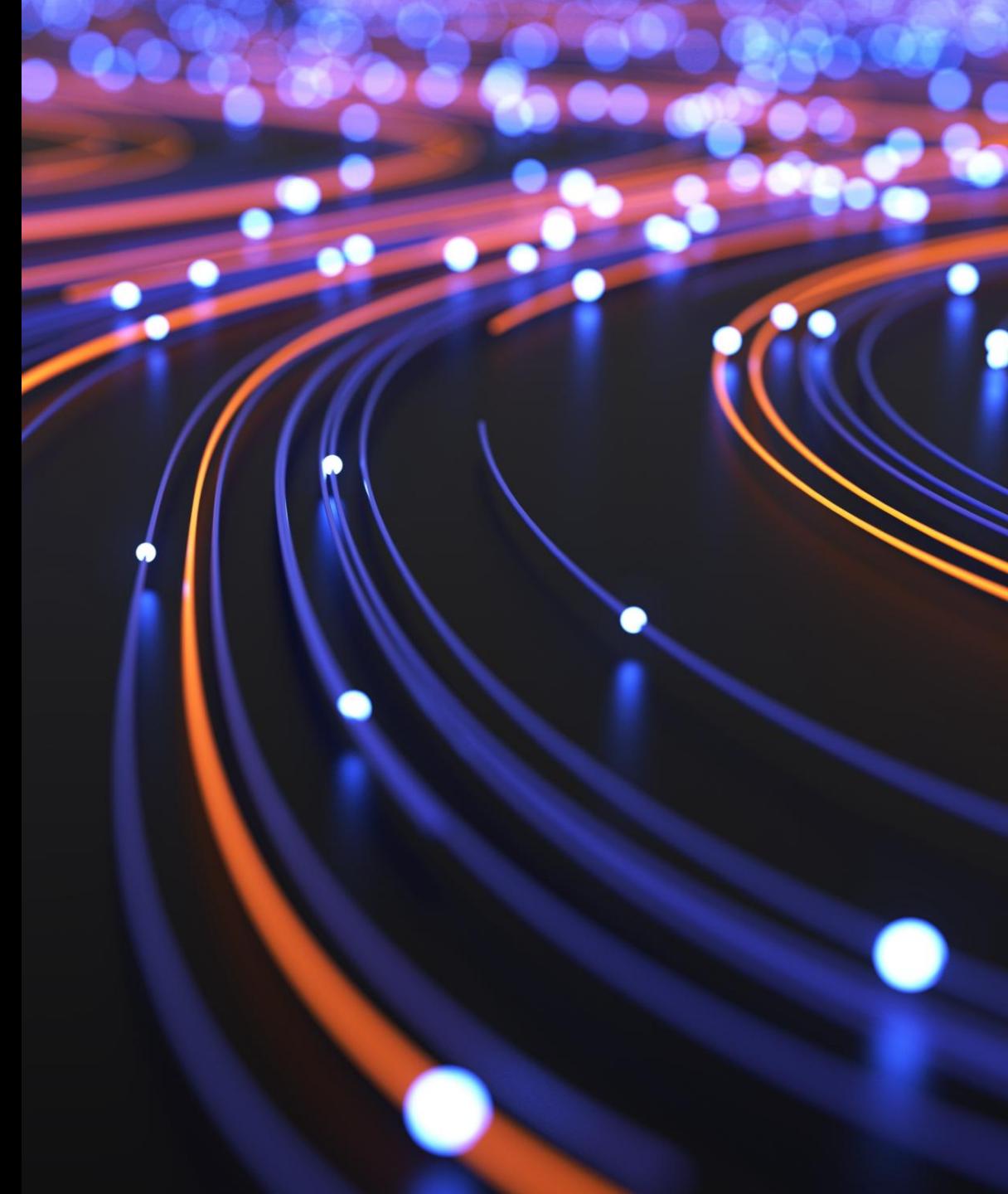
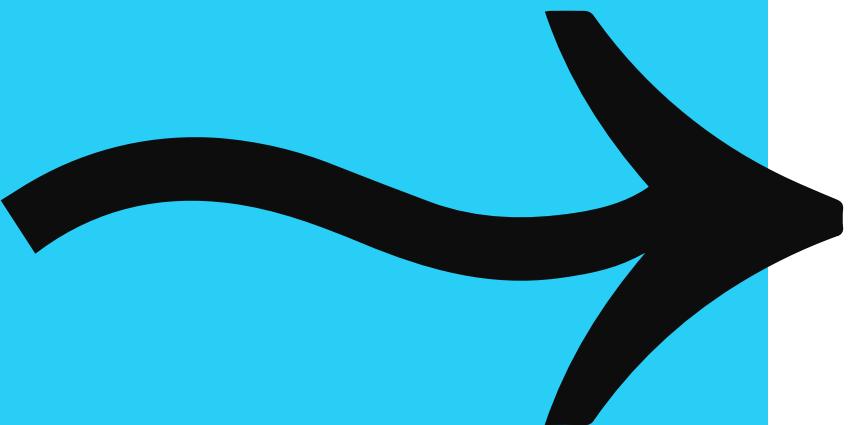


Behaviour-Driven Development



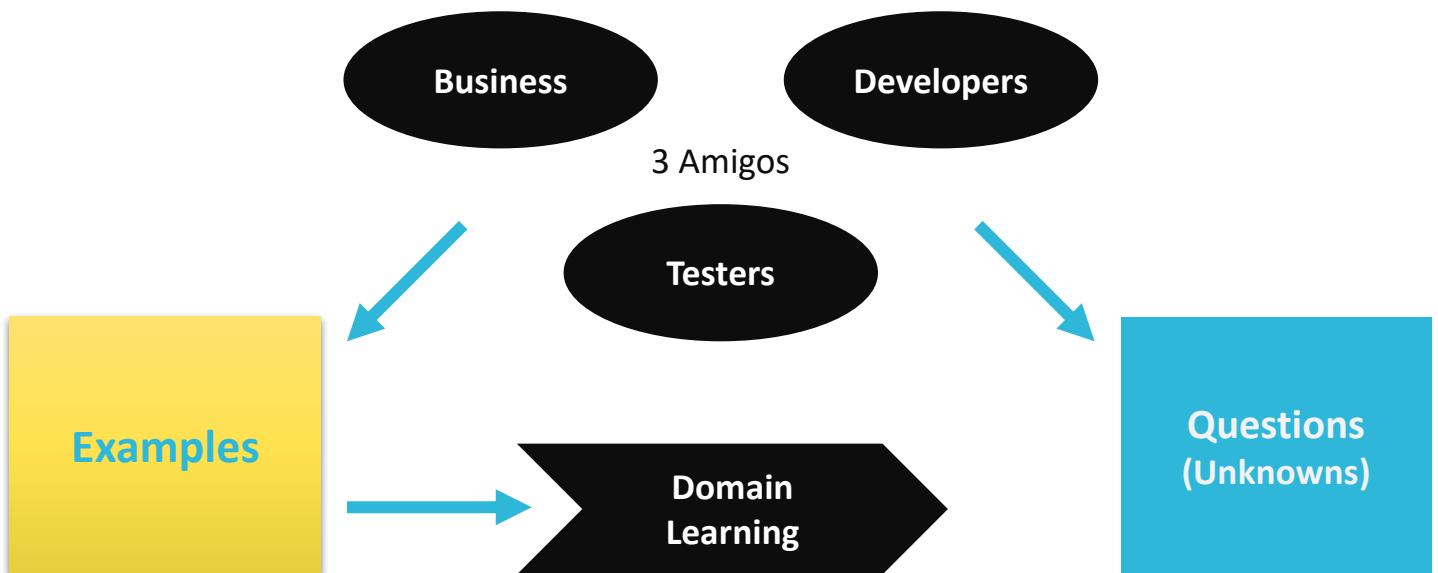
What is BDD?



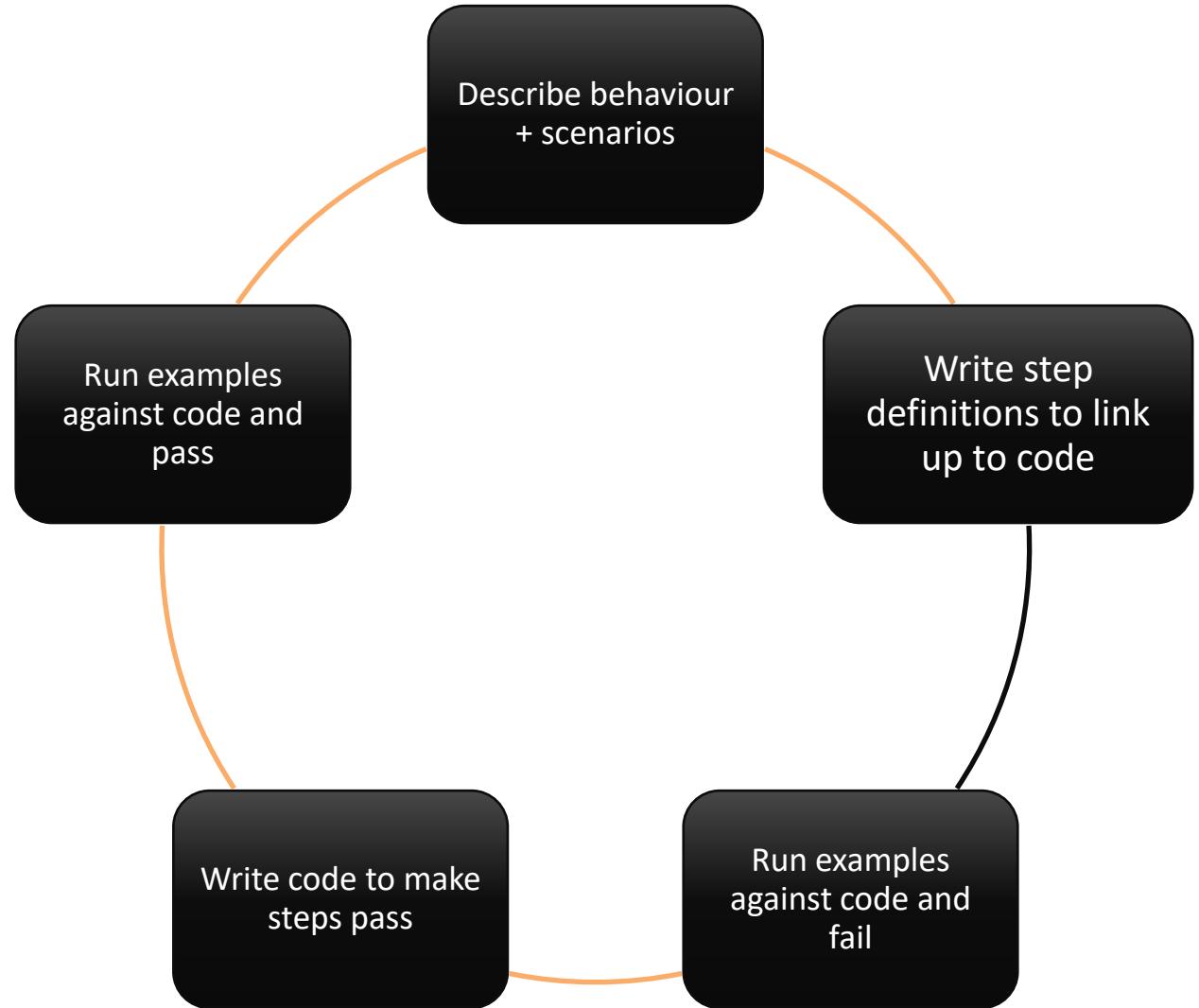
BDD is about conversation and collaboration

Language understandable to all stakeholders:

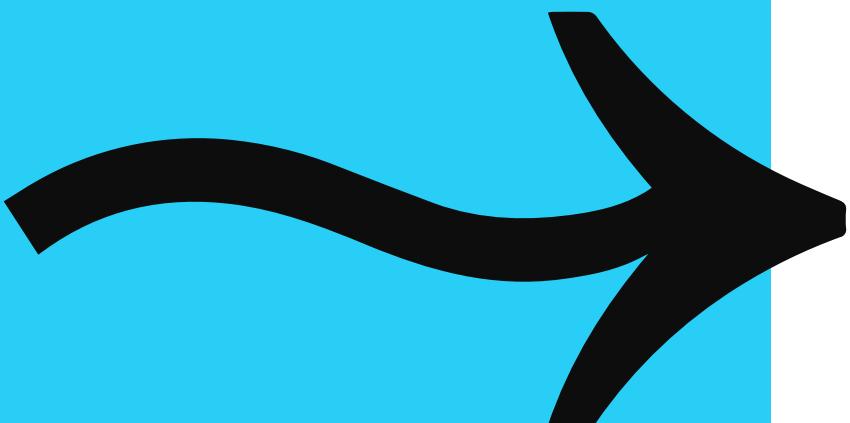
- Requirements = features
- Acceptance criteria = scenarios
- Scenarios illustrate how features work



BDD Process



Defining Scenarios



Situation	Specific to feature	Description
Scenario	<name>	Concise title
Given	<assumption/context1>	Assuming a current state or context
And	<assumption/context2>	Additional context clauses
When	<event occurs>	When specific event(s) occur
And	<additional events occur>	
Then ensure this	<outcome occurs>	The following result(s)/outcome(s) should happen
And this	<additional outcome occurs>	

Cucumber and Gherkin



Cucumber is a tool that supports BDD

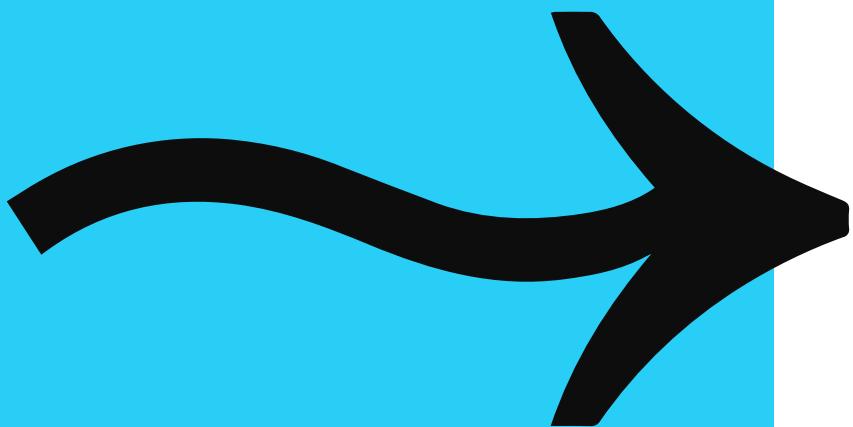
Cucumber reads plain-text specifications and checks that the software behaves as expected.

The scenarios must follow the syntax rules, called **Gherkin**.

Scenarios are broken into step definitions:

```
Given I have £100 in checking account
And I have £20 in saving account
When I transfer £15 from checking account to savings account
Then I should have £85 in checking account
And I have £35 in saving account
```

Writing Scenarios



1. Write the 'happy path' scenario:

Given my shopping basket is empty

When I add the book "BDD is Fun" to the basket

Then the shopping basket contains 1 copy of "BDD is Fun"

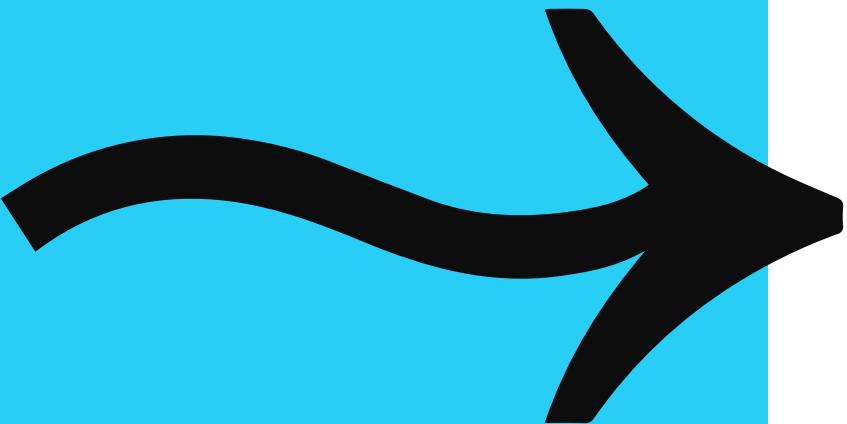
2. Then add alternative 'edge' scenarios:

Given my shopping basket contains the book "BDD is Fun"

When I add another copy of book "BDD is Fun" to the basket

Then the shopping basket contains 2 copies of "BDD is Fun"

Best practices



- Write in present tense (timeless and no ambiguity)
 - ✓ Given user is logged in, When the "Add" button is clicked Then...
 - ✗ Given a user had logged in When "Add" is being clicked, Then...
- Always use business language
- Express the clauses as readable sentences
- Keep to one feature per story, and keep to maximum of 12 scenarios per feature
- Use only the Gerkin words – "or" doesn't exist!
- Capitalise Gherkin keywords Given, When, Then... Capitalise titles too

Imperative vs declarative



Imperative Style

- Describes the exact steps needed to perform an action.
- Focuses on how something is done which leads to low-level, procedural test steps.

Given the user is on the login page

And the user enters "bob@qa.com" in the email text field

And the user enters "password123" in the password text field

And the user clicks the "Login" button

Then the user should see the "Add details" button

Declarative Style

- Describes behaviour at a higher level without specifying exact steps.

Given the user is logged in

Then the user should see the "Add details" button

Using backgrounds



Use Background to avoid repeating the same clauses continually.

- Keep your Background section vivid, short and relevant
- For a lengthy Background can use higher-level steps or splitting the feature file

Feature: Add items to shopping cart

Scenario: Add a book to cart
Given the user is logged in
And the shopping cart is empty
When the user adds a book
Then the cart contains 1 item

Scenario: Add a laptop to cart
Given the user is logged in
And the shopping cart is empty
When the user adds a laptop
Then the cart contains 1 item

Feature: Add items to shopping cart

Background:
Given the user is logged in
And the shopping cart is empty

Scenario: Add a book to cart
When the user adds a book
Then the cart contains 1 item

Scenario: Add a laptop to cart
When the user adds a laptop
Then the cart contains 1 item

Validating Scenarios



Ask these questions:

- ✓ What is the intent?
- ✓ Have we understood the expected behaviour?
- ✓ Is the expected behaviour clearly expressed?
- ✓ Can the rule be clearly understood when reading the scenario, and is there enough detail?
- ✓ How many rules are defined in the scenario?
Remember, it should ideally just be one!

An example... Define features

Feature: Calculator

![Calculator] (<https://specflow.org/wp-content/uploads/2020/09/calculator.png>)

Simple calculator for adding two**numbers**

Link to a feature: [Calculator](SpecFlowProject4/Features/Calculator.feature)

** **Further read * **: *[Learn more about how to generate Living Documentation] (<https://docs.specflow.org/projects/specflow-livingdoc/en/latest/LivingDocGenerator/Generating-Documentation.html>)***

@mytag

Scenario: Add two numbers

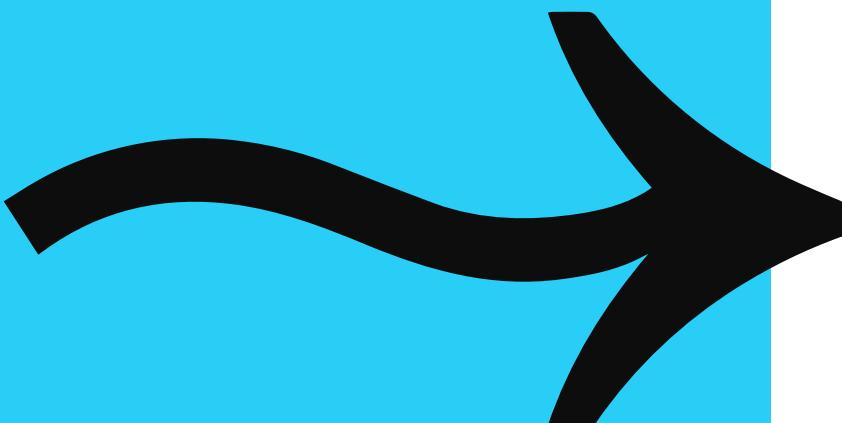
Given the first number is 50

And the second number is 70

When the two numbers are added

Then the result should be 120

Step definitions



```
@Given("the balance is (.*)")
public void setBalance(double bal)
{
    account = new Account(bal);
}

@When("the user withdraws (.*)")
public void withdrawAmount(double amount)
{
    account.Withdraw(amount);
}

@Then("the balance is (.*)")
public void testResult(double bal)
{
    Assert.assertEquals(bal, account.getBalance());
}
```

Then write the code to pass the tests

Multiple test values

Scenario: Add Lots of numbers

Given the first number is <n1>

And the second number is <n2>

When the two numbers are added

Then the result should be <n3>

Examples:

n1	n2	n3
1	2	3
11	12	23
21	22	43
31	32	63

Parameterised
tests

Scenario: Multiply two numbers

Given the first number is 5

And the second number is 6

When the two numbers are multiplied

Then the result should be 30

Add more
scenarios

An example... write the steps

```
using Microsoft.VisualStudio.TestTools.UnitTesting;
using TechTalk.SpecFlow;
[Binding]
public sealed class CalculatorStepDefinitions {
    private Calculator calculator;
    private readonly ScenarioContext _scenarioContext;

    public CalculatorStepDefinitions(ScenarioContext scenarioContext) {
        _scenarioContext = scenarioContext;
        calc = new Calculator();
    }

    [Given("the first number is (.*)")]
    public void GivenTheFirstNumberIs(int number) {
        calculator.firstNumber = number;
    }

    [Given("the second number is (.*)")]
    public void GivenTheSecondNumberIs(int number) {
        calculator.secondNumber = number;
    }
}
```

Can also use `_scenarioContext` to store values:
`_scenarioContext["result"] = calculator.Add();`

```
int result;
[When("the two numbers are added")]
public void WhenTheTwoNumbersAreAdded() {
    result = calculator.Add();
}

[When("the two numbers are multiplied")]
public void WhenTheTwoNumbersAreMultiplied() {
    result = calculator.Multiply();
}

[Then("the result should be (.*)")]
public void ThenTheResultShouldBe(int expectedResult) {
    Assert.AreEqual(expectedResult, result);
}
```

Automation



- Developers can use a **Cucumber** engine to run Gherkin. This engine may link to a framework to talk to browsers, simulators etc.
- Mobile developers use tools like **Appium**
- Web developers can use **Selenium** to hook into browsers. Step definitions run commands via Selenium Api.

Lab

Afterwards, we'll discuss what you thought...

