Automation with Playwright

PRACTICAL INFORMATION

COURSE CONTENTS – DAY 1

01

Basics – How to get started

Includes installation, configuration and everything you need to get started.

02

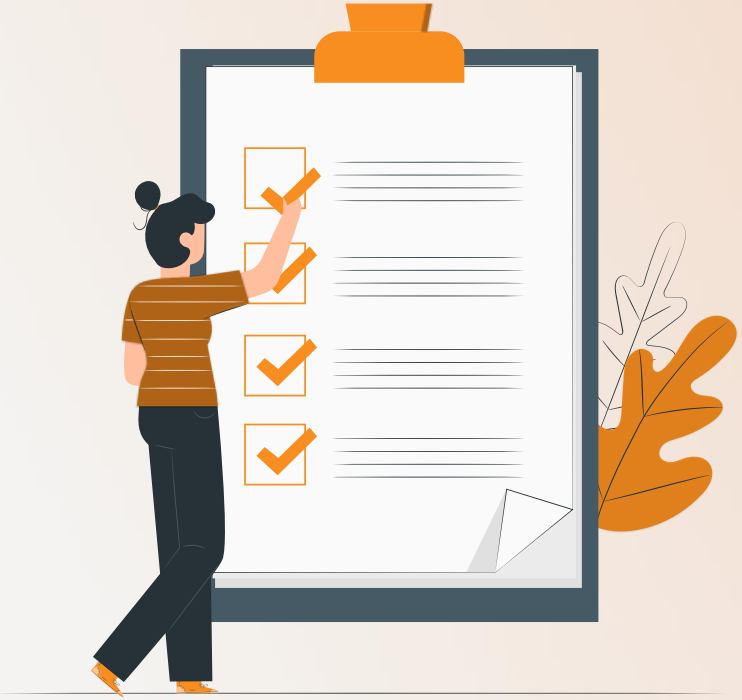
Navigation and Interaction

Locate and click elements. How to use forms and text while controlling elements.

03

API Tests

Introduction to using playwright to perform API tests and learning the fundamentals of APIs.



COURSE CONTENTS – DAY 2

04 Automation in General

05 Automation of Manual Tests

06 Structuring

07 Maintainability

08 Test-first Methodologies

QUESTIONS & EVALUATION

- Questions are welcome throughout the course
- Evaluation forms

PARTICIPANT PRESENTATIONS

- Your name?
- Experience with software testing?
- Expectations of the course?

1. BASICS

BASICS – HOW TO GET STARTED

- Learn how to install Playwright
- Learn about basic configuration to get started

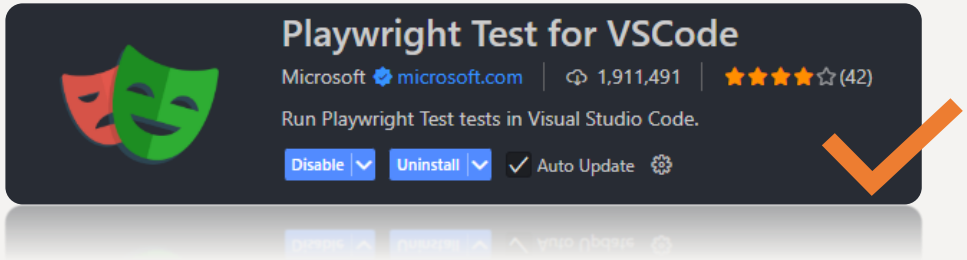
COURSE REQUIREMENTS

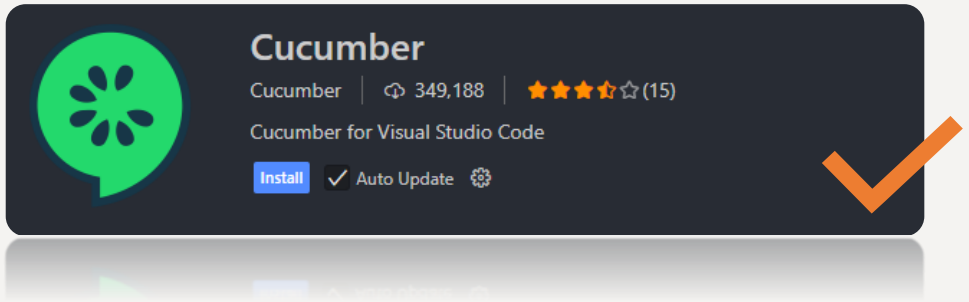
- What do we need so we can get started?
  - Selecting a testing framework
  - Prepare our IDE (Integrated Development Environment)
  - Prepare NodeJS project
  - Install and configure Playwright

SETTING UP OUR IDE

01 Required extensions

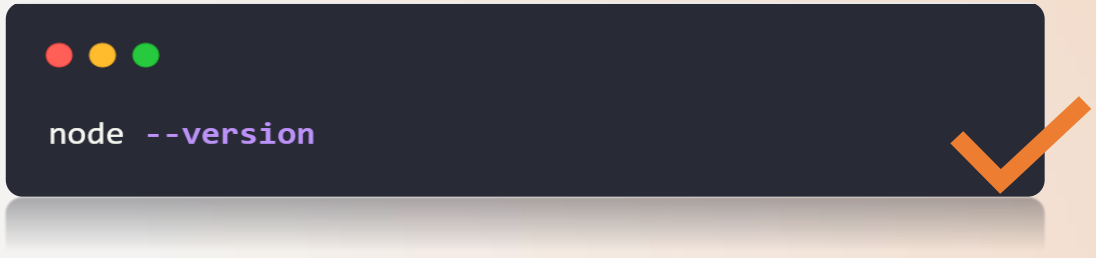
The below extensions will be used throughout the course, as they provide quality of life additions.

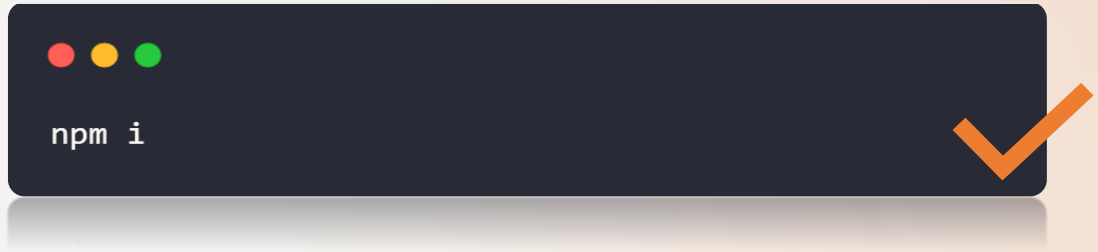




02 Setting up NPM for the project

The exercises throughout the course is running on NodeJS.





INSTALLING PLAYWRIGHT

- Install dependencies
  - Playwright: the browser automation library
  - TypeScript + @types/node: for TypeScript support
  - *npx playwright install*: downloads browser binaries (Chromium, Firefox, Webkit)

```
npm install playwright typescript @types/node --save-dev
```

```
npx playwright install
```

BROWSER CONTEXT AND PAGES

A **Browser Context** is Playwrights clever answer for isolated browser instances.

- Isolated, temporary browser profile
- With its own cookies
- Its own local / session storage
- Its own cache
- Its own permissions (camera, geo location)
- Its own logic

Browser Context ensures the following:

- ✓ Reproducibility: repeated runs always start "fresh".
- ✓ Zero leakage: scripts never contaminate your real browser profile.
- ✓ Parallel testing: you can have 10 contexts each pretending to be a different user.
- ✓ Cleaner automation: you control the environment fully (locale, timezones, device)

```
const browser = await playwright.chromium.launch();
const context = await browser.newContext();
```

BROWSER CONTEXT AND PAGES

- A **Page Object** is where all the action happens.
- Each Browser Context can have multiple pages.
- A Page refers to a single tab within the Browser Context

Things you can do with a Page:

- Navigate to URLs: `page.goto('URL')`
- Fill fields: `page.fill()`
- Click buttons: `page.click()`
- Take screenshots: `page.screenshot()`
- Extract data: `page.textContent()`
- Wait for elements: `page.waitForSelector()`

```
const page = await context.newPage();
```

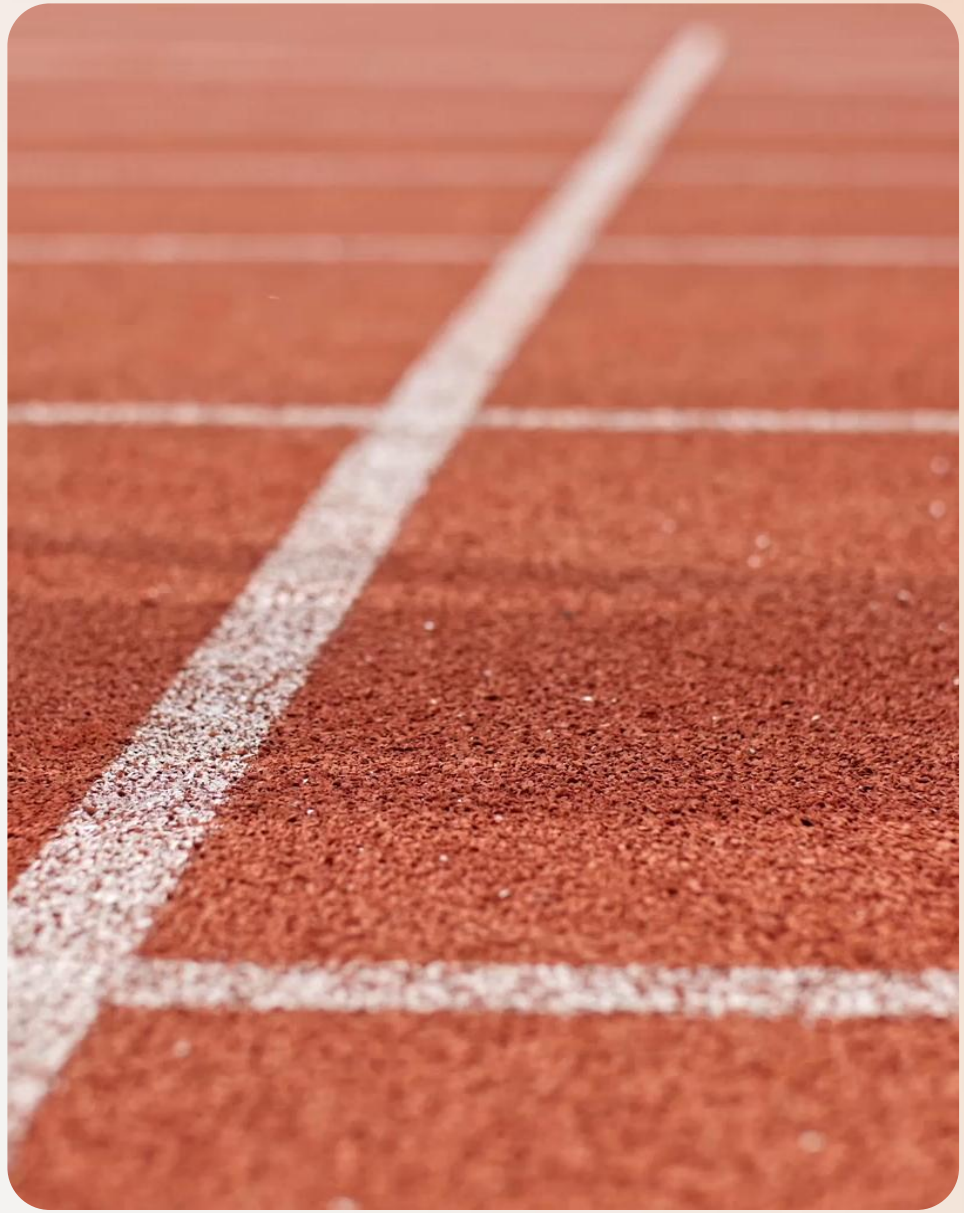
EXAMPLE

```
const page = await context.newPage();


await page.goto('http://example.com');

await page.locator('#search').fill('query');


await page.locator('#submit').click();

console.log(page.url());
```

EXERCISE 01 (15 min.)

- **Goals**:
  - Run your first Playwright script
  - Launch a browser
  - Open a page
  - Take a screenshot

- **Locate** 'INSTRUCTIONS.MD'



SUMMARY FOR BROWSER CONTEXT + PAGE

Concept	What it does	Analogy	Why it matters
Browser	The real browser engine	Apartment Building	Runs playwright automation
Context	An isolated browser session	An apartment within in the building	Keeps tests and scripts independent
Page	A browser tab inside the context	Individual rooms within the apartment	Where you interact with the elements

FIXTURES IN PLAYWRIGHT

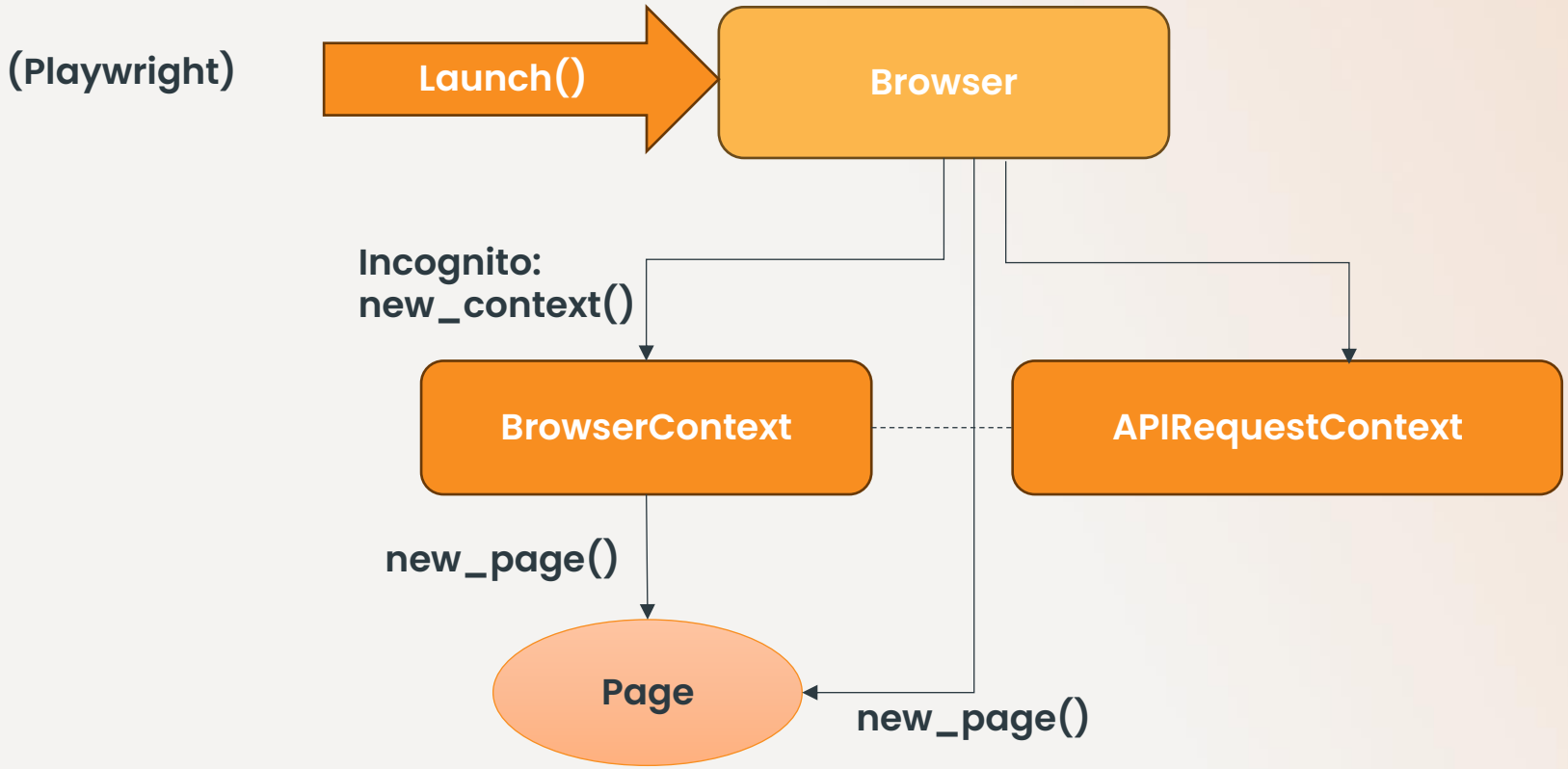
Fixtures
A reuable object that Playwright create for you, injects into your test, and cleans up automatically.

- Built-in fixtures
  - `page` – a fresh Page (Tab) in a new context
  - `context` – a fresh BrowserContext
  - `browser` – a Browser instance
  - `request` – an APIRequestContext

TS fixtures_example.ts

```ts
// Individual fixtures
const browser = await pw.chromium.launch();
const context = await browser.newContext();
const page = await context.newPage();

// @playwright/test
test('my test', async ({ page }) => {
  // no need to create browser/context/page manually
  await page.goto('https://example.com');
});
```

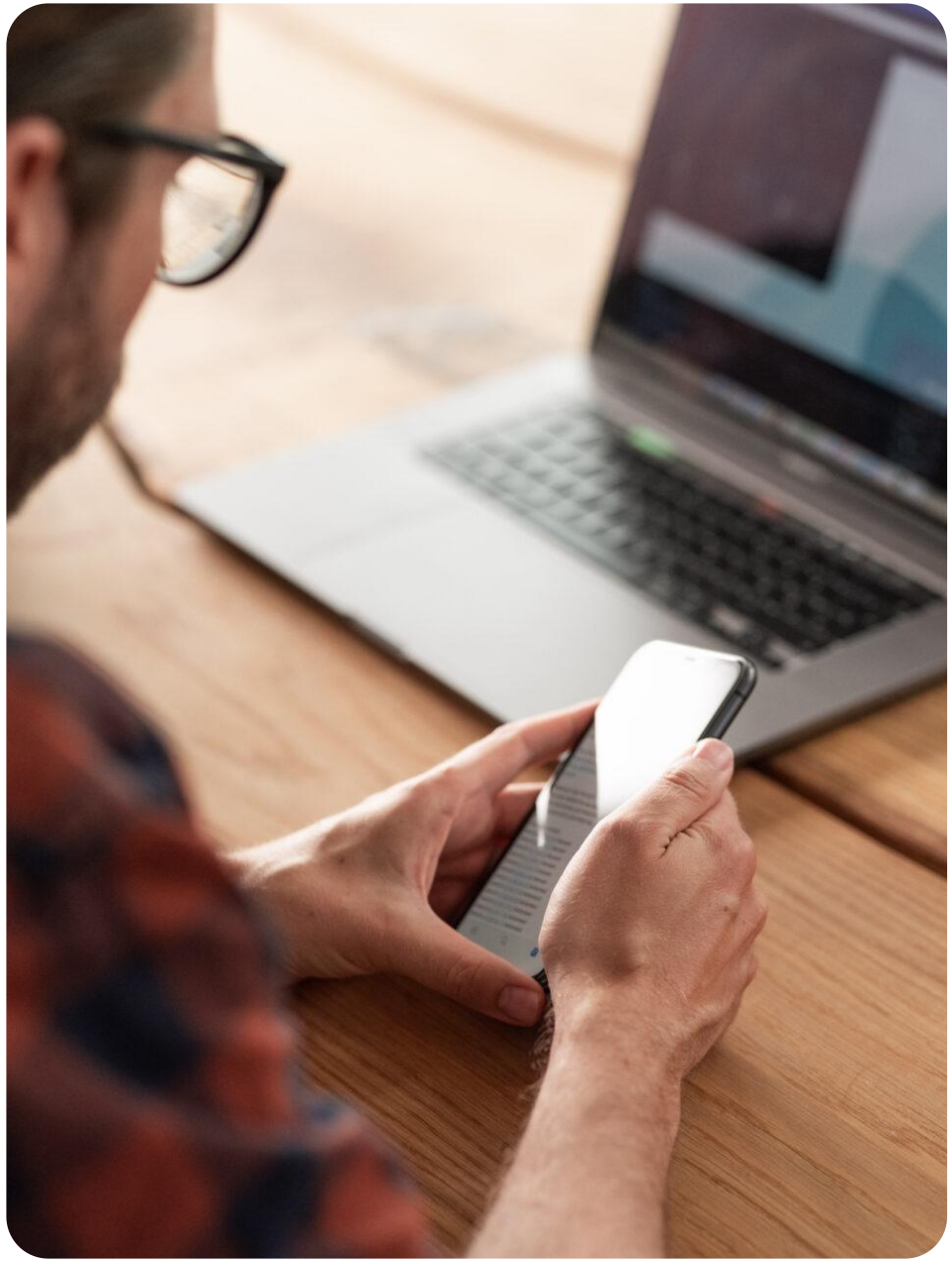
FIXTURES IN PLAYWRIGHT



2. Navigation and Interaction

NAVIGATION AND INTERACTION

- Get familiar with Selectors and Locators
- Understand how to use assertions in Playwright
- Learn about various actions such as Clicks and how to interact and control elements.



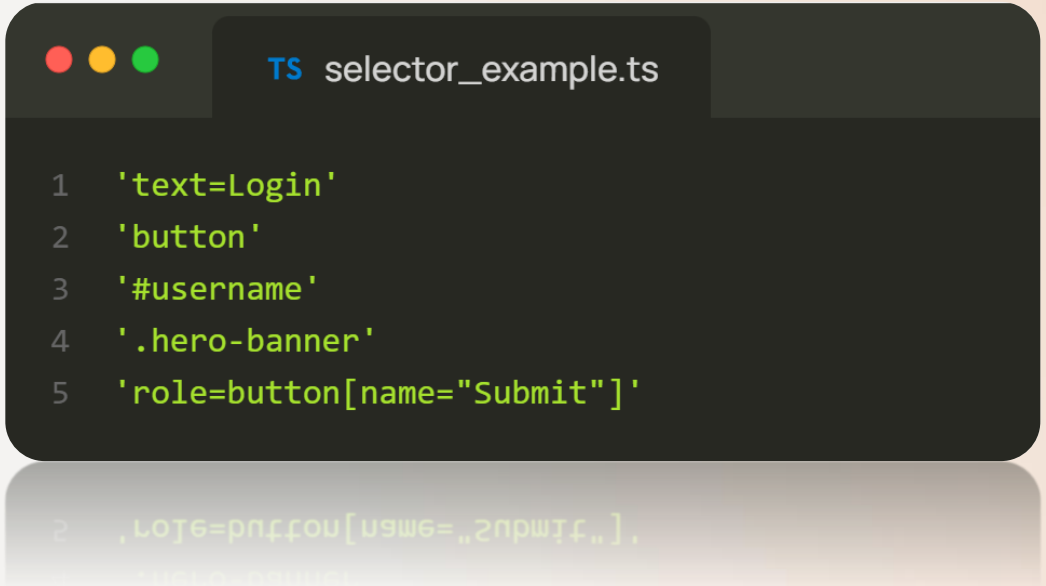
2. Navigation and Interaction

SELECTORS & LOCATORS

- Learn how to interact with text fields and inputs
- Learn how to fill out form elements

WHAT ARE SELECTORS IN PLAYWRIGHT

- A *selector* is a string that identifies an element on the page, such as:
  - CSS selector
  - Text locator
  - Role locator
  - XPath (not recommended, but available)



Selectors
A selector is just a description of how to find something

TYPES OF SELECTORS

- **CSS Selectors**
  - The most common, powerful and fast
- **Text locators**
- **Role locators** (Recommended for accessibility)
  - Robust, readable, reflects actual user-facing elements
- **Test IDs** (Best for stable tests)
- **Xpath**
  - Available, but generally discouraged

Playwright Recommendation
Playwright recommends that when using a selector, you should use something visible, like a placeholder, testid etc.

WHY SELECTORS ALONE ARE NOT ENOUGH

- Selectors return zero or many elements
- They don't provide the interaction by themselves

- This tells Playwright *what,* but not *how* to interact with it
- This is where **Locators** come in

WHAT IS A LOCATOR?

— A **Locator** is a "smart handle" to an element

— **Locators**:

- ✓ Auto-wait for elements
- ✓ Retry on failure
- ✓ Track DOM changes
- ✓ Are lazy (they evaluate when used)

1. You create a locator from a selector:

TS locator_example.ts

```ts
const button = page.locator('#submit');
```

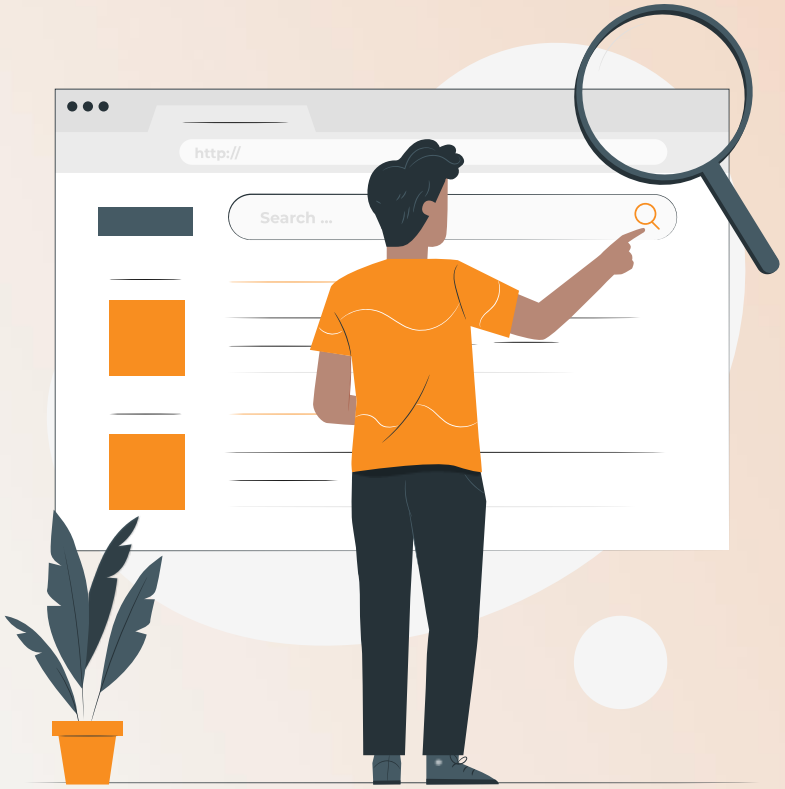
2. Then you can interact with it

TS locator_example.ts

```ts
await button.click();
await expect(button).toBeVisible();
await button.fill('hello');
```

LOCATOR EXAMPLES

- page.getByRole() – Find elements by **ARIA role** (buttons, links, inputs, etc.)
- page.getByText() – Find a non-interactive element by its **visible text**
- page.getByLabel() – Find a form field by its **associated <label>**
- page.getByPlaceholder() – Find a form field by its **placeholder text**
- page.getByAltText() – Find an element (e.g., <img>, <area>) by its **alt text**
- page.getByTitle() – Find an element by its **title attribute**
- page.getByTestId() – Find an element by **test id attribute** (e.g., data-testid)
- page.locator() – Find elements using **CSS selector** or **XPath**



RECOMMENDED LOCATOR STRATEGIES

👍 Best Practice

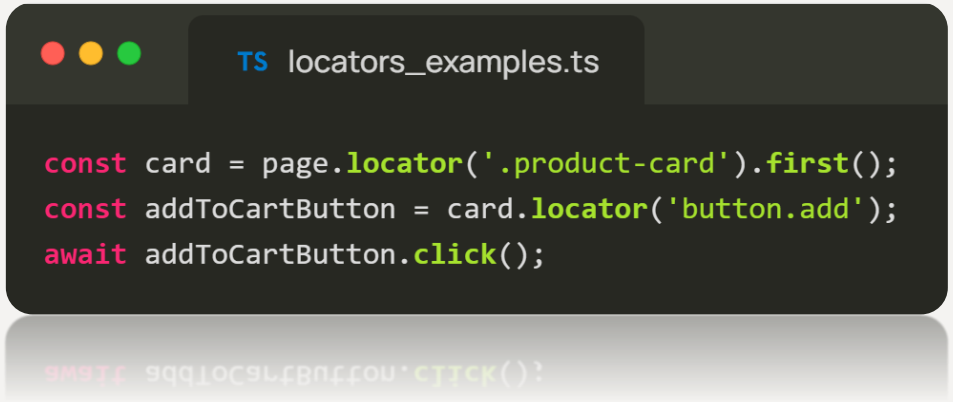
1. Role selectors
   - Most stable +accessible
2. Test IDs
   - When you control the application
3. CSS selectors
   - Clean and simple

👎 Avoid when possible

- XPath
- Deep nested selectors (`div > div > div > ul > li a`)

CHAINING LOCATORS

- `page.locator('PARENT').locator('CHILD')`
- **Why chain locators?:**
  - Chaining lets you narrow down elements by selecting children of already-located elements.
- **How to chain locators?:**



- **Benefits**:
  - ✓ More readable than complex CSS
  - ✓ More resilient to UI changes
  - ✓ Allows composing element-specific logic

ACTION: CLICK()

- An **Action** that simulates a human clicking the mouse
- **Actions** are methods of the Locator object

- **Parameters**
  - button - "left", "middle", "right", None
  - modifiers - ["Shift", "Control", "Alt", "Meta"]
  - force – True, False

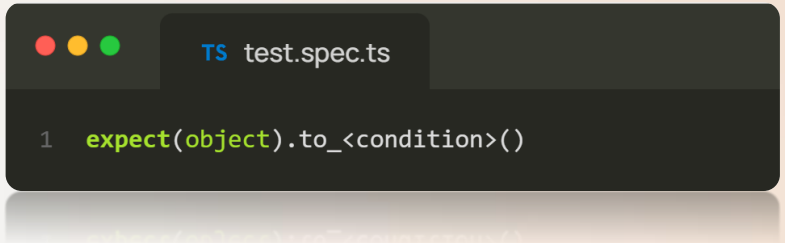
CLICK – EXAMPLES

TS actions_examples.ts

```ts
page.locator('button').click()
```

HOW DO WE VERIFY A STATE IN PLAYWRIGHT?

- We use **assertions** through `expect`
- Assertions check that the application is in the **expected state**
- If the expectation is not met -> Playwright retries until timeout -> then fails the test
- **Why assertions matter:**
  - ✓ Validate UI behavior
  - ✓ Catch regressions
  - ✓ Ensure the page actually did what we expected
  - ✓ Make tests meaningful



- Where **object** can be:
  - Locator: An element
  - Page: Full page state
  - APIResponse: result of a request
- **Common conditions**
  - toBeVisible()
  - toHaveText()
  - toHaveURL()
  - toHaveAttribute()

Assertions
Playwright assertions are **auto-waiting**, meaning:
They wait for the conditions to be true before failing

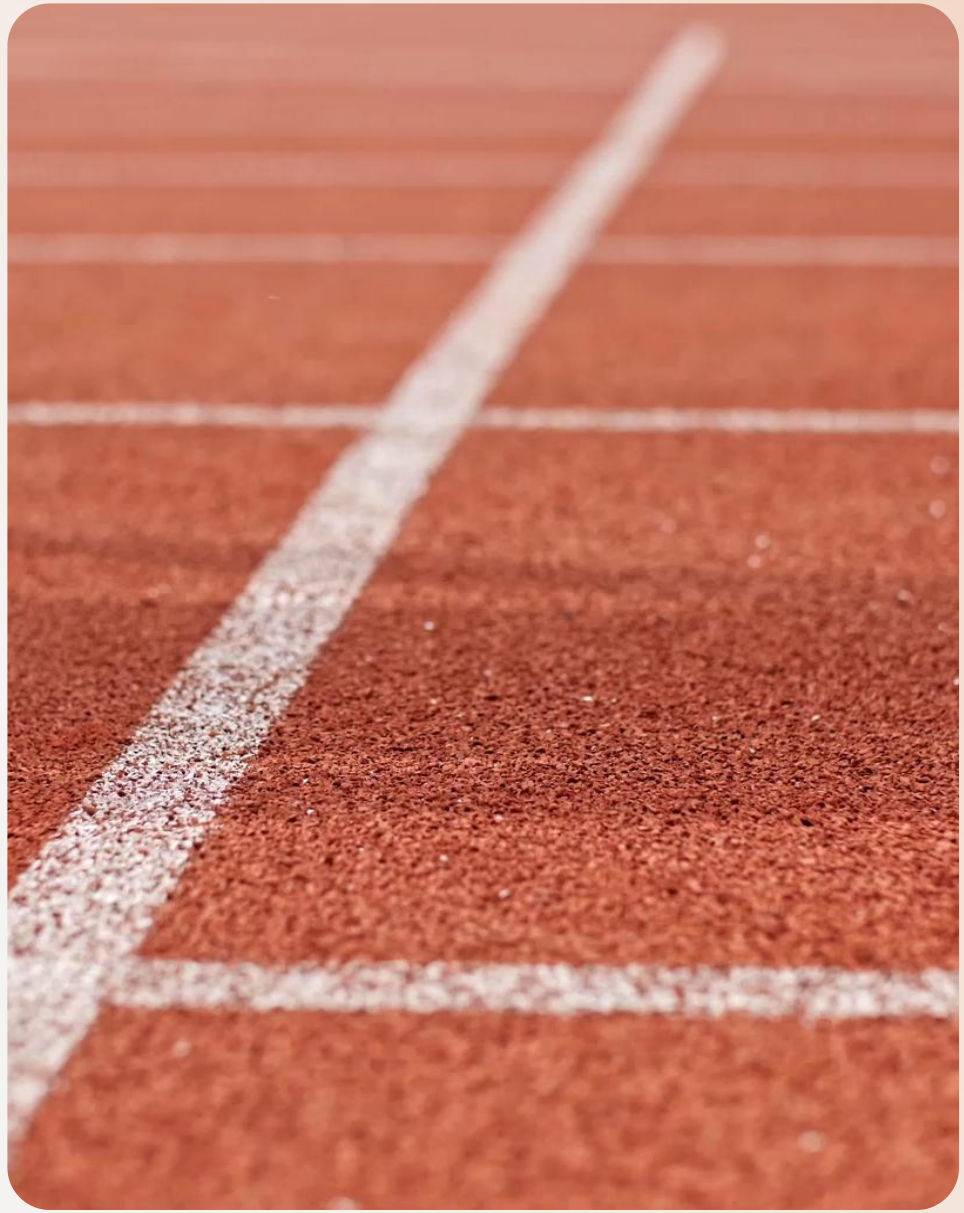
EXAMPLES OF ASSERTIONS

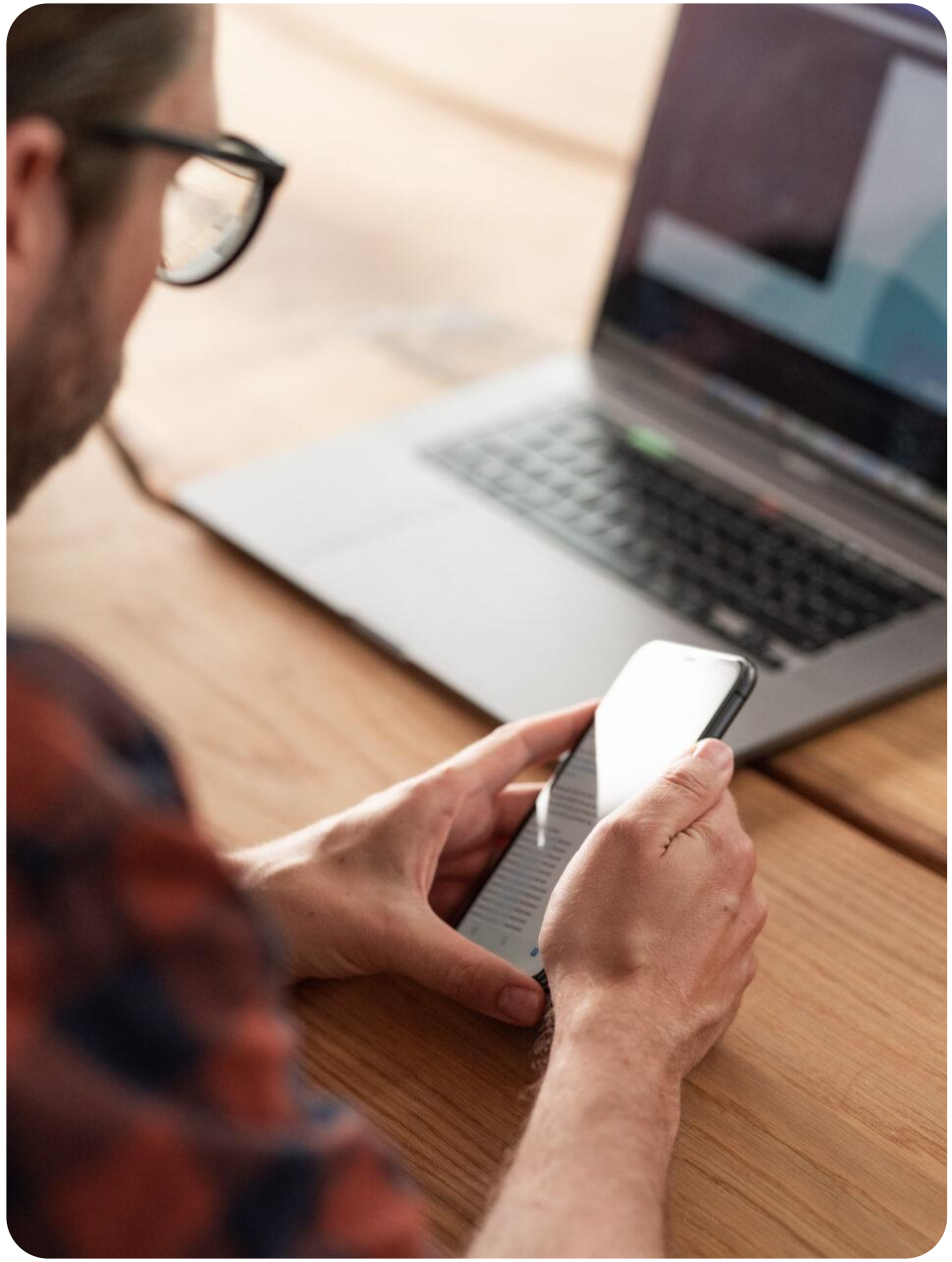
- Notes
  - Assertions almost always take a **Locator**, ApiResponse, Page or Snapshot.
  - Expect(page) is useful for verifying navigation
  - Failures show meaningful diff output in the CLI
  - Built-in retries remove the need for waits

EXERCISE 02 (20 min.)

- **Goals:**
  - Navigate around the site until you find the correct objective
  - Locate the course and click various items
  - Assert values on page

- **Locate** 'INSTRUCTIONS.MD'





2. Navigation and Interaction

TEXT AND FORMS

- Learn how to interact with text fields and inputs
- Learn how to fill out form elements

ACTION: FILL()

- `locator.fill(value)`
- `fill()` replaces the existing value of an input field

- **When to use?**
  - ✓ Fast and deterministic input
  - ✓ Most form fields
  - ✓ API-like direct text replacement

- **Characteristics:**
  - Clears the field automatically
  - Types the value in instantly (not character-by-character)
  - Ensures the element is ready (visible + enabled) before filling
  - Works for inputs and text areas

ACTION: PRESS()

- `locator.press(key)`
  - *Simulates pressing a single key or a combo*
- **Common use cases:**
  - Submitting forms with `Enter`
  - Keyboard shortcuts
  - Navigation keys (ArrowUp, ArrowDown)
  - Selecting all, copying, pasting

TS actions_examples.ts

```ts
await page.locator('#search').press('Enter');
await page.locator('#input').press('Control+A');
```

- **Notes**
  - Uses keyboard events exactly like a real user
  - Does *not* insert text (use `fill()` or ~~`type()`~~ for that)

ACTION: PRESSSEQUENTIALLY()

- `locator.pressSequentially(text)`
  - Types text **one character** at a time, including keydown/keyup for each character

- **Why use this?**
  - Simulates realistic typing
  - Useful for fields with:
    - Autocomplete
    - Input masking
    - Type-ahead search
    - JavaScript validation on each keystroke

ACTION: PRESSSEQUENTIALLY()

— Differences vs `fill()`

fill()	pressSequentially()
Instant insertion	Types character-by-character
No typing events	Emits full keyboard events
Good for stable API-like filling	Good for UI that reacts to typing

ACTION: CLEAR()

- `locator.clear()`
  - Clears the input field **without typing**.

TS actions_examples.ts

```ts
await page.locator('#search').clear();
```

- **When to use?:**
  - When you want to remove the text without replacing it
  - For fields where typing or replacing may trigger undesired events.
  - For textareas or inputs with reactive formatting

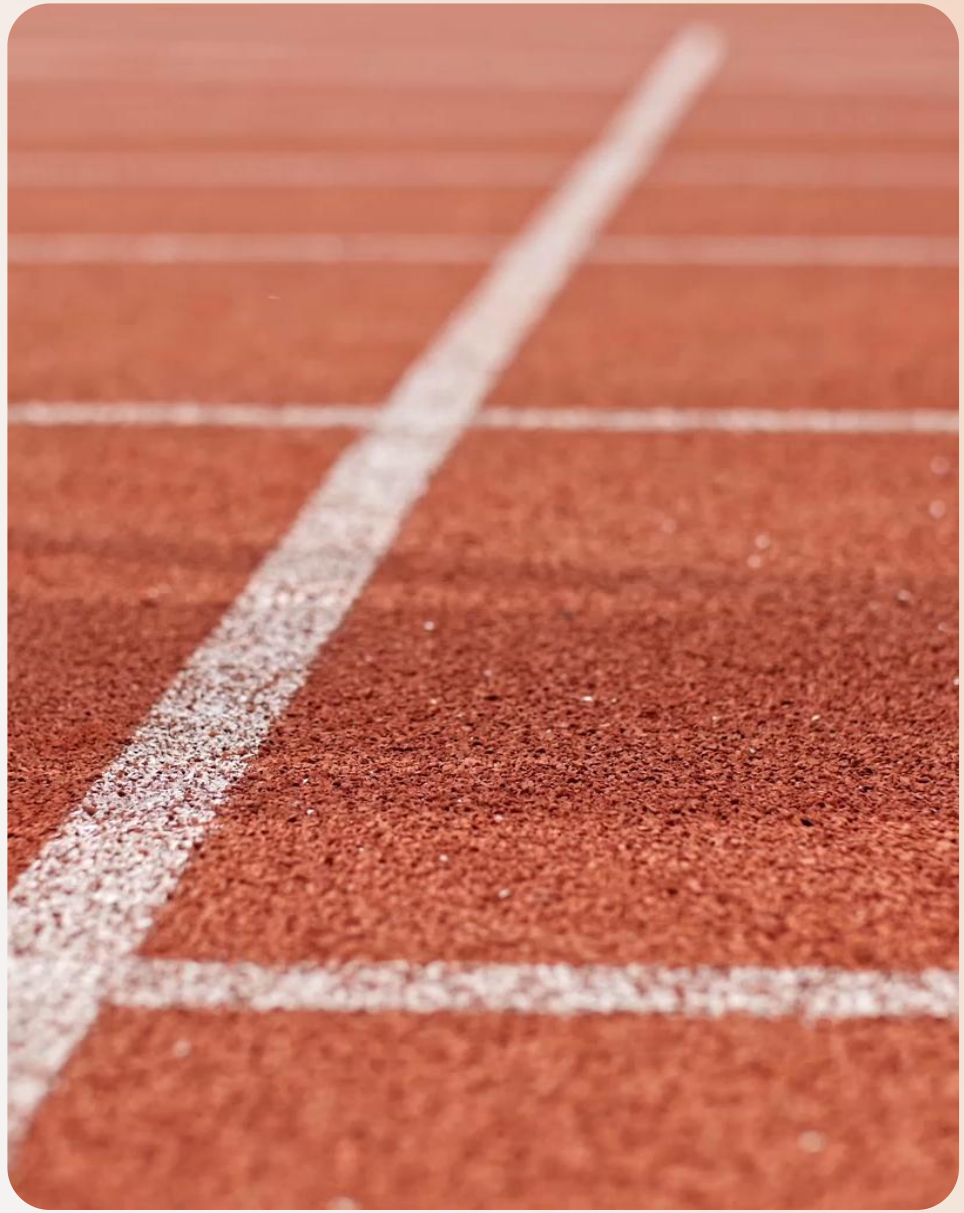
Notes

Some UI frameworks override default behavior; In those cases `fill('')` or `press('Control+A')` + `press('Backspace')` may be needed. But for most forms, `clear()` works as expected.

EXERCISE 03 (35 min.)

- **Goals:**
  - Get a better understanding of the @playwright/test module
  - Create a test that locates input fields and fills them.
  - Use Playwright to locate a button and click on it
  - Use assertions (expects) that the outputs shows the correct values

- **Locate** 'INSTRUCTIONS.MD'



ACTION: HOVER()

- `locator('SELECTOR').hover()`
- **Hovering is essential for UI interactions such as:**
  - Dropdown menus
  - Tooltips
  - Hidden elements that appear on hover
  - Animated hover effects
- **Usage:**

TS actions_examples.ts

```
await page.locator('.menu-item').hover();
```

- **What Playwright does automatically**
  - ✓ Waits for the element to visible
  - ✓ Moves the mouse to the element's center
  - ✓ Triggers browser hover events (mouseover, mouseenter)
  - ✓ Works with animated and dynamic UIs
- **When to use?:**
  - Opening a hidden menu
  - Triggering tooltips
  - Revealing hover-based buttons (e.g., delete icons in lists

ACTION: FILTER()

- `locator('SELECTOR').filter({options})`
- Filters refine locators by adding *constraints*

- **Common usage:**

```ts
page.locator('button').filter({ hasText: 'Remove' });
```

- **Filter options:**
  - hasText: matches text content
  - has: contains another locator
  - nth: select item by index
  - locator.first() / .last()

- **Why we use filters:**
  - ✓ Avoid fragile XPath
  - ✓ Express intent clearly
  - ✓ Precise targeting in dynamic lists

```ts
// button with text
await page.locator('button').filter({ hasText: 'Login' }).click();

// list item containing an image
await page.locator('li').filter({ has: page.locator('img') }).first();
```

LOGGING IN PLAYWRIGHT

— **How can logging help us?**

- Track test execution
- Debug failures
- Record steps or data
- Monitor long test runs

— **Benefits**:

- ✓ Cleaner logs
- ✓ Central logging configuration
- ✓ Consistent output across tests
- ✓ Can easily add file logging later

TS logger.ts

```ts
import { createLogger, transports, format } from 'winston';

export const logger = createLogger({
    level: 'info',
    format: format.combine(
        format.simple()
    ),
    transports: [
        new transports.Console(),
    ],
});
```

LOGGING IN PLAYWRIGHT

— **Improve your logging with severity channels**

- Debug
- Info
- Warning
- Error
- Critical

TS logger.ts

```ts
import { createLogger, transports, format } from 'winston';

const { combine, printf, timestamp } = format;

const timestampFormat = timestamp({
    format: () => {
        const date = new Date();
        const pad = (n: number) => n.toString().padStart(2, '0');

        const day = pad(date.getDate());
        const month = pad(date.getMonth() + 1);
        const year = date.getFullYear();
        const hours = pad(date.getHours());
        const minutes = pad(date.getMinutes());

        return `${day}-${month}-${year} ${hours}:${minutes}`;
    },
});

export const logger = createLogger({
    level: 'info',
    format: combine(
        timestampFormat,
        printf(({ timestamp, level, message }) => {
            return `[${timestamp}] ${level.toUpperCase()}: ${message}`;
        })
    ),
    transports: [new transports.Console()],
});
```

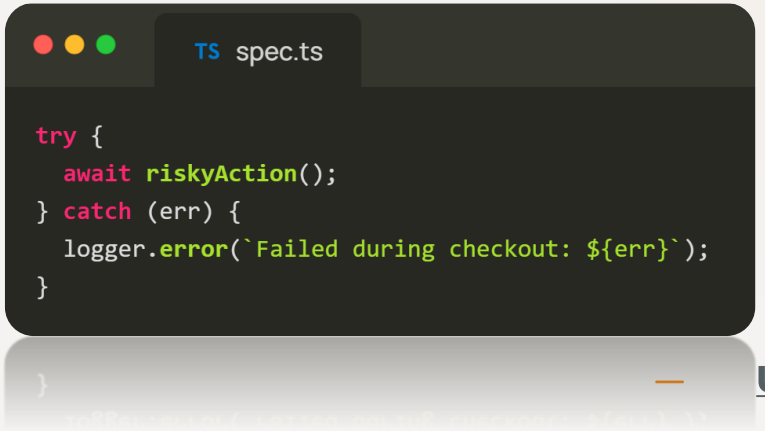
HOW TO USE LOGGING?

TS spec.ts

```ts
import { logger } from '../utils/logger';

test('example with logging', async ({ page }) => {
  logger.info('Navigating to homepage');
  await page.goto('https://example.com');

  logger.info('Hovering over menu');
  await page.locator('.menu').hover();

  logger.info('Test finished');
});
```

LOGGING BEST PRACTICES

- **Use consistent log level**
  - **error** – something fails and needs immediate attention
  - **warn** - Something unexpected happened; not fatal
  - **info** - High-level operational messages
  - **debug** - Detailed diagnostic information
- **Include timestamps**
  - E.g., 'DD-MM-YYYY HH:mm:ss'
- Make logs structured and readable
- Avoid logging sensitive data

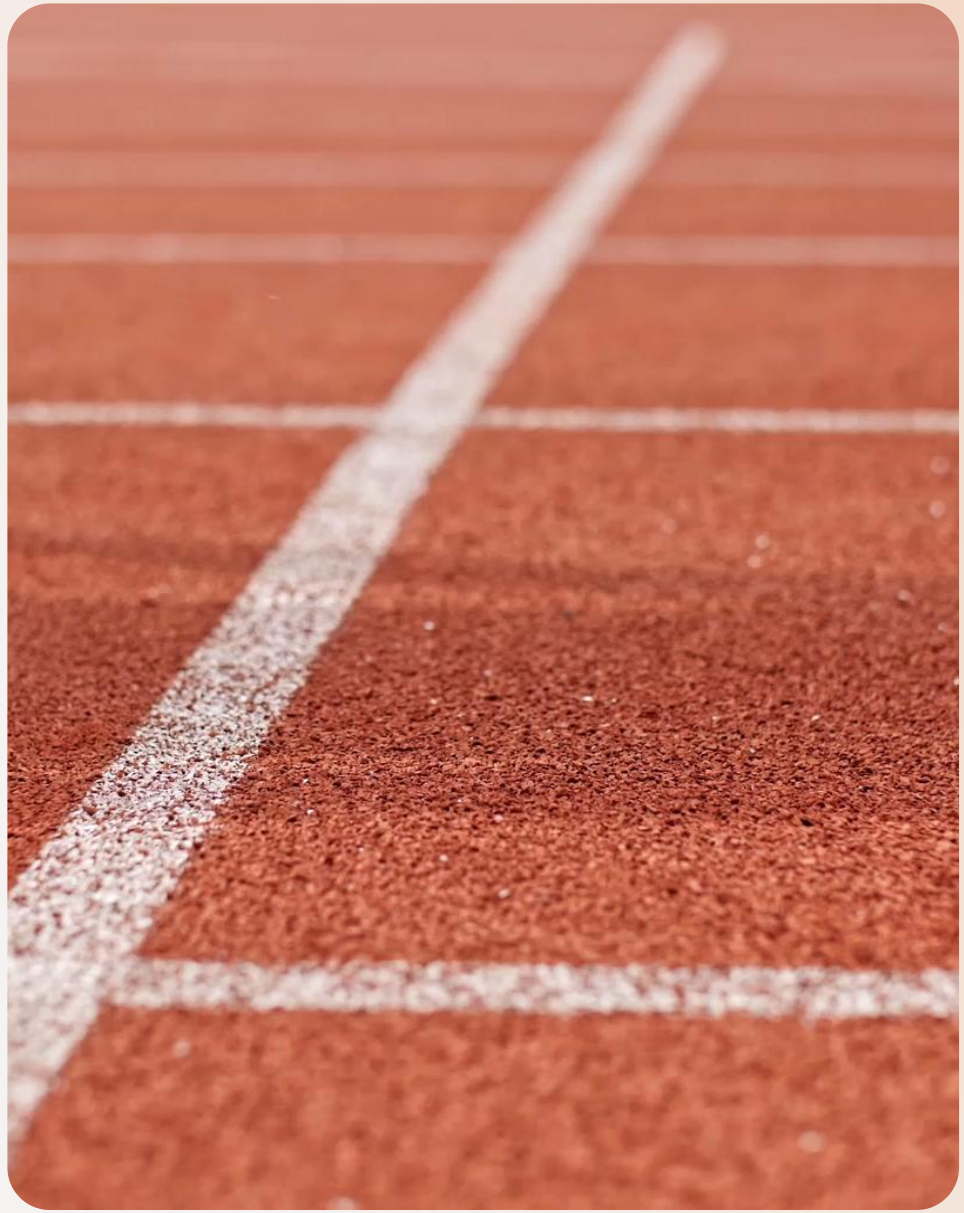


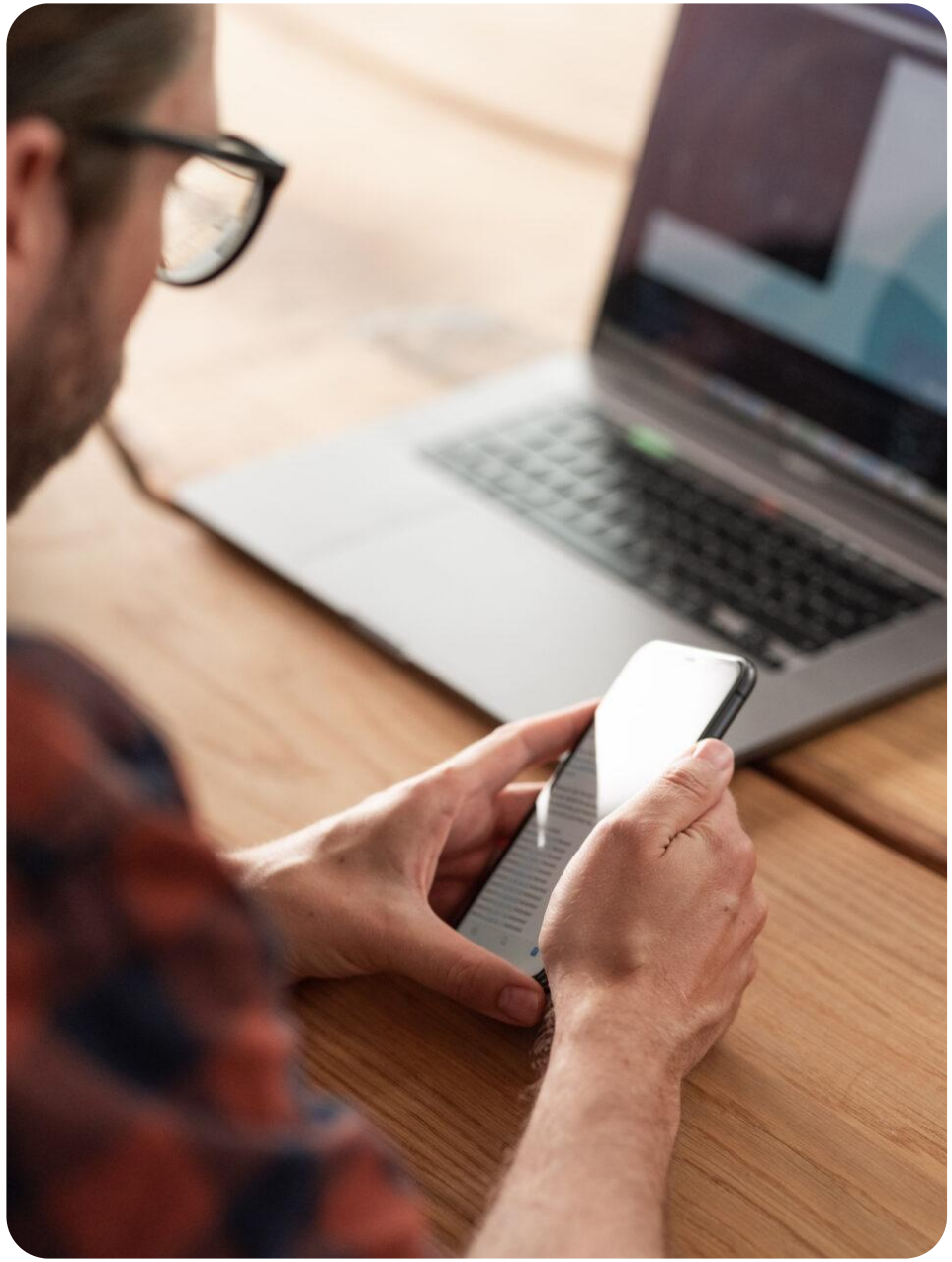
- **Log only what is useful**
  - Too much logging = noise
  - Too little logging = missing context
  - Avoid:
    - Logging every DOM click
    - Logging every selector
    - Repeating the same data many times
- **Use logging inside Try/Catch blocks**
  - This will make debugging 10x easier

EXERCISE 04 (15 min.)

- **Goals:**
  - Go back to exercise 2 and implement logging within each test step.

- **Locate** 'INSTRUCTIONS.MD'





2. Navigation and Interaction

CONTROL ELEMENTS

— Learn how to interact with radio buttons, checkboxes and dropdowns

ACTION: CHECK()

- `locator('SELECTOR').check()`

- **Key Characteristics**
  - Works only if element is *checkable*
  - Throws an error if:
    - Element is disabled
    - Element is not visible
    - Element is not a checkbox or radio

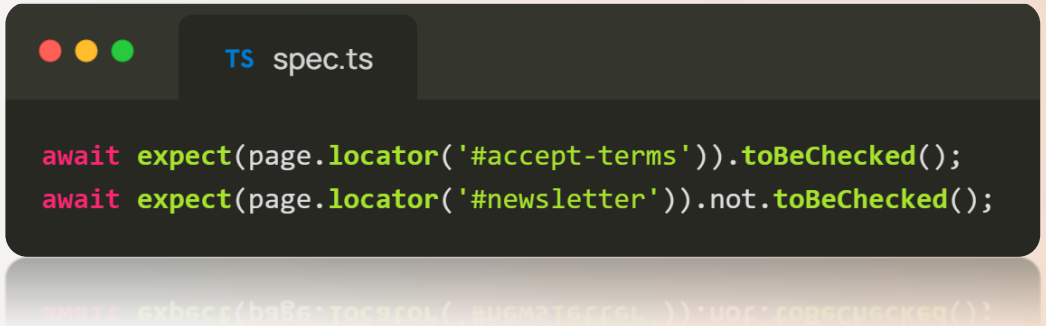
TS spec.ts

```ts
await page.locator('#accept-terms').check();
// OR
await page.getByRole('checkbox').setChecked(true);
```

- **When to use:**
  - ✓ For checkboxes
  - ✓ For radio button selections

VERIFY: CHECK()

- **Use assertions to verify check state**



- **When to verify?:**
  - After calling .check()
  - When you expect something to be either checked or unchecked.

ACTION: SELECTOPTION()

- `locator('SELECTOR').selectOption('OPTION')`
- Used to select a value from a <select> dropdown

TS spec.ts

```ts
await page.locator('#currency').selectOption('USD');
```

Ways to select an option:

TS spec.ts

```ts
// By Value
page.locator('').selectOption('usd');

// By Label
page.locator('').selectOption({ label: 'US Dollar' });

// By Index
page.locator('')).selectOption({ index: 2 })
```

VERIFY: SELECTOPTION()

— **Why verification matters:**

- Dropdowns often trigger logic (currency updates, filtering, etc)
- Ensures correct UI state before proceeding
- Catches wrong defaults or missing options

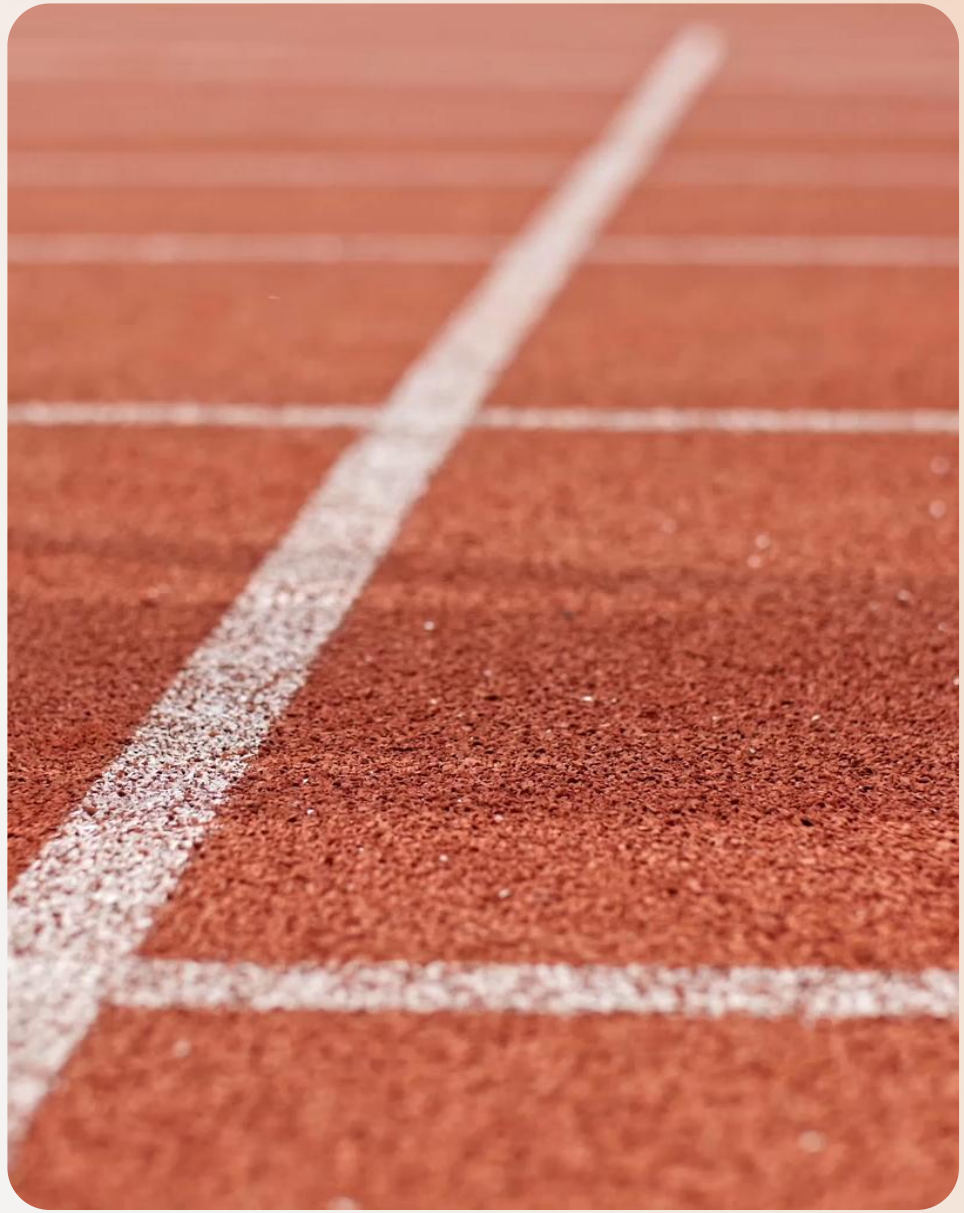
TS spec.ts

```
// Standard single checked value
await expect(page.locator('#currency')).toHaveValue('USD');

// Check selected label (using option locator)
await expect(page.locator('#currency option:checked')).toHaveText('US Dollar');

// For multi-select
await expect(page.locator('#countries')).toHaveValues(['DK', 'NO']);
```

EXERCISE 05 (35 min.)

- **Goals:**
  - Use playwright to add a new note by using all the techniques that we have learned so far.
  - Create multiple test scripts.

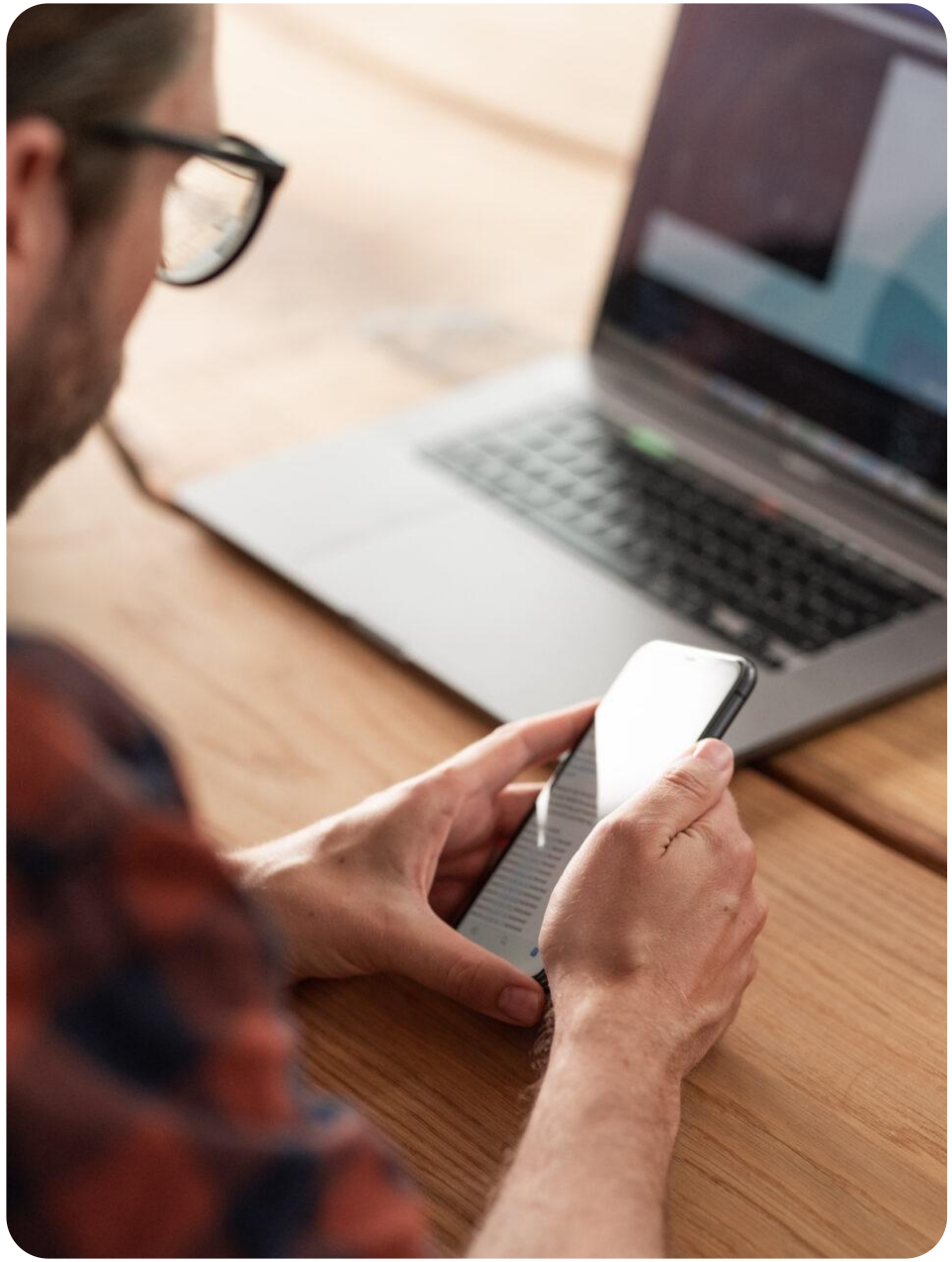
- **Locate** 'INSTRUCTIONS.MD'



3. API Testing

API TESTING

- APIs and API testing fundamentals
- API testing in Playwright



3. API Testing

API & API TESTING FUNDAMENTALS

- Learn what an API is
- Dive into the understanding of REQUESTs and RESPONSEs and how to use them correctly

WHAT IS AN API?

- API = Application Programming Interface
  - A structured way for two systems to communicate
    - APIs expose endpoints
    - You send **requests**
    - You receive **responses**

API
An API is like a resturant menu.
You order → the kitchen returns a dish.

HTTP REQUESTS (BASICS)

- API typically use HTTP

- **Common Methods**

Method	Meaning
GET	Retrieve data
POST	Create something
PUT	Replace something
PATCH	Update partially
DELETE	Remove something

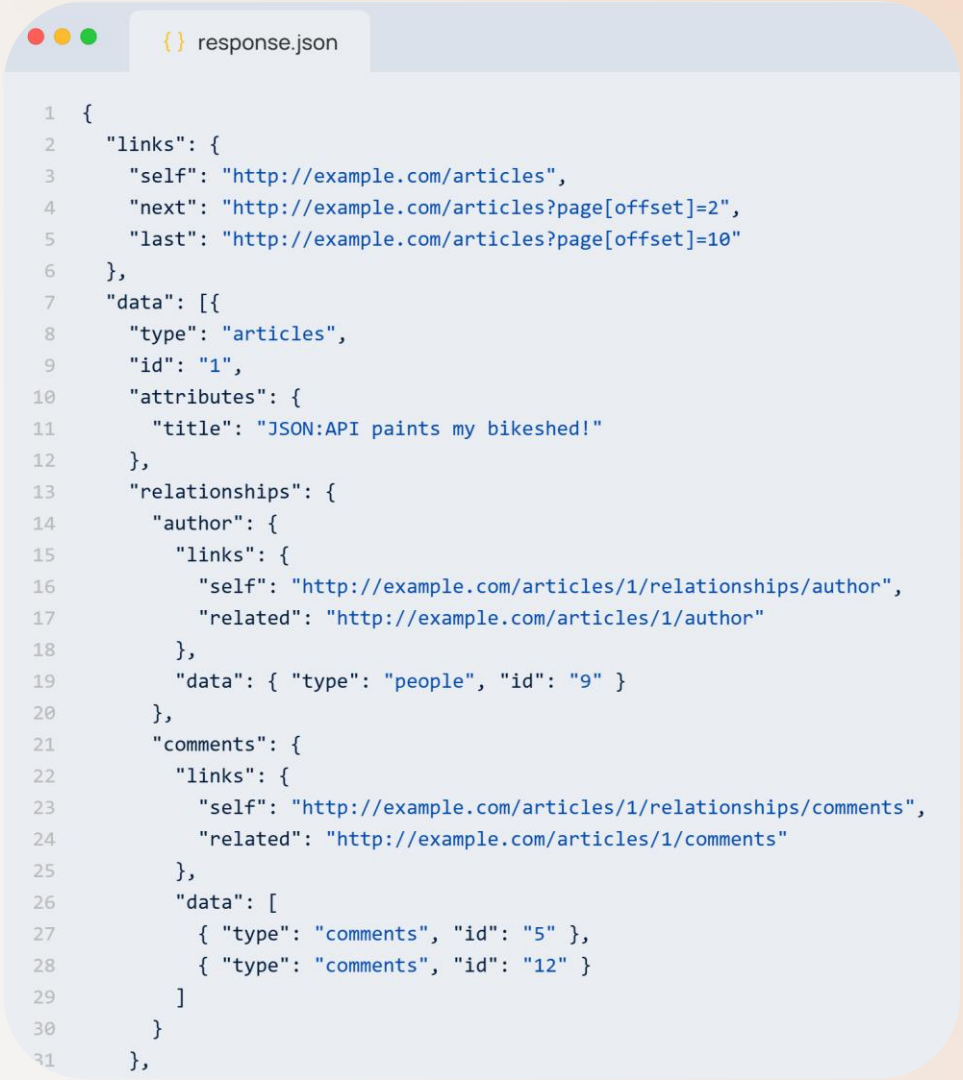
- **Requests usually contain:**
  - URL
  - Method
  - Headers
  - Body (optional)

HTTP RESPONSES (BASICS)

- Responses includes status codes:
- **Status Codes:**
  - 200 → OK
  - 201 → Created
  - 400 → Bad Request
  - 401 → Unauthorized
  - 404 → Not Found
  - 418 → I'm a Teapot!
  - 500 → Server Error
- **Body**: Usually in JSON

RESPONSE FORMATS

- **JSON**
  - Almost all modern APIs return **JSON**.
- **Use Cases:**
  - **Easy to learn and adopt:** Same syntax as JavaScript
  - **Fast and easy to validate:** YAML tends to be fussier about small errors.
  - **Compact**: More compact than YAML
  - **Secure**: More secure than YAML



JSON | YAML

JSON.json

```json
{
  "employees": [
    {
      "id": "40159",
      "name": "John",
      "technology": "Cloud computing",
      "title": "Engineer",
      "team": "Development"
    }
  ]
}
```

yaml file

```yaml
employees:
  - id: '40159'
    name: John
    technology: Cloud computing
    title: Engineer
    team: Development
```

SERVICE DESCRIPTION

- A service description is a formal, machine-readable document that explain:
  - What endpoints exists
  - What they accept
  - What they return
  - The expected request/response formats
  - Authentication rules
  - Error codes
- A service description is the *blueprint* of an API.
- **Most common formats**
  - OpenAPI / Swagger
    - The industry standard for REST APIs
  - WADL
    - Older, XML-based alternative (rare today)

Other formats you may encounter
WSDL, OData, RAML, API Blueprint, etc

WADL

Web Application Description Language

- Describes
  - Resources (including path)
  - Methods
  - Parameters
- XML file

wadl.xml

```
<application>
    <doc title="Web Finance Service WADL" />

    <resources base="http://web-finance.azurewebsites.net">

        <resource path="/api-rest/authenticate">
            <method id="login" name="POST">
                <doc title="AuthenticationRest.login" />
                <request>
                    <param name="username" default="string" type="xs:string" required="true" style="plain" />
                    <param name="password" default="string" type="xs:string" required="true" style="plain" />
                </request>
            </method>
        </resource>

        <resource path="/api-rest/registration">
            <method id="register" name="POST">
                <doc title="AuthenticationRest.register" />
                <request>
                    <param name="name" default="string" type="xs:string" required="true" style="plain" />
                    <param name="username" default="string" type="xs:string" required="true" style="plain" />
                    <param name="password" default="string" type="xs:string" required="true" style="plain" />
                </request>
            </method>
        </resource>

        <resource path="/api-rest/bankclient">
            <method id="getByUsername" name="GET">
                <doc title="BankClientControllerRest.getByUsername" />
                <request>
                    <param name="username" style="query" type="xs:string" required="true" />
                </request>
            </method>
        </resource>

    </resources>
</application>
```

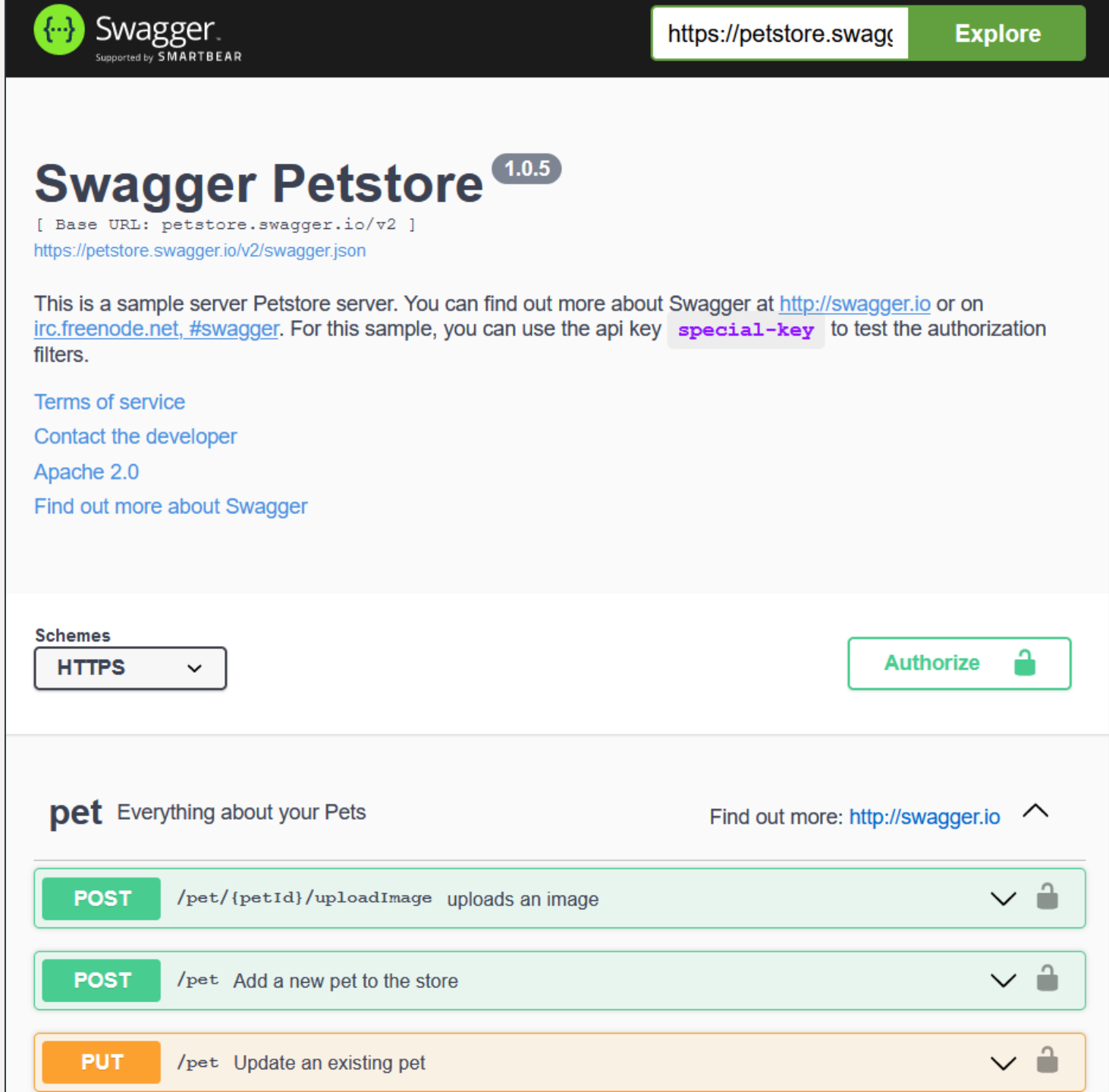
OPENAPI (SWAGGER)

- JSON or YAML file
- Translates into an interactive webpage, where API methods are described and available to try
- Can be used for validation

OPENAPI

- Webpage as described on previous slide

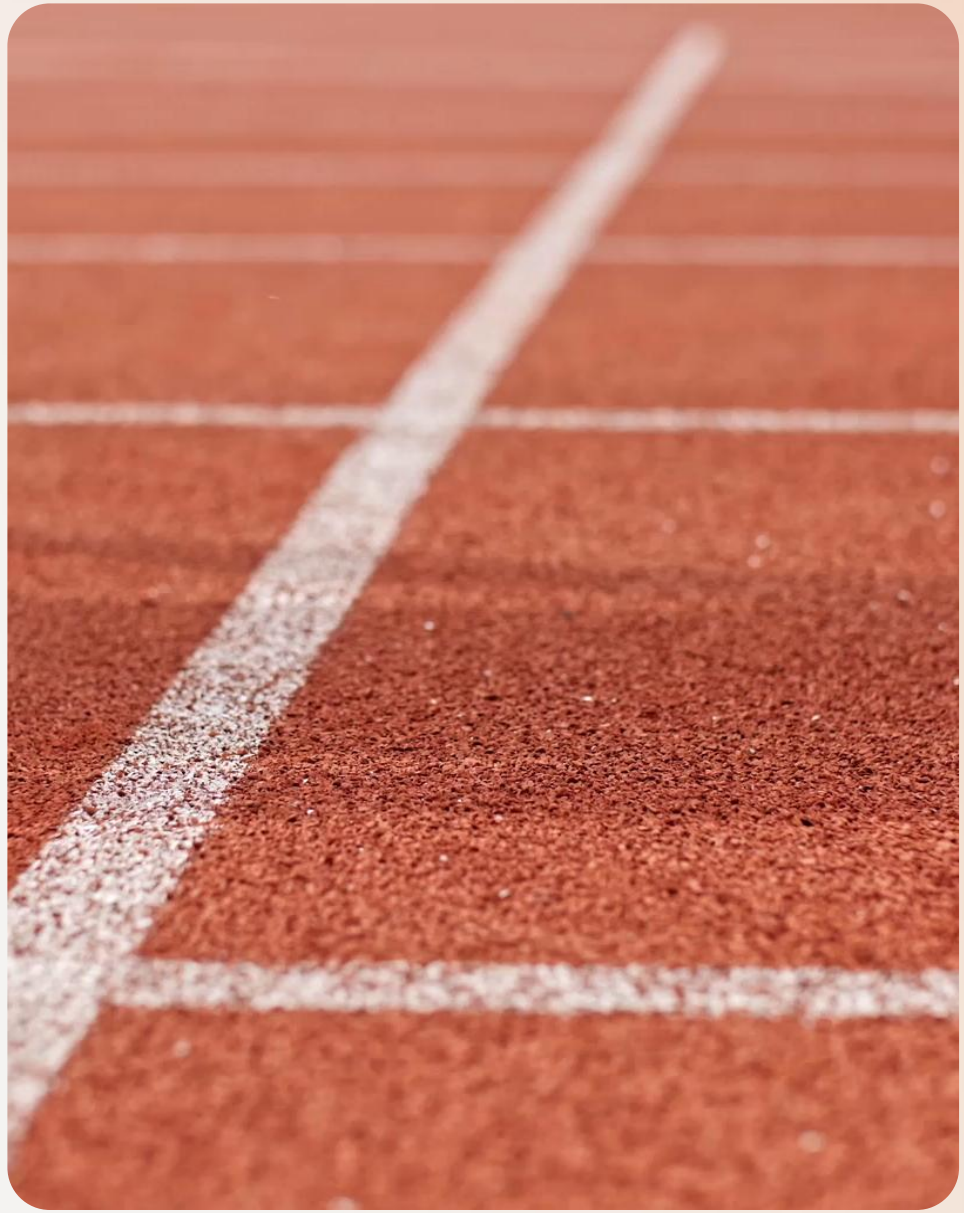
- Live Demo:



EXERCISE BONUS (10 min.)

- **Goals:**
  - Create a new note using only the swagger provided

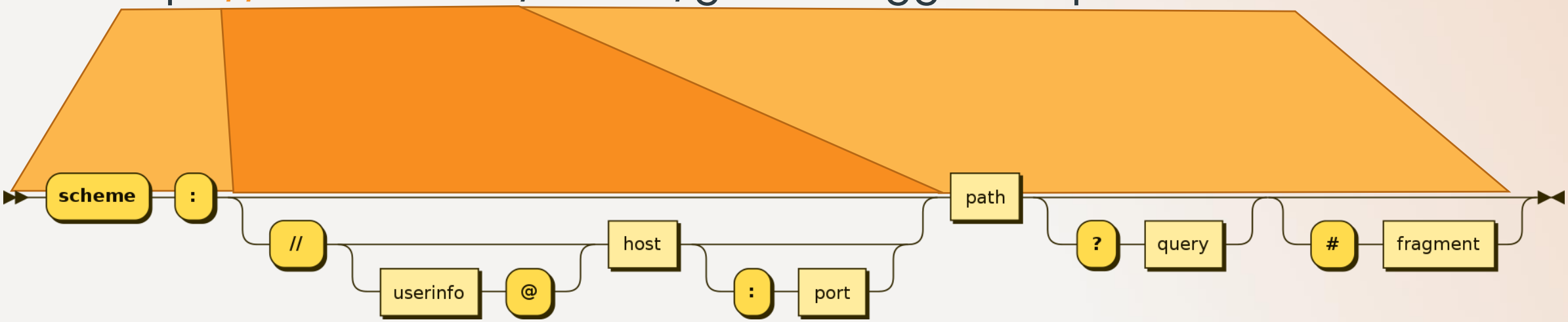
- **Locate** 'INSTRUCTIONS.MD' in BONUS EXERCISE DAY 1



WHAT DO WE NEED?

- **Where is the API?**
  - URL
- **How do we talk to it?**
  - HTTP

https://testhuset.dk/kursus/grundlaeggende-postman



PARAMETERS

- **Query Parameters:**
  - Used for filtering, sorting or searching
  - Placed after the ? In the url.

```bash
GET {host}/api-rest/product/phone?brand=sony&year=2021
```

- **Path Parameters:**
  - Used to identify a specific resource

```makefile
GET {host}/api-rest/product/{brand}/{series}

GET {host}/api-rest/product/samsung/galaxyS
```

PARAMETERS

- **Headers & Cookies**
  - Used to send **metadata**, such as authentication tokens, request IDs, or session data.

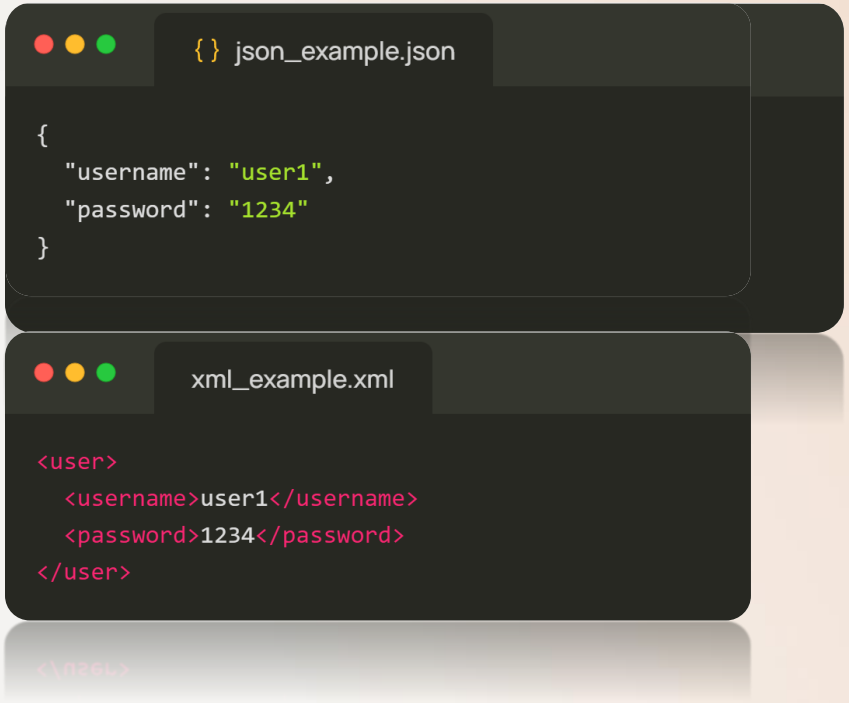
Parameters:
Different parameter types serve different purposes: **filters, identifiers,** and **metadata** all live in different parts of a request

PARAMETERS IN REQUEST BODIES

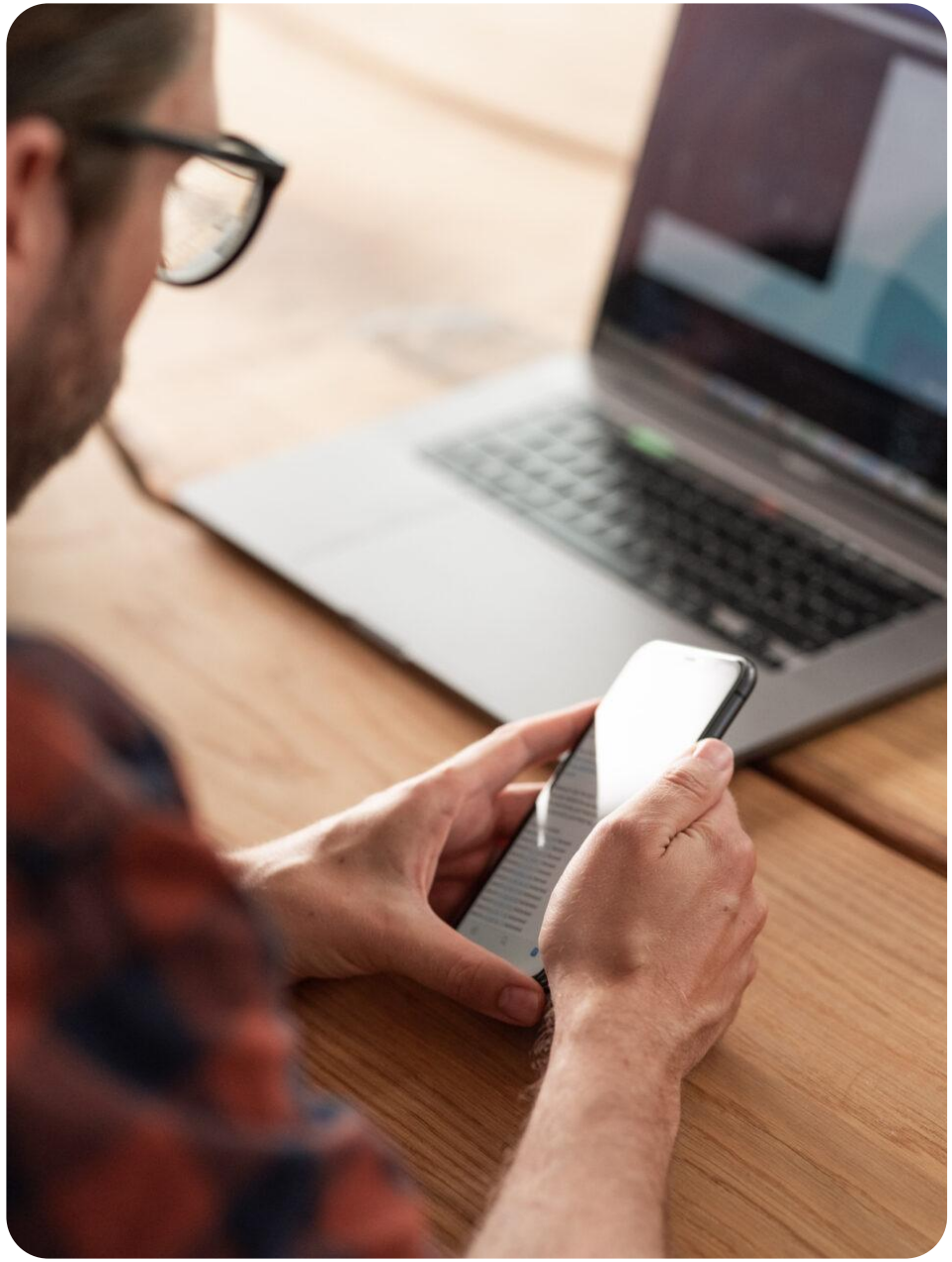
What are Request Body Parameters?

- Body parameters are used when you need to send **structured data**, not just filters or identifiers
- Common in **POST**, **PUT** and **PATCH**

Common Body Formats:



Notes
Query + Path parameters = simple text values
Body Parameters = structured data used to create/upload resources



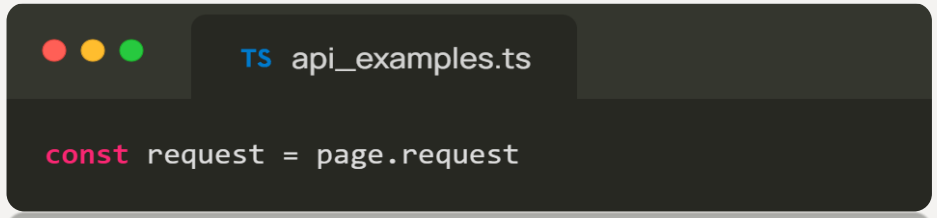
3. API Testing

USING API'S IN PLAYWRIGHT

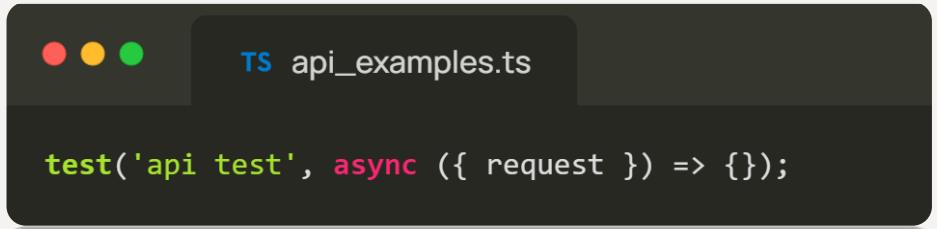
- Take the knowledge we have learned about APIs for test implementation.
- Play around with Playwrights API Testing engine

PLAYWRIGHT APIREQUESTCONTEXT

- Playwright provides a fixture called:



- And used in tests:



- **This object lets you send:**
  - GET
  - POST
  - PUT
  - PATCH
  - DELETE

Playwright APIRequestContext
All the above has built-in auto-waiting and response helpers

SENDING A SIMPLE REQUEST

— Playwright makes API requests extremely easy, you don't need axios or fetch.

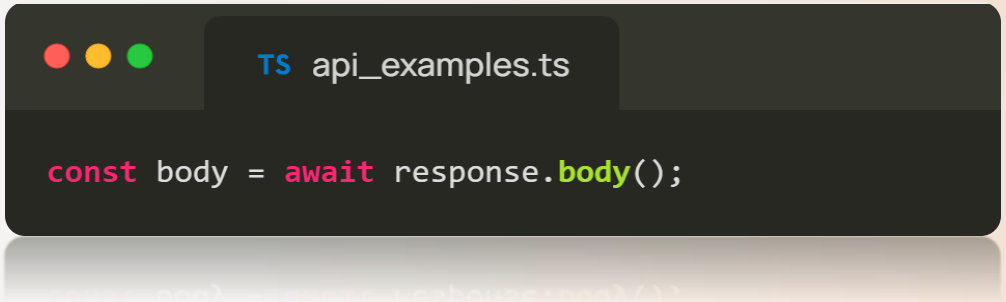
TS api_examples.ts

```ts
test('GET users', async ({ request }) => {
  const response = await request.get('https://api.example.com/users');

  expect(response.ok()).toBeTruthy();

  const data = await response.json();
  logger.info(data);
});
```

READING THE RESPONSE

There's a few options when wanting to read the response body:

- `response.body()`
  - Images
  - Binary files
  - PDFs
  - Downloads



READING THE RESPONSE

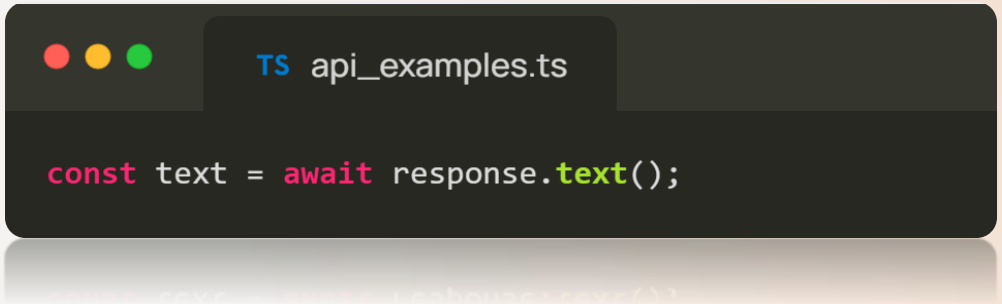
There's a few options when wanting to read the response body:

- `response.json()`
  - Parses the body as JSON
  - Automatically deserializes into an object
  - Most common for REST APIs

READING THE RESPONSE

There's a few options when wanting to read the response body:

- `response.text()`
  - HTML
  - Logs
  - Text-based responses

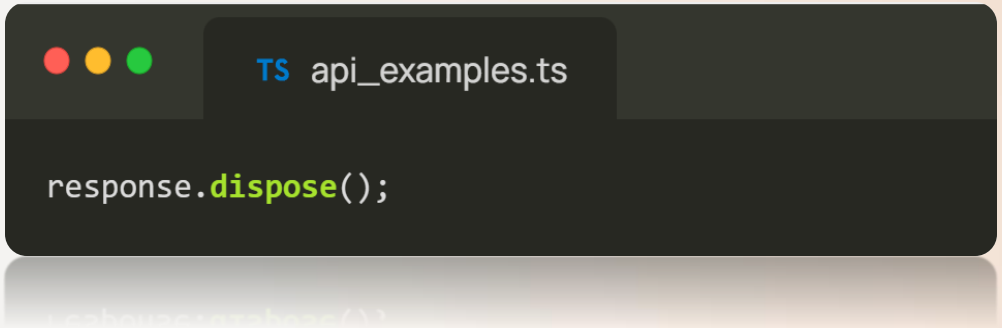


response.text()
Returns the body as a plain text string

READING THE RESPONSE

There's a few options when wanting to read the response body:

- `response.dispose()`
  - Large responses
  - Repeated tests
  - Reducing memory pressure



response.dispose()
Frees the stored response body from memory

VALIDATING API RESPONSES

— **Typical assertions:**

TS api_examples.ts

```
expect(response.status()).toBe(200);

const json = await response.json();
expect(json.name).toBe('John');
expect(json.active).toBe(true);
```

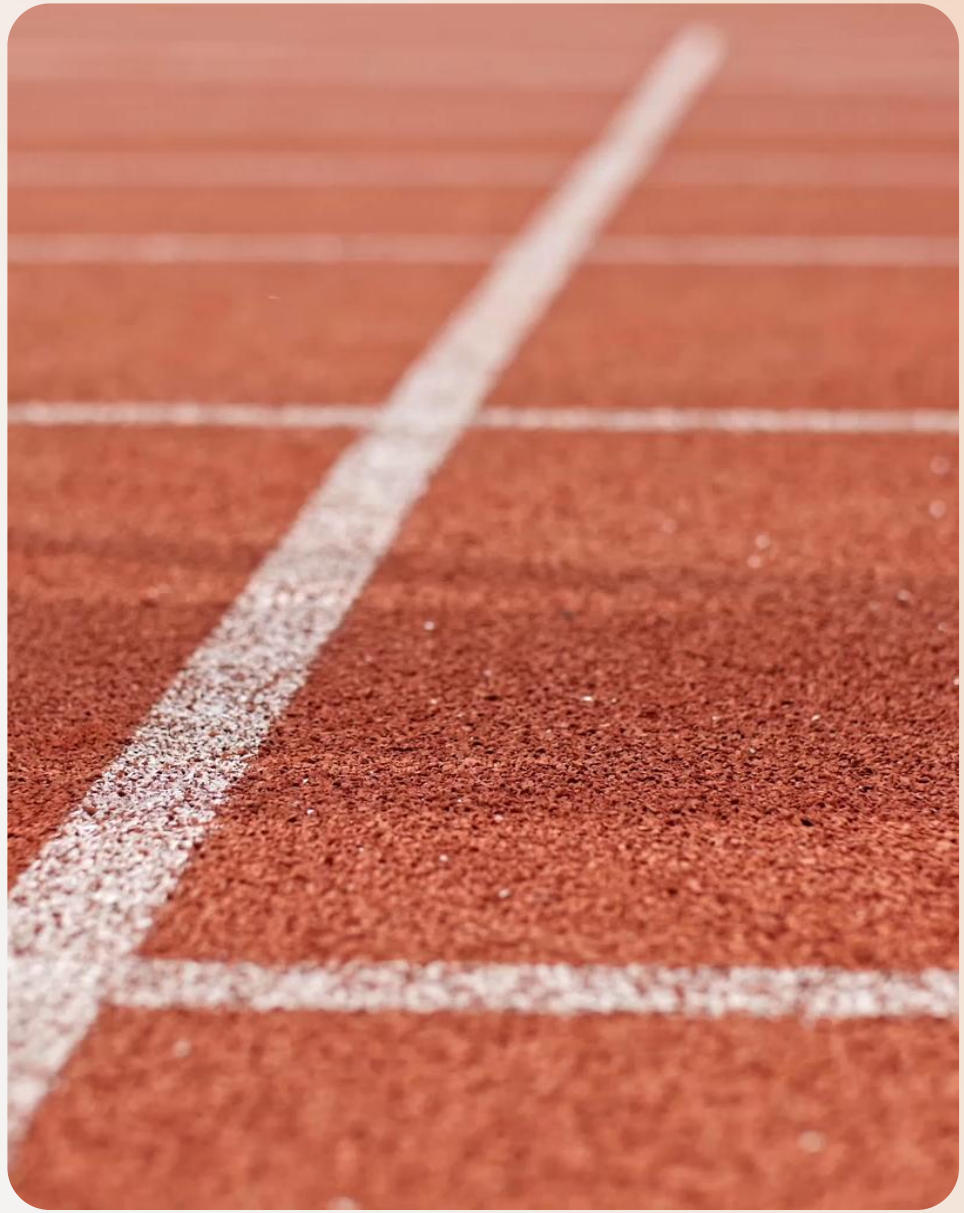
— **You can also assert:**

- Headers
- Text
- Raw body bytes
- Cookies

EXERCISE 06 (20 min.)

- **Goals:**
  - Learn how to perform a simple API request using Playwright
  - Understand how to call a health-check endpoint
  - Parse and validate JSON responses
  - Use assertions to verify correct API behavior

- **Locate** 'INSTRUCTIONS.MD' in exercise_6 for DAY 1



AUTHENTICATION

— Basic Auth:

TS api_examples.ts

```ts
await request.get('/admin', {
  headers: {
    Authorization: `Basic ${btoa('user:pass')}`
  }
});
```

— Bearer Token:

TS api_examples.ts

```ts
await request.get('/me', {
  headers: {
    Authorization: `Bearer ${token}`
  }
});
```

AUTHENTICATION – GOOD PRACTICE

TS api_examples.ts

```ts
test.beforeAll(async ({ request }) => {
  const response = await request.post(`${baseUrl}/users/login`, {
    form: { email, password },
    headers: { Accept: "application/json" },
  });

  expect(response.ok()).toBeTruthy();

  const body = await response.json();

  token = body.data.token;
});
```

SENDING POST

TS api_examples.ts

```ts
const response = await request.post('/users', {
  data: {
    name: 'Sarah',
    email: 'sarah@example.com'
  }
});

expect(response.status()).toBe(201);
```

COMBINING UI + API

— A common workflow

1. Create user via API
2. Login with UI
3. Assert page

EXERCISE 07 (35 min.)

- **Goals:**
  - Authenticating against a REST API.
  - Using the returned auth token to call a protected endpoint
  - Making basic assertations on the response

- **Locate** 'INSTRUCTIONS.MD' in BONUS EXERCISE DAY 1