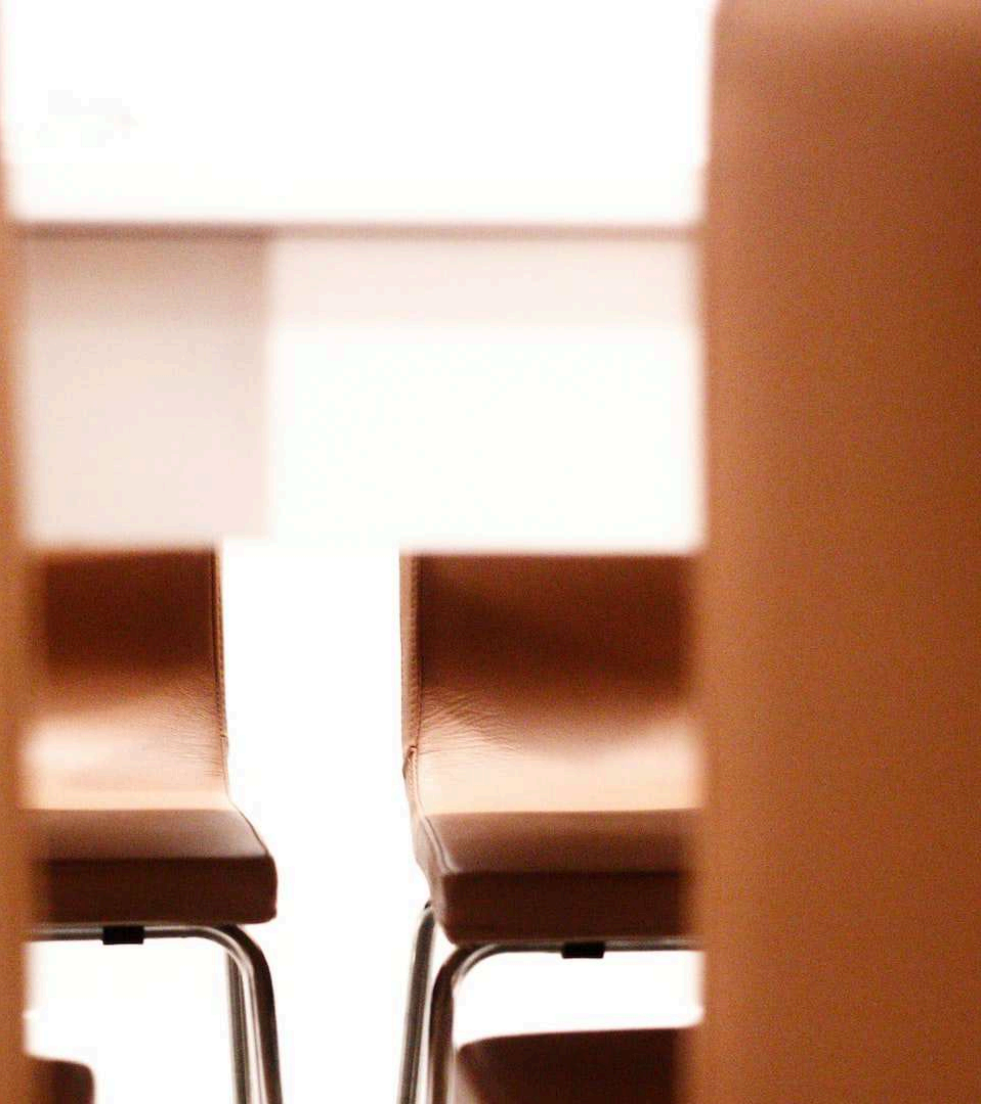
Testing Applications with

Playwright and Typescript



Traininig

- Training goals and program
- Questions, discusion, expectations
- Elastic program contents

Mateusz Kulesza

Your Trainer

- Senior Software Developer,
- Team Leader, Scrum Master
- Project Manager
- Consultant and trainer

Experience

Multiple years of commercial experience in:

- HTML5, CSS3, SVG, EcmaScript 5 i 6
- jQuery, underscore, backbone.js
- Angular.JS, Angular2, React, RxJS, Flux
- Webpack, Karma, Jasmine, Jest, Vitest ...
- Selenium, Protractor, Cypress, **Playwright**
- NodeJS, Express, MongoDB, **Typescript**

Why test the code?

Well, what do we need it for?

Why test the code?

- faster errors detection
- allows for agile programming (refactoring)
- tests are live and vivid documentation
- avoids regression
- the testable code is more reusable
- We can assess the performance of some of the application
- we can check the code covering (dead code)
- tests reduce the hidden complexity of the code
- saving time and money

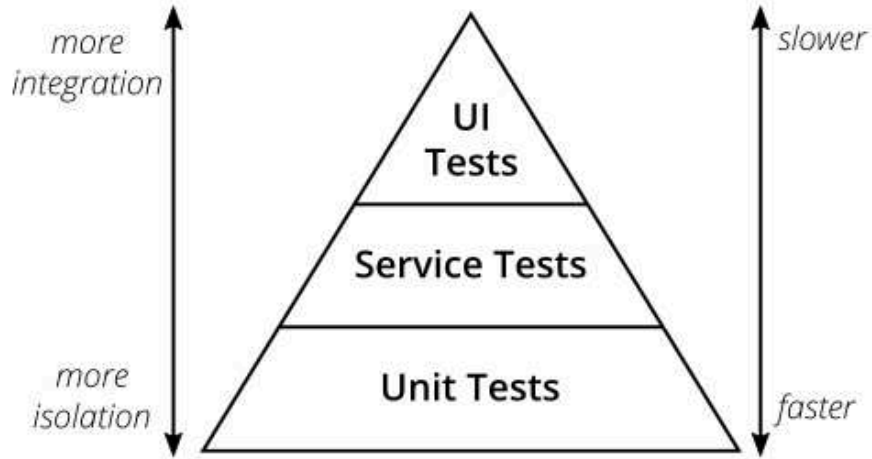
Types of tests

Test pyramid

- static analysis
- unit test
- integration
- Service tests
- Interface tests
- End2end - Functional

Test pyramid

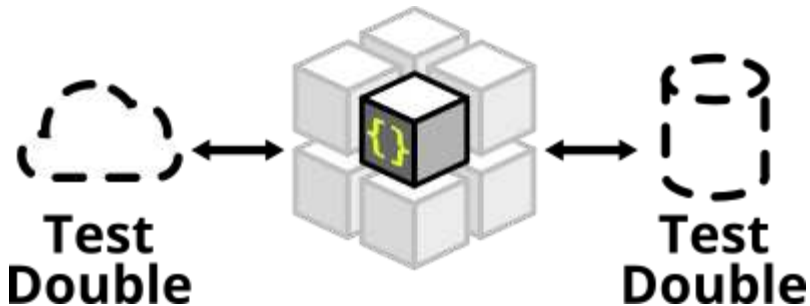
pros and cons of each layer of testing



- source:

  https://martinfowler.com/articles/practical-test-pyramid.html

Unit tests



In the unit test, we substitute false collaborators

Unit tests

Tests can be divided into the following variants:

- path analysis
- use of equivalence classes (test sets)
- Testing of boundary values
- syntactic testing (Garbage in - Garbage Out)

Substitutes - Stubs i Mocks

- **Dummy** - They are transferred but are never used.They are usually used only to fill in lists of parameters.
- **fakes** - They actually have working implementation, but usually use shortcuts that make them not suitable for production (e.g. database in memory).
- **Stubs** - They provide ready answers to the connections made during the test, usually without responding to anything but what is programmed for the test.
- **Spy** - These are shortcuts that also record certain information based on how they were called.One such form may be the e-mail service that records the number of messages sent.
- **Mock** - They are pre -programmed with expectations that create the specification of connections to be obtained.They can report an exception if they receive an unexpected connection and are checked during verification to make sure they have received all the expected calls.

Integration tests

Integration tests are carried out to assess the compliance of the system or component with specific functional requirements

They occur after unit tests, but before validation tests.Modern applications usually consist of many cooperating systems, so you should check whether communication between them is not disturbed.

Integration tests will use unit -tested modules, grouping them into larger aggregates.

Static tests

There are plenty of ways to make mistakes in the application.

One of the most common sources of errors is associated with typos and incorrect types.

Transferring a sequence to a function that expects a number or a typo effect in a logical statement are stupid mistakes that we have never been to commit, but this happens all the time.

- Code structure - Linter
- Code correctness - Compiler

Unit tests - BDD style

In the BDD style (Behavior Driven Design) the names of the tests should describe the behaviors expected by the user

```
describe("The module we test", () ⇒ {
  it("should do x", () ⇒ {
    expect(operacja()).toEqual("expected result");
  });
});
```

Raport:

```
ERROR: The module we test should do x

Expected 'undefined' to equal 'expected result'
```

Pattern Arrange/Act/Assert

Convention called given-when-then. or Arrange-act-assert.We divide the test into three sections:

- **Arrange** / given... - description of the sample situation and initial conditions,
- **Act** / when... - The action we perform at the tested class facility (one call call)
- **Assert** / then ... Then - verification of the results.

Rule of F.I.R.S.T

Good quality tests are

- **Fast** - Fast: Start (subset) tests quickly (because you will be running it all the time)
- **Independent** - Independent: no tests depend on others, so you can start any subset in any order
- **Repeatable** - Repeated: run n times, get the same result (to help isolate errors and enable automation)
- **Self-checking** - the test can automatically detect whether it has passed (without checking the results by man)
- **Timely** - Written around the same time as the tested code (with TDD, written as the first!)

Zombie Testing

One behavior at once
Baby steps - We are slowly increasing the complexity of the test.Early green light is priceless feedback.

Z.O.M.B.I.E.S

- Z – Zero
- O – One
- M – Many (or More complex)

- B – Boundary Behaviors
- I – Interface definition
- E – Exercise Exceptional behavior

- S – Simple Scenarios, Simple Solutions

Zero cases - The first test scenarios relate to the simple conditions of the created object.

When determining zero cases, watch out for the design of the interface and capturing border behavior in test scenarios.

Then "simmer", both the code and tests by adding cases

Special border case - Testing the behavior desired during the transition from scratch to **one**.

When the border behavior between zero and one (and maybe return to zero from one) are captured in tests, go to abstraction dealing with more complex scenarios and management of many elements.

New **boundary conditions are created**.

Finally, you consider **unique things** that can happen.

Test Driven Design

tests first

Test-Driven Development is a technique that helps create a production code that is easy to use and which does not have excess implementation.

Starting always with the test, we are more sure that the whole code is tested.

As the tests become more detailed, the production code becomes more general.

It is important to remember that TDD is not a technique that aims to create good tests (although this is certainly a positive side effect).

"Tests are a tool that helps you write a better production code" — **Kent Beck, autor książek o TDD**

Cykl
Red-Green-Refactor

Red

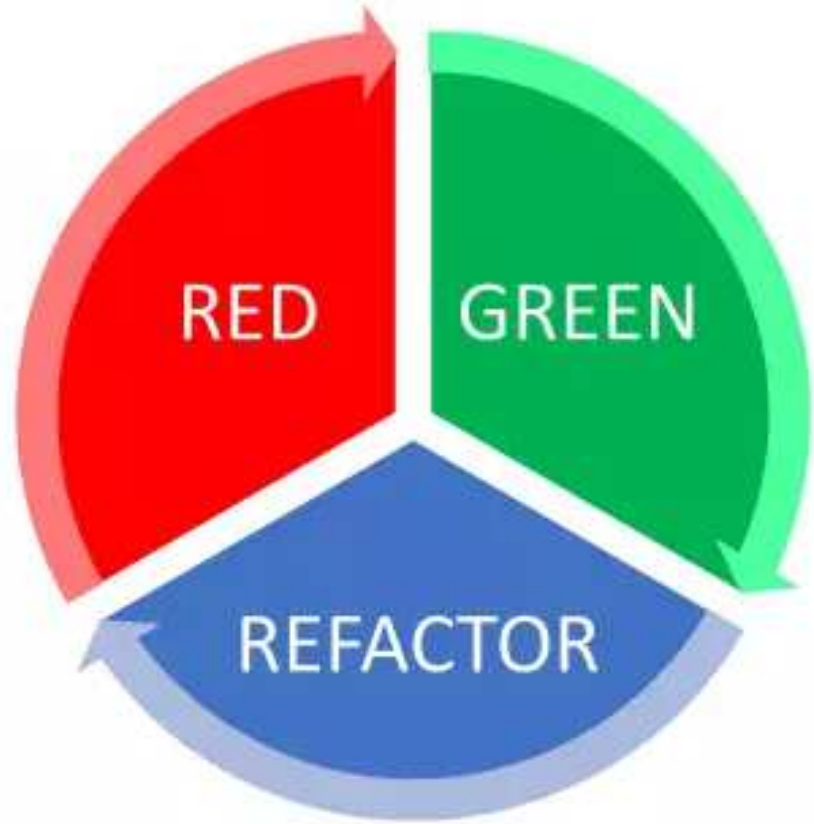
– We write tests that do not pass

Green

– We write a minimum code that undergoes tests

Refactor

– We improve the quality and readability of the code



And we repeat ...

Cycle of Red-Green-Refactor

The Idea TDD assumes that we break the code for adding code to three phases:

- Red – We create a test that starts but does not pass.
- Green – We add a code that makes new tests successfully.
- Refactoring – Improving the quality of the written code (or tests)

In TDD, when we write a new test, first **it must fail** - red.

Writing tests that always go through is another way to get stuck during TDD.

Regression detection

- **Regression** – The phenomenon of errors in the software after the intended change in some part of the program code (e.g. after making a correction for another error).

The result of these changes may be the wrong operation of another program function, which in previous versions worked properly.

To detect regression during the development of the program, you must conduct **regression testing**.

Usually, regression tests are related to the restart of the test set, which previously ended correctly.

It aims to reveal potential problems arising from the changes.

Playwright

Any browser • Any platform • One API

- Auto-wait. Playwright waits for elements to be actionable prior to performing actions. It also has a rich set of introspection events. The combination of the two eliminates the need for artificial timeouts - the primary cause of flaky tests.
- Web-first assertions. Playwright assertions are created specifically for the dynamic web. Checks are automatically retried until the necessary conditions are met.
- Tracing. Configure test retry strategy, capture execution trace, videos, screenshots to eliminate flakes.

Powerful Tooling

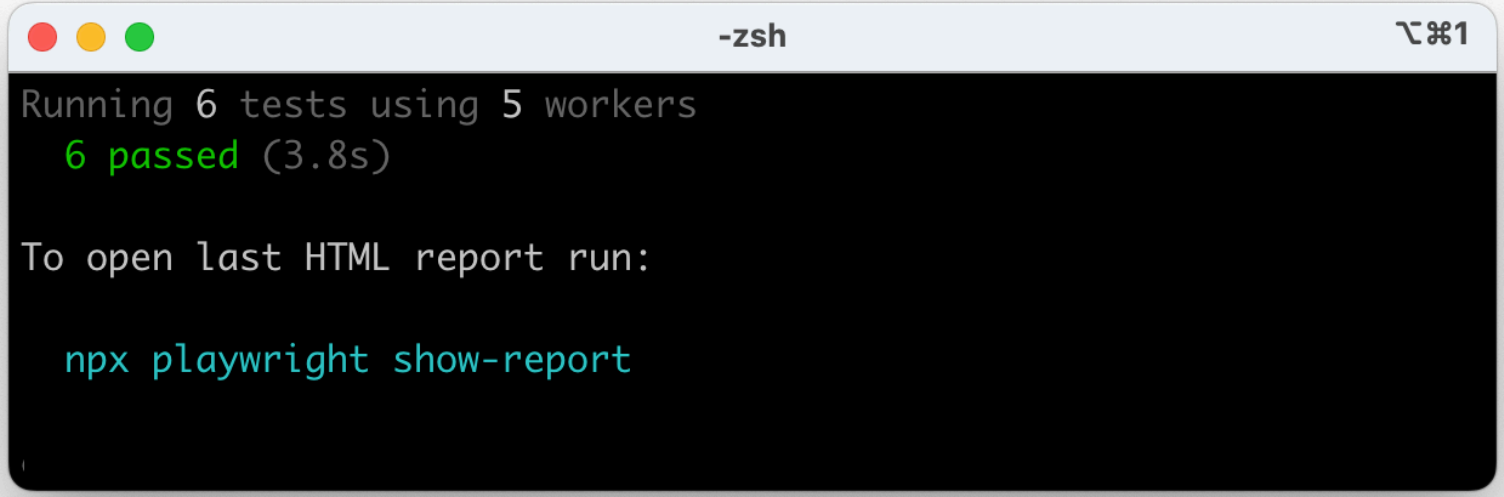
- Codegen. Generate tests by recording your actions. Save them into any language.
- Playwright inspector. Inspect page, generate selectors, step through the test execution, see click points, explore execution logs.
- Trace Viewer. Capture all the information to investigate the test failure. Playwright trace contains test execution screencast, live DOM snapshots, action explorer, test source, and many more.

Installation Guide

```
node -v
npm -v
git -v

npm init playwright@latest
```

```
npx playwright test
```



HTML Test Reports

```
npx playwright test --reporter=list

npx playwright test --reporter=line

npx playwright test --reporter=html


npx playwright show-report
```

Projects

```
import { defineConfig, devices } from '@playwright/test';

export default defineConfig({
  projects: [
    {
      name: 'chromium',
      use: {  ...devices['Desktop Chrome'] },
    },
//  ...
```

```
npx playwright test --project=firefox
```

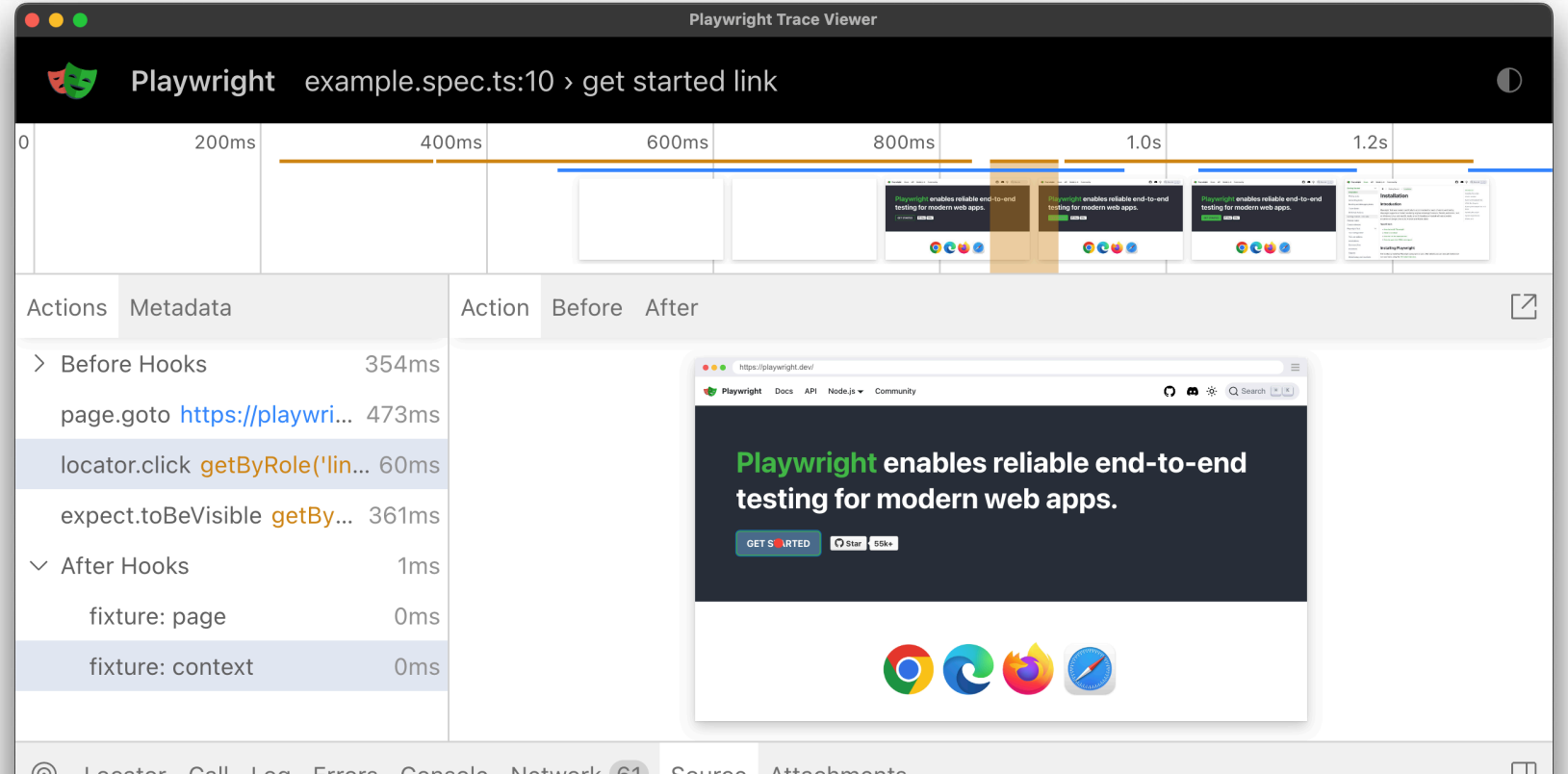
Package.json

```
{ //  ...
    "scripts":{
        "test:dev": "npx playwright test --project=chromium",
        "test:ui": "npx playwright test --ui",
        "test": "npx playwright test"
         //  ...
```

```
npm run test:dev
```

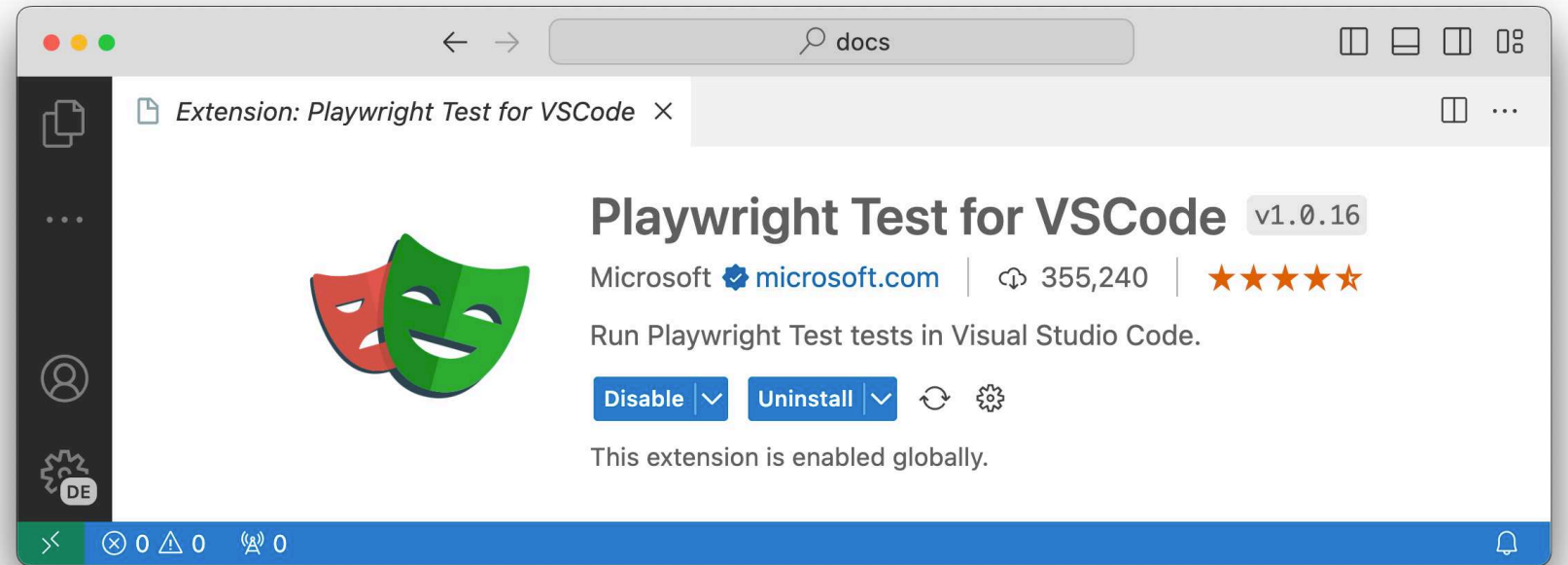
Playwright UI

npx playwright test --ui

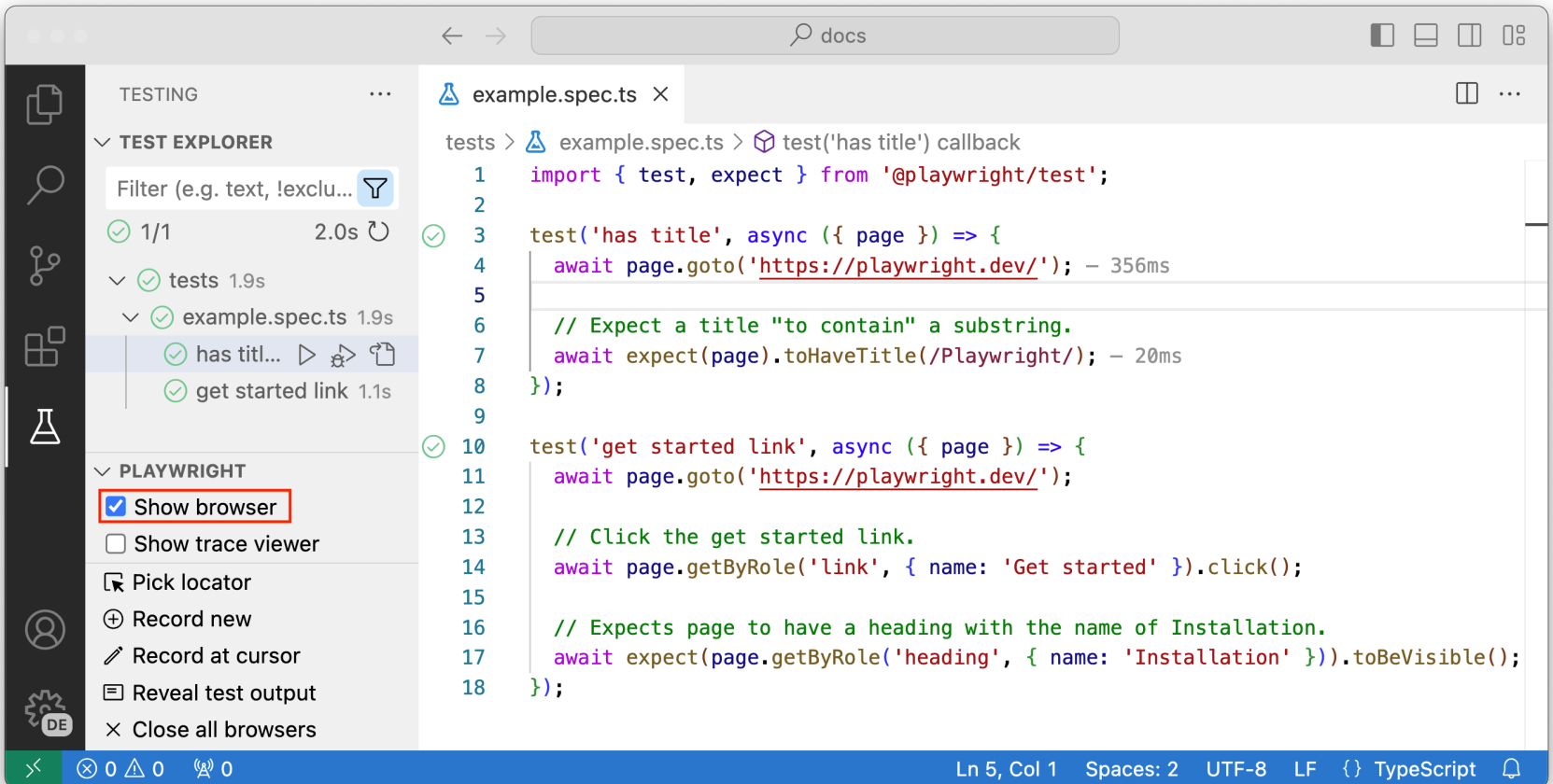


VS Code extension

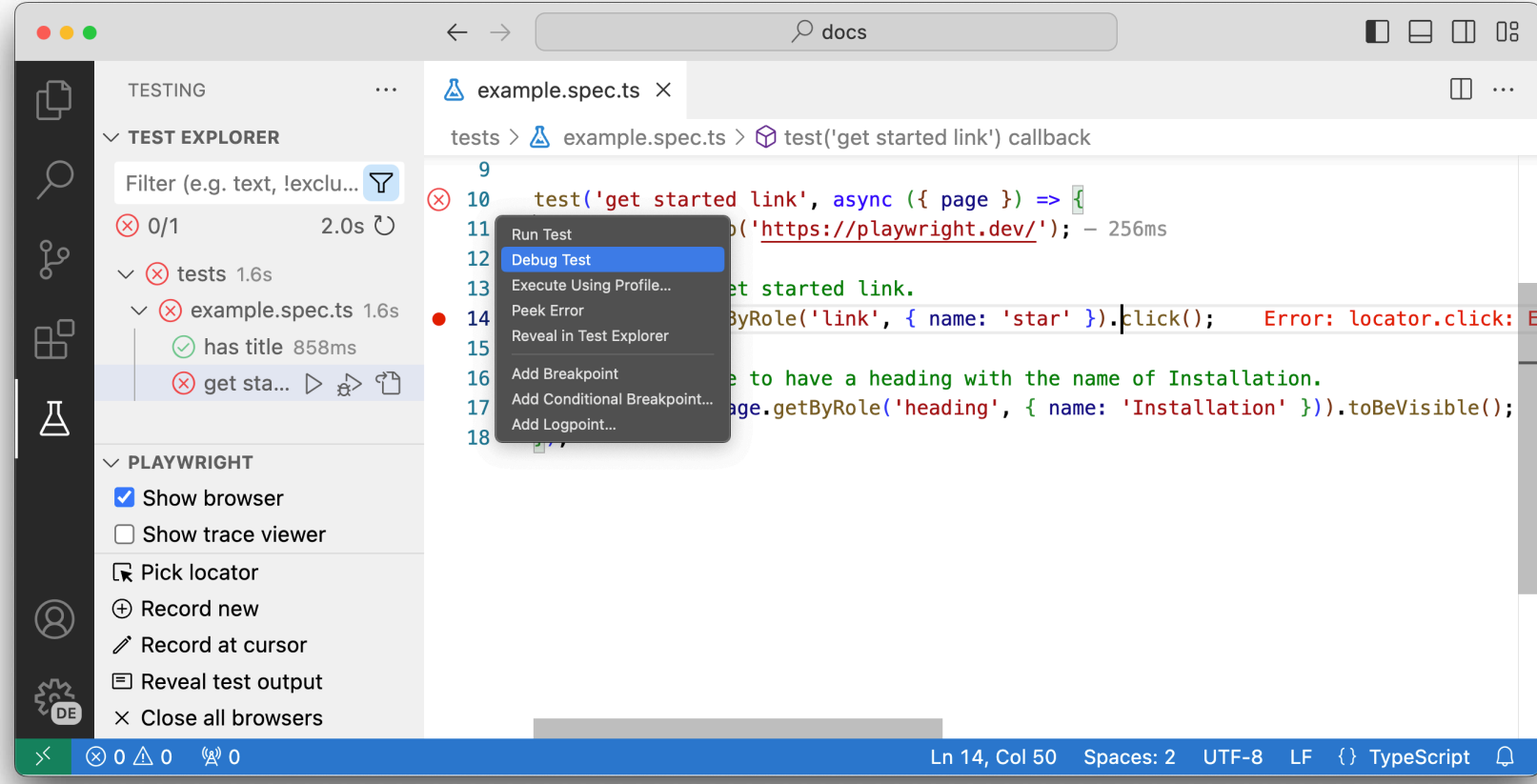
https://marketplace.visualstudio.com/items?itemName=ms-playwright.playwright



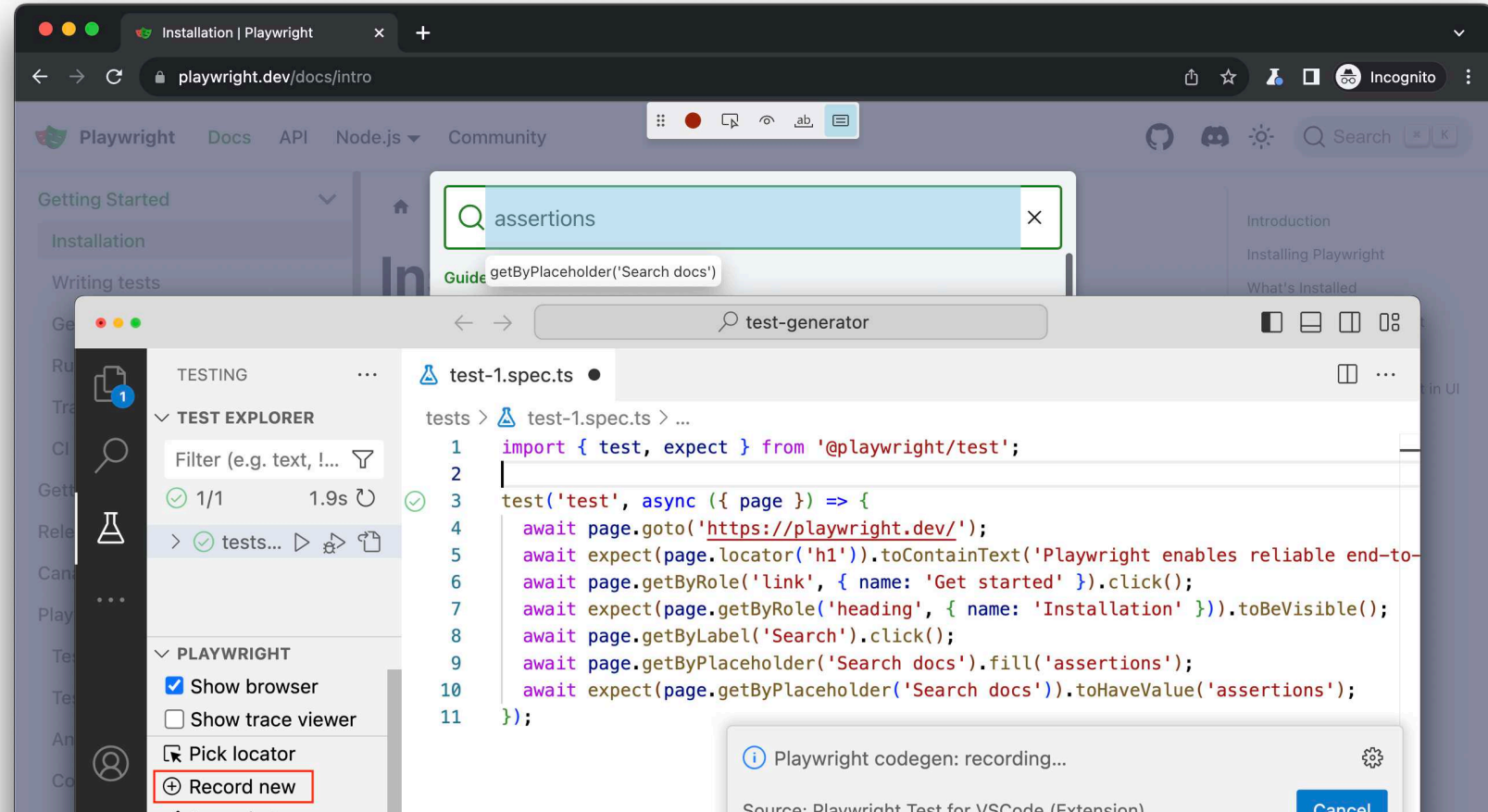
Run Tests and Show Browsers



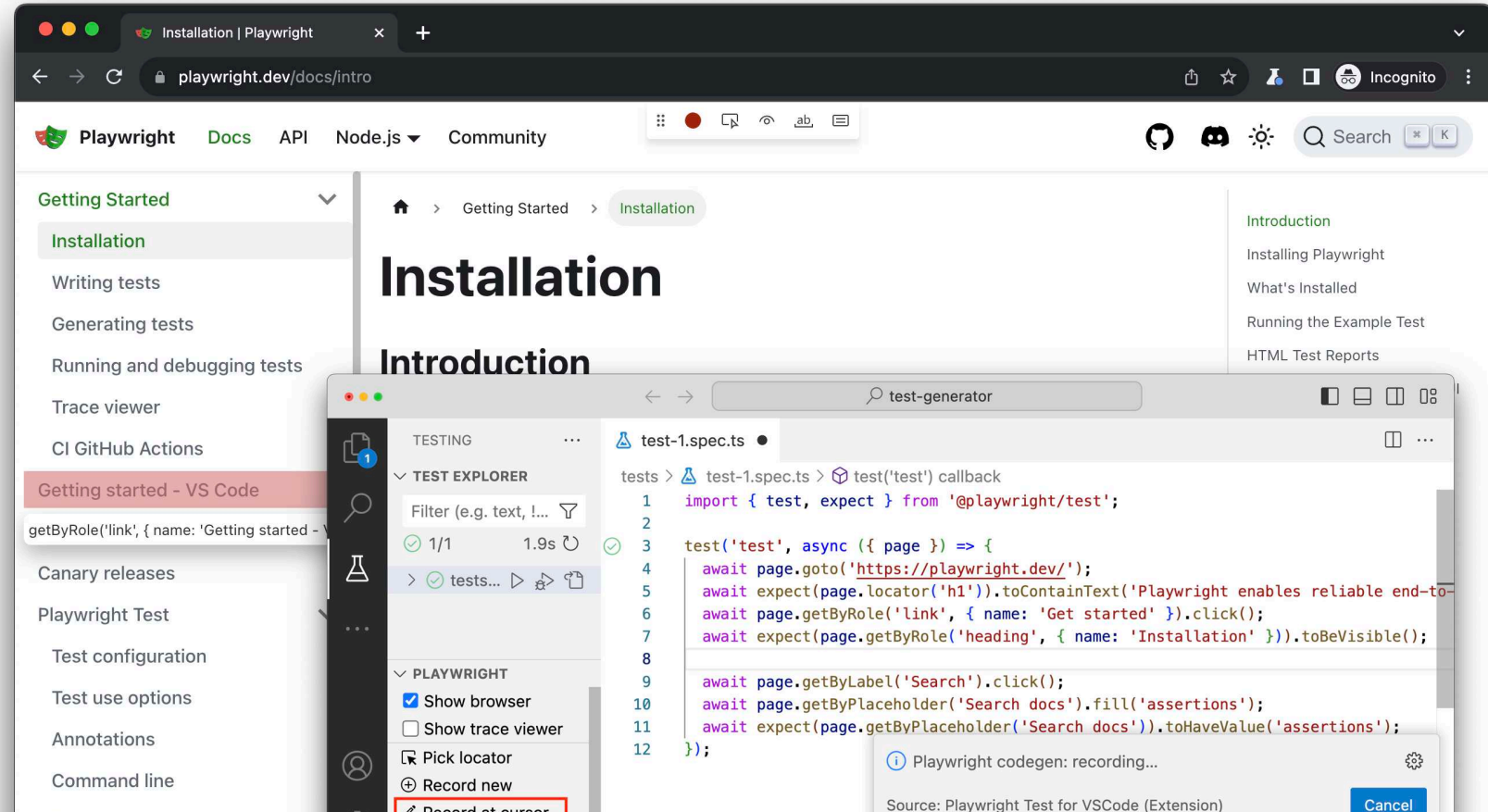
Debug tests


docs
TESTING
TEST EXPLORER
Filter (e.g. text, !exclu...
0/1
2.0s
tests 1.6s
example.spec.ts 1.6s
has title 858ms
get sta...
PLAYWRIGHT
Show browser
Show trace viewer
Pick locator
Record new
Record at cursor
Reveal test output
Close all browsers
example.spec.ts
tests > example.spec.ts > test('get started link') callback
9
10 test('get started link', async ({ page }) => {
11 ('https://playwright.dev/'); — 256ms
12
13 et started link.
14 yRole('link', { name: 'star' }).click(); Error: locator.click: E
15
16 to have a heading with the name of Installation.
17 age.getByRole('heading', { name: 'Installation' })).toBeVisible();
18
Run Test
Debug Test
Execute Using Profile...
Peek Error
Reveal in Test Explorer
Add Breakpoint
Add Conditional Breakpoint...
Add Logpoint...
Ln 14, Col 50
Spaces: 2
UTF-8
LF
TypeScript

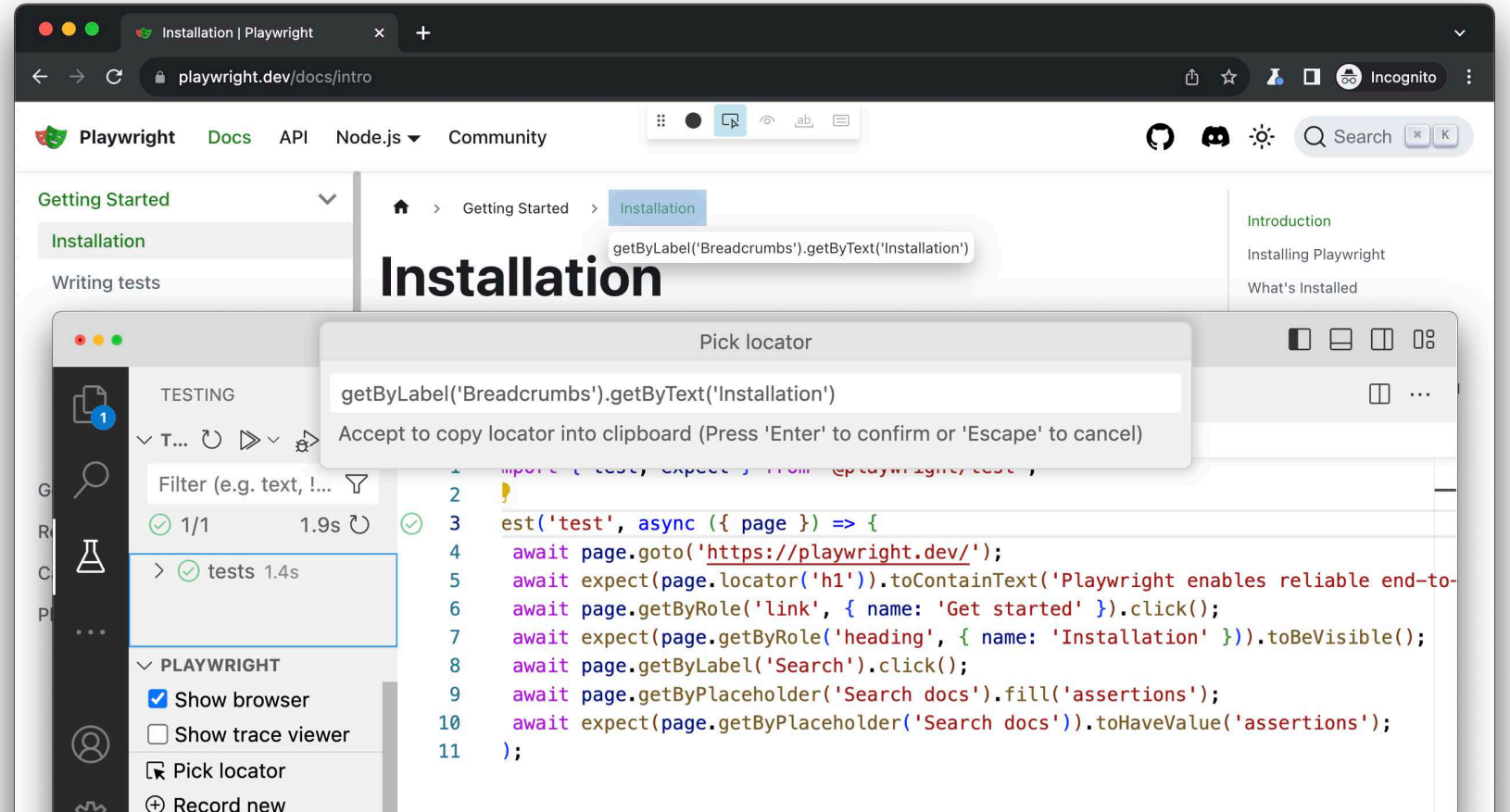
Record tests



Record tests at cursor



Pick locator



Good Practices

Test user-visible behavior

Automated tests should verify that the application code works for the end users, and avoid relying on implementation details such as things which users will not typically us

Make tests as isolated as possible

Each test should be completely isolated from another test and should run independently with its own local storage, session storage, data, cookies etc.

Avoid testing third-party dependencies

Only test what you control. Don’t try to test links to external sites or third party servers that you do not control.

Testing with a database

If working with a database then make sure you control the data. Test against a staging environment and make sure it doesn’t change.

Best Practice

Use locators

Find elements on the webpage using locators. Locators come with auto waiting and retry-ability

Use chaining and filtering

Locators can be chained to narrow down the search to a particular part of the page.

Prefer user-facing attributes to XPath or CSS selectors

Your DOM can easily change so having your tests depend on your DOM structure can lead to failing tests

Generate locators

Playwright test generator will look at your page and figure out the best locator, prioritizing role, text and test id locators.

Use web first assertions

Assertions are a way to verify that the expected result and the actual result matched or not.

Debugging

- The VS Code extension - gives you a great developer experience when writing, running, and debugging tests.
- The test generator can generate tests and pick locators for you.
- The trace viewer gives you a full trace of your tests - you can view the timeline, inspect DOM snapshots for each action, view network requests and more.
- The UI Mode let's you explore, run and debug tests with a time travel experience complete with watch mode - run, view, watch and debug each test.
- Typescript in Playwright works out of the box and gives you better IDE integrations. Your IDE will show you everything you can do and highlight when you do something wrong.
- You can share and review test trace - `playwright show-trace trace.zip` https://trace.playwright.dev/

Fixtures

Fixture	Type	Description
page	Page	Isolated page for this test run.
context	BrowserContext	Isolated context for this test run. The page fixture belongs to this context as well. Learn how to configure context.
browser	Browser	Browsers are shared across tests to optimize resources. Learn how to configure browser.
browserName	string	The name of the browser currently running the test. Either chromium, firefox or webkit.
request	APIRequestContext	Isolated APIRequestContext instance for this test run.

Browser, Context and Page

```
// Creates a new browser context. It won't share cookies/cache with other browser contexts.
const alice = browser.newContext()
const bob = browser.newContext()

// Openes new tab
const alicePage = await alice.newPage();
const bobsPage = await bob.newPage();

// Navigation
await bobsPage.goto('https://fakechat.com');
await alicePage.goto('https://fakechat.com');

// Listen for tab to be opened
const messagePopupPage = bobsPage.waitForEvent('popup');

// Send message from alice
await alicePage.getByRole('link',{ name:'Message Bob' }).click()

const popup = await messagePopupPage;

// Recieve message from alice
expect(await popup.getByText('message from alice')).toBeVisible()
```

Web First Locators

- `page.getByRole()` - by ARIA accessibility attributes.
- `page.getByText()` - by text content.
- `page.getByLabel()` - a form control by associated label's text.
- `page.getByPlaceholder()` - by placeholder.
- `page.getByAltText()` - by its text alternative.
- `page.getByTitle()` - an element by its title attribute.
- `page.getByTestId()` - an element based on its data-testid attribute (other attributes can be configured).

Use the code generator to generate a locator, and then edit it as you'd like.

```
await page.getByPlaceholder('User Name')
    .fill('John');

await page.getByLabel('Password')
    .fill('secret-password');

await page.getByTestId('a-special-button').click();

const buttonLocator page.getByRole('button', {
    name: 'Sign in'
})

buttonLocator.hover();
buttonLocator.click();

await expect(page.frameLocator('#my-frame')
    .getByText('Welcome, John!'))
    .toBeVisible();
```

Actions

- `locator.click()` - will click the label and automatically focus the input field;
- `locator.fill()` - will fill the input field;
- `locator.inputValue()` - will return the value of the input field;
- `locator.selectText()` - will select text in the input field;
- `locator.setInputFiles()` - will set files for the input field with type=file;
- `locator.selectOption()` - will select an option from the select box.

Assertions

Assertions with convenience methods for creating assertions that will wait and retry until the expected condition is met.

```
// Page Assertions
await expect(page).toHaveURL(/.*\/login/);
await expect(page).toHaveTitle(/.*checkout/);
await expect(page).toHaveScreenshot('image.png');

// Locator Assertions
await page.locator('#submit-button').click();
await expect(page.locator('.status')).toHaveText('Submitted');
await expect(page.getByText('Hidden text')).toBeAttached();

// A specific element is visible.
await expect(page.getByText('Welcome')).toBeVisible();
// At least one item in the list is visible.
await expect(page.getByTestId('todo-item').first()).toBeVisible();

// Generic Assertions
expect('abc').toEqual(expect.any(String));
expect([1, 2, 3]).not.toEqual(expect.arrayContaining([1, 4]));
expect({ foo: 1, bar: 2 }).toEqual(expect.objectContaining({ foo: 1 }));

// Inversion Assertions
await expect(page.getByText('Welcome')).not.toBeVisible();
```

CSS / XPath

```
await page.locator('css=button').click();
await page.locator('xpath=//button').click();

await page.locator('button').click();
await page.locator('//button').click();

// Waits for either confirmation dialog or load spinner.
await page.locator(
    `//span[contains(@class, 'spinner__loading')]|//div[@id='confirmation']`
).waitFor();
```

Extended CSS selectors

```
await page.locator('article:has-text("Playwright")').click();
await page.locator('#nav-bar :text("Home")').click();
await page.locator('button:visible').click();
await page.locator('article:has(div.promo)').textContent();

// Clicks a <button> that has either a "Log in" or "Sign in" text.
await page.locator('button:has-text("Log in"), button:has-text("Sign in")').click();

// Fill an input to the right of "Username".
await page.locator('input:right-of(:text("Username"))').fill('value');

// Click a button near the promo card.
await page.locator('button:near(.promo-card)').click();

// Click the radio input in the list closest to the "Label 3".
await page.locator('[type=radio]:left-of(:text("Label 3"))').first().click();

// Wait until all three buttons are visible
await page.locator(':nth-match(:text("Buy"), 3)').waitFor();

// Click last button
await page.locator('button').locator('nth=-1').click();
```

Locating multiple, nested and filtering

```
const addToCartBtn = page.getByRole('button', { name: 'Add to cart' })
        .and(page.getByTestId('purchase-btn'));

const products =  page.getByRole('listitem')

expect(products.locate(page.getByRole('heading')))
    .toHaveText(['apple', 'banana', 'orange']);

const product = products
       .filter({ hasText: /Product 2/ })
    // .filter({ has: page.getByRole('heading', { name: 'Product 2' }) })


product.locate(addToCartBtn).click();

await expect(page.getByRole('listitem')
    .filter({ hasNotText: 'Out of stock' }))
    .toHaveCount(5);
```

Page Object Pattern

```
export class TodoPage {
  readonly inputBox = this.page.locator('input.new-todo');
  readonly todoItems = this.page.getByTestId('todo-item');

  constructor(public readonly page: Page) {}

  async goto() {
    await this.page.goto('https://demo.playwright.dev/todomvc/');
  }

  async addToDo(text: string) {
    await this.inputBox.fill(text);
    await this.inputBox.press('Enter');
  }

  async remove(text: string) {
    const todo = this.todoItems.filter({ hasText: text });
    await todo.hover();
    await todo.getByLabel('Delete').click();
  }

  async removeAll() {
    while ((await this.todoItems.count()) > 0) {
      await this.todoItems.first().hover();
      await this.todoItems.getByLabel('Delete').first().click();
```

Downloads

```
// Start waiting for download before clicking. Note no await.
const downloadPromise = page.waitForEvent('download');
await page.getByText('Download file').click();
const download = await downloadPromise;

// Wait for the download process to complete and save the downloaded file somewhere.
await download.saveAs('/path/to/save/at/' + download.suggestedFilename());
```

Snapshot / screenshot testing

```
const locator = page.getByRole('button');

expect(await page.textContent('.hero__title')).toMatchSnapshot('hero.txt');

await expect(locator).toHaveScreenshot('image.png');
```

```
npx playwright test --update-snapshots
```

API Mocking

```
test("mocks a fruit and doesn't call api", async ({ page }) => {
  // Mock the api call before navigating
  await page.route('*/**/api/v1/fruits', async route => {
    const json = [{ name: 'Strawberry', id: 21 }];
    await route.fulfill({ json });
  });
  // Go to the page
  await page.goto('https://demo.playwright.dev/api-mocking');

  // Assert that the Strawberry fruit is visible
  await expect(page.getByText('Strawberry')).toBeVisible();
});
```

Authentication

```
const authFile = 'playwright/.auth/user.json';

setup('authenticate', async ({ page }) => {
  // Perform authentication steps. Replace these actions with your own.
  await page.goto('https://github.com/login');
  await page.getByLabel('Username or email address').fill('username');
  await page.getByLabel('Password').fill('password');
  await page.getByRole('button', { name: 'Sign in' }).click();
  // Wait until the page receives the cookies.
  //
  // Sometimes login flow sets cookies in the process of several redirects.
  // Wait for the final URL to ensure that the cookies are actually set.
  await page.waitForURL('https://github.com/');
  // Alternatively, you can wait until the page reaches a state where all cookies are set.
  await expect(page.getByRole('button', { name: 'View profile and more' })).toBeVisible();

  // End of authentication steps.

  await page.context().storageState({ path: authFile });
});
```

Session storage

```
// Get session storage and store as env variable
const sessionStorage = await page.evaluate(() ⇒ JSON.stringify(sessionStorage));
fs.writeFileSync('playwright/.auth/session.json', sessionStorage, 'utf-8');

// Set session storage in a new context
const sessionStorage = JSON.parse(fs.readFileSync('playwright/.auth/session.json', 'utf-8'));
await context.addInitScript(storage ⇒ {
  if (window.location.hostname ⩵ 'example.com') {
    for (const [key, value] of Object.entries(storage))
      window.sessionStorage.setItem(key, value);
  }
}, sessionStorage);
```

API Testing

```
test('should create a bug report', async ({ request }) => {
  const newIssue = await request.post(`/repos/${USER}/${REPO}/issues`, {
    data: {
      title: '[Bug] report 1',
      body: 'Bug description',
    }
  });
  expect(newIssue.ok()).toBeTruthy();

  const issues = await request.get(`/repos/${USER}/${REPO}/issues`);
  expect(issues.ok()).toBeTruthy();
  expect(await issues.json()).toContainEqual(expect.objectContaining({
    title: '[Bug] report 1',
    body: 'Bug description'
  }));
});
```

Request context

```
test.beforeAll(async ({ playwright }) => {
  apiContext = await playwright.request.newContext({
    // All requests we send go to this API endpoint.
    baseURL: 'https://api.github.com',
    extraHTTPHeaders: {
      // We set this header per GitHub guidelines.
      'Accept': 'application/vnd.github.v3+json',
      // Add authorization token to all requests.
      // Assuming personal access token available in the environment.
      'Authorization': `token ${process.env.API_TOKEN}`,
    },
  });
});
```

Api Configuration

```
import { defineConfig } from '@playwright/test';
export default defineConfig({
  use: {
    // All requests we send go to this API endpoint.
    baseURL: 'https://api.github.com',
    extraHTTPHeaders: {
      // We set this header per GitHub guidelines.
      'Accept': 'application/vnd.github.v3+json',
      // Add authorization token to all requests.
      // Assuming personal access token available in the environment.
      'Authorization': `token ${process.env.API_TOKEN}`,
    },
  }
});
```

JavaScript

Refresher

+ ES6

Built-in objects

- Number, Nan, Infinity, undefined, null
- eval(), isFinite(), isNaN(), parseFloat(), parseInt(), decodeURL(), encodeURL()
- Object, Function, Boolean, Error(s)
- Number, Math, Date
- String, RegExp
- Array, Object
- JSON
- console

Flow-control instructions

```
if( some_condition == true ){
  // ...
} else {
  // ...
}
```

```
if( some_condition )
operacja ;
```

```
var result = some_condition == true? 'Truthy' : 'Falsy';
```

Switch

```
switch( some_expression ){

  case "some value":
    // ...
  break;

  default:
    // ...
}
```

While / Do While loop

```
while( some_condition ){
  // Skip to next iteration
  continue;
  // Break out of the loop
  break;
}
```

```
do {
  // Skip to next iteration
  continue;
  // Break out of the loop
  break;

} while( some_condition )
```

For Loops

```
for(var i; i ⩽ 10; i++) {
  // Skip to next iteration
  continue;
  // Break out of the loop
  break;
}
// i == 10
```

```
var array = [1,2,3];

for( var i in array ) {
  console.log( i, array[i] )
}
// i == 2
```

Iterating over object keys

```
var obj = { a:1, b: 2};

for(var key in obj){

  // Checking if key belongs to object
  // or to one of its prototype parents
  if(obj.hasOwnProperty(key)) {

    console.log(key)

  }
}
```

Operators

```
// Mathematical operators
1 - 2 + 3 * 4 / 5

// Concatenating strings
"ala" + "ma" + "kota"

// Logical operators
true && false || !true

// Binary operators
1<<2  1>>2 ~1

// Comparison operators
== ≠ ⩽ ⩾ === !===

// Type operators
typeof x
x instanceof X
```

Comparison operators

```
// Comparison with typecase
"123" == 123 // true

// Strict Comparison (no typecast)
"123" === 123 // false
```

Type conversion

```
parseFloat("123.564") // 123.564

parseInt("123.8") // 123

(123).toString() // "123"

(123).toString(2) // "1111011"

(-2.345).toFixed(2) // "-2.35"

"The answer is " + 42 // "The answer is 42"
"37" + 7 // "377"
"37" - 7 // 30
  + "37" // 37

// Beware of how floating points are stored in memory:
0.1 + 0.2 // 0.30000000000000004
```

Functions

Named functions - function declaration

```
function add(a,b){
  return a + b
}
```

Anonymous function - function expression

```
var add = function(a,b){
  return a + b
}
```

Imediately Invoked Function

```
(function(outerParam){
  var innerVariable = "123";

  console.log( outerParam + innerVariable )
})("run with this param")
```

Closures

```
var variable = "value"

function function(parameter){
  return parameter + variable
}

function("from closure")
// "value from closure"
```

Variable scope and hoisting

JavaScript variables doesn’t jave block scope, but **function scope**:

```
if ( true ) {
  var x = 5;
}
console.log(x); // 5
```

We can use variables that are declared later in code - its so called **"hoisting"**:

```
console.log(y); // exception: y is not defined

console.log(x); // undefined
var x = 5;
```

Functional programming

Build in functional array methods:

```
Array.prototype.every
Array.prototype.some
Array.prototype.forEach
Array.prototype.map
Array.prototype.filter
Array.prototype.reduce
Array.prototype.reduceRight
```

Function combinators Function is first class member - can be passed as variable:

```
function combine( f, g ){
  return function(x){
    return f( g( x ) )
  }
}
```

`this` context and context binding

```
function WhatIsThis(){
  console.log(this)
}

WhatIsThis() // Window

new WhatIsThis() // Object()

var obj = { test: WhatIsThis, option: 123 }
obj.test() // { test: WhatIsThis, option: 123 }

WhatIsThis.apply({ value: '42' }) // { value: '42' }
```

Obiect oriented programming

```
function Person(name){

  this.name = name;

  this.sayHello = function(){
    return "Hi, I am " + this.name
  }
}

var alice = new Person('Alice')

alice.sayHello() // "Hi, I am Alice"
```

Prototypal programming

```
function Person(name){
  this.name = name;
}

Person.prototype.sayHello = function(){
  return "Hi, I am " + this.name
}

var bob = new Person('Bob')
bob.sayHello() // "Hi, I am Bob"
```

Prototypal inheritance

```
function Employee(name, salary){
  // this.name = name;
  Person.apply(this, arguments)
  this.salary = salary
}
Employee.prototype = Object.create(Person.prototype)

Employee.prototype.doWork = function(){
  return "Done. I would like my "  +this.salary
}

var tom = new Person('Tom',1200)

tom.sayHello() // "Hi, I am Tom"
tom.doWork() // "Done. I would like my 1200"
```

Exception handling

```
try {
  throw new Error('Upss ... ');

} catch(err) {
  throw new Error('Failed again ;-( ');

} finally {
  alert('Giving up. Unhandled Error')

}
```

Timed / Asynchronous Events

```
var handler = setTimeout( function(){
  // Call this  "not sooner than" ~16 miliseconds
}, 16)
// handler == 1

// Stop the timeout
clearTimeout(handler)
```

```
var handler = setInterval( function(){
  // Call this *around* every 1000ms / 1 second
}, 1000)

// Stop interval
clearInterval(handler)
```

Async'hronousness

What will happen? In which order? How much will it take?

```
document.onkeyup = (e) ⇒ console.log(e.key)

console.log(1)

setTimeout(() ⇒ {
  console.log(2)
}, 0)

Promise.resolve(3).then(console.log)

console.log(4)

start = Date.now()
while( start + 5000 > Date.now() ) {}

console.log(5)
```

The asynchronous code is NOT performed "in order" it appears in

Working with asynchronous code

Asynchronous operation - executes as soon as main JavaScript thread is idle

```
setTimeout(()⇒{
  console.log('First')

},0)

console.log('Last')

// Last
// First
```

Callback - function to be called

To get data out of async operation we provide a function as a parameter.

Function gets called inside with data once operation completes and data is available.

```
function echo(message, callback){
  setTimeout(function(){
    callback('  ...  ' + message)
  }, 1500)
}

echo('Anybody there?', function(response){
  console.log(response)
})

// around 1.5 second later:
// "  ...  Anybody there?"
```

XMLHttpRequest

```
let url = 'https://jsonplaceholder.typicode.com/'

function fetchUserById(id, callback) {
    var xhr = new XMLHttpRequest();

    xhr.open("GET", url + "users/" + id);

    xhr.onloadend = function (event) {
        var res = this.responseText;
        data = JSON.parse(res);
        callback(data);
    };
    xhr.send();

    // return ??
}
```

Pyramid of Doom

Callback Hell

```
function reziseFiles(src, callback){
  // Search files
  fs.readdir(src, (err, files) => {
    if (err) return callback(err,null)
    files.forEach( (filename, fileIndex) => {

      // Get file size
      gm(src + filename).size( (err, size) => {
        if (err) return callback(err,null)

        // Aspect
        aspect = (size.width / size.height)
          widths.forEach((width, widthIndex) => {
            height = Math.round(width / aspect)
            destPath = dest + 'w' + width + '_'

            // Resize
            this.resize(width, height)
            .write(destPath + filename, (err) => {
              if (err) return callback(err,null)
            })
          })
      })
```

Promise

- Asynchronous operations container

Promises allows to avoid nested callback functions. Promise is a "future value container", which allows to chain operations over data thats available asynchronously

```
var lazyVal = new Promise(function (resolve, reject) {
  setTimeout(function () {
    resolve("Lazy Value");
  }, 2000);
});

lazyVal
  .then(function addBang(value) {
    return value + "!";
  })
  .then(function render(valueWithBang) {
    console.log(valueWithBang);
  })
  .catch(function handleError(err) {
    console.error("Promise rejected with " + err);
  });
```

Fetch API

Modern HTTP communication with Promises API

```
let url = 'https://jsonplaceholder.typicode.com/'

let p = fetch(url + 'users/?q=a', {
    method:'GET',
    headers:{
        'Content-Type':'application/json'
    }
});

p.then((res) => {
    res.json().then((data) => {
        console.log(data)
    })
})
```

Nested Promises

and Pyramid of Doom Strikes Again ...

```
let url = 'https://jsonplaceholder.typicode.com/'

const p = fetch(url+'users/?q=a');
p.then((res) => {
    res.json().then((data) => {
        console.log(data)

        p = fetch(url+'users/' + data[0].id)
        .then((res) => {
            res.json().then((data) => {
                console.log(data)
            })
        })
    })
})
```

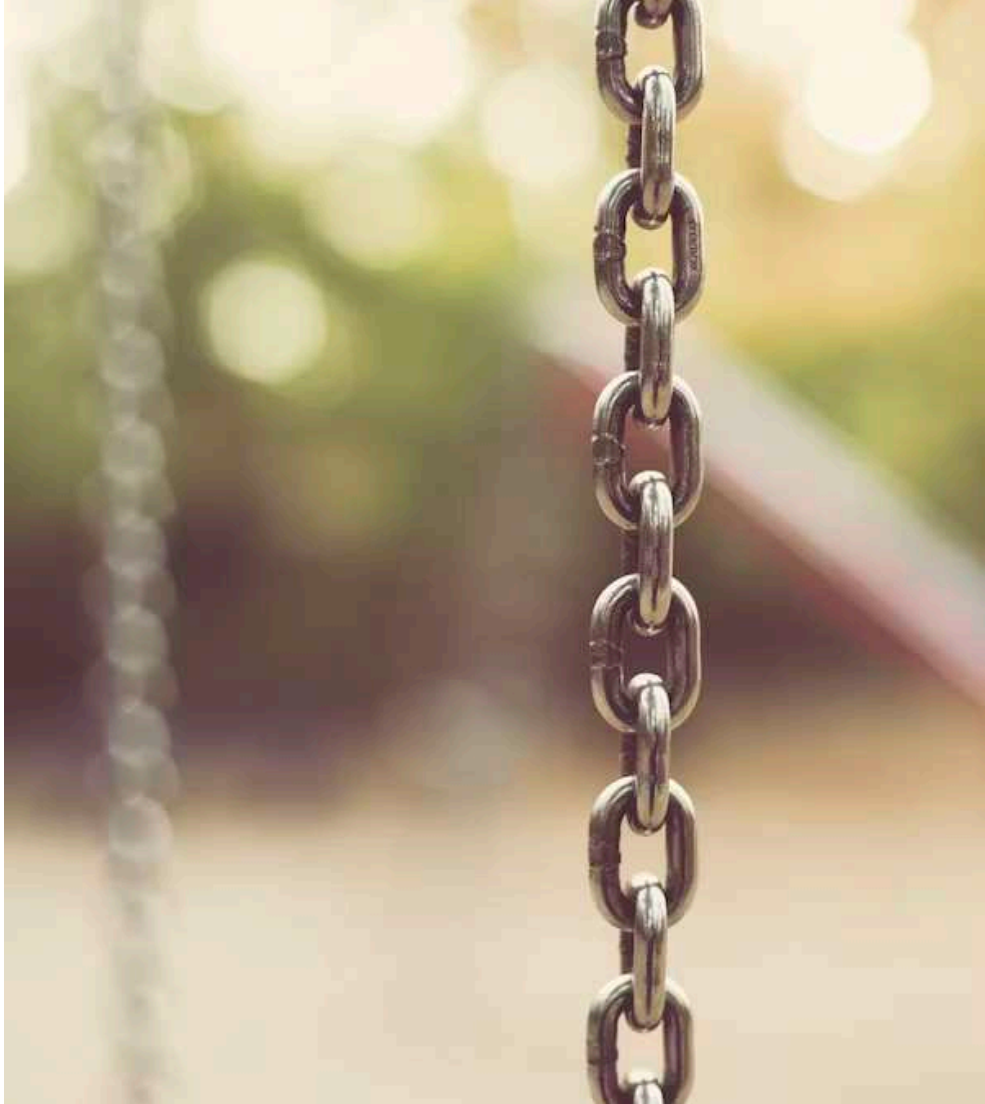
Promise Chain

```
let url = 'https://jsonplaceholder.typicode.com/'

findUsers = query => fetch(url+'users/?q='+query)
  .then( res  => res.json() )

fetchUser = id => fetch(url+'users/' + id)
  .then( res  => res.json() )

searchUsers('anne')
 .then( users => fetchUser(users[0].id) )
 .then( user => console.log(user))
```



Async / Await

and async try / catch

```
async function createNewDoc() {
  // post a new doc
  let response = await db.post({});
  // find by id
  return db.get(response.id); // auto-await return
}

async function main() {
  try {
    let doc = await createNewDoc();
    console.log(doc);
  } catch (err) {
    console.log(err);
  }
}

main() // returns Promise, so:
.then(()⇒console.log('success'))
.catch((error)⇒console.log(error))
```

Async/Await

Just add "async" prefix to wrap function with a promise and add "await" for function to wait for promises inside

```
async function iReturnAPromise() {
  const result = await someAsyncOperation();
  const result2 = await someAsyncOperation();

  // Will return the result wrapped in a promise!
  return result + result2;
}

iReturnAPromise().then(console.log)
```

Async/Await

```
async function iReturnAPromise() {
  try {
    const result = await someAsyncOperation();
    const result2 = await someAsyncOperation();
    return result + result2;

  // It will catch asynchronous (awaited!) exceptio
  } catch (err) {
    throw new Error("Upss..");
  }
}
```

Asynchronous iterator

```
function idMaker() {
  let index = 0;
  return {
    next: async function () {
      return {
        value: index++,
        done: false,
      };
    },
  };
}

let it = idMaker();

console.log((await it.next()).value); // '0'
console.log((await it.next()).value); // '1'
console.log((await it.next()).value); // '2'
```

Iterables and Iterators

```
const myInfiniteIterator = {
  value: 0,
  [Symbol.asyncIterator]: function () {
    return {
      next: async () => {
        return {
          value: this.value++,
          done: false
        };
      },
    };
  },
};
```

Generator async

```
async function* idMaker() {
  let index = 0;
  while (true) {
    yield index++;
  }
}

const it = idMaker();

console.log((await gen.next()).value); // '0'
console.log((await gen.next()).value); // '1'
console.log((await gen.next()).value); // '2'

// Or simply:
for await (let i of it) {
  if (i > 10) break;
}
```

Selected topics from ES6

** "EcmaScript 2015" ** Standard - known as ES6

Another update to the JavaScript standard after EcmaScript 5. It is available in almost all new browsers except Internet Explorer 11 and a few others.

Current browser support can be checked at: http://kangax.github.io/compat-table/es6/

New kinds of variables

```
//block variables - let
if(true){
  let localVar = 1
}
localVar // Exception - local not defined

// stałe - const
const PARAM = 123
const obj = {value:123}

PARAM = "change"
// TypeError: Assignment to constant variable.

obj.value = "ups!" // Written by reference!
```

Template Strings

```
 // Strings are single-line - break lines
 // requires an escape character before enter

var tekst = " \
";
```

Since ES6 we have multi-line strings with variable interpolation!

```
let variable = 123;
let tekst = ` Title:
  Second line "${ variable + variable }"
`;
// Title:
//    Second line "246"
//
```

Operator spread

```
function f(x, y = 12) {136 / 5000
    // default y value (if y === undefined)
    return x + y;
}
f(3) === 15;

function f(x, ...y) {136 / 5000
  // y is an array of the remaining values
  return x * y.length;
}
f(3, "hello", true) === 6;

function f(x, y, z) {
 return x + y + z;
}
// pass each element of the array separately
f(...[1, 2, 3]) === 6;
```

Arrow functions

```
  const func = (a, b) => {
    return a + b;
  }
```

One parameter allows you to omit the parentheses. The expression in the function body allows you to omit return and braces

```
  const multiply3 = x => x * 3;
  multiply3(2) == 6
```

Lexical `this`

```
var bob = {
  _name: "Bob",
  _friends: [],
  printFriends() {
    this._friends.forEach(f =>
      console.log(this._name + " knows " + f));
  }
};
```

They do not replace **function**. Cannot be used with **new**

Default function parameters

```
 function f(x, y=12) {
  // y is 12 if not passed (or passed as undefined)
  return x + y;
}
f(3) == 15
```

Destructuring

```
// list matching
var [a, , b] = [1,2,3];

// swap values
[ b, a ] = [a, b ]

// object matching
var { op: a, lhs: { op: b }, rhs: c }
 = getASTNode() // i.e. { op: ‘a’,  lhs: {op: ‘b’ }, rhs: ‘c’ }}

// object matching shorthand
var {op, lhs, rhs} = getASTNode()

// Can be used in parameter position
function g({name: x}) {
 console.log(x);
}
g({name: 5})
```

Maps and Sets

```
// Maps allows you to use any reference as a key
var map = new Map()
map.set( obj, 'secret_value' )

 // The sets check the uniqueness of the added values
var set = new Set()

set.add('x')
// Set(1) {"x"}
set.add('x')
// Set(1) {"x"}
```

Symbols

```
const SPECIAL_TOKEN = Symbol('TOKEN_NAME')

 // Symbol Can Be Used As Key
 map.set (SPECIAL_TOKEN, 'secret')

 // Symbol value is just a description
 map.get ('TOKEN_NAME') == undefined

 // The reference is a unique key,
 // which cannot be "faked
 map.get (SPECIAL_TOKEN) == 'secret'

 // Predefined symbols let you run hidden mechanisms
 // eg Symbol.iterator
```

Iterators

```
let fibonacci = {
  [Symbol.iterator]() {
    let pre = 0, cur = 1;
    return {
      next() {
        [pre, cur] = [cur, pre + cur];
        return { done: false, value: cur }
      }
    }
  }
}

for (var n of fibonacci) {
  // truncate the sequence at 1000
  if (n > 1000)
    break;
  console.log(n);
}
```

Generators

```
var fibonacci = {
  [Symbol.iterator]: function*() {
    var pre = 0, cur = 1;
    for (;;) {
      var temp = pre;
      pre = cur;
      cur += temp;
      yield cur;
    }
  }
}

for (var n of fibonacci) {
  // truncate the sequence at 1000
  if (n > 1000)
    break;
  console.log(n);
}
```

Classes - "syntax sugar" on prototype

```
class Person{

  constructor(name){
    this.name = name;
  }

  static member = 123

  method(){
    // ...
  }
  field = "value"
}
var pete = new Employee('Pete', 1200 )
pete instanceof Person // true
```

Inheritance (prototypes)

```
class Employee extends Person{

  constructor(name,salary){
    super(name)
  }

  method(){
    super.method()
  }
}

var tom = new Employee('Tom', 1200 )

tom instanceof Employee // true
tom instanceof Person // true
```

Modules

```
// lib/math.js
export function sum(x, y) {
  return x + y;
}
export var pi = 3.141593;

export default { sum, pi };

// app.js
import * as math from "lib/math";
// default:
// import math from "lib/math";

console.log("2π = " + math.sum(math.pi, math.pi));

// otherApp.js
import {sum, pi} from "lib/math";

console.log("2π = " + sum(pi, pi));
```

Working with a browser

Object model of the document - home

```
// Active tab in the browser
// "Global" object Window:
window;

// Main document or e.g. iframe
document;

// Header
document.head; // <head> ... </head>

// Main body
document.body; // <body> ... </body>
```

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width,
    <title>Document</title>
  </head>
  <body>
    <div class="container">
      <div class="row">
        <div class="col">
          <h1>Welcome</h1>
          <p>
            Lorem ipsum dolor sit amet consectetur adip
            laborum, placeat ad labore impedit aspernat
            vero qui, rerum, accusamus eos incidunt, de
            asperiores!
          </p>
        </div>
      </div>
    </div>
    <script src="scripts/main.js"></script>
    <script src="scripts/plugins.js"></script>
  </body>
</html>
```

DOM API

Finding elements

```
document.getElementById("unikalny")
// <div id="unique">
```

"Live" selectors - follow the changes to the house:

```
document.getElementsByTagName("div")
// [<div>, <div>, <div>]

document.getElementsByClassName(".klas")
// [<div class="klas">, <div class="klas">]
```

Static selectors

```
document.querySelector(".list .items:first-child")
// <li class="item">

document.querySelectorAll(".list .items")
// [<ul class="items">]
```

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width,
    <title>Document</title>
  </head>
  <body>
    <div class="container">
      <div class="row">
        <div class="col">
          <h1>Welcome</h1>
          <p>
            Lorem ipsum dolor sit amet consectetur adip
            laborum, placeat ad labore impedit aspernat
            vero qui, rerum, accusamus eos incidunt, de
            asperiores!
          </p>
        </div>
      </div>
    </div>
    <script src="scripts/main.js"></script>
    <script src="scripts/plugins.js"></script>
  </body>
</html>
```

DOM API

Modifying elements

The properties DOM can be modified through attributes or directly:

```
element.id = "123";
element.setAttribute("id", 123);
element; // <div id="123">

element.setAttribute("style",
                     "border: 1px solid red");

element.style.borderColor = "blue";

element; // <div style="border:1px solid blue">
```

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width,
    <title>Document</title>
  </head>
  <body>
    <div class="container">
      <div class="row">
        <div class="col">
          <h1>Welcome</h1>
          <p>
            Lorem ipsum dolor sit amet consectetur adip
            laborum, placeat ad labore impedit aspernat
            vero qui, rerum, accusamus eos incidunt, de
            asperiores!
          </p>
        </div>
      </div>
    </div>
    <script src="scripts/main.js"></script>
    <script src="scripts/plugins.js"></script>
  </body>
</html>
```

DOM API

Creating elements

```
// Object oriented creation of elements
var of elements("div");

// Creating a structure through HTML
div.innerHTML = "<p>Treść akapitu w divie</p>";

// Building a structure objectively
var paragraph = document.createElement("p");
p.textContent = "Other paragraph";

div.appendChild(paragraph);
document.body.appendChild(div);
```

Method with a template

```
<template id="tpl"> <b>Alice</b> has a cat </templa
<script>
  const tpl = document.getElementById('tpl')
  const elem = tpl.content.clone(true)
  targetEl.parent.insertBefore(targetEl, elem)
</script>
```

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width,
    <title>Document</title>
  </head>
  <body>
    <div class="container">
      <div class="row">
        <div class="col">
          <h1>Welcome</h1>
          <p>
            Lorem ipsum dolor sit amet consectetur adip
            laborum, placeat ad labore impedit aspernat
            vero qui, rerum, accusamus eos incidunt, de
            asperiores!
          </p>
        </div>
      </div>
    </div>
    <script src="scripts/main.js"></script>
    <script src="scripts/plugins.js"></script>
  </body>
</html>
```

DOM API

Listening to events

```
element.onclick = function (event) {
  console.log(event);
}

element.addEventListener("click", (event) => {
  console.log(event);
});
```

Adding and removing "listeners" of events

```
const handler = (event) => {
  console.log(event.target.value);
}
element.addEventListener("keyup", handler);

element.removeEventListener("keyup", handler);
```

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width,
    <title>Document</title>
  </head>
  <body>
    <div class="container">
      <div class="row">
        <div class="col">
          <h1>Welcome</h1>
          <p>
            Lorem ipsum dolor sit amet consectetur adip
            laborum, placeat ad labore impedit aspernat
            vero qui, rerum, accusamus eos incidunt, de
            asperiores!
          </p>
        </div>
      </div>
    </div>
    <script src="scripts/main.js"></script>
    <script src="scripts/plugins.js"></script>
  </body>
</html>
```

Bubbled events

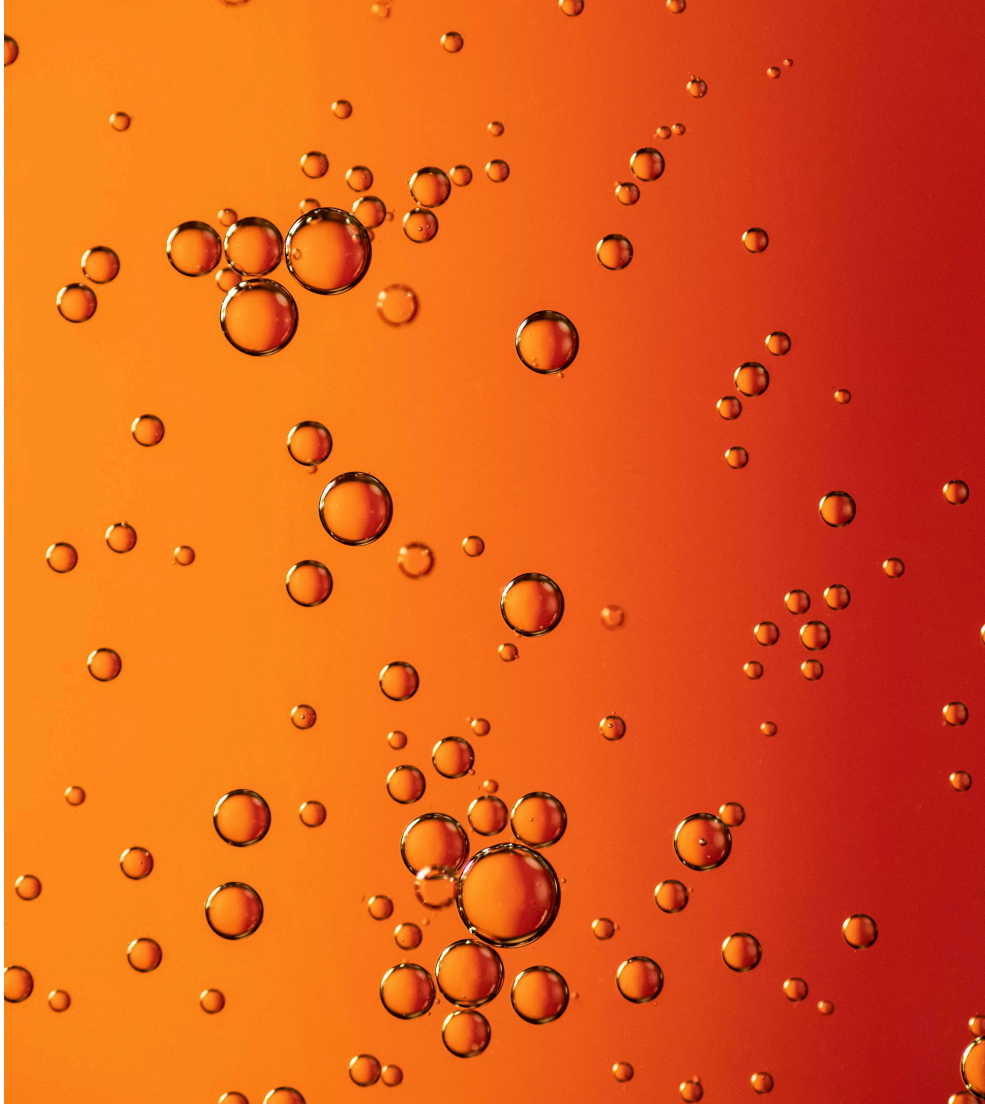
```
<div id="container">
   <div class="list-group">
     <div id="child" class="list-group-item">
      <!— Użyszkodnik Kliknął tutaj —>
```

Direct

```
// we can catch the event from the clickdocument.ge
  .addEventListener("click", (event) => {
  // event.target == child;
  // event.currentTarget == child;
  // event.currentTarget == event.target;
});
```

Delegated

```
// or above the element that bubbles the event
document.getElementById("container")
  .addEventListener("click", (event) => {
  // event.bubbling == true
  // event.target == child;
  // event.currentTarget == container;
  // event.currentTarget! = event.target;
});
```



Attributes data-*

and dataset

```
<div data-user-id="1">...</div>
```

```
elem.getAttribute("data-user-id");
// "1"

elem.dataset;
// DOMStringMap {userId: "1"}

elem.dataset.userId;
// "1"
```

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width,
    <title>Document</title>
  </head>
  <body>
    <div class="container">
      <div class="row">
        <div class="col">
          <h1>Welcome</h1>
          <p>
            Lorem ipsum dolor sit amet consectetur adip
            laborum, placeat ad labore impedit aspernat
            vero qui, rerum, accusamus eos incidunt, de
            asperiores!
          </p>
        </div>
      </div>
    </div>
    <script src="scripts/main.js"></script>
    <script src="scripts/plugins.js"></script>
  </body>
</html>
```

Match through the CSS selector

Find children who meet the selector:

```
element.querySelectorAll(".child");
// NodeList <Element []>
```

Whether the element is fulfilled by the CSS selector:

```
element.matches("[data-user-id]");
// true
```

Return an element or its ancestor that the CSS selector meets:

```
element.closest("[data-user-id]");
// element, element.parent ... parent.parent
```

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width,
    <title>Document</title>
  </head>
  <body>
    <div class="container">
      <div class="row">
        <div class="col">
          <h1>Welcome</h1>
          <p>
            Lorem ipsum dolor sit amet consectetur adip
            laborum, placeat ad labore impedit aspernat
            vero qui, rerum, accusamus eos incidunt, de
            asperiores!
          </p>
        </div>
      </div>
    </div>
    <script src="scripts/main.js"></script>
    <script src="scripts/plugins.js"></script>
  </body>
</html>
```

Delegating events

```
<div class="container">
  <!-- When we want to know which item -->
  <div class="list-item" data-id="123">
    <div class="trigger">
      <!-- But click is here! -->
```

```
// We listen to the level 'container':
document
  .querySelector('.container')
  .addEventListener('click', e => {

    // We find an element withdata-id
    const realTarget = e.target
                        .closest('[data-id]')

    if(realTarget){
      const id = realTarget.dataset.id; // !
    }
  })
```

jQuery - This is the JavaScript library designed to simplify the passage and manipulation of the HTML DOM tree

And also to support events, CSS, Ajax and other browser APIs ...

It is free open source software that uses the Liberal MIT license.

As of August 2022, 77% of 10 million of the most popular websites use JQuery. —Wikipedia https://api.jquery.com/

```
<script src="https://ajax.googleapis.com/ajax/libs/jquery/3.7.1/jquery.min.js"></script>
```

PS:

- https://youmightnotneedjquery.com/
- https://vanilla-js.com/ 😉

jQuery

selectors and operators

```
$(".item").attr("href",'https://www.com/');
// parent
$("#w1:parent").html('Hey <em>yo</em>')
// contains
$("h4:contains(Course)").text('Opis');

$('h2').parent().find('h1').css({
  backgroundColor: 'gray',
  fontSize: '24px'
});

// jQuery addClass
$('.button').addClass('active');

// jQuery removeClass
$('.button').on('mouseleave', evt => {
   let e = evt.currentTarget;
   $(e).removeClass('active');
});

// jQuery .toggleClass
$('.choice').toggleClass('highlighted');
```

ATTRIBUTES / CSS

Attributes
- .attr()
- .prop()
- .removeAttr()
- .removeProp()
- .val()

CSS
- .addClass()
- .css()
- jQuery.cssHooks
- jQuery.cssNumber
- jQuery.escapeSelector()
- .hasClass()
- .removeClass()
- .toggleClass()

Dimensions
- .height()
- .innerHeight()
- .innerWidth()
- .outerHeight()
- .outerWidth()
- .width()

Offset
- .offset()
- .offsetParent()
- .position()
- .scrollLeft()
- .scrollTop()

Data
- jQuery.data()
- .data()
- jQuery.hasData()
- jQuery.removeData()
- .removeData()

MANIPULATION

Copying
- .clone()

DOM Insertion, Around
- .wrap()
- .wrapAll()
- .wrapInner()

DOM Insertion, Inside
- .append()
- .appendTo()
- .html()
- .prepend()
- .prependTo()
- .text()

DOM Insertion, Outside
- .after()
- .before()
- .insertAfter()
- .insertBefore()

DOM Removal
- .detach()
- .empty()
- .remove()
- .unwrap()

DOM Replacement
- .replaceAll()
- .replaceWith()

TRAVERSING

Filtering
- .eq()
- .even()
- .filter()
- .first()
- .has()
- .is()
- .last()
- .map()
- .not()
- .odd()
- .slice()

Miscellaneous Traversing
- .add()
- .addBack()
- ~~.andSelf()~~
- .contents()
- .each()
- .end()

Tree Traversal
- .children()
- .closest()
- .find()
- .next()
- .nextAll()
- .nextUntil()
- .parent()
- .parents()
- .parentsUntil()
- .prev()
- .prevAll()
- .prevUntil()
- .siblings()

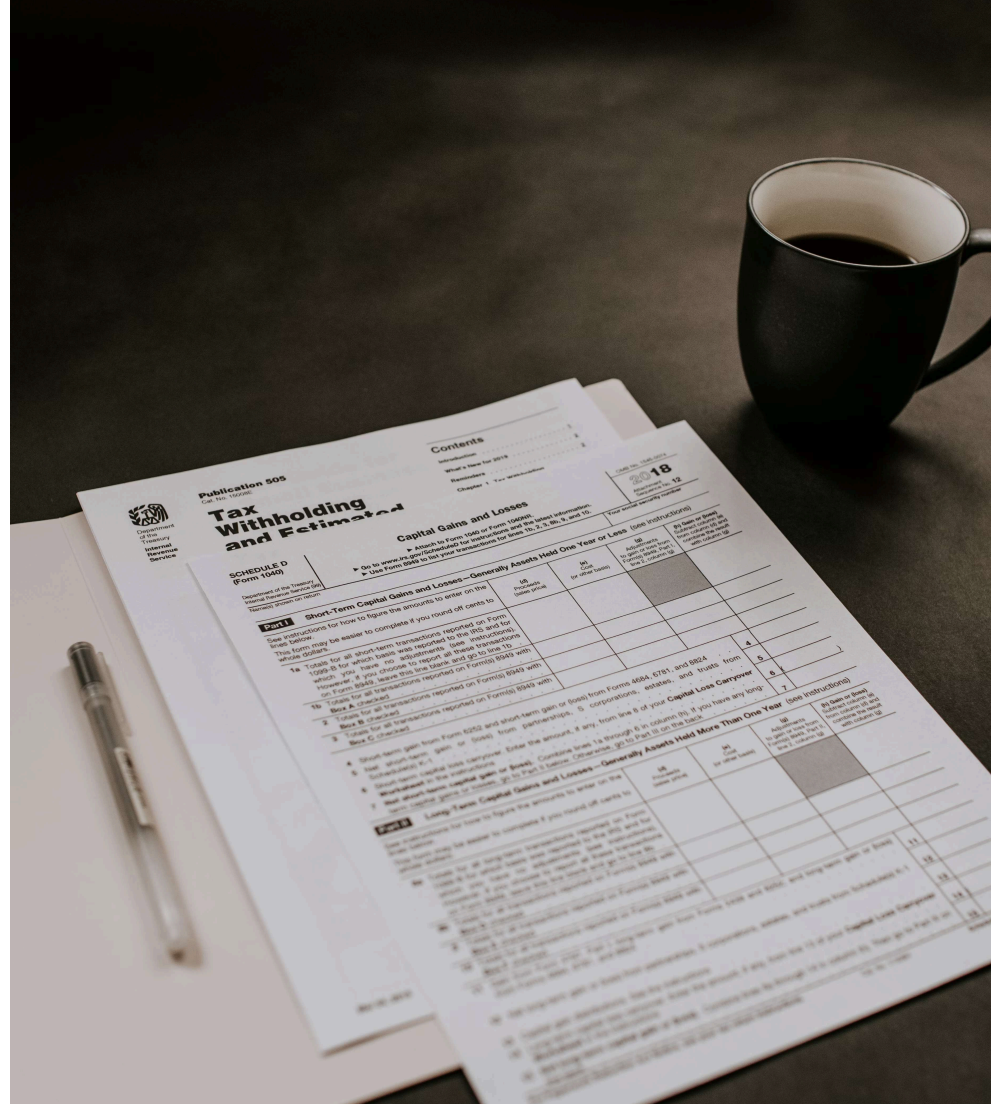
https://oscarotero.com/jquery/

Forms

One of the main interaction points between the user and the site or application.

The forms allow users to enter data that is sent to the web server to process and store them ...

... or used on the client side to immediately update the interface.



Form

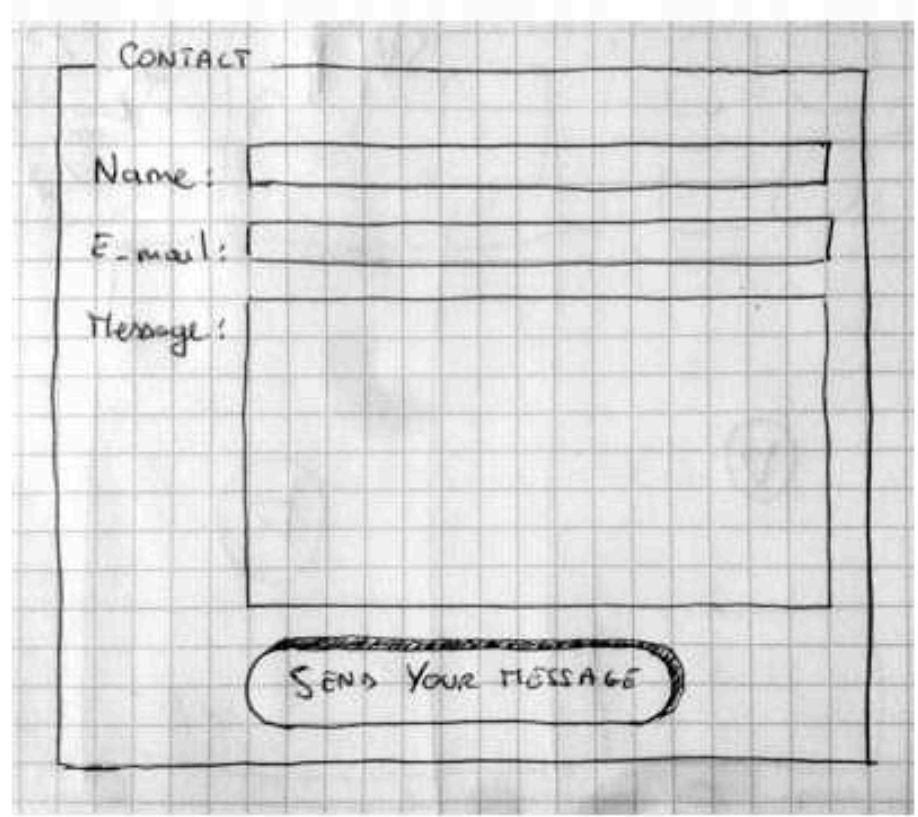
Sent by the browser to the server:

```
<form action="/server.php" method="post">…</form>
```

Supported by JavaScript in the browser:

```
<form onsubmit="handleFormSubmit()">
 <div>
     <label for="name">Name:</label>
     <input type="text" id="name" name="name" />
 </div>
 <div>
     <label for="mail">Email:</label>
     <input type="email" id="mail" name="email" />
 </div>
 <div>
     <label for="msg">Message:</label>
     <textarea id="msg" name="message"></textarea>
 </div>

 <button type="submit">Send</button>
</form>
```



Formularz

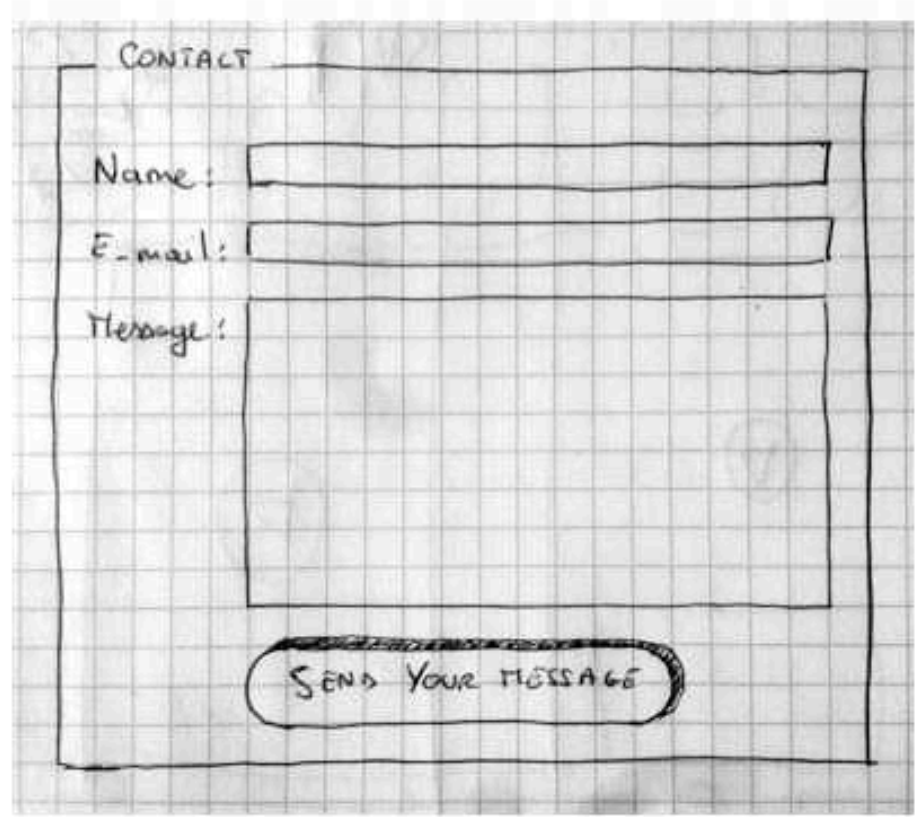
i JavaScript

```
document.querySelector('form')
.addEventListener('submit', (event) => {

    const formData = new FormData(e.target)
    formData.append("secretField", "12345");

    for (const [key, value] of formData) {
        //   ...
    }

    console.log(formData)
})
```



Upload files

in JavaScript

File transfer field

```
<input type="file"
       id="uploader"
       multiple
       accept="image/*"
       style="visibility:hidden" />
```

We can call hidden input from the outside:

```
<button onclick="uploader.click()">
    Click and upload photos
</button>

<div id="preview">
    <!— Show miniatures here —>
</div>
```

```
const uploader = document.getElementById("uploader");
const preview = document.getElementById("preview");

fileElem.onchange = function () {
  const files = uploader.files;

  for (let i = 0; i < files.length; i++) {
    const file = files[i];

    if (!file.type.startsWith("image/")) continue;


    const img = document.createElement("img");
    img.classList.add("obj");
    img.file = file;
    // Zakładając, że "preview" jest DIVem
    preview.appendChild(img);

    const reader = new FileReader();
    reader.onload = (e) => {
      img.src = e.target.result;
    };
    reader.readAsDataURL(file);
  }
}
```

TypeScript

TypeScript

TypeScript is a programming language with JS-based syntax.

It adds static and strong typing to enable us to write safer and more reliable code.

TS compiles primarily against JavaScript, but can also be used with WebAssembly (AssemblyScript) or in environments where the compilation process is imperceptible to us (ts-node, deno).

Why do we need

TypeScript?

- Static typing
- Type Inference
- Documentation and contracts
- Better Tooling (Intelisence / IDE hints)

JavaScript vs TypeScript

This is JavaScript code:

```
function test() {
    return 1 + 2;
}
```

This is TypeScript* code:

```
function test() {
    return 1 + 2;
}
```

* - without strict mode enabled

TypeScript vs JavaScript

TypeScript is, informally, a superset of JavaScript.

Any code written in JS should also be valid code in TS.

Thus, all JS elements are available in TypeScript, such as:

- classes,
- symbols,
- generators / iterators,
- Proxy,
- and others.

TypeScript vs JavaScript

With TypeScript, you can just use anything there is available in the current version of JavaScript.

Also, the developers of TypeScript decided to provide some extra features that are yet to be in JS, for example:

- decorators,
- constructor class properties,
- and others.

Installation

Install compiler globally (for current user)

```
npm install -g typescript
```

should be accessible via `$PATH` env variable, so anywhere:

```
tsc --help
```

TypeScript Compiler

Compile to disk:

```
tsc file.ts
tsc file.ts --watch
```

Adding types

```
function add(a: number, b: number) {
    return a + b;
}
add(1, 2); // Ok
add(1, "2"); // Error!
add("1", "2"); // Error!
add(); // Error!
add([], {}); // Error!
```

Strict mode

```
add(null, undefined); // Ok?
```

```
tsc --strict file.ts
```

Strict mode

Strict mode enables the following compiler options:

- **noImplicitAny** - Expressions that do not have a type will not become any, they will only cause an error
- **noImplicitThis** - this without a specific type is invalid,instead of becoming any
- **alwaysStrict** - all files are understood in ECMAScript "use strict" mode
- **strictBindCallApply** - the types of the arguments passed to bind, call and apply are more carefully checked
- **strictNullChecks** - stop accepting variables and arguments null and undefined as if they were valid values (like in the previous example)
- **strictFunctionTypes** - function arguments are not bivariate
- **strictPropertyInitialization** - all class fields must be initialized

tsconfig.json

You dont have to remember all compiler arguments. You can use tsconfig.json Generate default config and use `tsc` without extra arguments

```
tsc --init --strict
```

and then just

```
tsc
```

Playground

You can test the code examples and the build result on the square play "prepared by the creators of TypeScipt. You will find it under at:

https://www.typescriptlang.org/play/index.html

Example Web project

```
// project1.ts
const username = window.prompt("What is your name:");
window.alert(`Hello, ${username}`);
```

```
<!DOCTYPE html>
<html lang="en">
<body>
    <!-- Notice use of  ..js -->
    <script src="project1.js"></script>
</body>
</html>
```

```
tsc project1.ts
```

Example node project

```
const user = process.argv[2] || 'Stranger';
console.log(`Hello, ${user}!`);
```

```
tsc --strict node.ts
node node.js
```

If you are missing types:

```
npm install -g @types/node
```

Typescript tools

for productivity

Example node project

```
const user = process.argv[2] || 'Stranger';
console.log(`Hello, ${user}!`);
```

```
tsc --strict node.ts
node node.js
```

If you are missing types:

```
npm install -g @types/node
```

Tools for node

Compile and run:

```
npm i -g nodemon ./dist/app.js

npm i -g ts-node ./src/app.ts
npm i -g ts-node-dev ./src/app.ts
```

```
ts-node --compiler-options '{"strict": true}' file.ts
```

Do not use them in production. Check documentation for caveats!

Debugging NodeJS

```
debugger

node --enable-source-maps --inspect-port=[host:]port index.js
node --enable-source-maps --inspect[=[host:]port] index.js

ts-node-dev --respawn --enable-source-maps --inspect [--inspect-brk] -- ./path/to/app.ts
```

Example frontend - Webpack.config.js

```
const path = require("path");
module.exports = {
  entry: "./src/index.ts",
  output: {
    filename: "bundle.js",
    path: path.resolve(__dirname, "dist"),
  },
  module: {
    rules: [
      {
        test: /\.tsx?$/,
        use: "ts-loader",
        exclude: /node_modules/,
      },
    ],
  },
  resolve: {
    extensions: [".tsx", ".ts", ".js"],
  },
};
```

Example frontend - vite

```
npm create vite@latest my-super-app -- --template vanilla-ts
```

```
npm create vite@latest my-super-app -- --template react-ts
```

Example fullstack - NextJS

```
npx create-next-app@latest
```

Typescript - JavaScript Types

Primitives:

- boolean
- number
- string
- symbol
- bigint
- null
- undefined

Complex types:

- object
- Array

Typescript - extra types

- tuple,
- enum,
- void,
- any,
- Object,
- never,
- unknown.

boolean

One of the most basic types. Represents a boolean value, that is, true or false: true, false:

```
const isThisBookOkay: boolean = true;
isThisBookOkay = true
isThisBookOkay = false
```

number

The numbers in TypeScript are the same as those in JS (TypeScript only adds types).

They are floating point numbers.

There is also the use of hexadecimal, octal and binary literals:

```
const result: number = 123 + 0×beef + 0b1111 + 0o700;
// 49465
```

string

Strings are strings of characters identical to those in JS.

This is probably the most popular data type and without it you wouldn’t be able to build any applications.

We pass strings in single (') or double (") quotation marks

```
const firstName: string = "Mateusz";
const lastName: string = "Kulesza";
const hello: string = `Hello,
${firstName} ${lastName}!`;
// Hello,\nMateusz Kulesza!
```

Any

In some situations, not all type information is available or its declaration would take an inappropriate amount of effort.

In these cases, we might want to opt-out of type checking.

```
declare function getValue(key: string): any;

// OK, return value of 'getValue' is not checked
const str: string = getValue("myString");

// Any propagates down to members:
let looselyTyped: any = {};
let d = looselyTyped.a.b.c.d;
//  ^ = let d: any
```

Void

void is a little like the opposite of any: the absence of having any type at all.

You may commonly see this as the return type of functions that do not return a value:

```
function warnUser(): void {
  console.log("This is my warning message");
}
```

Declaring variables of type void is not useful because you can only assign `null`

(only if `--strictNullChecks` is not specified) or `undefined` to them

Null and Undefined

In TypeScript, both `undefined` and `null` actually have their types named undefined and null respectively. Much like `void`, they're not extremely useful on their own:

```
// Not much else we can assign to these variables!
let u: undefined = undefined;
let n: null = null;
```

By default null and undefined are subtypes of all other types. That means you can assign null and undefined to something like number.

However, when using the `--strictNullChecks` flag, null and undefined are only assignable to `unknown`, any and their respective types

This helps avoid many common errors. In cases where you want to pass in either a string or null or undefined, you can use the union type `string | null | undefined.`

Never

The never type represents the type of values that never occur. For instance, never is the return type for a function that always throws an exception or one that never returns. Variables also acquire the type never when narrowed by any type guards that can never be true.

The never type is a subtype of, and assignable to, every type; however, no type is a subtype of, or assignable to, never (except never itself). Even any isn't assignable to never.

```
// Function returning never must not have a reachable end point
function error(message: string): never {
  throw new Error(message);
}

// Inferred return type is never
function fail() { return error("Something failed"); }

// Function returning never must not have a reachable end point
function infiniteLoop(): never {  while (true) {} }
```

Object

object is a type that represents the non-primitive type, i.e. anything that is not number, string, boolean, bigint, symbol, null, or undefined.

With object type, APIs like Object.create can be better represented. For example:

```
declare function create(o: object | null): void;
// OK
create({ prop: 0 });
create(null);

create(42);
```

Array

TypeScript, like JavaScript, allows you to work with arrays of values. Array types can be written in one of two ways.

```
// Way 1: Type[]
let list: number[] = [1, 2, 3];Try
The second way uses a generic array type, Array<elemType>:

// Way 2: Array<Type>
let list: Array<number> = [1, 2, 3];

// This is NOT and Array! This is a tuple!
let notAList: [number] = [1]
```

Tuple

Tuple types allow you to express an array with a fixed number of elements whose types are known, but need not be the same.

```
// Declare a tuple type
let x: [string, number];
// Initialize it
x = ["hello", 10]; // OK


// Initialize it incorrectly
x = [10, "hello"]; // Error
```

More on tuples: https://en.wikipedia.org/wiki/Tuple

Type assertions

Type assertions are a way to tell the compiler **“trust me, I know what I’m doing.”** A type assertion is like a type cast in other languages, but it performs no special checking or restructuring of data. It has no runtime impact and is used purely by the compiler. TypeScript assumes that you, the programmer, have performed any special checks that you need.

```
// the as-syntax:
let someValue: unknown = "this is a string";

let strLength: number = (someValue as string).length;Try

//  “angle-bracket” syntax:
let someValue: unknown = "this is a string";

let strLength: number = (<string>someValue).length;
```

About Number, String, Boolean, Symbol and Object

It can be tempting to think that the types `Number`, `String`, `Boolean`, `Symbol`, or `Object` are the same as the lowercase versions recommended above. These types do not refer to the language primitives however, and almost never should be used as a type.

```
const s:String = 'text' // Wrong

const s:string = 'text' // Correct!
```

Complex types - Inline

Complex object types can be defined inline

```
const item: { name:string, value:number } = {name:'item', value: 42 }

function printItem(item:  { name:string, value:number }) {
    return item.name + ' : ' + item.value
}

// Or can be aliased for reuse
type Item = { name:string, value:number };
```

Declaring complex types - type alias vs interface

Unlike an `interface` declaration, which always introduces a named object type, a `type` alias declaration can introduce a name for any kind of type, including primitive, union, and intersection types.

```
type aString = string

type x = string | number
type y = { field: string }

interface Z { field: string }
```

Interfaces can also be used for merging declarations. On that later...

Named Interfaces

Simple interface

```
interface User{
    id: number
    name: string
    active: boolean
}
```

Optional Properties

Not all properties of an interface may be required. Some exist under certain conditions or may not be there at all. These optional properties are popular when creating patterns like “option bags” where you pass an object to a function that only has a couple of properties filled in.

```
interface SquareConfig {
  color?: string;
  width?: number;
}

const square: SquareConfig = {}
square.color // undefined
square.color.toUpperCase() // Error!

if(square.color){
    quare.color.toUpperCase() // OK
}
```

Optional Properities and Optional Chaining

At its core, optional chaining lets us write code where TypeScript can immediately stop running some expressions if we run into a `null` or `undefined`.

```
// insead of
let x = foo && foo.bar && foo.bar.baz()

// you can write
let x = foo?.bar?.baz();

// Actually it only checks for undefined or null. Not 'falsy' values like "0" or ""
let x = foo === null || foo === undefined ? undefined : foo.bar.baz();
```

Non-Null Asserion operator

A new `!` post-fix expression operator may be used to assert that its operand is non-null and non-undefined in contexts where the type checker is unable to conclude that fact.

```
let x = foo!.bar!.baz();
```

Operation `x!` produces a value of the type of x with `null` and `undefined` excluded.

Similar to the forms `<T>x` and `x as T`, the `!` non-null assertion operator is simply removed in the emitted JavaScript code.

Nullish Coalescing

You can think of this feature - the ?? operator - as a way to “fall back” to a default value when dealing with null or undefined. When we write code like

```
let x = foo ?? bar();

// instead of:
let x = foo !== null && foo !== undefined ? foo : bar();
```

Readonly properties

Some properties should only be modifiable when an object is first created.

```
interface Point {
  readonly x: number;
  readonly y: number;
}
// After the assignment, x and y can’t be changed.
let p1: Point = { x: 10, y: 20 };
p1.x = 5; // error!
// Error: Cannot assign to 'x' because it is a read-only property.
```

Readonly is for properties only. Variables use const whereas properties use readonly.

Indexable Types

Similarly to how we can use interfaces to describe function types, we can also describe types that we can "index into" like `a[10]`, or `ageMap["daniel"]`. Indexable types have an index signature that describes the types we can use to index into the object, along with the corresponding return types when indexing.

```
interface StringArray {
  [index: number]: string;
}

let myArray: StringArray;
myArray = ["Bob", "Fred"];

let myStr: string = myArray[0];
```

Extracting types

```
type UserID = User["id"];
function findUser(id: User["id"]): User { /* … */ }
```

Declaration merging

Interfaces with same name are merged allowing for extending existing types.

```
interface Xyz {
  a: number;
}

interface Xyz {
  b: string;
}

const res: Xyz = {
  a: 1,
  b: "foo",
};
```

Functions

```
// Named function
function add(x, y) {
  return x + y;
}

// Anonymous function
let myAdd = function (x, y) {
  return x + y;
};
```

Typing the function

```
function add(x: number, y: number): number {
  return x + y;
}

let myAdd = function (x: number, y: number): number {
  return x + y;
};
```

Writing the function type

We can write complete function signature 'outside'

```
let myAdd: (baseValue: number, increment: number) => number = function (
  x: number,
  y: number
): number {
  return x + y;
};
```

Function Signature as Type

When function has its type defined then arguments and return type can be inferred:

```
let myAdd2: (baseValue: number, increment: number) => number = function (x, y) {
  return x + y;
};
```

Function alias and interface

We can extract function type altogether

```
type addFunc: (baseValue: number, increment: number) => number;
let myAdd:addFunc = (x, y) => x + y;

interface SearchFunc {
  (source: string, subString: string): boolean;
  (source: string, subString: string, startPos:number): boolean; // overload
}
const mySearch: SearchFunc = (heap, needle, start) => heap.substring(needle, start);
```

Optional and default parameters

```
function buildName(firstName: string, lastName: string) {}

let result1 = buildName("Bob"); // error, too few parameters
let result2 = buildName("Bob", "Adams", "Sr."); // error, too many parameters
```

We can make params optional or give them defaults

```
function buildName(firstName: string = 'Guest', lastName?: string) {}

let result1 = buildName(); // OK, firstname == 'Guest'
let result2 = buildName("Bob"); // OK, lastName == undefined
```

Rest parameters

```
function sum(initialValue = 0, ...rest : number[]): number {
    return rest.reduce( ( sum, x ) => sum + x , initialValue)
}
```

Function this context

```
interface Elem{
    onclick: (this:Elem, event:Event) ⇒ void
}
```

Enums

```
enum InvoiceStatus {
  SUBMITTED, // 0
  APPROVED, // 1
  PAID, // 2
}

function getStatusLabel(status: InvoiceStatus) {
  switch (status) {
    case InvoiceStatus.SUBMITTED:
      return "invoice was submitted";
    case InvoiceStatus.APPROVED:
      return `it's approved`;
    case InvoiceStatus.PAID:
      return "invoice paid";
  }
}

// Two-way mapping
const val = InvoiceStatus.SUBMITTED; // 0
const statusName = InvoiceStatus[val]; // 'SUBMITTED
```

Initialize enums

```
enum AnotherEnum {
  SUBMITTED = 123,
  APPROVED, // 124
  PAID, // 125
  SOMETHING_ELSE = 1000 + 24, // expression
  MORE, // 1025
}

enum FileAccess {
  None = 1 << 0,
  Read = 1 << 1,
  Write = 1 << 2,
  Execute = 1 << 3,
  ReadWrite = FileAccess.READ | FileAccess.WRITE;
}
```

String enums

```
const UserRole = {
  USER: "user",
  ADMIN: "admin",
  MODERATOR: "moderator",
};

function getPermissionsFor(
  role: Role,
): number | undefined {
  switch (role) {
    case Role.ADMIN:
      return 123;
    case Role.USER:
      return undefined;
    default:
      const _never:never = role; // exhaustiveness check
      return _never
  }
}
```

Function literal overloads

```
function create(name: "User"): User;
function create(name: "Admin"): Admin;
function create(name: "Moderator"): Moderator;
function create(name: string): Character {
// …
}
const m = create("Moderator"); // m is a Moderator
```

Type intersections and unions

```
type A = {
  a: string;
  b: number;
};
type B = {
  b: number;
  c: string;
};

type Union = A | B;
type Intersection = A & B;
```

Recursive types

```
type User = {
  name: string;
  friends: Array<User>;
};

type Json =
| null
| string
| number
| boolean
| Json[]
| { [name: string]: Json };
```

Literal types

A literal is a more concrete sub-type of a collective type. What this means is that "Hello World" is a string, but a string is not "Hello World" inside the type system.

```
const anyText:string = 'Hello!'
anyText = 'Goodbye!' // OK

const helloText: 'Hello' = 'Hello'
helloText = 'Goodbye!' // Error!

// alternatively:
// const helloText = 'Hello' as const
```

Literal unions

You can alias union of multiple possible literals

```
type Easing = "ease-in" | "ease-out" | "ease-in-out";

class UIElement {
  animate(dx: number, dy: number, easing: Easing) {
    if (easing === "ease-in") {
    } else if (easing === "ease-out") {
    } else if (easing === "ease-in-out") {
    } else {
      // It's possible that someone could reach this
      // by ignoring your types though.
    }
  }
}
let button = new UIElement();
button.animate(0, 0, "ease-in");
button.animate(0, 0, "uneasy"); // Error!
```

Union alternative syntax

```
type RequestCredentials =
  | "omit"
  | "same-origin"
  | "include";

let cred: RequestCredentials;
cred = "omit"; // OK
cred = "inclade"; // Error!
```

Indexed types and unions

```
type Headers = "Accept" | "Authorization" /* … */;
type RequestConfig = {
  [header in Headers]: string;
};

const requestConfig: RequestConfig = {
  Accept: "…",
  Authorization: "…",
};
```

Classes

```
class Person {
  name:string

  constructor(){}

  method(){}
}
```

Class uninitialized properties

```
class SomeClass{
  initializedProperty = 'some default text';

  uninitializedProperty:string // Error in strict mode!

  optionalProperty?:string // undefined when instance is created

  // You can make compiler accept it but its not safe
  trustMeIKnowWhatIAmDoingProperty!:string

  constructor(){
    // You have to remember to initialize it yourself
    this.trustMeIKnowWhatIAmDoingProperty = 'its initialized now'
  }
}
```

Class properties access modifiers

```
class IAmPartiallyAccesible{
  // fields are publicly accesible by default
  /* public  */ name = ''

  // only accessible in instances of this class
  private _secretData = ''

  // accessible in instances of this class and sub-classes
  protected _overrideMe = ''

  // Cannot be changed after instance is created
  readonly _cantChangeMe = ''

}

class AnotherClass extends IAmPartiallyAccesible{
  method(){
    this._secretData // Error
    this.name // OK
    this._overrideMe // OK
  }
}
```

Initialize fields with arguments

```
class ExampleDataDTO{
  public name: string = ''; private timestamp: string = '';

  constuctor(name:string, timestamp:string){
    this.name = name; this.timestamp = timestamp
  }
}
```

Can be written as

```
class SimplerDataDto{
  constructor(public name:string, private timestam:string){}

  toJSON(){
    return { name: this.name } // fields are populated from arguments!
  }
}
```

Any modifier works: public, private, protected, readonly.

Class inheritance

```
class Employee extends Person {
  name:string

  constructor(){
    // parent constructor call is required!
    super()
  }

  method(){
    // optionally can call overriden methods
    super.method()
  }
}
```

Class can implement interface or type

```
interface IDoStuff {
    // optionally can define constructor signature:
    new (name:string): IDoStuff
    name: string
    method():void
}
type IHaveId = {id:string}

// you can implement multiple types
class StuffClass implments IDoStuff, IHaveId{
  name: string
  constructor(){}
  method(){}
}
```

Generics

The key motivation for generics is to document meaningful type dependencies between members. The members can be:

- Class instance members
- Class methods
- function arguments
- function return value

```
const x: Array<string> = ["a", "b", "c"];
```

Generic type example

```
type Ref<T> = {
  current: T;
};
const ref1: Ref<number> = { current: 123 };
const ref2: Ref<string> = { current: "aaa" };

function getValue<T>(ref: Ref<T>): T {
  return ref.current;
}
```

Generic class example

```
/** A class definition with a generic parameter */
class Queue<T> {
  private data = [];
  push(item: T) { this.data.push(item); }
  pop(): T | undefined { return this.data.shift(); }
}
```

Generics and constraints

Those use generics but do not offer any type safety:

```
function parse<T>(name: string): T {}
function serialize<T>(name: T): string {}
```

practivally same as:

```
declare function parse(name: string): any;
const something = parse('something') as TypeOfSomething;
```

Generics and constraints

Use generic type in two places to add constraints:

```
const id = <T>(x: T): T => x;
const result = id<number>(1);

// typeof result == number
```

Here generics provide constraints between argument and return value

Generic interface

`new` keyword declares constructor signature - object can be constructed with `new`

```
interface Constructable<T> {
  new (...args: any[]): T;
}
```

Generics and type constraints - extends

Generic works like `unknown` type until type is given or inferred. Type can be constraint to only subtypes

```
type ObjWithName = { name: string };

function printName<T extends ObjWithName>(arg: T) {}

printName({ name: "Kate" }); // OK
printName({ name: "Michael", age: 22 }); //  OK
printName({ age: 22 }); // Error!
```

Multi-type generics

```
function makePair<T, U>(arg1: T, arg2: U): [T, U] {
  return [arg1, arg2];
}

function defaults<T extends object, U extends T>(
  obj1: T,obj2: U,
) {
  return { ...obj1, ...obj2 };
}
defaults({a:0, b:1}, { a: 123  });
// OK, { a: 123, b:1 }
defaults({a:0, b:1}, { a: 123, c: 234 });
// Error U contains 'c' witch T does not!
```

Type Compability

```
let str: string = "Hello";
let num: number = 123;

str = num; // ERROR: `number` is not assignable to `string`
num = str; // ERROR: `string` is not assignable to `number`
```

Structural typing / duck-typing

```
interface Point {
    x: number,
    y: number
}

class Point2D {
    constructor(public x:number, public y:number){}
}
let p: Point;
// OK, because of structural typing
p = new Point2D(1,2);
```

Structural overloads

```
interface Point2D {
    x: number; y: number;
}
interface Point3D {
    x: number; y: number; z: number;
}
var point2D: Point2D = { x: 0, y: 10 }
var point3D: Point3D = { x: 0, y: 10, z: 20 }
function iTakePoint2D(point: Point2D) { /* do something */ }

iTakePoint2D(point2D); // exact match okay
iTakePoint2D(point3D); // extra information okay
iTakePoint2D({ x: 0 }); // Error: missing information `y`
```

Variance

Variance is an easy to understand and important concept for type compatibility analysis.

For simple types Base and Child, if Child is a child of Base, then instances of Child can be assigned to a variable of type Base. – This is `polymorphism 101`

Compability can take multiple "flavours":

- Covariant : (co aka joint) only in same direction
- Contravariant : (contra aka negative) only in opposite direction
- Bivariant : (bi aka both) both co and contra.
- Invariant : if the types aren’t exactly the same then they are incompatible.

Unions

You can make type from union of types

```
type Id = number | string

interface Square {
    kind: "square"; size: number;
}

interface Rectangle {
    kind: "rectangle"; width: number; height: number;
}
type Shape = Square | Rectangle;
```

Type Guards

```
function parseInt(arg: string | number) {
  if (typeof arg === "string") {
    arg.toUpperCase()
  } else {
    arg.toPrecision(2)
  }
}
```

Type Guards - instanceof

```
class Character { name!: string;  }
class User extends Character { age!: number; }
class Enemy extends Character { hp!: number; }

declare let ch: Character;

if (ch instanceof User) {
  ch.age; // OK
  ch.name; // OK
} else if (ch instanceof Enemy) {
  ch.hp; // OK
  ch.name; // OK
} else {
  ch.name; // OK
}
```

Type Guards - in operator

```ts
type Character = { name: string; };
type User = { name: string; age: number; };
type Admin = { name: string; age: number; role: string; };
declare let x: Character | User | Admin;

if ("age" in x) {
  x; // User | Admin
}
if ("role" in x) {
  x; // Admin
}
```

Reusable type guards

```
type SingleValue = { value: string };
type ManyValues = { options: Array<SingleValue> };

function isSingleValue(val: SingleValue | ManyValues): val is SingleValue {
  return ( "value" in val && typeof val.value === "string" );
}
declare const obj: any;

if (isSingleValue(obj)) {
  // obj is a SingleValue
} else {
  // any
}
```

Discriminated unions

If all members of union share a property but each have unique literal value, we can discrimiate:

```
function area(s: Shape) {
    if (s.kind === "square") {
        // Now TypeScript *knows* that `s` must be a square ;)
        // So you can use its members safely :)
        return s.size * s.size;
    }
    else {
        // Wasn't a square? So TypeScript will figure out that it must be a Rectangle
        // So you can use its members safely :)
        return s.width * s.height;
    }
}
```

Exhaustiveness check

```
type Shape = Square | Rectangle | Circle // !!!

function area(s: Shape) {
    if (s.kind === "square") {
        return s.size * s.size;
    }
    else if (s.kind === "rectangle") {
        return s.width * s.height;
    }
    else {
        // We can check for that with 'never':
        const _exhaustiveCheck: never = s;
        // ERROR : `Circle` is not assignable to `never`
    }
}
```

Extracting types

```
const defaultConfig = {
  port: 3000,
  host: "localhost",
};
type Config = typeof defaultConfig;
```

Type Mapping

Type alias allows you to extract or derive new types from existing ones

```
interface Payload{ id:string, name:string, extra:string }

type someKeys = 'id' | 'name'
type PartialPayload = {
  // new Type borrows only few keys and types from original type
  [key in someKeys]: Payload[key]
}
```

Type Mapping with generics

Using generics makes mapping reusable

```
interface Payload{ id:string, name:string, extra:string }

type Partial<T> = {
  // take all keys and all types, but make them optional! (?)
  [key in keyof T] ? : T[key]
}
const changedFields: Partial<Payload> = { name: 'I am enough!' }
```

You dont have to define `Partial` yourself. Its built-in typescript type!

Conditional types

Type can differ depending on generic parameter types

```
type R = T extends U ? X : Y;

// Simple usage example:
type IsBoolean<T> = T extends boolean ? true : false;
type t01 = IsBoolean<number>; // false
type t02 = IsBoolean<string>; // false
type t03 = IsBoolean<true>; // true
```

Conditional types and unions

```
type NonNullable<T> = T extends null | undefined
  ? never
  : T;

type t04 = NonNullable<number>; // number
type t05 = NonNullable<string | null>; // string
type t06 = NonNullable<null | undefined>; // never


type StringsOnly<T> = T extends string ? T : never;
type Result = StringsOnly<"abc" | 123 | "ghi">;
// "abc" | never | "ghi", therefore only "abc" | "ghi"
```

Using decorators - example

```
import { JsonController, Param, Body, Get, Post, Put, Delete } from 'routing-controllers';

@JsonController('/users')
export class UserController {

  @Get('/')
  getAll(@Req() request: any, @Res() response: any) {
    return []
  }
}
useExpressServer(app, {
  routePrefix: '/api',
  controllers: [UserController], // we specify controllers we want to use
});
```

Decorators - validation

https://github.com/typestack/class-validator

```
enum Roles {
    Admin = "admin",
    User = "user",
    Guest = "guest",
}

class GetUsersQuery {

    @IsPositive()
    limit: number;

    @IsAlpha()
    city: string;

    @IsEnum(Roles)
    role: Roles;

    @IsBoolean()
    isActive: boolean;

}

@Get("/users")
```

Decorators - parsing

https://github.com/typestack/class-transformer

```
export class User {
  firstName: string;
  lastName: string;

  @Exclude({ toPlainOnly: true }) password: string;

  @Type(() => Album) albums: Album[];

  getName(): string {
    return this.lastName + ' ' + this.firstName;
  }
}

@Controller()
export class UserController {
  post(@Body() user: User) {
    console.log('saving user ' + user.getName());
  }
}
```