UNIVERSITÀ DEGLI STUDI DI NAPOLI
FEDERICO II

End-to-End Testing

***Luigi Libero Lucio Starace**, Ph.D.*
luigiliberolucio.starace@unina.it

May 25, 2023

Web Technologies

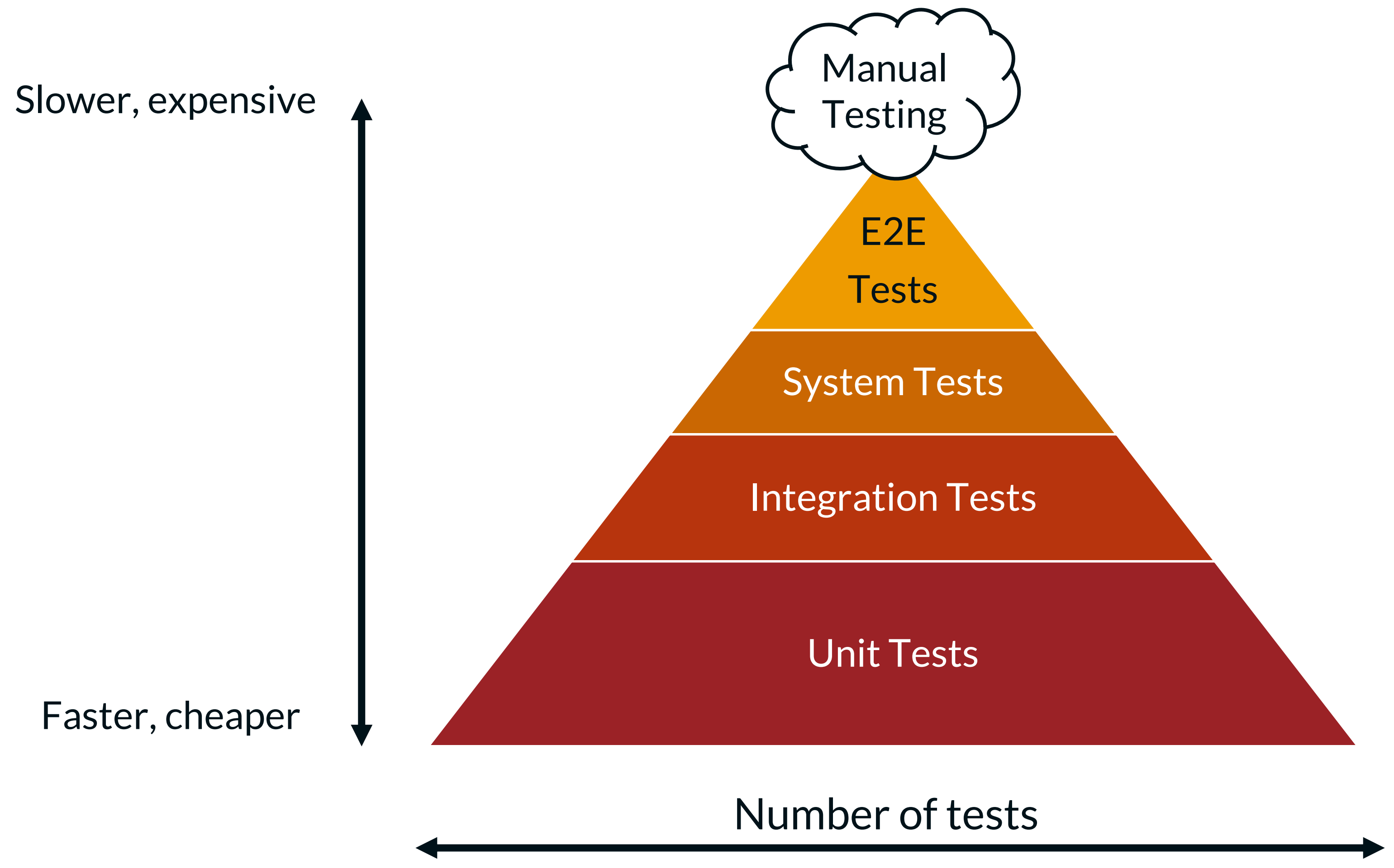
End-to-End (E2E) Testing

Goal:

- Test the system as a **whole**
- From the **point of view** of its intended **end users**

Focus is on **realistic usage flows**, starting from the **external interfaces** of the system

The Testing Pyramid



E2E Testing – Web Apps

- Web pages are the external interface
- End users are humans using a web browser
- E2E tests interact with the web pages, and check that they change correctly as a result of the interactions

GUI-level E2E Testing

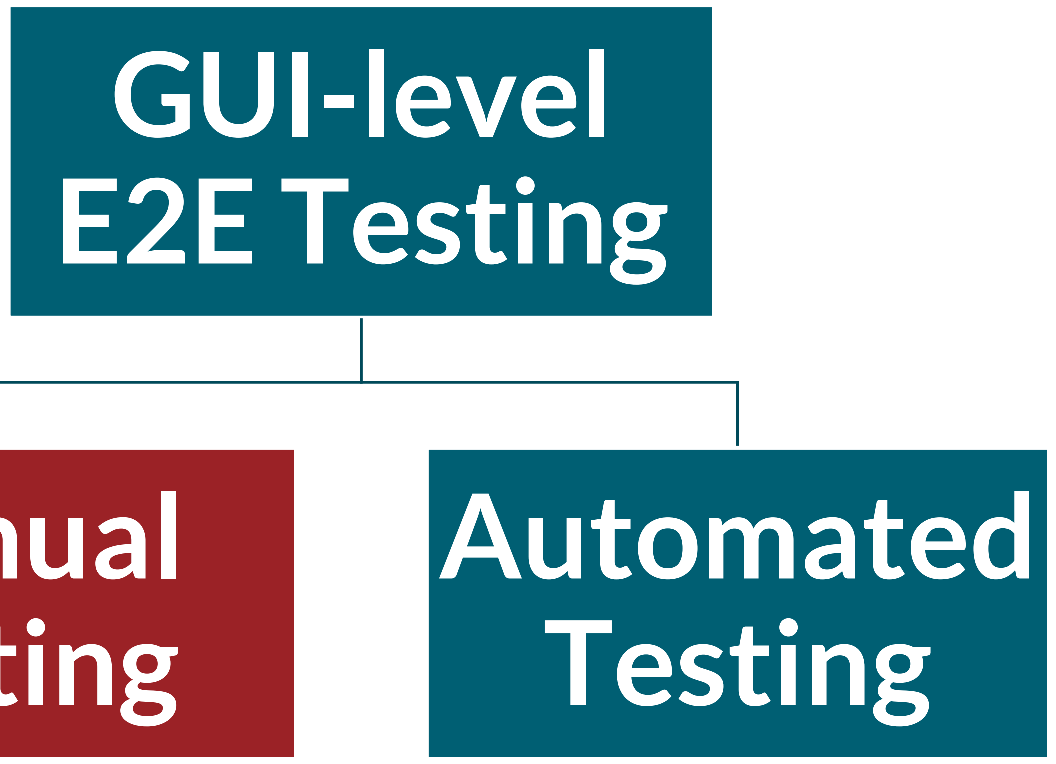
Key Steps

The GUI-level E2E testing loop for a usage scenario:

1. **Select** the GUI element to interact with (button, text field, …)
2. **Interact** with it (click, press, scroll, pinch-in, pinch-out, fill, etc…)
3. (Optionally) **Check** that resulting GUI state is correct
4. Return to 1, until the entire usage scenario is complete

GUI-level E2E Testing Approaches

- Two approaches exist: Manual Testing and Automated Testing



Manual GUI-level E2E Testing

- Humans **manually** interact with the GUI
- Typically follow **pre-defined test scripts**
- No particular skills required
- Tedious, time-consuming, error-prone activity
- Sometimes, **exploratory approaches** are used to define the interactions
    - Testers are not given a pre-defined script to follow, but are free to interact with the software according to their own sensibility
- Does not scale well

Automated GUI-level E2E Tests

- Test Software **automatically** interacts with the GUI
- Once tests are in place, they can be re-executed many times
- Tipically way **less expensive** than manual testing in the long run
- Some **skills required** to develop tests
- Test Software needs to be properly **maintained**

Automated GUI-level E2E Tests

Two factors are crucial in automated E2E GUI-level tests:

- How can Test Software determine which GUI elements it needs to interact with?
- How can we obtain executable GUI-level tests?

Automated GUI-level Testing: Selecting GUI Elements

How can Test Software select (**locate**) which elements to interact with?

Three main approaches exist:

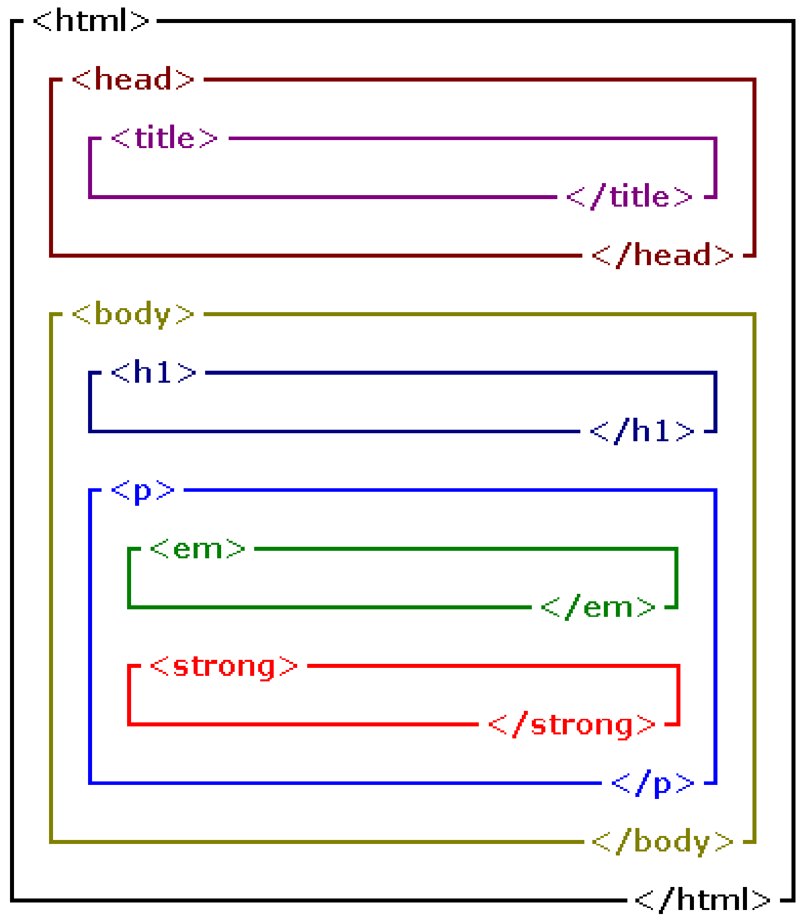
- Absolute screen coordinates
- Layout-based selectors
- Visual-based selectors

Automated GUI-level Testing: Locators based on screen coords

- Simplest approach
- Based on absolute on-screen coordinates
- E.g., click on coords (x,y), swipe from (x, y) to (x’, y’), etc…
- Generally, leads to **fragile** tests
    - Layout changes are likely to break the tests
    - Device changes (e.g.: using a bigger monitor) are likely to break the tests
    - Resulting code not very easy to understand

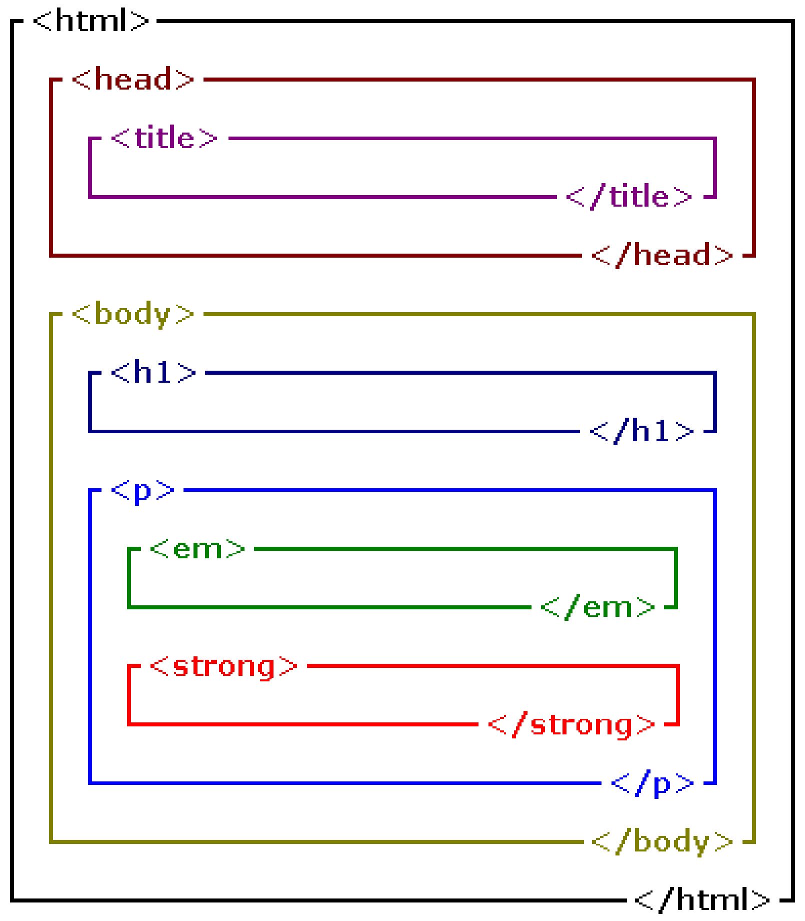
Automated GUI-level Testing: Layout-based Locators

- Most GUIs can be represented as structured documents
  - E.g.: web pages are HTML documents, Android GUIs are XML files
- Target elements are determined leveraging unambiguous layout properties (e.g.: ids) or query language expressions (e.g.: XPATH)



Automated GUI-level Testing: Layout-based Locators

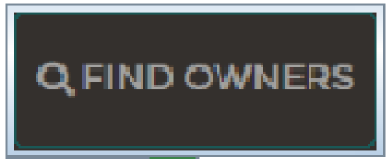
- Lead to more robust tests w.r.t. coordinate-based approaches
  - Tests do not depend on the device, but only on GUI structure
  - A lot depends on how the selectors are specified!

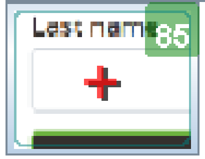


Automated GUI-level Testing: Visual-based Selectors

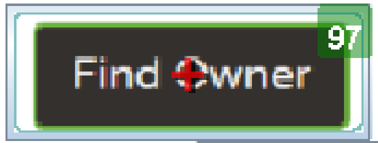
- Determines which element to interact with based on screen captures and image-matching algorithms
- Does not depend on the structure of the GUI, but is tightly coupled with the look and feel of the GUI
- Can be more robust than Layout-based selectors
- Example of visual-based automation tools include Sikuli, Eyeautomate Studio, etc…

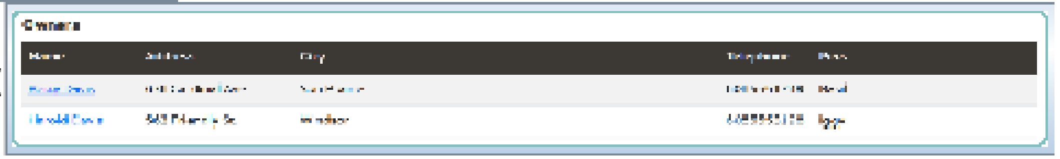
Visual-based Selectors: Example

`click(`  `)`

`click(`  `)`

`type("Davis")`

`click(`  `)`

`assert(exists(`  `))`

Automated GUI-level Testing: How to obtain tests?

Three main approaches exist to obtain GUI-level E2E Tests

- **Programmable Testing**
  - Test code is manually developed by programmers using dedicated frameworks that support GUI interaction
- **Capture & Replay (C&R)**
  - Specialized tools **record** (**capture**) real user interactions and automatically generate test code **replicating** (**replaying**) those interactions
- **Fully-automated Test Generation**
  - Test code is **automatically generated** by specialized tools

Programmable Tests

- Programmers **write test code** that **simulates** user interactions with the GUI
- Leverage **dedicated frameworks** that allow to simulate end-user interactions with a GUI
    - **Selenium** (for web testing)
    - **Espresso** (for Android testing)
- These frameworks provide ways to programmatically **select** GUI elements, **interact** with them, and perform **assertions**

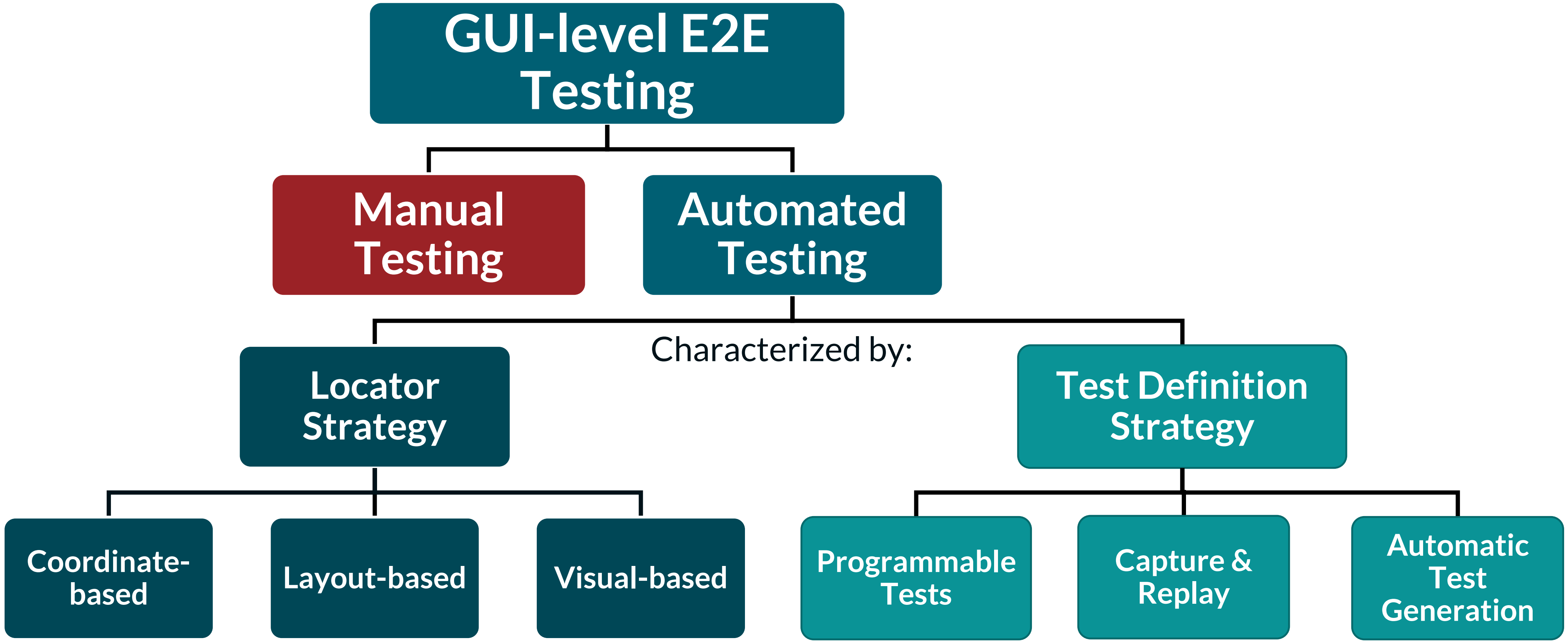
Capture and Replay (C&R)

- Programming skills not necessarily required (but may help!)
- A recording component keeps track of all user interactions
- A human user manually performs the use case **once**
- The recording component automatically generates executable tests (often as source code, using the same automation frameworks used for Programmable Tests)

Fully-automated Test Generation

- No human involvement necessary
- An automated tool systematically explores the application
- Automatically generates re-executable tests
- Hard to deal with the Oracle Problem ➔ Generic Assertions
  - I.e., no crashes on mobile apps, no 300/400/500 errors on web apps, …

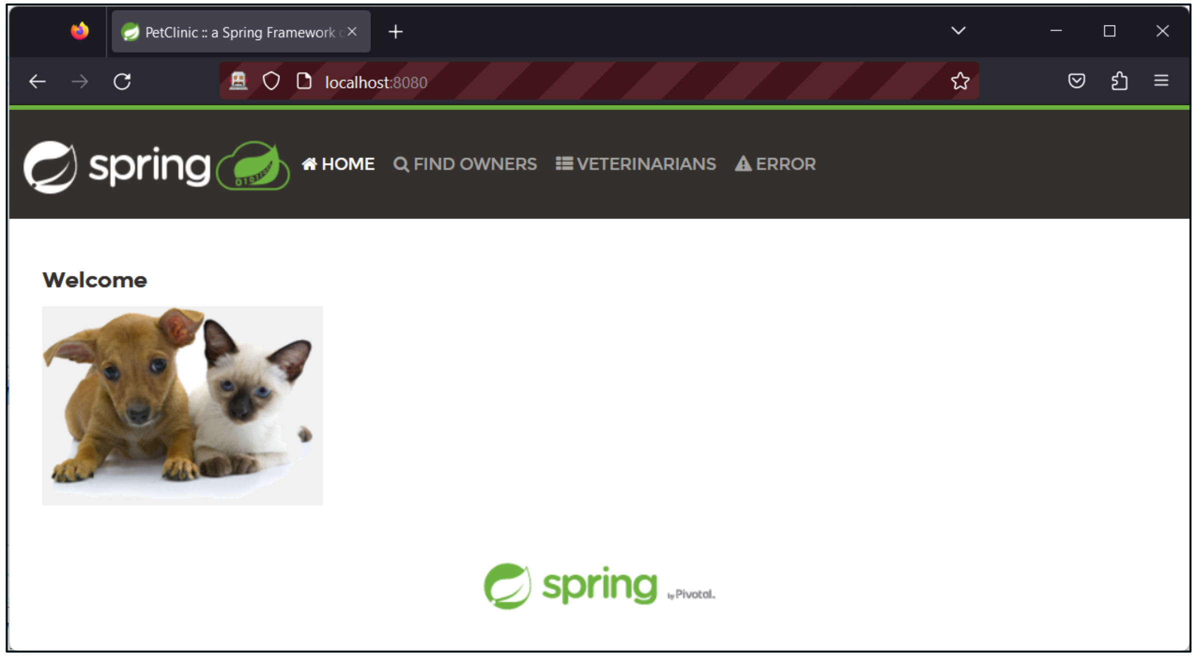
E2E GUI-level Testing: Overview



GUI-level E2E Testing in Practice

The PetClinic Spring Web App

- Open-source Web App
- Demo app for Spring Boot
- Will be our main test subject in the remainder of this session



Selenium

- Browser Automation Library: link
- Open-source
- Officially supports: Java, Python, C#, JS, Ruby
- Key component: **WebDriver**
- Supports all major browsers: Firefox, Chrome, Edge, Opera, Safari

Getting Started with Selenium

1. Install a Selenium library (examples in Maven/Java)

```
<dependency>
    <groupId>org.seleniumhq.selenium</groupId>
    <artifactId>selenium-java</artifactId>
    <version>4.8.0</version>
</dependency>
```

2. Install the required WebDrivers
   - Check out the official docs
   - Tools like WebDriverManager make the whole process painless

```
import io.github.bonigarcia.wdm.WebDriverManager;

WebDriverManager.edgedriver().setup();
```

Your First Selenium Web Test

- Selenium allows devs to programmatically interact with browsers
- A **Test Runner** is still needed (we'll use JUnit 5)
- Assertion Frameworks might be useful (we'll see some AssertJ)
- E2E GUI-level Tests aim at replicating realistic usage scenarios
- Keep in mind we're not doing unit testing
  - Common to have multiple **Act** phases and multiple **Assert** phases throughout the entire test

Your First Selenium Web Test

1. Create a WebDriver
   - Opens a Remotely-controlled Browser
   - The example uses MS Edge
2. Load the HomePage of PetClinic
3. Assert that the title contains «*PetClinic*»
4. Close the WebDriver

```java
import io.github.bonigarcia.wdm.WebDriverManager;
import org.junit.jupiter.api.Test;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.edge.EdgeDriver;

public class FirstTest {
    @Test
    void homePageTest(){
        WebDriverManager.edgedriver().setup();
        WebDriver driver = new EdgeDriver();
        driver.get("http://localhost:8080/");
        String pt = driver.getTitle();
        assertTrue(pt.contains("PetClinic"));
        driver.quit();
    }
}
```

Managing Setup and Teardown

- `@BeforeAll` tests we ensure that the proper browser driver is installed on our platform
- `@BeforeEach` test we create a new WebDriver instance and load the PetClinic homepage
- `@AfterEach` test, we properly quit the *marionette* browser

```java
public class PetClinicTests {
    WebDriver driver;
    @BeforeAll
    static void setupClass() {
        WebDriverManager.edgedriver().setup();
    }
    @BeforeEach
    void setupTest(){
        driver = new EdgeDriver();
        driver.get("http://localhost:8080/");
    }
    @AfterEach
    void teardownAfterTest(){
        driver.quit();
    }
    /* Actual Tests */
}
```

Selecting Elements

- Loading a web page and inspecting its title is cool
- … but we won’t go far without properly selecting elements!
- In Selenium, we can do this by using

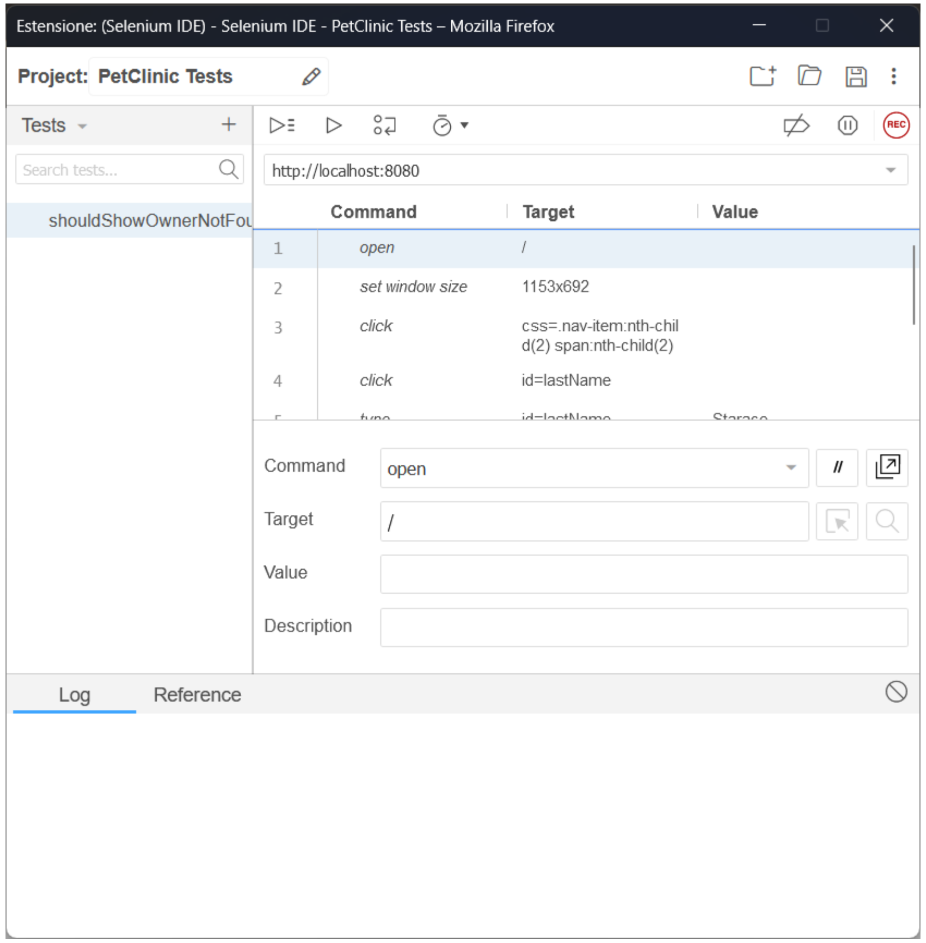
```
WebDriver driver = new EdgeDriver();
driver.get("http://mypage.org/");

WebElement element          = driver.findElement(selector);
List<WebElement> elements = driver.findElements(selector);
```

- Different **layout-based locators** are provided by the By class

Selenium IDE

- Open-source Capture & Replay E2E Web Test Automation
- Available here
- Based on Selenium
- Comes as a browser extension for Firefox/Chrome
- Tests can be directly replayed
- Tests can be exported as Selenium code (JUnit, Python, and more…)


Estensione: (Selenium IDE) - Selenium IDE - PetClinic Tests – Mozilla Firefox

Project: PetClinic Tests

Tests

Search tests...

shouldShowOwnerNotFou

http://localhost:8080

	Command	Target	Value
1	*open*	/	
2	*set window size*	1153x692	
3	*click*	css=.nav-item:nth-child(2) span:nth-child(2)	
4	*click*	id=lastName	

Command: open

Target: /

Value:

Description:

Log Reference

Challenges in GUI-Level Testing

- Flakyness
- Fragility

Dealing with Flakiness

- E2E web tests have a tendency to be **flaky**
- You might get different outcomes in multiple test executions
- Typically depends on external factors
  - Page loading was slower than usual and the test tried acting on an element before the page was fully loaded
- Solution:
  - Set adequate timeouts and wait for actionable elements to be loaded

Dealing with Fragility

- E2E GUI-level Tests have a tendency to be **fragile**
- Changes in the GUI are likely to break tests
- Using **good** locators is the most effective approach
    - Locators that are too dependant on the overall layout are more likely to break upon changes (e.g.: /html/body/div[2]/div/div/table/tr[3]/td[2]/a)
    - Locators that are too generic also are likely to break (i.e., to select a different element than the intended one) upon changes (e.g.: //a[3])

Guidelines for designing Locators

- If available and meaningful, alway prefer semantic properties (e.g.: name of an input field, id of a button, classes)
  - These are somewhat less likely to change over time
- Avoid dynamically generated attributes (e.g.: id="btn-348756")
- The way the software under test has been developed has a great impact on E2E testability
  - Adding semantic ids, names, and/or specific attributes to help locating elements goes a long way!

Dealing with E2E Test Evolution

- Still, despite our best efforts on proper locator design, E2E GUI-level test code will still need maintenance
- How maintainable is the test code we've written so far?
- Let's take another look:
  - Lots of repeated code (should a button change its id, we'll need to change all the tests interacting with that button)
  - Dependancy between test code and GUI layout
  - Tests are not very readable
- It's a good practice to separate test logic and test implementation details. How? Using **Page Objects**

The Page Object Design Pattern

- Break the dependancy between test logic (what do we need to do) and test implementation (how do we do that in a browser).
- Can be used with any UI technology
- Great example of **encapsulation**: hide unnecessary details from other parts of the software
- *If you have WebDriver API code in your tests, you're doing it wrong!*

The Page Object Design Pattern

```java
@Test
void searchOwnersAndNoResultFound(){
    WebElement fOwnLink = driver.findElement(By.partialLinkText("FIND OWNERS"));
    fOwnLink.click();
    WebElement findOwnersTextInput = driver.findElement(By.name("lastName"));
    findOwnersTextInput.sendKeys("Starace");

    driver.findElement(By.cssSelector("button.btn.btn-primary")).click();
    WebElement message = driver.findElement(By.cssSelector("span.help-inline"));

    assertThat(message.getText()).containsIgnoringCase("has not been found");
}
```

- This API is about HTML!

The Page Object Design Pattern

```
@Test
void searchOwnersAndNoResultFound(){
    new Navbar(driver).navigateToFindOwner();
    FindOwnersPage f = new FindOwnersPage(driver);
    f.searchForOwnerName("Starace");
    assertThat(f.getHelpMessage()).containsIgnoringCase("has not been found");
}
```

- This one is about the Application!