

# Architecture and Testing

Separation of Concerns. End-to-End Testing.



**SoftUni Team**  
Technical Trainers



**SoftUni**



Software University

<https://softuni.bg>

sli.do

**#js-advanced**

# Table of Contents

1. Application **Testing**
2. Testing with **Playwright**
3. **Architecture** and **Separating Concerns**
4. Live **Demo**





# Application Testing

Unit, Integration and End-to-End Testing

- **Unit tests** – cover separated functionality
  - E.g., test the **result of a function** with different input
- **Integration tests** – cover the communication inside and between entire modules
  - E.g., test if data coming from a **remote request** is correctly interpreted by the **business logic**
- **End-to-end (Functional) tests** – cover **all steps** that occur when the **user** performs an **action**, from the UI, to the DB, and back

- **Unit tests** are used to verify that a **piece of code** (function, class, etc.) operates correctly
- The tested code does **not** involve **external dependencies** (application state, other modules, external systems)
- They are fast to **write** and fast to **execute**
- Usually **created by the developer**, who is aware of the code specifics (**white-box** testing)
- Common tools include **Mocha**, **Chai**, **QUnit**, **Jasmine**, etc.

- **Integration tests** are used to check the communication between multiple code elements (functions, classes, entire modules)
- They often require the **inclusion of external dependencies** (other application modules, databases, remote resources)
- Relatively **complex to create** (due to the external dependencies)
- Can be delegated to a **separate team**, not involved in the writing of the code (**black-box** testing)
- Common tools include **Sinon**, **JMock**, **Mockito**, etc.

- **Functional tests** are used to run through the **entire application**, in a real environment
- Usually involves the whole **technological stack** (REST services, database operations, authentication, etc.)
- Depending on the expected outcome and tools used, their **complexity** is comparable to **integration tests**
- Mostly the concern of specialized **QA automation engineers**
- Common tools include **Selenium, Puppeteer, Cypress**, etc.





# Testing with Playwright

End-to-End Testing with a Headless Browser

# What is Playwright?

- A **complete suite** for **testing web applications** in a real environment – the **web browser**
  - Our application is executed inside a **"headless" browser**
  - User **input is simulated**, and the result is **monitored**
- Compatible with **Chromium**, **Firefox** and **WebKit**
- Available in **JavaScript**, **TypeScript**, **Python**, **C#** and **Java**
- Home page: <https://playwright.dev/>

- Install via **NPM** with **Chromium** support:

```
npm install --save-dev playwright-chromium
```

- **Note:** this will download **browser binaries** (~200 MB)
- Normal operation involves the following setup:



- Create **test.js** and enter the following code:

```
const { chromium } = require('playwright-chromium');  
(async () => {  
  const browser = await chromium.launch();  
  const page = await browser.newPage();  
  await page.goto('https://google.com/');  
  await page.screenshot({ path: `example.png` });  
  await browser.close();  
})();
```

- **Execute** via Node.js:

```
node test.js
```

- Combine with a **test-running framework** (e.g., Mocha and Chai)

```
const { chromium } = require('playwright-chromium');
const { expect } = require('chai');

let browser, page; // Declare reusable variables

describe('E2E tests', async function() {
  before(async () => { browser = await chromium.launch(); });
  after(async () => { await browser.close(); });
  beforeEach(async () => { page = await browser.newPage(); });
  afterEach(async () => { await page.close(); });
});
```

- Note:** make sure both the **REST service** and **web server** are running **before** executing tests

# Example: Loading Static Page

- Direct **navigation** – same as entering the URL in the **address-bar**

```
it('loads static page', async function() {  
  await page.goto('http://localhost:3000/');  
  await page.screenshot({ path: `index.png` });  
  await browser.close();  
});
```

- Visiting via clicking on links (<a>-tags)

```
await page.click('a[href="/register"]');  
await page.waitForNavigation();  
await page.waitForLoadState();  
// Perform operations on new page
```

# Example: Finding Elements

- CSS Selectors:

```
await page.click('button'); // Basic selector  
await page.click('article:has(div.promo)'); // Content-based
```

- Find element by **text content**:

```
// Case-insensitive, partial matches  
await page.click('text=Log in');  
  
// Case-sensitive, full match only  
await page.click('text="Log in"');
```

- Advanced usage: <https://playwright.dev/docs/selectors>

# Example: Verifying Content

- Obtain text content:

```
const content = await page.textContent(selector);
```

- Attribute value:

```
const val = await page.getAttribute(selector, attrName);
```

- Checkbox state:

```
const checked = await page.isChecked(selector);
```

- Visibility:

```
const visible = await page.isVisible(selector);
```



# Example: Form Input

- Text input:

```
await page.fill(selector, 'Peter');           // Text  
await page.fill(selector, '2020-02-02');      // Date  
await page.fill('text=First Name', 'Peter');  // Via Label
```

- Checkboxes and radio buttons:

```
await page.check(selector);  
await page.uncheck(selector);
```

- Select options (single and multiple values):

```
await page.selectOption(selector, 'blue');  
await page.selectOption(selector, ['red', 'greeb', 'blue']);
```

# Example: Request Handling

- Submit form and wait for response:

```
const [response] = await Promise.all([  
  page.waitForResponse('**/api/data'),  
  page.click('input[type="submit"]'),  
]);
```

- Request matching can be done with predicate:

```
page.waitForResponse(  
  response => response.url().includes(token))
```

- Obtain request body (to validate sent values):

```
const postData = JSON.parse(response.request().postData())
```

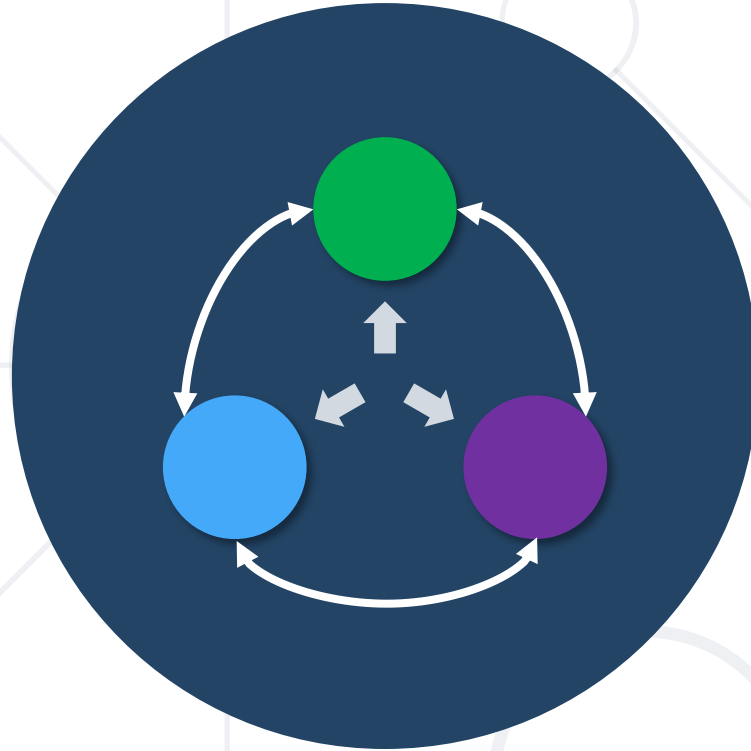
# Example: Response Mocking

- Setup request interception can return mock data:

```
await page.route('**/api/data', route => route.fulfill({  
  status: 200,  
  body: testData,  
}));
```

- Note: this must be configured before the form is submitted
- Abort requests (to prevent external calls or resource loading):

```
await page.route('**/*.{png,jpg,jpeg}',  
  route => route.abort());
```



# Separating Concerns

Writing Easy to Maintain Code

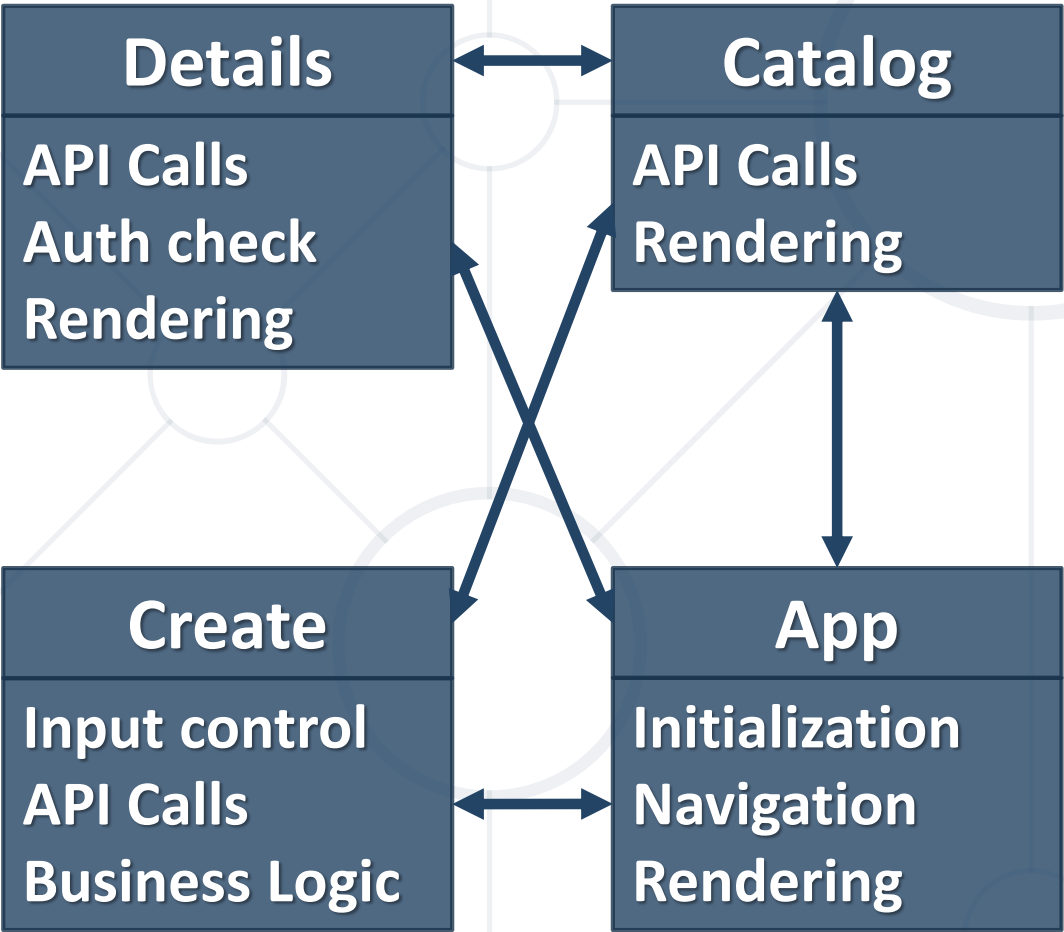
- **Multiple concerns** – parts of the application perform actions on **various domains** (e.g., DB calls, business logic, UI)
- This leads to **high coupling**:
  - **Low abstraction** level limits the size of the application
  - It's **difficult to change** one module without affecting the rest
  - **Code** steps are **repeated** out of necessity
  - It's **impractical to reuse** a module in another applications
  - The developer must be **aware of all specifics** of every module

# Goal of Separation of Concerns

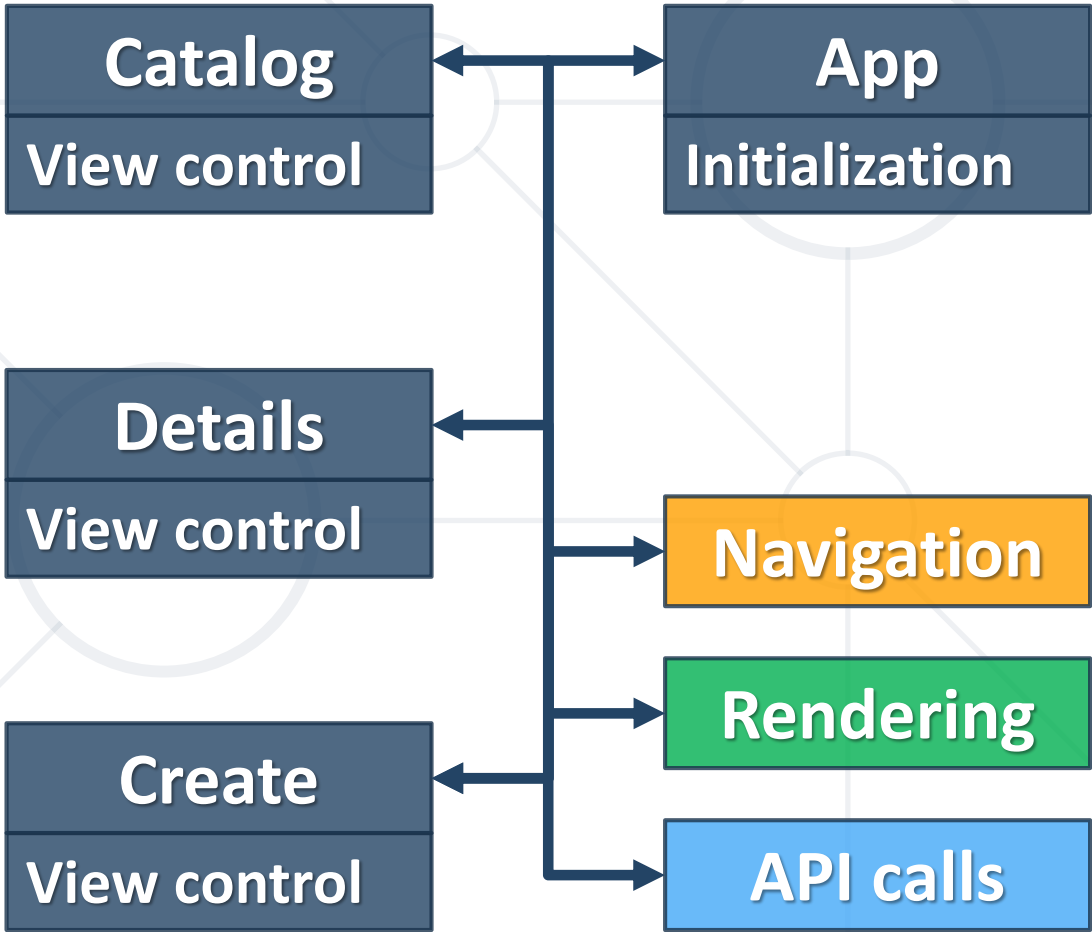
- Limit a unit of code (function, module) to a **single domain**
  - E.g., a method that **only visualizes** (renders) data on screen
- Implementation is **abstract from details**
  - E.g., the rendering function **does not concern** itself with the source of the data
- The developer **doesn't need to know** how a module operates internally in order to use it
- **Code reuse** is a secondary effect – **easier reasoning** is primary

- Common steps:
  - **Extract actions** over different domains in their own functions
  - Identify **similar actions** across different parts of the application
  - **Increase abstraction** of the extracted functions, so that they can be **used in more places** with minimal changes
  - Move functions from a single domain to a **separate module**
- **Don't overdo abstraction!** A good rule of thumb – increase abstraction **when** you need to **refactor** the code

## Multiple Concerns



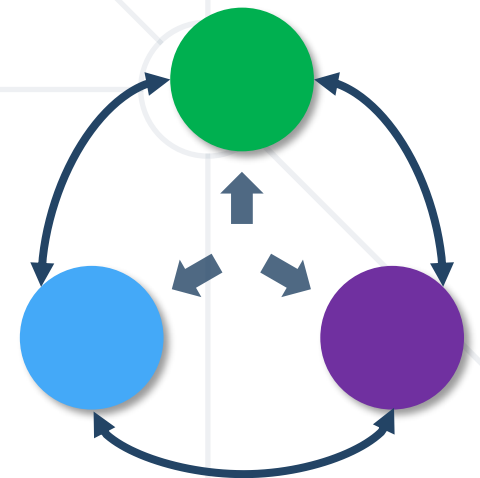
## Isolated Concerns





# Example Isolated Modules

- **Backend API** – specific to the used service
- **Request logic** – specific to the application business logic
- **Data manipulation** – specific to the application business logic
- **UI display and control**
- **Utility** functions



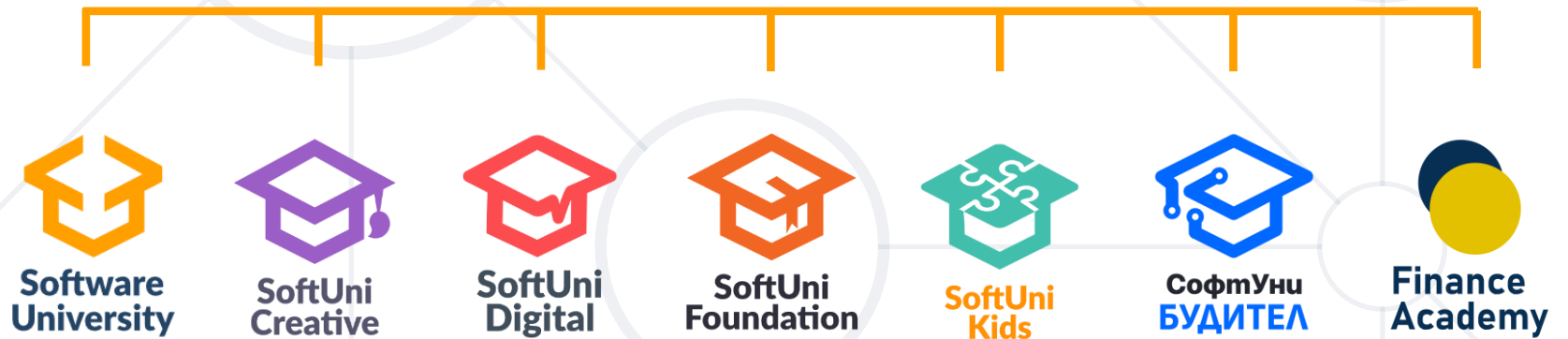
- Different **categories of tests** can be used at various stages of development
  - **Unit, Integration, End-to-end**
- **Playwright** is a **testing suite** for web apps
- By **separating code concerns** we make our programs **easier** to reason about
  - A **code unit** must be concerned only by a **single domain** (data, rendering, etc.)



# Questions?



SoftUni



# SoftUni Diamond Partners



- Software University – High-Quality Education, Profession and Job for Software Developers
  - [softuni.bg](http://softuni.bg), [softuni.org](http://softuni.org)
- Software University Foundation
  - [softuni.foundation](http://softuni.foundation)
- Software University @ Facebook
  - [facebook.com/SoftwareUniversity](https://facebook.com/SoftwareUniversity)



- This course (slides, examples, demos, exercises, homework, documents, videos and other assets) is **copyrighted content**
- Unauthorized copy, reproduction or use is illegal
- © SoftUni – <https://softuni.org>
- © Software University – <https://softuni.bg>

