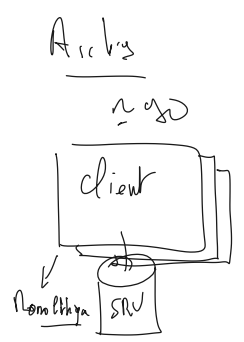
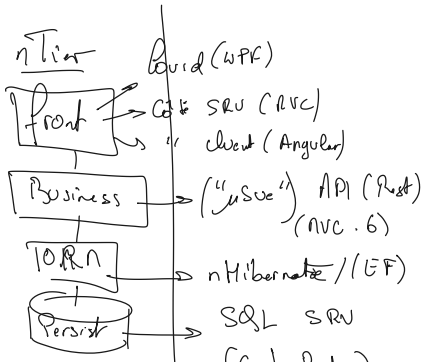
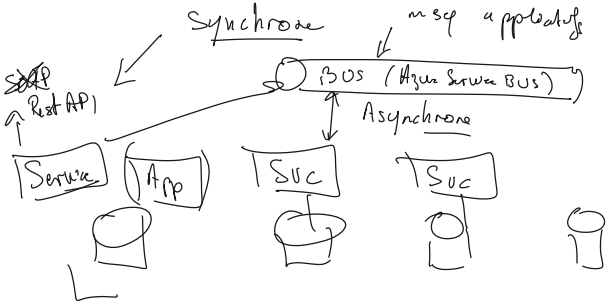
Bonjour tout le Monde





- Service Oriented Architecture - 2000 · 2010

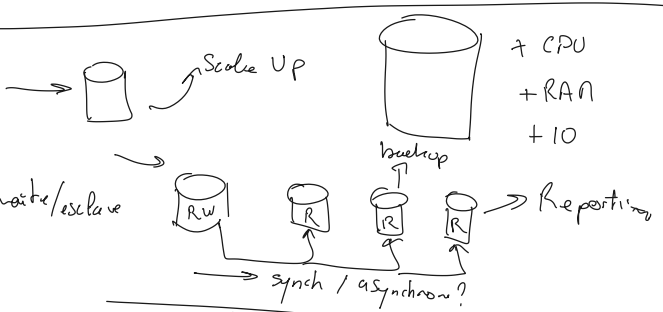


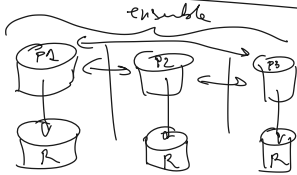
> 2010 Architectures en µSvc

- chaque µSvc est un projet
- Une Route vers 1 API
- Découplé du reste

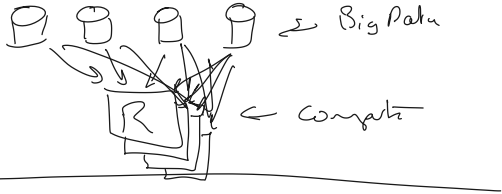
- Aucun point Central
  - Pas de BDD
  - Pas de relation / intégrité référentielle
  - Maintenant Disponible

- 1 front pour le piloter Tous

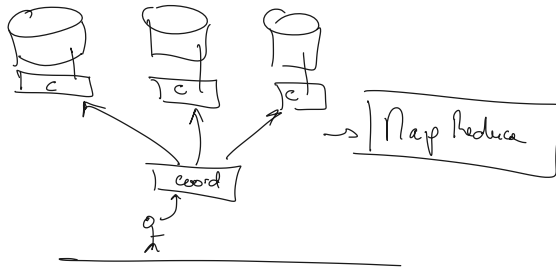




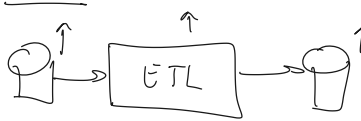
cluster Big Data



Big Compute (Hadoop)



Batch



a → z

Abatch.

A → Z

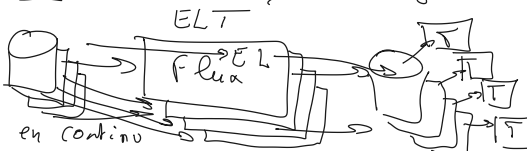
(3 jours Max)

Flux.

Apache

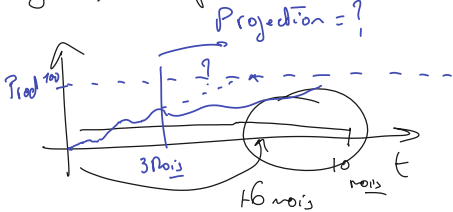
Nifi

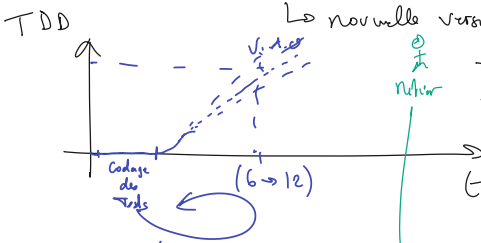
ELT



en continu

Agile, Waterfall





→ Tests
→ Code
→ Correction des ~~Tests~~ Code

Sur Bug → Reproduit le bug en Test
→ Corrige.
→ Évite les régressions

Cas d'tests / A Coder

→ Cas Normaux
→ essayer d'être exhaustif sur le cas de Test
→ Cas alternatifs → champ vide
→ tentative d'exploit (CSRF)
→ Cas d'erreurs → Plus de mémoire
→ erreur réseau
→ coupure en plein milieu ....

La nouvelle version

Refactor

→ Specs
→ Analyse → UML
→ Ecrire les Tests
↓
Squelette du code
" des Tests
→ Coder les Tests
→ "tdd" → Intégré d'un FW ex: NSTest JUnit PhpUnit

→ BDD Behaviour D. D.
→ Comportements
→ "déclaratif" lisible.

→ Implémenter
→ Tester
→ Corriger

Implémenter les Autres Types de Tests (Stress, Charge, UX, Intégrale, ...)

Types de tests :

• Accessibility testing
• Acceptance testing
• Black box testing : tester un système inconnu.
• End to end testing : tester un workflow fonctionel
• Functional testing : s'assurer qu'il fait exactement ce qui était prévu.
• Interactive testing : eq tests manuels
• Interface tests
• Integration testing : test en environnement équivalent à la prod dans son ensemble.
• Load testing
• Non functional testing : catégorie de tests d'interface (conformité), accessibilité, performance, ...
• Performance testing : recherche des meilleurs métriques, benchmark.
• Regression testing
• Sanity testing : contrôle que les bugs ciblés sont bien corrigés
• Security testing : contrôle face aux menaces
• Single user performance testing
• Smoke testing : valider les fonctions critiques d'un système, envers la stabilité.
• Stress testing : test selon des conditions exceptionnelles de charge, au dela des specs.
• Unit testing
• White-box testing : teste les entrées sorties d'un logiciel bien connu, concernant le design, l'utilisabilité, la sécurité, la stabilité.

3 Ways to Test
There are 3 ways you can do testing.
• Manual testing is the most hands-on type of testing and is employed by every team at some point. Of course, in today's fast-paced software development lifecycle, manual testing is tough to scale.
• Automated testing uses test scripts and specialized tools to automate the process of software testing.
• Continuous testing goes even further, applying the principles of automated testing in a scaled, continuous manner to achieve the most reliable test coverage for an enterprise. Keep reading to learn more about the differences between automated testing vs. manual testing and how continuous testing fits in.
• BDD
 • Frameworks
  ○ Cucumber (most popular)
  ○ Concordion.Net
   § Small open-source C#
   § Extends nunit
  ○ Concordion
   § Java based
  ○ BDDfy
   § Simple to use, compatible with test runners
   § . Net
  ○ Specflow
   § Part cucumber familly for. Net, Gherkin parser, most used in. Net
  ○ Gherkin : est le langage (DSL) : given, when, then
  ○ Quantum
   § Java based
  ○ Jbehave
   § Java based
  ○ Codeception
   § Php based
  ○ Zephyr scale
   § Free
   § Ruby, Java, Javascript, and C# (SpecFlow).
  ○ jDave : on top of JUnit
  ○ TestLeft
   § .NET, C#, Java, Jenkins, and more

- **• Unit testing**
  - La solution technique d'implémentation des tests. (MSTest, nUnit, ...)
  - Le socle des test en général.
- **• Interface tests**
  - S'appuie sur les Tests unitaires
  - Ou utilise des solutions externes de pilotage d'interface.
- MVC :
  - Modele : Testable via du test de l'ORM
  - Vue : non codable (test d'interface) -> solution ?
  - Controleur : Directement du code en test unitaire.
- • Load testing
  - Outils manuels (ex : sous visual studio) avec des graphiques etc...
  - Codable sous des tests unitaires.
- Test d'intégration :
  - teste un ensemble (plusieurs composants) a plus haut niveau
  - Codable avec les tests unitaires
  - Teste une fonction de la solution dans son ensemble (Front+back+db)
- Test fonctionnel :
  - teste un ensemble (plusieurs composants) a plus haut niveau
  - Codable avec les tests unitaires
  - Teste une fonction en particulier (ex : un formulaire de login)
- Performance testing - charge :
  - Temps de réponse dans divers scénarios (1 , 10, 1000 users)
    - Sommes nous conforme ? (performance) -> résultat : oui/non
  - Tel et tel report.
  - Charge :
    - Temps de réponse en fonction de la montée en charge (a partir de quand avons nous une limite ?) -> résulat : bench (courbe)
  - Single user : acceptabilité de l'app en temps de réponse dans son ensemble.
- Tests de sécurité :
  - Directement codé dans les tests unitaires
  - Implémentable en plus via des solutions externes (pentest) - OWASP Zap
- Tests de régression :
  - Complètement des tests unitaires (cas particulier)
  - Sanity test (santé, propreté) : que certaines régressions soient ok (pas toutes)
- Stress test :
  - Génère une fort montée en charge, et on évalue l'expérience utilisateur.
  - Etude de cas dans les pires scénarios de montée en charge.
- Smoke test :
  - Essayer de faire planter les systèmes sur longue saturation de charge.

- Accessibility testing
  - https://www.numerique.gouv.fr/publications/rgaa-accessibilite/documentation-rgaa/methodologie-test/
  - Plutôt de la recette : mal voyants, daltoniens, épileptiques, ...
- Acceptance testing
  - Au délà des specs, quel est le sentiment des utilisateurs finaux ?
- Black box testing : tester un système inconnu.
  - White-box testing : teste les entrées sorties d'un logiciel bien connu, concernant le design, l'utilisabilité, la sécurité, la stabilité.
  - Surcouche aux tests
- End to end testing : tester un workflow fonctionel
  - Codable (test d'interface) ou recettable

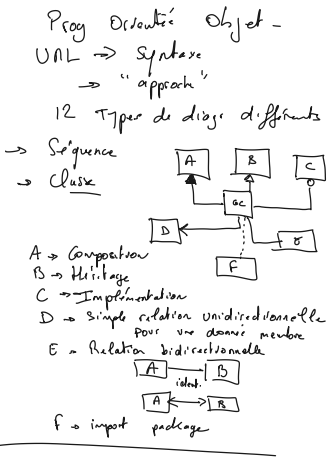
- Interactive testing : eq tests manuels
  - Recettage

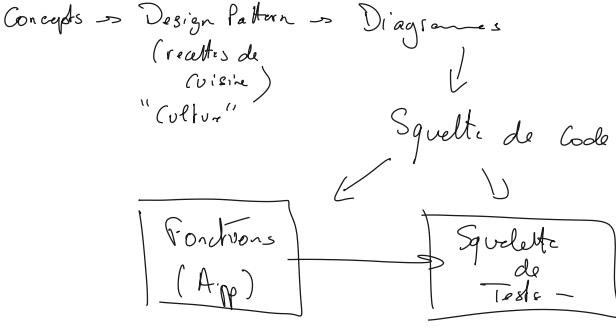
- Non functional testing : catégorie de tests d'interface (conformité), accessibilité, performance, ...
  - catégorie de tests de plus haut niveau (ensemble de tests)

Outils de test :

- Outils de tests unitaires (MSTest, Jasmine, ...)
  - Tests unitaires
  - Suites de tests
    - Ensemble de tests unitaires (déclaré)
- Outils interactifs de test (charge, recette, end to end, ...)
- Outils externes
  - Ex : Postman pour être plus efficace dans l'élaboration des tests d'API
    - Newman pour l'intégrer dans une CI
  - Scripts de pilotage d'interface (ex : WPf Interface obsolète)
-

UML
Patterns
Tests unitaires



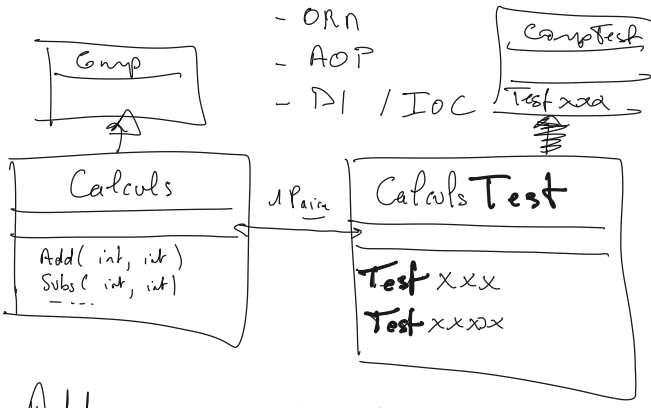


Design Pattern

book : Design Pattern

GoF "Gang of 4" ≃ 23 patterns

- architectures → - MV*
  - MVC
  - MVP
  - MVVM
- GRASP
- ORM
- AOP
- DI / IoC



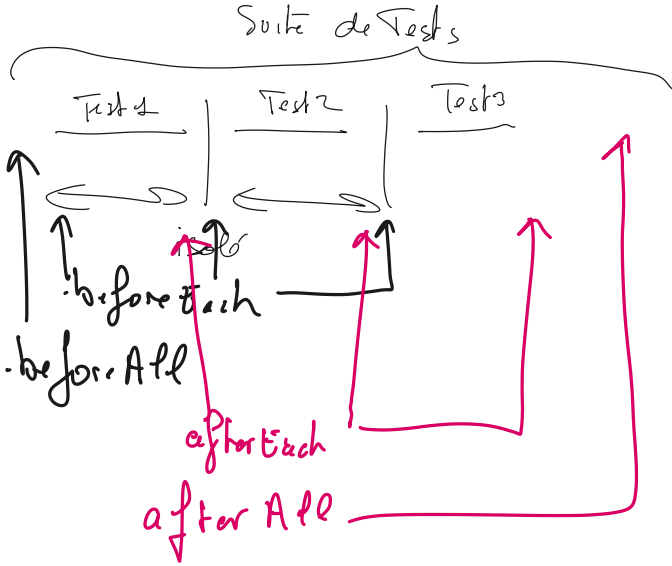
Add → TestAdd

Tous les cas de Tests ?

↳ n Méthodes _

1 .. n Tests Unitaires dans 1 Methode

m methodes de Test pour la meth



Exercice :

```
/* Specs :
    Divide doit diviser normalement avec perte selon le type
    Doit lever une exception sur division demandée non conforme
    Doit avoir un comportement cohérence sur nombres négatifs
*/
0 références
public static float Divide( float a, float b )
{
    throw new NotImplementedException("Division à implémenter");
}
```

Référence : https://docs.microsoft.com/fr-fr/visualstudio/test/improve-code-quality?view=vs-2022

Article xUnit (BDD en .Net) : https://github.com/ttutisani/Xunit.Gherkin.Quick

MSTest → .net Framework (up to 4.x)

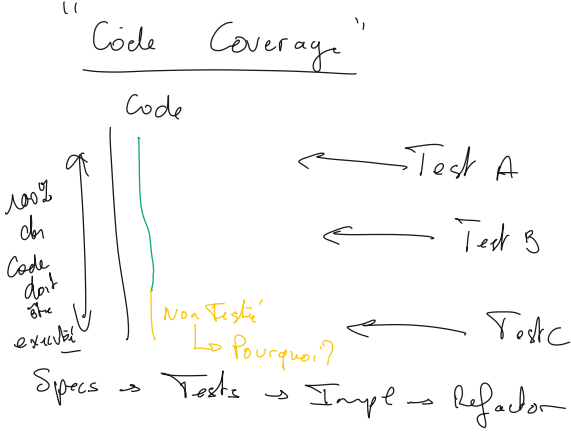
MSTest V2 → .net Standard = .net FW
.net 6+
.net core 1,2,3

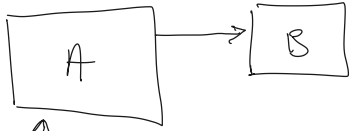
nUnit → portage de JUnit etc

xUnit → FW open source basé sur nUnit
selon les dernières bonnes pratiques.

https://www.lambdatest.com/blog/nunit-vs-xunit-vs-mstest/

Bonnes pratiques pour l'écriture des tests :
https://docs.microsoft.com/fr-fr/dotnet/core/testing/unit-testing-best-practices





Test Unitaire

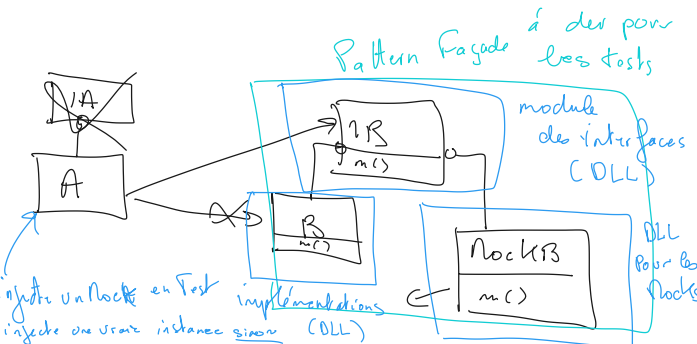
"le Comportement de A sera dépendant du Comportement de B"

Test Unitaire :

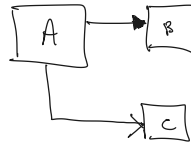
→ Test A indep du reste

Tests d'intégration

→ Ensemble



```
class Voiture
{
    private IMoteur moteur;

    public Voiture()
    {
        moteur = new Moteur();
    }

    public Voiture(IMoteur moteur)
    {
        this.moteur = moteur;
    }
```



Pattern Factory

Vehicule Factory :

```
V = new Voiture(m);
m = new Moteur
return V;
```

=> Pour les tests, le "type d'instance" doit être "paramétrable". (Pas dans ce code)

Pattern Inversion of Control → DI

base

"On décide du Type d'instance"

```
A a = new A();
```

IoC

On Demande un Type d'instance

```
IA a = System("A");
```

A en Prod

RockA en Tests

Mock :

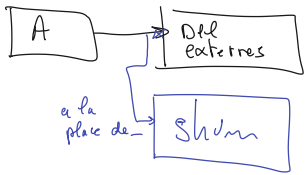
Différences : https://knplabs.com/fr/blog/mocks-fakes-stubs-dummy-et-spy-faire-la-difference

- Classe qui ressemble au Composant dépendant
- Générique -
  - ne fonctionne rien
  - peut avoir des valeurs dans ses membres -

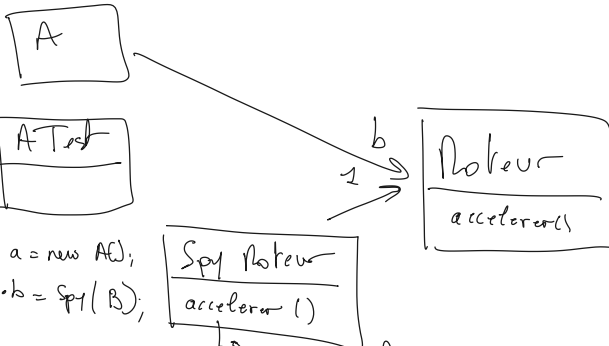
Déclinaisons :

- Dummy
  Mock qui est toujours nulle,
- Stub
  Une Implémentation qui correspond à ce que j'attends
- Fake
  Une vraie Implémentation (maquette) qui fonctionne toujours bien, avec un comportement bien connu -

(Windows) : Shim



- Spy = Catégorie de Mock
  - Qui enregistre les Appels (pour analyse, reporting, ...)
  - peut être une façade construite sur un vrai composant



Fixtures

→ Collection de données —

ex: test sur un dico de mots de passe —

Une "Fixture" est un jeu de données de Test —

"Test doubles"

→ Classe "dupliquée" pour les tests

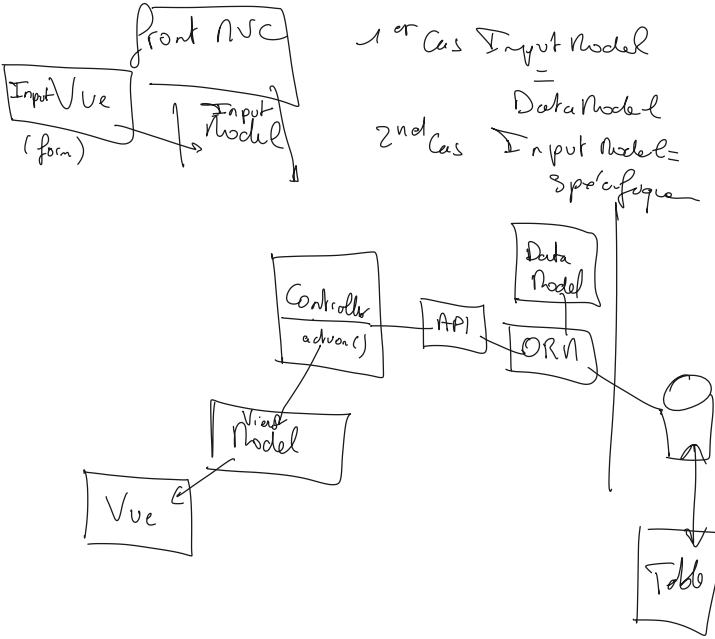
(Fake, stub, spy, etc....)

Exercice récapitulatif Test MS.Net :

- Créer une classe de tests unitaires
- En initialisation globale, obtenir une instance vers une connexion vers cette base (OleDb, Data...) vers la chaine de connexion https://www.nboost.eu/formation/comptoirs/
  - Les droits sont en lecture seule.
- Par test :
  - Faire un test qui vérifie le nombre de lignes dans une table
  - Faire un second test qui valide que nous sommes bien en lecture seule.

https://docs.microsoft.com/fr-fr/aspnet/core/test/integration-tests?view=aspnetcore-6.0
https://docs.microsoft.com/fr-fr/dotnet/core/testing/

```
using System.Data.SqlClient;

namespace TestProjectSqlAzure
{
    [TestClass]
    public class SqlAzureTest
    {

        public static SqlConnection cnx = null;
        public static TestContext context = null;


        [ClassInitialize()]
        public static void ClassInitialize(TestContext context)
        {
            SqlAzureTest.context = context;
            cnx = new SqlConnection("Server=tcp:nboost59.database.windows.net,1433;Initial Catalog=stage;Persist Security Info=False;User
ID=stagiaire;Password=Pa$$w0rd;MultipleActiveResultSets=False;Encrypt=True;TrustServerCertificate=False;Connection Timeout=30;");
        }

        [TestInitialize]
        public void TestInitialize()
        {
            cnx.Open();
        }

        [TestCleanup]
        public void TestCleanup()
        {
            cnx.Close();
        }

        [TestMethod()]
        public void TestMethod1()
        {
            var cmd = new SqlCommand("select count(*) from comptoirs.Categories", cnx);
            var rdr = cmd.ExecuteReader();
            rdr.Read();
            int res = rdr.GetInt32(0);
            Assert.AreEqual(8, res, "8 catégories attendues");
        }
    }
}
```



Test MVC en mode intégration :
https://docs.microsoft.com/fr-fr/aspnet/core/test/integration-tests?view=aspnetcore-6.0#test-app-prerequisites

(xUnit)

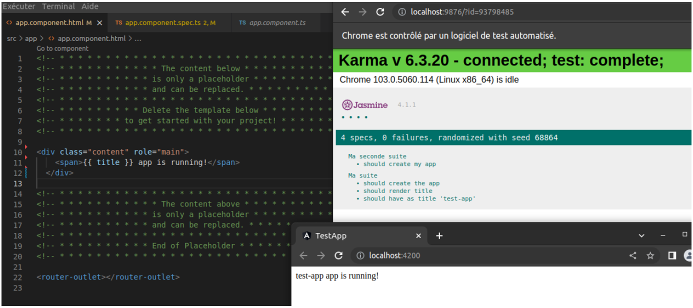
```C#
public class BasicTests
    : IClassFixture<WebApplicationFactory<RazorPagesProject.Startup>>
{
    private readonly WebApplicationFactory<RazorPagesProject.Startup> _factory;

    public BasicTests(WebApplicationFactory<RazorPagesProject.Startup> factory)
    {
        _factory = factory;
    }

    [Theory]
    [InlineData("/")]
    [InlineData("/Index")]
    [InlineData("/About")]
    [InlineData("/Privacy")]
    [InlineData("/Contact")]
    public async Task Get_EndpointsReturnSuccessAndCorrectContentType(string url)
    {
        // Arrange
        var client = _factory.CreateClient();

        // Act
        var response = await client.GetAsync(url);

        // Assert
        response.EnsureSuccessStatusCode(); // Status Code 200-299
        Assert.Equal("text/html; charset=utf-8",
            response.Content.Headers.ContentType.ToString());
    }
}
```

Angular

- ng new app
- cd app
- ng test
- ng serve -o

Exercices :

- Voir le comportement de Karma en temps réel
- Modifier la page pour l'épurer au maximum sans casser les specs.



Exercice :
Appliquez par défaut du code coverage dans votre projet
Implémenter du code dans le app.component qui ne soit pas appelé pendant les tests
(exemple :

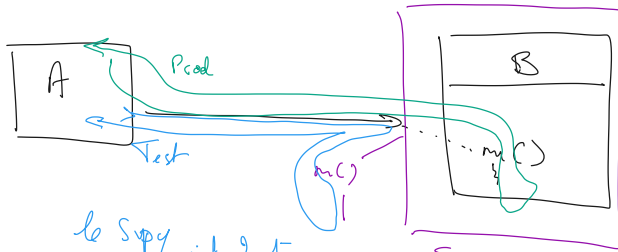
```
sayHello(name: string): void {
    console.log(`Hello ${name}`);
  }
```

)
lancez le karma runner
Vérifiez au fil des devs que le coverage est mis à jour en continu (surveillance manuelle via un autre navigateur)

Test de service :

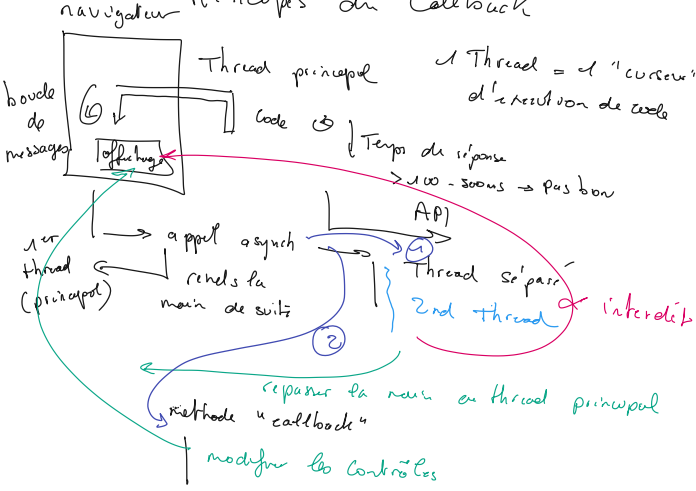
1. Créer un composant : ng generate component <component-name>
2. Créer un service : ng generate service <service-name>
3. Créer un test qui reçoive une instance du service

Jasmine : https://jasmine.github.io/tutorials/your_first_suite
Les matchers : https://jasmine.github.io/api/edge/matchers.html



le Spy intercepte ("stubbe") la classe et ses méthodes

Principes du callback



Exercice de tests unitaires sur Services avec Async :

1. Créer un service (ce qui va engendrer les tests unitaire de base - un modèle)
2. Ajouter une méthode au service qui renvoie une valeur simple en synchrone (code de base) : getUn() : return 1;
3. Implémentez un test unitaire qui valide la valeur (tout à fait basique) (expect(getUn()).toBe(1) par ex
4. Ajoutez une implémentation Observable de getUn en getUnAsync() Observable<Integer>
5. Implémenter un second test unitaire qui valide le 1 en async.

Aide :
Sync / Async :

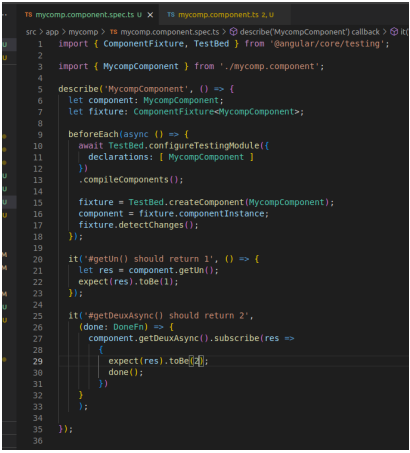
```
it('#getValue should return real value', () => {
   expect(service.getValue()).toBe('real value');
 });

  it('#getObservableValue should return value from observable',
    (done: DoneFn) => {
    service.getObservableValue().subscribe(value => {
      expect(value).toBe('observable value');
      done();
    });
  });
```

Ajout de la fonction Observable dans un service :

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

getHeroes(): Observable<Hero[]> {
  const heroes = of(HEROES);
  return heroes;
}
```

src > app > mycomp > TS mycomp.component.ts > ...

```ts
import { Component, OnInit } from '@angular/core';
import { MyserviceService } from 'src/app/myservice.service';
import { Observable, of } from 'rxjs';

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

  constructor(myservice : MyserviceService) { }
/*
  ngOnInit(): void {
  } */

  getUn(): number {
    return 1;
  }

  getDeuxAsync(): Observable<number> {
    return of(2);
  }

}
```



Ecri
<vo

-
-
-
-

Exercice :

- Les appels client http sont très fréquents
- dans le cadre des tests, nous ne devons pas nous connecter à une API en http.
    - Implémentez un service qui fasse un appel http qui renvoie un nombre (1), ou tout ce que vous voulez.
    - Implémentez son test pour appliquer un spy et une valeur constante à la place de cet appel http.

Implémentation du service : https://angular.io/tutorial/toh-pt6#get-heroes-with-httpclient

```
/** GET heroes from the server */
getHeroes(): Observable<Hero[]> {
  return this.http.get<Hero[]>(this.heroesUrl)
}
```

Implémentation du test :
https://angular.io/guide/testing-services#testing-http-services

ne composant et un test qui demande un prénom et affiche "Hello rénom>"
ez un composant vide

- https://angular.io/tutorial/toh-pt1#create-the-heroes-component

chez le dans la page principale

- https://angular.io/tutorial/toh-pt1#show-the-hero-object

utez un champ de saisie

- https://angular.io/tutorial/toh-pt1#edit-the-hero
- https://angular.io/tutorial/toh-pt3#add-the-input-hero-property

lémentez le test en injectant une valeur dans le champ de saisie, et fiant le texte affiché

- https://angular.io/guide/testing-components-scenarios#component-binding

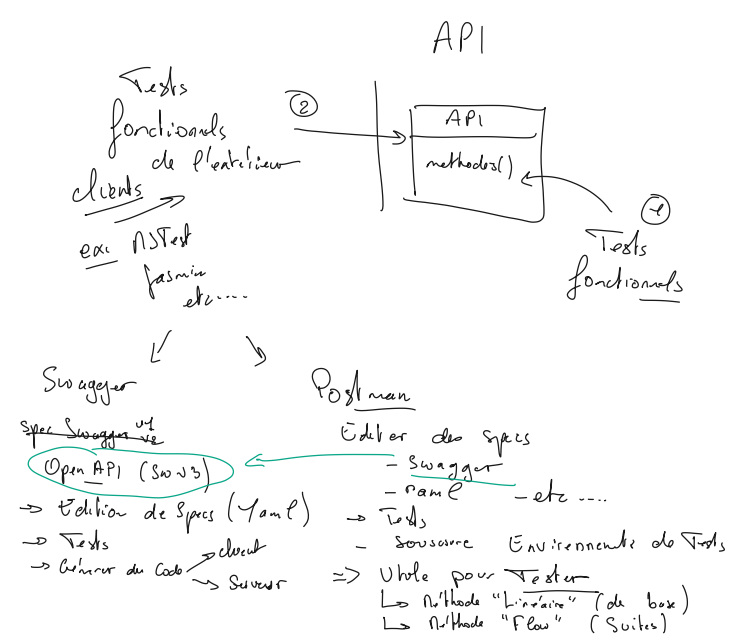

```
export class TestComponent implements OnInit {
 prenom: string = "";

 constructor() { }

 ngOnInit(): void {
 }

}
```

Méthode de tests sous Angular :
https://angular.io/guide/testing
Debug des tests :
(Depuis Karma) : https://angular.io/guide/test-debugging

Référence Jasmine :
https://jasmine.github.io/api/4.2/global
FAQ :
https://jasmine.github.io/pages/faq.html

Autres refs, tutos, etc...
https://guide-angular.wishtack.io/angular/testing/unit-testing/test-driven-development

---



Postman → Editeur | en ligne — en Equipe
on Prem ↑ Sync

→ Git, etc ....

→ Export de WorkSpace
"forme de build"
⇓
TWP

CI

→ Tests — unitaires
— int
— API → newman (en console)
↳ OK / Pas OK !

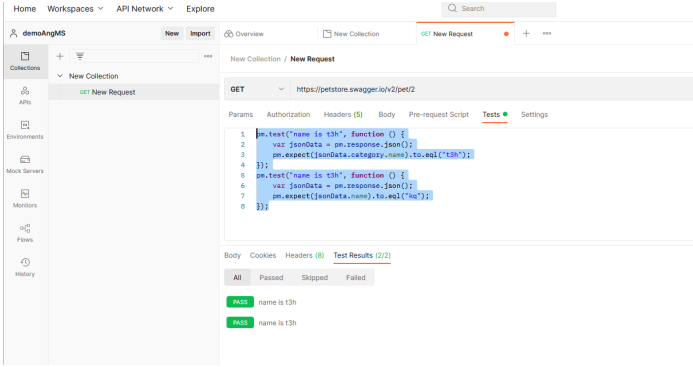
API / Tests

Requête(s) — Verbe — Params
— Auth — Body (query)
— Headers

↳ Réponse — Code (200/3xx/4xx/5xx)
— Headers
— Body

↳ Effectuer les Tests (Javascript)
→ Style BDD —

Exercice Postman :
Ecrire des tests unitaires sur une API



Référence vers l'objet pm :
https://learning.postman.com/docs/writing-scripts/script-references/postman-sandbox-api-reference/