

A futuristic car crash test scene. A robot dummy is seated in the driver's seat of a dark-colored car. The car is crashing through a wall, with debris flying everywhere. In the background, a large yellow radiation symbol is visible on a screen. The scene is lit with blue and red neon lights.

Tester c'est douter ?

TESTER ★
C'EST DOUTER ★



★★★★ **COMMITSTRIP.COM** ★★★★★

Coder sans harnais de test : bonne idée ?

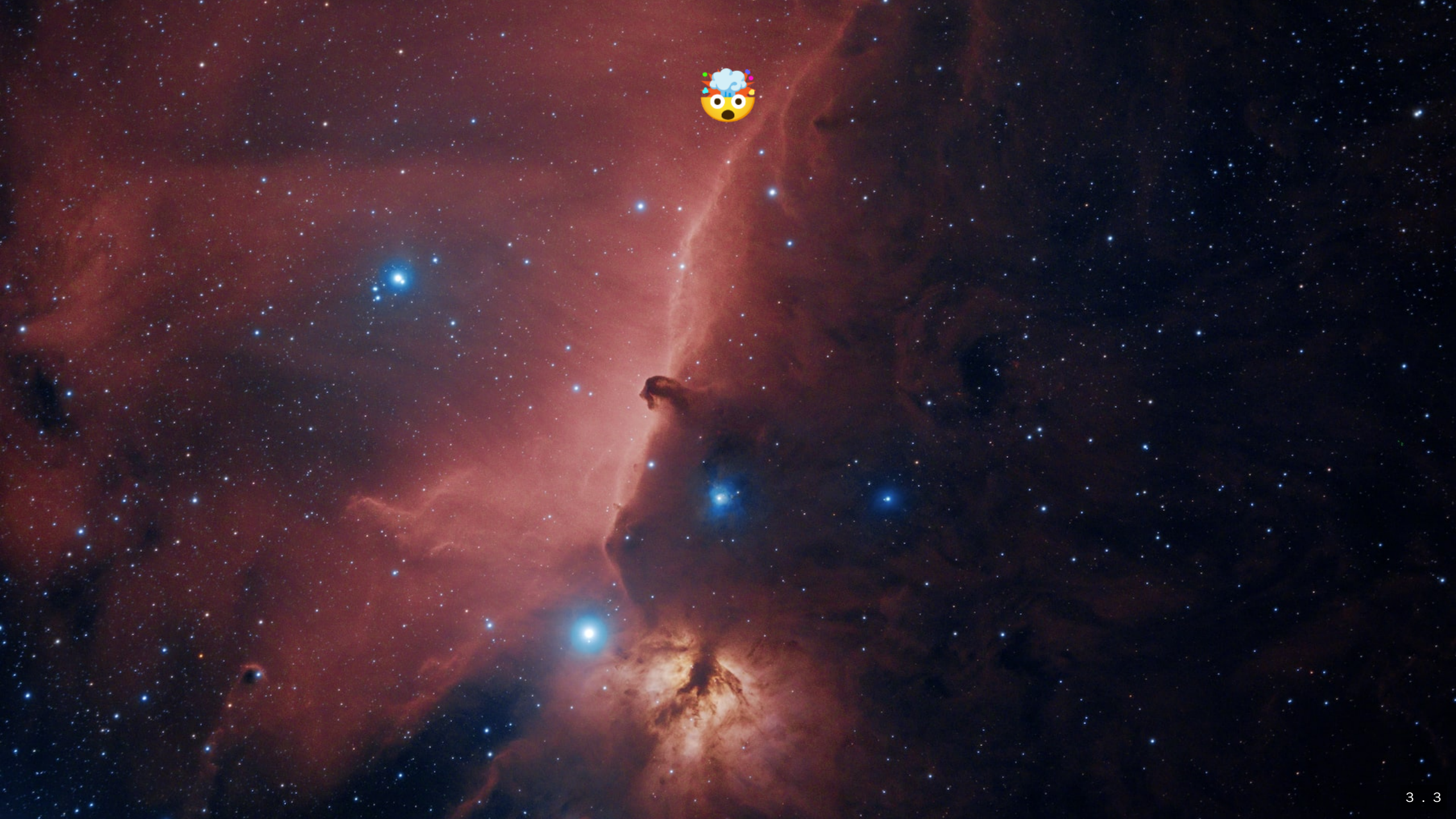


La galaxie des tests



La galaxie des tests

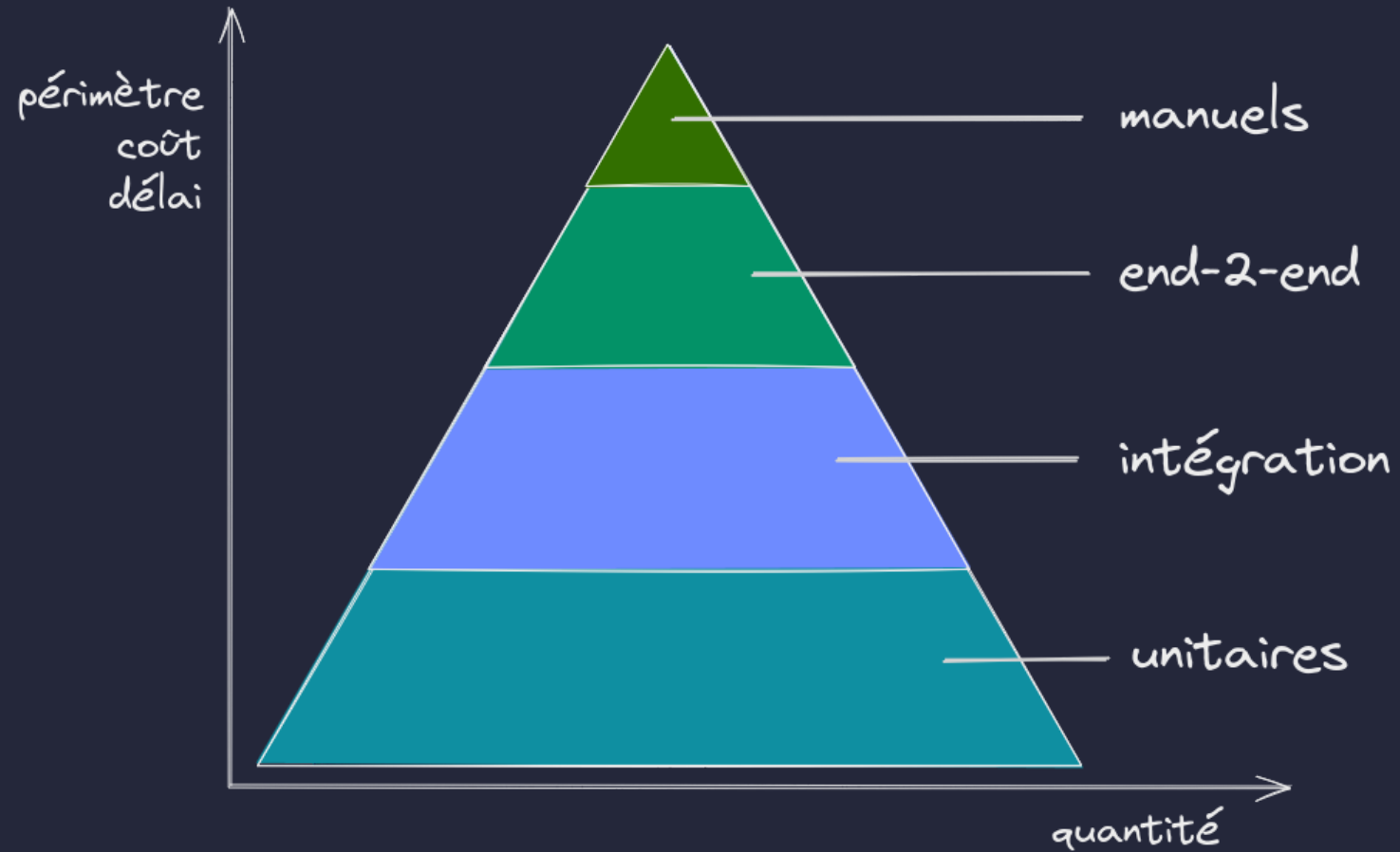
- Les tests manuels
- Les tests de charge
- Les tests end-to-end
- Les tests de contrat
- Les tests d'intégrations
- Les tests unitaires



La galaxie des tests

- ~~Les tests manuels~~
- ~~Les tests de charge~~
- ~~Les tests end-to-end~~
- ~~Les tests de contrat~~
- Les tests d'intégrations
- Les tests unitaires

La pyramide



Propriétés désirables

Pour une **suite de test** ^[1]

- Prédicatif du **succès du déploiement**
- Soutient les **évolutions du code en cours**
- Apporte un **feedback rapide**
- Minimise son **coût d'appropriation**

Les tests unitaires

F.I.R.S.T

-  Fast // Rapide
-  Independent // Indépendant
-  Repeatable // Répétable
-  Self-validating // Auto-validé
-  Thorough // Complet



Fast // Rapide

- on veut un feedback rapide
- on veut rester concentré
- test lent = pas exécuté $\Rightarrow$ test inutile



Independant // Indépendant

- une seule raison d'échouer
- pas de dépendance extérieure
 - système de fichier, bdd, random, date, ...
 - autre test



Repeatable // Répétable

- Toujours le même résultat
- Sinon, pas de confiance dans le test $\Rightarrow$ test inutile



Self-validating // Auto-validé

- la validation est automatique
- les conditions de validation sont incluses dans le test



Thorough // Complet

- les cas d'usage nominaux (*happy path*)
- les cas aux limites (*edge case*)
- les cas d'erreurs ultimes
 - base de données HS
 - HTTP 500 sur une API REST



Bonus : Simple et lisible

- un seul niveau
- pas de structures (if/boucles)

En pratique

On a besoin au minimum :

- d'**une syntaxe** pour écrire définir les tests
- d'instructions pour vérifier les résultats : **les assertions**
- d'**un outil d'exécution**

Idéalement, on pourra aussi s'aider :

- d'un outil de mesure du code testé : **le coverage**
- d'instructions pour faciliter l'isolation : **mocks/doublures de test**

Quelques outils

- Jest, pour *javascript*, *typescript*
- JUnit / AssertJ (assertions) / Jacoco (coverage) / Mockito (mocks) pour *java*

Certains langages récents intègrent nativement des outils de tests :

- Rust
- Elixir

Ok, et je met ça où ?

- ça dépend de votre outil
- généralement séparé du code de production $\Rightarrow$ garder des limites claires $\Rightarrow$ ne pas envoyer du code de test en production par erreur

Exemples :

- Jest : `__tests__`
- Java : `src/test/java`

Exécution

```
mvn test
```

```
[INFO] -----
[INFO] T E S T S
[INFO] -----
[INFO] Running feature.DescriptionTest
[INFO] Tests run: 3, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.362 s -- in feature.DescriptionTest
[INFO] Running feature.ContentTest
[INFO] Tests run: 5, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.018 s -- in feature.ContentTest
[INFO] Running feature.SandboxTest
/* ***** Lorem ipsum ***** */
/* * This is a test * */
/* * With a multiline description * */
/* * * */
/* * */ System.out.println('Code to decorate'); /* * */
/* ***** */
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.007 s -- in feature.SandboxTest
[INFO] Running feature.BorderTest
[INFO] Running feature.BorderTest$BorderLeftTest
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.060 s -- in feature.BorderTest$BorderLeftTest
[INFO] Running feature.BorderTest$BorderBottomTest
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.007 s -- in feature.BorderTest$BorderBottomTest
[INFO] Running feature.BorderTest$BorderRightTest
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.013 s -- in feature.BorderTest$BorderRightTest
[INFO] Running feature.BorderTest$BorderTopTest
[INFO] Tests run: 3, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.020 s -- in feature.BorderTest$BorderTopTest
[INFO] Tests run: 0, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.164 s -- in feature.BorderTest
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 18, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 6.588 s
[INFO] Finished at: 2025-05-07T22:28:37+02:00
[INFO]
```

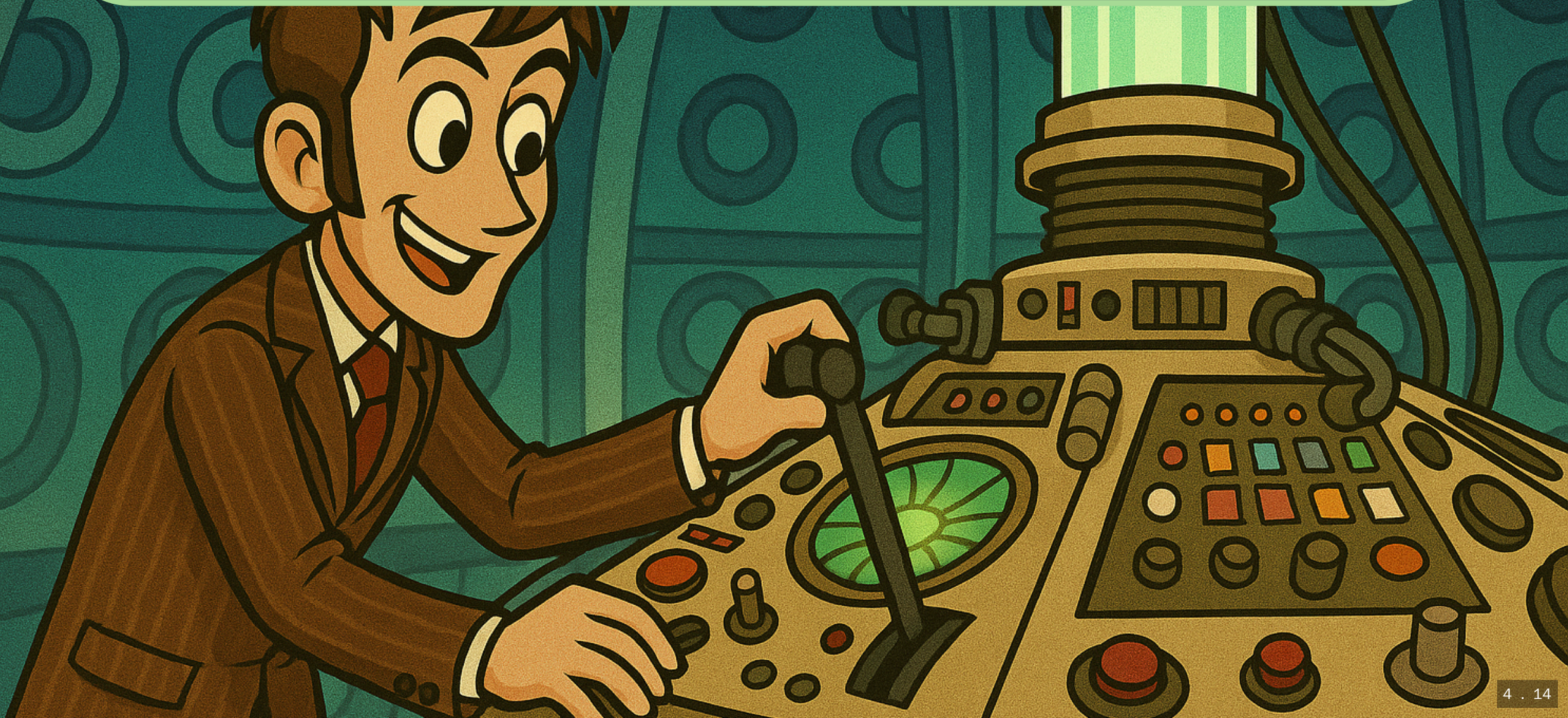
Mesure de la couverture du code

```
code-decorator > org.gitlab.mbarberot.code.decorator > CodeDecorator.java

CodeDecorator.java

1. package org.gitlab.mbarberot.code.decorator;
2.
3. import java.util.ArrayList;
4. import java.util.List;
5.
6. import static java.util.stream.Collectors.joining;
7. import static java.util.stream.Collectors.toList;
8. import static org.gitlab.mbarberot.code.decorator.BorderStyle.javaStyle;
9. import static org.gitlab.mbarberot.code.decorator.TextUtils.maxLineLength;
10. import static org.gitlab.mbarberot.code.decorator.TextUtils.splitToLines;
11.
12. public class CodeDecorator {
13.
14.     private final String code;
15.     private final String title;
16.     private final String description;
17.     private final BorderStyle borderStyle;
18.     private final int contentLength;
19.
20.     public CodeDecorator(String code) {
21.         this(code, "", "", javaStyle());
22.     }
23.
24.     public CodeDecorator(String code, String title) {
25.         this(code, title, "", javaStyle());
26.     }
27.
28.     public CodeDecorator(String code, String title, String description) {
29.         this(code, title, description, javaStyle());
30.     }
31.
32.     public CodeDecorator(String code, String title, String description, BorderStyle borderStyle) {
33.         this.code = code;
34.         this.title = title == null ? "" : title;
35.         this.description = description == null ? "" : description;
36.         this.borderStyle = borderStyle;
37.         this.contentLength = calculateContentLength();
38.     }
39.
40.     private int calculateContentLength() {
41.         return Math.max(
42.             maxLineLength(splitToLines(code)) + borderStyle.escapeLength(),
43.             maxLineLength(splitToLines(description))
44.         );
45.     }
46. }
```

Allons-y !



Anatomie d'un test

Trois étapes :

- Arrange
- Act
- Assert

Anatomie d'un test

```
1    @Test
2    void rollDice_return1() {
3        /* Arrange */
4        var randomGenerator = mock(RandomGenerator.class);
5        when(randomGenerator.nextInt()).thenReturn(1);
6        var dice = new Dice(randomGenerator);
7
8        /* Act */
9        int result = dice.roll();
10
11       /* Assert */
12       assertThat(result).isEqualTo(1);
13   }
```

Les assertions

- pour vérifier le résultat obtenu
- idéalement une seule par test
- produisant un rapport de test utile

Les assertions (exemple)

En JS/TS avec Jest (Documentation)

```
expect (age) .toEqual (34)  
expect (age) .not .toBeLessThan (64)  
expect (password) .toMatch (" [A-Za-z0-9]+")
```

En Java avec AssertJ (Documentation)

```
assertThat (age) .isBetween (18, 100);  
assertThat (wordList) .containsExactlyInAnyOrder ("foo", "border");  
assertThat (password) .matches (" [a-zA-Z0-9+-$*!]+")
```

Le coverage

- mesure du code exercé par les tests
 - utile pour détecter les zones non testées
 - attention à la quête impossible des 100%
-
- Exemple de rapport de coverage

Les mocks

- cas particulier des **doublures de test** ^[1]
- permettent de donner rapidement une implémentation différente
- on peut faire des assertions sur leur utilisation

1. Pensez doublures comme des cascadeurs dans les films

Les mocks (exemple)

```
@Test
void rollDice_return1_withMock() {
    /* Arrange */
    var randomGenerator = mock(RandomGenerator.class);
    when(randomGenerator.nextInt()).thenReturn(1);
    var dice = new Dice(randomGenerator);

    /* Act */
    int result = dice.roll();

    /* Assert */
    assertThat(result).isEqualTo(1);
    verify(randomGenerator, times(1)).nextInt();
}
```

-  Documentation Jest : Mock
-  Documentation Mockito : Mock

Les doublures de test

- permettent également de donner une implémentation différente
- avantage : réutilisable dans d'autres tests ou dans le code de production
- inconvénient : plus difficile de faire des assertions dessus

Les doublures de test communes

- **Dummy**
- **Fake**
- **Stub**
- **Spy**
- **Mock**

Les doublures de test (exemple)

```
@Test
void rollDice_return1_withTestDouble() {
    /* Arrange */
    RandomGenerator randomGenerator = () -> 1;
    var dice = new Dice(randomGenerator);

    /* Act */
    int result = dice.roll();

    /* Assert */
    assertThat(result).isEqualTo(1);
}
```

Mocks vs. autres doublures de test ?

- mocks/spies : faciles à écrire avec bibliothèques de mocking, souvent utilisés au besoin, difficiles à maintenir sur de grandes suites de test
- dummy/fake/stub : peuvent être écrits sans bibliothèques de mocking, souvent réutilisable et plus faciles à maintenir que des mocks sur des suites de tests grandissantes
- spies écrits à la main : faciles à écrire et maintenir sur des petites classes

Maintenir le code de test

*Vous devez mettre autant de soin dans le code de test
que dans le code de production*
— Mathieu Barberot

- Le code doit rester facile à lire
- Les **code smells** existent aussi dans les tests

 Code smells - Refactoring Guru

Nommer tests

- Rapport de test compréhensible
- Javascript : intitulés dans les `describe()`, `test()`, `it()`
- Java JUnit
 - `@DisplayName`
 - `src/test/resources/junit-platform.properties:`
`junit.jupiter.displayName.generator.default`



Display Names - JUnit

Description d'assertions

- Parfois utiles
 - comparaison d'un nombre avec un autre
 - comparaison de boolean

Assertion description - AssertJ

```
TolkienCharacter frodo = new TolkienCharacter("Frodo", 33, Race.HOBBIT);  
// failing assertion, remember to call as() before the assertion!  
assertThat(frodo.getAge()).as("check %s's age", frodo.getName()).isEqualTo(100);
```

```
[check Frodo's age] expected:<100> but was:(33)
```

Limiter les assertions

- plusieurs assertions = plusieurs raisons d'échouer
- tests fragiles
- signe qu'on teste trop de chose d'un seul coup

⇒ limiter le nombre d'assertions à 1 ou 2 ⇒ faire plusieurs tests pour dispatcher les assertions en trop

Exemple : Avant refactoring

```
@Test
void player_can_fight() {
    // Arrange
    var kevin = new Joueur("Kevin", 5, 100, 75, 30);
    var slime = new Monstre("Slime", 5, 30, 25);

    // Act
    combatService.attack(kevin, slime);

    // Arrange
    assertThat(slime.getHp()).isZero();
    assertThat(slime.isDead()).isTrue();
    assertThat(kevin.getXp()).isZero();
    assertThat(kevin.getLevel()).isEqualTo(6);
}
```

Exemple : Après refactoring

```
@Test
void monster_can_die_in_combat() {
    // Arrange
    var kevin = new Joueur("Kevin", 5, 100, 75, 30);
    var slime = new Monstre("Slime", 5, 30, 10);

    // Act
    combatService.attack(kevin, slime);

    // Arrange
    assertThat(slime.getHp()).isZero();
    assertThat(slime.isDead()).isTrue();
}
```

Exemple : Après refactoring

```
@Test
void player_gain_xp_on_monster_death() {
    // Arrange
    var kevin = new Joueur("Kevin", 5, 100, 75, 30);
    var slime = new Monstre("Slime", 5, 30, 10);

    // Act
    combatService.attack(kevin, slime);

    // Arrange
    assertThat(kevin.getXp()).isEqualTo(85);
}
```

Exemple : Après refactoring

```
@Test
void player_can_level_up_on_monster_death() {
    // Arrange
    var kevin = new Joueur("Kevin", 5, 100, 75, 30);
    var slime = new Monstre("Slime", 5, 30, 25);

    // Act
    combatService.attack(kevin, slime);

    // Arrange
    assertThat(kevin.getLevel()).isEqualTo(6);
}
```

Eviter la duplication

- quand les tests sont quasiment des copier/coller

Exemple : Avant refactoring

```
@Test
void formatsDateToDayMonthYear() {
    // Arrange
    // Act
    String formatted = parse("2023-01-01").format(ofPattern("dd/MM/yyyy"));
    // Assert
    assertThat(formatted).isEqualTo("01/01/2023");
}

@Test
void formatsDateToMonthNameAndDay() {
    // Arrange
    // Act
    String formatted = parse("2024-12-25").format(ofPattern("MMMM d"));
    // Assert
    assertThat(formatted).isEqualTo("December 25");
}

@Test
void formatsDateToIso() { /* ... */ }
```

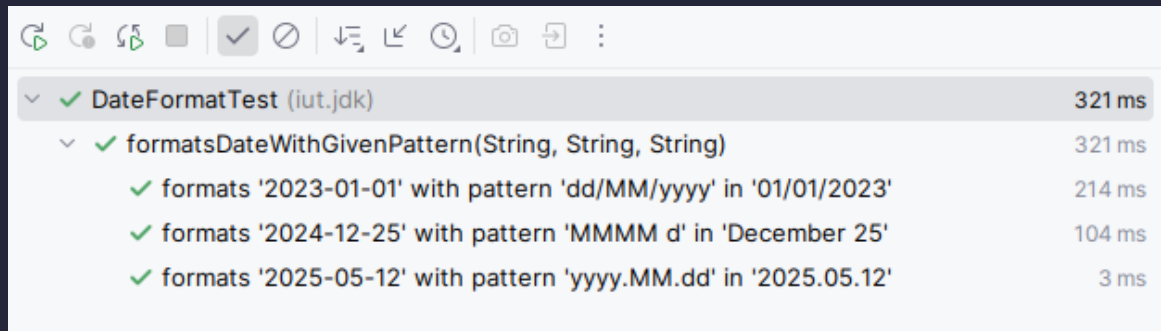
Exemple : Tests paramétrés

```
@ParameterizedTest(name = "formats '{0}' with pattern '{1}' in '{2}'")
@CsvSource({
    "2023-01-01,      dd/MM/yyyy,      01/01/2023",
    "2024-12-25,      MMMM d,          December 25",
    "2025-05-12,      yyyy.MM.dd,      2025.05.12"
})
void formatsDateWithGivenPattern(String input, String pattern, String expected) {
    // Arrange
    LocalDate date = parse(input);

    // Act
    String formatted = date.format(ofPattern(pattern, Locale.US));

    // Assert
    assertThat(formatted).isEqualTo(expected);
}
```

Résultat



A screenshot of a Java IDE's test runner window. The window has a toolbar at the top with icons for running, debugging, and other actions. Below the toolbar, a tree view shows the test results. The root node is 'DateFormatTest (iut.jdk)' with a green checkmark and a duration of 321 ms. It is expanded, showing a sub-node 'formatsDateWithGivenPattern(String, String, String)' with a green checkmark and a duration of 321 ms. This sub-node is further expanded, showing three individual test cases, each with a green checkmark and a duration: 'formats '2023-01-01' with pattern 'dd/MM/yyyy' in '01/01/2023' (214 ms), 'formats '2024-12-25' with pattern 'MMMM d' in 'December 25' (104 ms), and 'formats '2025-05-12' with pattern 'yyyy.MM.dd' in '2025.05.12' (3 ms).

✓ DateFormatTest (iut.jdk)	321 ms
✓ formatsDateWithGivenPattern(String, String, String)	321 ms
✓ formats '2023-01-01' with pattern 'dd/MM/yyyy' in '01/01/2023'	214 ms
✓ formats '2024-12-25' with pattern 'MMMM d' in 'December 25'	104 ms
✓ formats '2025-05-12' with pattern 'yyyy.MM.dd' in '2025.05.12'	3 ms

Factoriser // Arrange

- clarifier le setup du test
- réutiliser le code dans plusieurs tests

Example

```
@Test
void anItemCanBeAddedIntoACart() {
    // Arrange
    Cart cart = new Cart();

    // Act
    cart.addItem(new Item(1, "Keyboard", 100));

    // Assert
    assertThat(cart.getItems()).hasSize(1);
}
```

BeforeEach / AfterEach

```
private Cart cart;

@BeforeEach
void setUp() {
    cart = new Cart();
}

@Test
void anItemCanBeAddedIntoACartUsingBeforeEach() {
    // Arrange
    // Act
    cart.addItem(new Item(1, "Keyboard", 100));

    // Assert
    assertThat(cart.getItems()).hasSize(1);
}
```

BeforeAll / AfterAll

- ⚠ cela peut rompre l'indépendance de vos tests !
- peut être considéré comme un code smell

Limitations

- Duplication possible entre plusieurs suites de tests
- Séparation du code = moins de lisibilité
- Des tests peuvent ne pas utiliser la totalité du beforeEach

Factories

- Factorisation plus générique car réutilisable entre les suites
- Nommage des fonctions pour la lisibilité
- Plusieurs méthodes pour configurer le strict minimum à chaque test

Factories

```
class CartFactory {  
    public static Cart createEmptyCart() {  
        return new Cart();  
    }  
}  
  
@Test  
void multipleItemsCanBeAddedIntoACartUsingFactory() {  
    // Arrange  
    Cart emptyCart = CartFactory.createEmptyCart();  
  
    // Act  
    emptyCart.addItem(new Item(1, "Keyboard", 100));  
    emptyCart.addItem(new Item(2, "Mouse", 50));  
  
    // Assert  
    assertThat(emptyCart.getItems()).hasSize(2);  
}
```

Aller plus loin

- Les personas
 - pousser le concept de factories
 - incarner des types d'utilisateurs
 - requiert une coordination de l'équipe

Persona (Wikipedia)

Persona (exemple)

```
class LillyFactory {  
    public static Cart createCart() {  
        Cart cart = new Cart();  
        cart.addItem(new Item(1, "Gamer Keyboard", 200));  
        cart.addItem(new Item(2, "Gamer Mouse", 100));  
        return cart;  
    }  
}  
  
@Test  
void lillyHasAnExpensiveCart() {  
    // Arrange  
    // Act  
    Cart lillysCart = LillyFactory.createCart();  
  
    // Assert  
    assertThat(lillysCart.getTotal()).isGreaterThan(250);  
}
```

Les tests d'intégration

Les critères

- Tester que les différentes parties sont bien branchées

Exemple:

- Je tourne le volant $\Rightarrow$ les roues tournent

La règle

- Comme les tests unitaires
- Mais avec moins de contraintes

F.I.R.S.T

-  Fast
 - Autant que possible
-  Independant
 - On accepte des dépendances, mais maîtrisées
 - Utiliser une base de données éphémère
 - Utiliser des faux services tiers...

F.I.R.S.T

-  Repeatable
 - Toujours !
-  Self-validated
 - Toujours !
-  Thorough
 - Le *happy path* et des *edge cases*

Avec quels outils ?

Pour une SPA :

- Playwright
- Cypress
- Jest

Avec quels outils ?

Pour une API :

- Bruno / Postman (ou équivalent)
- Playwright
- Hurl / JetBrains Http Client
- RestAssured *pour Java*

Tester une SPA

1. Installer et initialiser Playwright ^[1]

```
npm init playwright@latest
```

1. <https://playwright.dev/docs/intro>

Tester une SPA

1. Installer et initialiser Playwright
2. Écrire un test

```
test('Home page has the right title in page metadata', async ({ page }) => {  
  // Arrange  
  // Act  
  await page.goto('/');  
  
  // Assert  
  await expect(page).toHaveTitle("Keyboard Factory");  
});
```

Tester une SPA

1. Installer et initialiser Playwright
2. Écrire un test
3. Exécuter une suite de test

```
npx playwright test
```

Tester une SPA

Q

All 8Passed 8Failed 0Flaky 0Skipped 0

5/15/2025, 2:30:46 PMTotal time: 14.8s

▼ catalog.spec.ts

✓ A catalog has itemschromium2.2s

catalog.spec.ts:3

✓ A catalog has itemsfirefox3.6s

catalog.spec.ts:3

▼ home.spec.ts

✓ Home page has the right title in page metadatachromium2.2s

home.spec.ts:4

✓ Home page has the right title in headerchromium1.0s

home.spec.ts:14

✓ Home page shows the catalogchromium1.2s

home.spec.ts:24

✓ Home page has the right title in page metadatafirefox4.0s

home.spec.ts:4

✓ Home page has the right title in headerfirefox1.7s

home.spec.ts:14

✓ Home page shows the catalogfirefox1.6s

home.spec.ts:24

Tester une API REST

⇒ Documenter l'API

```
@OpenApi(  
    path = "/rover",  
    methods = {HttpMethod.POST},  
    summary = "Create a rover",  
    requestBody = @OpenApiRequestBody(  
        content = {@OpenApiContent(from = RoverCreationDto.class)},  
        required = true  
    ),  
    responses = {@OpenApiResponse(status = "201")}  
)  
public static void create(Context ctx) {  
    /* ... */  
}
```

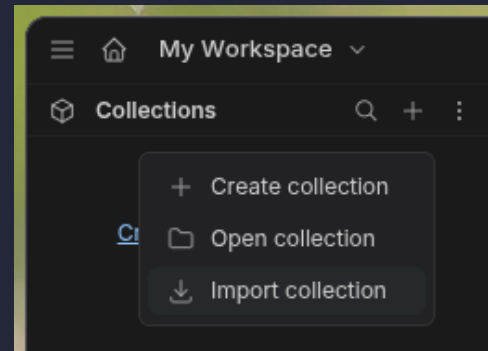
Tester une API REST

⇒ Documenter les structures de données

```
public record CreateActionDto(  
    @OpenApiRequired  
    @OpenApiDescription("Le nom du rover")  
    String rover,  
    @OpenApiRequired  
    @OpenApiDescription("L'action à effectuer : forward, backward, turn_left, turn_right")  
    String action  
) {}
```

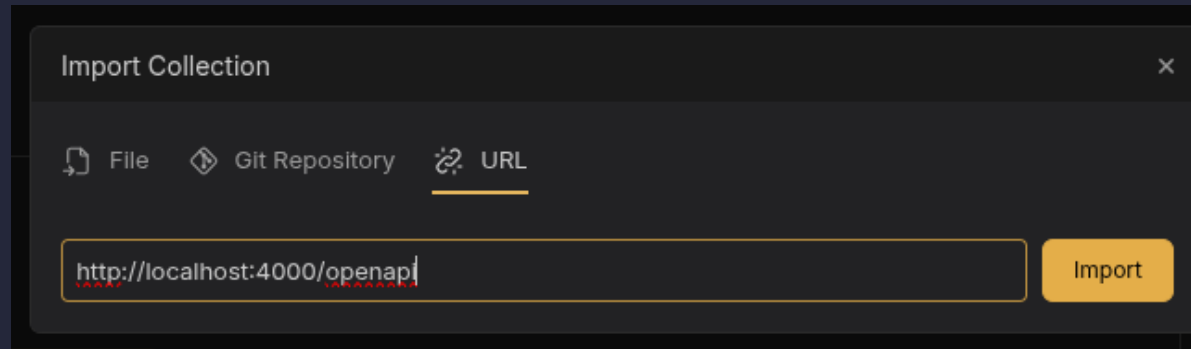
Tester une API REST

⇒ Initialiser un client REST

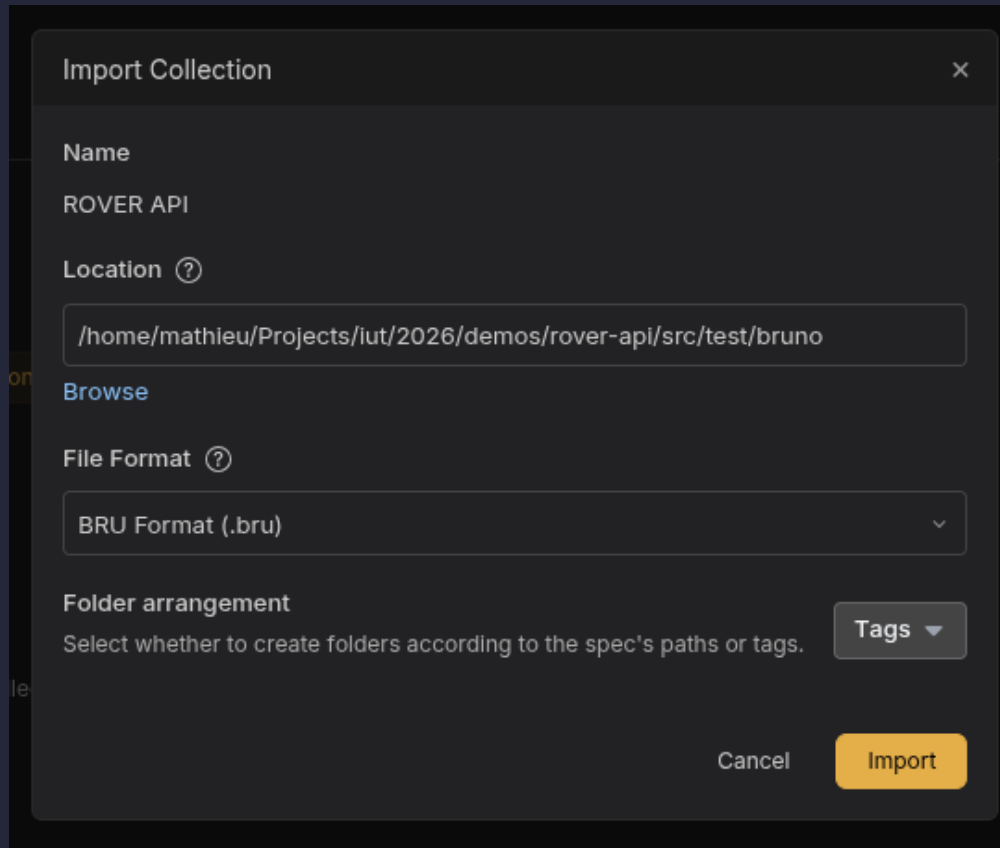


Tester une API REST

⇒ Importer via la documentation OpenAPI



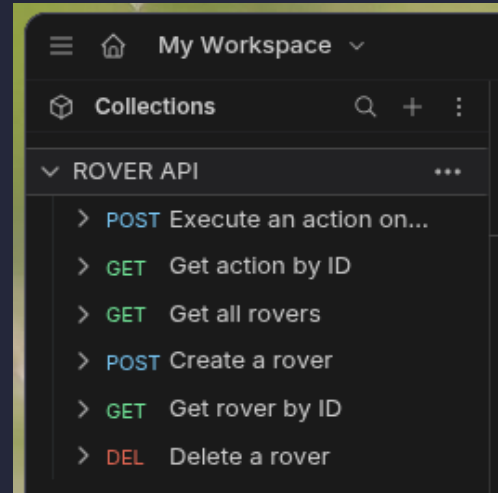
Tester une API REST



The image shows a dark-themed 'Import Collection' dialog box with a close button (X) in the top right corner. It contains the following fields and options:

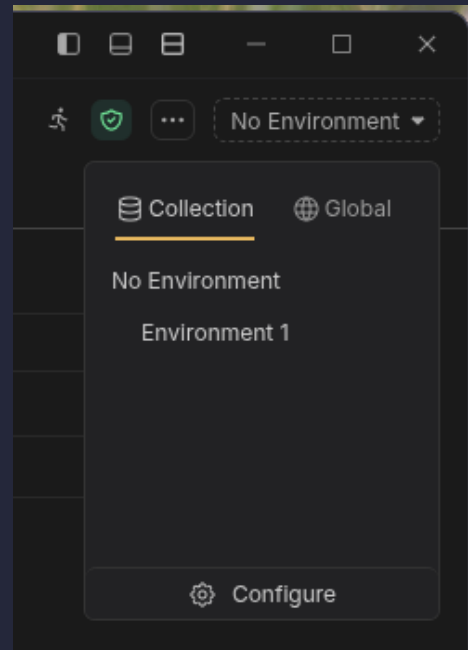
- Name:** ROVER API
- Location ?**: A text input field containing the path `/home/mathieu/Projects/iut/2026/demos/rover-api/src/test/bruno`. Below the field is a blue 'Browse' link.
- File Format ?**: A dropdown menu currently showing 'BRU Format (.bru)' with a downward arrow.
- Folder arrangement**: A section with the text 'Select whether to create folders according to the spec's paths or tags.' and a 'Tags' button with a downward arrow.
- Buttons:** 'Cancel' and 'Import' buttons at the bottom right.

Tester une API REST

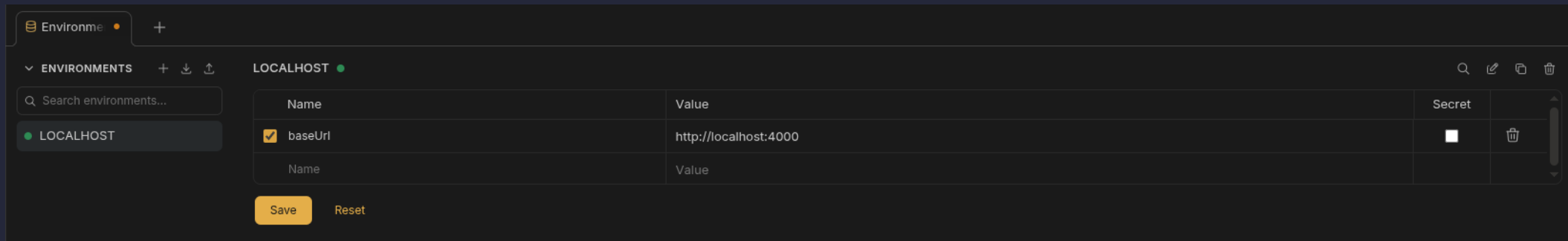


Tester une API REST

⇒ Configurer l'environnement



Tester une API REST



Tester une API REST

⇒ Exécuter une requête

The screenshot displays a REST client interface with a dark theme. At the top, there are tabs for 'GET Get all rovers' and '200 Response'. The main area shows a GET request to `{{baseUrl}}/rover`. Below the request bar, there are tabs for 'Params', 'Body', 'Headers', 'Auth', 'Vars', 'Script', 'Assert', 'Tests', 'Docs', and 'Settings'. The 'Params' tab is active, showing a table with two columns: 'Name' and 'Value'. Below the table is a 'Path' field and a 'Bulk Edit' button. On the right side, the 'Response' tab is active, showing a JSON response. The response is a list containing one object with fields 'id', 'name', 'direction', and 'position'. The status bar at the top right of the response panel indicates '200 OK', '17ms', and '146B'.

```
GET {{baseUrl}}/rover
```

Name	Value
Name	Value

Path ? Bulk Edit

```
[
  {
    "id": "c8f48335-042c-41f6-b602-8a0486941f14",
    "name": "R2D2",
    "direction": "north",
    "position": {
      "x": 0,
      "y": 0
    }
  }
]
```

200 OK 17ms 146B

Tester une API REST

⇒ Ajouter des assertions

The screenshot shows a REST client interface with a GET request `GET {{baseUrl}}/rover`. The response is a 200 OK status with a 27ms response time and 146B of data. The 'Assert' tab is active, displaying a table of assertions:

Expr	Operator	Value	
☒ <code>res.status</code>	<code>equals</code>	<code>200</code>	
☒ <code>res.body</code>	<code>isEmpty</code>		
☒ <code>res.body[0].name</code>	<code>equals</code>	<code>"R2D2"</code>	
Expr	<code>equals</code>		

The 'Tests' tab is also active, showing the following assertions:

- `res.status: eq 200`
- `res.body[0].name: eq "R2D2"`
- `res.body: isEmpty`

Tester une API REST

⇒ Ou carrément écrire des tests en JS

The screenshot displays a REST client interface with a dark theme. At the top, there are tabs for 'GET Get all rovers' and '200 Response'. The main area shows a GET request to `{{baseUrl}}/rovers`. Below the request, there are tabs for 'Params', 'Body', 'Headers', 'Auth', 'Vars', 'Script', 'Assert', 'Tests', 'Docs', and 'Settings'. The 'Tests' tab is active, showing two JavaScript test functions:

```
1 test("is successful", function () {
2   expect(res.getStatus()).toEqual(200);
3 });
4
5 test("has R2D2 rover", function () {
6   const body = res.getBody();
7
8   expect(body).not.toBe.empty;
9   expect(body[0].name).toEqual("R2D2");
10
11 });
```

On the right side, there are tabs for 'Response', 'Headers', 'Timeline', and 'Tests'. The 'Tests' tab is active, showing the results of the tests:

- Tests (2), Passed: 2, Failed: 0
 - is successful
 - has R2D2 rover
- Assertions (3), Passed: 3, Failed: 0
 - res.status: eq 200
 - res.body: isEmpty
 - res.body[0].name: eq "R2D2"

At the top right of the right panel, the status '200 OK' is displayed along with '12ms' and '146B'.

Merci de votre attention

