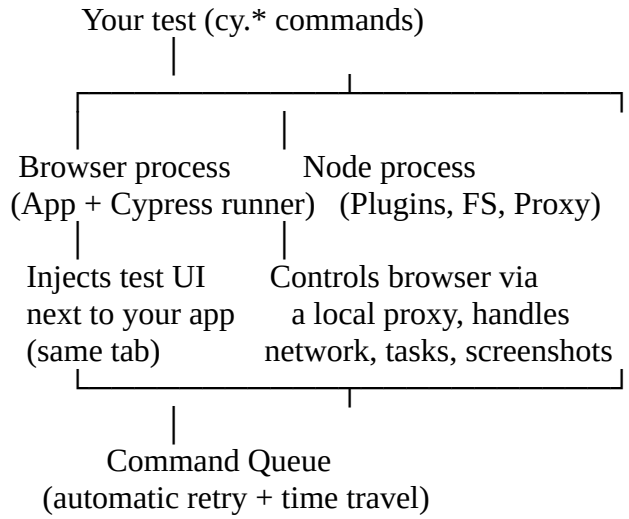
Cypress is a JavaScript/TypeScript test framework for **web UI** and **API** testing. It's unique because it runs **inside the browser** alongside your app, giving it deep access to the DOM, network, timers, and console—plus a live GUI runner with time-travel debugging. It also supports **Component Testing** (React/Vue/Angular/etc.) and **E2E testing**.

```
cypress/
  e2e/          # your tests
  fixtures/     # test data
  support/      # custom commands, hooks
  downloads/    # (during tests)
cypress.config.js
```

Architecture:



In-browser execution: Cypress injects its runner **into the same page** as your app (same event loop). That's why it sees DOM changes immediately and can time-travel in the GUI.

- **Local Node side:** A separate Node process handles **file system**, **screenshots/videos**, **`cy.task()`**, and runs a **built-in HTTP proxy** to observe/stub network (`cy.intercept()`).
- **Command Queue & Retry-ability:** Every `cy.*` command is queued; Cypress auto-retries until assertions pass or time out—this eliminates most `sleep()`/wait flakiness.
- **Auto-waiting:** Cypress implicitly waits for elements to appear/be actionable and for page/network stability (no manual explicit waits most of the time).

Lifecycle of a single command

1. You write `cy.get('#username').type('abhishek')`.
2. Cypress enqueues it; when executed, it queries the DOM repeatedly until `#username` exists and is interactable.
3. It performs `.type()`, then logs a snapshot for time-travel.

4. Assertions chained with `.should(...)` keep retrying until they pass or the default timeout is hit.

```
cy.visit('/path')                 // navigate
cy.get('selector')                // query
cy.contains('text')               // by text
cy.click() .type() .select()      // actions
cy.should('be.visible')           // assertions (chai)
cy.intercept('METHOD', 'url')     // network spy/stub
cy.viewport(1280, 720)            // responsive
cy.screenshot() .log()            // debug/report
cy.task('name', args)             // run Node code
```

What is `package.json`?

- `package.json` = **the manifest file of your project**.
- It tells npm:
  - Project name, version, description.
  - Which packages your project needs to run.
  - Which scripts you can run (`npm start`, `npm test`, etc.).
- When you install packages, npm writes them here so anyone else can run `npm install` and get the same dependencies.

`npx` just finds the package in your `node_modules` and runs it.

`npx cypress run`

- Runs Cypress tests **headlessly** (no visible browser UI).
- Executes all tests automatically and outputs results in the terminal.
- Used in **CI/CD pipelines** or automated scripts.

`cypress/fixtures/`

- Stores **static test data** (JSON, text, images, etc.).
- You can load this data into your tests with `cy.fixture()`.
- Useful for **mocking data** instead of hitting a live API.

```
cy.fixture('login_data').then((data) => {
  cy.get('#username').type(data.username)
  cy.get('#password').type(data.password)
})
```

`cypress/integration/` or `cypress/e2e/`

- This is where **your test files** live.

- In **Cypress v10+**, `integration` is replaced by `e2e`.
- You can organize tests into subfolders by feature/module.

Example:

```
cypress/e2e/login.spec.js
cypress/e2e/cart/cart.spec.js
```

`cypress/support/` Purpose

- This folder is where you put **global configurations**, **reusable commands**, and **hooks** that Cypress will **automatically load before running your tests**.
- Any code here is **not a test itself** — it's test *setup* or helper code.
- The goal: **Keep your test files clean and DRY** (Don't Repeat Yourself).

`cypress/support/e2e.js` (Cypress v10+)

(In Cypress < v10, this file was `index.js`)

What it does:

- Runs **before every single spec file**.
- Perfect for:
    - Importing custom commands (`commands.js`).
    - Setting up global hooks (`before`, `beforeEach`).
    - Importing reusable libraries (e.g., `faker`, `dayjs`).
    - Configuring global behaviors (e.g., ignoring uncaught exceptions).

```
// cypress/support/e2e.js

// Import custom commands
import './commands'

// Ignore Cypress failing on app errors
Cypress.on('uncaught:exception', (err, runnable) => {
  return false // prevents test from failing
})

// Global beforeEach
beforeEach(() => {
  cy.log('This runs before each test')
})
```

`cypress/support/commands.js`

- This is where you **create custom Cypress commands**.

- Custom commands help you reuse actions across tests without repeating steps.

Example:

```
// cypress/support/commands.js

// Custom login command
Cypress.Commands.add('login', (username, password) => {
  cy.visit('/login')
  cy.get('#username').type(username)
  cy.get('#password').type(password)
  cy.get('#loginBtn').click()
})
```

1️⃣ describe()

- Groups related tests together.
- Think of it as a **chapter heading** for a group of test cases.

```
describe('Login Feature', () => {
  // tests related to login go here
})
```

it()

- Defines a **single test case**.
- Reads like: “It should do this…”.

```
it('should login with valid credentials', () => {
  cy.visit('/login')
  cy.get('#username').type('admin')
  cy.get('#password').type('1234')
  cy.get('#loginBtn').click()
  cy.contains('Welcome').should('be.visible')
})
```

before()

- Runs **once** before **all** the tests in the `describe()` block.
- Good for **global setup** (e.g., DB connection, global login).

```
before(() => {
  cy.log('This runs once before all tests')
})
```

beforeEach()

- Runs **before every single test** in that `describe()` block.
- Good for **repetitive setup** (e.g., visiting a page before each test).

```
beforeEach(() => {
  cy.visit('/login')
})
```

after()

- Runs **once** after **all** the tests are done.
- Good for cleanup (e.g., closing DB connection, deleting test data).

```
after(() => {
  cy.log('This runs once after all tests')
})
```

---

6 afterEach()

- Runs **after every single test** in the `describe()` block.
- Good for **cleanup after each test** (e.g., clearing cookies/local storage).

```
afterEach(() => {
  cy.clearCookies()
})

describe('Login Feature', () => {

  before(() => {
    cy.log('=== Runs once before all tests ===')
  })

  beforeEach(() => {
    cy.visit('/login')
  })

  it('should login with valid credentials', () => {
    cy.get('#username').type('admin')
    cy.get('#password').type('1234')
    cy.get('#loginBtn').click()
    cy.contains('Welcome').should('be.visible')
  })

  it('should show error for invalid credentials', () => {
    cy.get('#username').type('wrong')
    cy.get('#password').type('wrong')
    cy.get('#loginBtn').click()
    cy.contains('Invalid username or password').should('be.visible')
  })

  afterEach(() => {
    cy.log('=== Cleanup after each test ===')
  })

  after(() => {
    cy.log('=== Runs once after all tests ===')
  })
})
```

Cypress commands are **asynchronous — you can’t just store the value in a variable like normal JS. You have to use `.then()` or `.invoke()`.

```
cy.get('.my-selector')
  .invoke('text')
  .then((text) => {
    cy.log('Element text is: ' + text)
    // you can also assert:
    expect(text.trim()).to.equal('Expected Value')
  })
```

If it’s an `<input>` or `<textarea>`, you use `.invoke('val')` instead of `.text()`.

`.invoke('text')` works for **visible text inside an element**.

- `.invoke('val')` works for **values inside inputs**.
- Always use `.trim()` if there might be extra spaces or line breaks.

`.contains()` in Cypress

Think of `.contains()` as "**find element by text content**".

```
cy.contains('Some Text')                 // Finds element containing text
cy.contains('button', 'Login')           // Finds <button> that has 'Login'
cy.get('.my-class').contains('Some Text')    // Within this selector, find text

// Click button that has "Submit" text
cy.contains('button', 'Submit').click()

// Check hidden text
cy.contains('Secret Text', { matchCase: false, includeShadowDom: true })
```

**

`expect()` in Cypress

This is **Chai assertion syntax**, used inside `.then()` or `.should()`.

Common forms:

```
expect(actualValue).to.equal(expectedValue)
expect(actualValue).to.include(substring)
expect(actualValue).to.contain(substring)
expect(actualValue).to.have.length(3)
expect(actualValue).to.match(/regex/)
```

How `.should()` works with Cypress

`.should()` is Cypress’ built-in way to assert directly in the chain.

```
cy.get('.welcome-msg').should('contain', 'Welcome') // partial match
cy.get('.welcome-msg').should('have.text', 'Welcome Abhishek') // exact match
cy.get('#username').should('have.value', 'Abhi123') // for input fields
```

Key difference:

- `.should()` works directly on Cypress commands.
- `expect()` is for **manual assertions** when you already have the value in `.then()`.

```
it('Example test', () => {
  // Click by text
  cy.contains('button', 'Login').click()

  // Check partial match
  cy.get('.welcome-msg').should('contain', 'Welcome')

  // Check exact match
  cy.get('.welcome-msg').should('have.text', 'Welcome Abhishek')

  // Using expect after extracting text
  cy.get('.welcome-msg').then(($el) => {
    const text = $el.text().trim()
    expect(text).to.include('Abhishek')
  })
})
```

Exact text match

Using `.should()`:

```
cy.get('.welcome-msg')
  .should('have.text', 'Welcome Abhishek')
```

Partial text match

Using `.should()`:

```
cy.get('.welcome-msg')
  .should('contain', 'Welcome')
```

Element is visible

Using `.should()`:

```
cy.get('#profile-picture')
  .should('be.visible')
```

Button enabled / disabled

Using `.should()`:

```
// Enabled
cy.get('#submit-btn').should('be.enabled')

// Disabled
cy.get('#submit-btn').should('be.disabled')
```

Assertion	With `.should()` (Retries)	With `.then() + expect()` (No jQuery)
Exact text match	`cy.get('.msg').should('have.text', 'Hello World')`	`cy.get('.msg').then($el => { const txt = $el[0].innerText.trim(); expect(txt).to.equal('Hello World'); })`
Partial text match	`cy.get('.msg').should('contain', 'Hello')`	`cy.get('.msg').then($el => { const txt = $el[0].innerText.trim(); expect(txt).to.include('Hello'); })`
Visible	`cy.get('#avatar').should('be.visible')`	`cy.get('#avatar').then($el => { const visible = $el[0].offsetWidth > 0 && $el[0].offsetHeight > 0; expect(visible).to.be.true; })`
Hidden	`cy.get('#avatar').should('not.be.visible')`	`cy.get('#avatar').then($el => { const visible = $el[0].offsetWidth > 0 && $el[0].offsetHeight > 0; expect(visible).to.be.false; })`
Enabled	`cy.get('#submit').should('be.enabled')`	`cy.get('#submit').then($el => { expect($el[0].disabled).to.be.false; })`
Disabled	`cy.get('#submit').should('be.disabled')`	`cy.get('#submit').then($el => { expect($el[0].disabled).to.be.true; })`
Have attribute	`cy.get('input').should('have.attr', 'placeholder', 'Search')`	`cy.get('input').then($el => { expect($el[0].getAttribute('placeholder')).to.equal('Search'); })`
CSS value match	`cy.get('.btn').should('have.css', 'color', 'rgb(255, 0, 0)')`	`cy.get('.btn').then($el => { expect(getComputedStyle($el[0]).color).to.equal('rgb(255, 0, 0)'); })`
Element count	`cy.get('.item').should('have.length', 5)`	`cy.get('.item').then($el => { expect($el.length).to.equal(5); })`

**
By default, Cypress commands like `cy.get()` return a **jQuery object**, even if there's only 1 element.

Rule of Thumb:

- Need `.text()`, `.attr()`, `.css()` → use `$el` directly (jQuery object).
- Need `.innerText`, `.value`, `.click()` → unwrap with `$el[0]` (DOM element).

Implicit Assertions

These are **built into Cypress commands** — you chain them using `.should()` or `.and()`.

Syntax

```
cy.get(selector).should('be.visible');
cy.get(selector).should('have.text', 'Login');
cy.get(selector)
  .should('be.visible')
  .and('contain.text', 'Login');
```

Key points:

- Cypress automatically **retries** the command + assertion until it passes or times out.
- `.and()` is just like `.should()` but for chaining **more assertions**.

Explicit Assertions

These are **manually written** assertions from libraries like **Chai, Sinon, or jQuery**, usually inside `.then()`.

Syntax

```
cy.get(selector).then($el => {
  expect($el).to.be.visible; // Chai BDD style
  expect($el.text()).to.include('Login'); // Partial match
});

cy.get(selector).then($el => {
  assert.equal($el.text(), 'Login', 'Text matches'); // Chai TDD style
});
```

What is a Fixture?

- In Cypress, **fixtures** are files (usually JSON) stored in the `cypress/fixtures/` folder.
- They store **test data** you can reuse across multiple tests.
- Cypress loads them using `cy.fixture()`.

Passing Fixture Data to Tests with `before()`

You can load the fixture **once before all tests**.

```
describe('Login Suite', () => {
  let loginData;

  before(() => {
    cy.fixture('login_data').then(data => {
      loginData = data;
    });
  });
```

```
describe('Data-driven Login Tests', () => {
  before(() => {
    cy.fixture('users').as('usersData');
  });

  it('runs login test for each user', function () {
    this.usersData.forEach(user => {
      cy.visit('https://example.com/login');
      cy.get('#username').type(user.username);
      cy.get('#password').type(user.password);
      cy.get('#loginBtn').click();
      cy.get('.welcome-msg').should('contain.text', 'Welcome');
    });
  });
});
```

✅ Here:

this.usersData contains the array from users.json

.forEach() runs the same steps for each set of credentials.

Bonus: Using `it()` per Data Set

If you want **separate test results** for each data set instead of one loop.

```
describe('Separate data-driven tests', () => {
  before(() => {
    cy.fixture('users').as('usersData');
  });

  it('runs login for all users', function () {
    this.usersData.forEach(user => {
      it(`Login test for ${user.username}`, () => {
        cy.visit('https://example.com/login');
        cy.get('#username').type(user.username);
        cy.get('#password').type(user.password);
        cy.get('#loginBtn').click();
        cy.get('.welcome-msg').should('contain.text', 'Welcome');
      });
    });
  });
});
```

or import test data as import then use directy test.forEach()
**

Creating a Custom Command

📄 **cypress/support/commands.js**

```
Cypress.Commands.add('login', (username, password) => {
  cy.get('#username').type(username);
  cy.get('#password').type(password);
  cy.get('#loginBtn').click();
```

```
});
```

Now, in your test:

```
describe('Login Test', () => {
  it('should login with valid user', () => {
    cy.visit('/login');
    cy.login('testuser', 'password123');
    cy.get('.welcome-msg').should('contain.text', 'Welcome');
  });
});
```

Scope	Meaning	Where to Put It	When to Use
Global	Available in *all* Cypress tests automatically.	`cypress/support/commands.js`	For common actions like `login`, `logout`, `addToCart`.
Local	Defined only inside a single test file or `describe()` block.	Inside the test file itself.	For one-off actions that are not reused across multiple tests.

Example — Local Scope

📄 **tests/login.spec.js**

```
function login(username, password) {
  cy.get('#username').type(username);
  cy.get('#password').type(password);
  cy.get('#loginBtn').click();
}

describe('Local login test', () => {
  it('login with local function', () => {
    cy.visit('/login');
    login('testuser', 'password123'); // only works in this file
  });
});
```

Example — Global Scope

📄 **cypress/support/commands.js**

```
Cypress.Commands.add('verifyText', (selector, text) => {
  cy.get(selector).should('contain.text', text);
});
```

✅ Now `cy.verifyText()` works in **any** test without importing anything.

Use semantic selectors if test IDs aren’t available

- Use `id` → `#username`
- Use `name` → `[name="email"]`

- Use visible text with `contains()` for static content

```
cy.contains('button', 'Login').click();
```

Avoid brittle selectors

Bad

```
cy.get('.btn.red.small').click(); // Style-based
cy.get('div > button:nth-child(2)').click(); // Structure-based
```

Why bad? Changes in CSS or HTML layout will break the test.

Use aliases for reusability

```
cy.get('[data-testid="username"]').as('usernameField');
cy.get('@usernameField').type('testuser');
```

Recommended Selector Priority

1. `data-testid` / `data-cy` (best)
2. `id` (if stable)
3. `name` (if stable)
4. `aria-label` or `alt` (for accessibility)
5. Visible text with `.contains()`
6. Class (only if stable and unique)
7. XPath (last resort)

Clear cookies and local storage

```
afterEach(() => {
  cy.clearCookies();
  cy.clearLocalStorage();
});
```

State Management Between Tests

Cypress **resets state** automatically between tests for:

- The DOM
- Cookies
- Local storage (unless `preserve` is used)

f you **need to share state** (like a token):

- Store it in Cypress environment variables or aliases:

```
before(() => {
  cy.request('POST', '/login', { user: 'admin', pass: 'pass' })
    .then((resp) => {
```

```
      Cypress.env('authToken', resp.body.token);
    });
});

beforeEach(() => {
  cy.setCookie('authToken', Cypress.env('authToken'));
});
```

Why We Need It cy.wrap()

Cypress commands (`cy.get()`, `cy.request()`, etc.) run asynchronously and are put into a command queue.
If you have a **plain JS value** (like a number, string, object, or promise), Cypress won't automatically handle it in its chain unless you wrap it.

Here, `cy.wrap(myName)` tells Cypress:

> "Treat `myName` as a Cypress subject so I can run Cypress assertions and commands on it."

Example with Objects

```
it('wrap object', () => {
  const user = { name: 'Abhi', role: 'admin' };

  cy.wrap(user).its('role').should('eq', 'admin');
});
```

Without `cy.wrap()`, you'd just have a normal JS object — no `.its()` or `.should()` available.

Example with Promises

Cypress can automatically resolve promises if you wrap them.

```
it('wrap a promise', () => {
  const promise = new Promise((resolve) => {
    setTimeout(() => resolve(42), 1000);
  });

  cy.wrap(promise).should('eq', 42); // Cypress waits for promise to resolve
});
```

Without `cy.wrap()`, Cypress wouldn't wait for that promise.

cypress.config.js

const { defineConfig } = require('cypress');

module.exports = defineConfig({
  e2e: {
    baseUrl: 'https://example.com', // Base URL for tests
    viewportWidth: 1280,   // Browser width
    viewportHeight: 720,   // Browser height

```
  env: {                    // Environment variables
    username: 'admin',
    password: 'pass123'
  },
  setupNodeEvents(on, config) {
    // Add plugins/events here
  }
 }
});
```

Environment Variables

- Stored in `env` inside `cypress.config.js`.
- Accessible via:

```
Cypress.env('username') // returns "admin"
```

Reading a **JSON file** inside Cypress tests.

- Writing back to a **JSON file** from a Cypress test.
- Using **Node.js runtime via Cypress tasks** to do file I/O (because Cypress tests run in browser context and can’t write files directly).

Defining Tasks in `cypress.config.js`

```
const fs = require('fs');

const { defineConfig } = require('cypress');

module.exports = defineConfig({
  e2e: {
    setupNodeEvents(on, config) {
      // Task: Read JSON
      on('task', {
        readJson(filePath) {
          return new Promise((resolve, reject) => {
            fs.readFile(filePath, 'utf8', (err, data) => {
              if (err) return reject(err);
              resolve(JSON.parse(data));
            });
          });
        },

        // Task: Write JSON
        writeJson({ filePath, jsonData }) {
          return new Promise((resolve, reject) => {
            fs.writeFile(filePath, JSON.stringify(jsonData, null, 2), (err) => {
              if (err) return reject(err);
              resolve({ success: true });
            });
          });
        }
      });
    }
  }
```

```
});
```

- Here:
  - **`readJson(filePath)`** reads and returns JSON data.
  - **`writeJson({ filePath, jsonData })`** writes JSON back to file.
  - These run in the **Node.js process** (outside browser), so they can use `fs`.

---

4 Test File `e2e/readWriteJson.cy.js`

```
describe('Read and Write JSON Example', () => {

  it('Reads JSON, modifies it, and writes back', () => {
    const jsonPath = 'cypress/fixtures/testData.json';

    // 1. Read JSON via cy.fixture
    cy.fixture('testData.json').then((data) => {
      cy.log(`Username from fixture: ${data.username}`);
    });

    // 2. Read JSON via task (Node)
    cy.task('readJson', jsonPath).then((data) => {
      cy.log(`Current count: ${data.count}`);

      // Modify data
      data.count += 1;
      data.lastUpdated = new Date().toISOString();

      // 3. Write back JSON
      cy.task('writeJson', { filePath: jsonPath, jsonData: data }).then((result)
=> {
        expect(result.success).to.be.true;
      });
    });
  });

});
```

---

5 How This Works

- **`cy.fixture()`** → Quick way to load **static** data at runtime (read-only).
- **`cy.task()`** → Runs in Node.js, so you can read/write dynamically.
- In this example:
  1. Fixture is read (read-only).
  2. Same file is read with `cy.task('readJson')` for dynamic changes.
  3. Updated data is saved with `cy.task('writeJson')`.

Why Not Just Use `cy.writeFile()`?

- `cy.writeFile()` **can write JSON directly** without defining a task, but it's limited to the Cypress project folder.
- Using tasks with `fs` gives **full Node control** and allows reading/writing **any path**, plus more complex logic.

Read and Update JSON File

```
const jsonPath = 'data.json';

// Example JSON: { "count": 1, "name": "Abhi" }

// Read JSON
const jsonData = JSON.parse(fs.readFileSync(jsonPath, 'utf8'));
console.log('Before update:', jsonData);

// Update value
jsonData.count += 1;
jsonData.lastUpdated = new Date().toISOString();

// Write back
fs.writeFileSync(jsonPath, JSON.stringify(jsonData, null, 2));
console.log('JSON updated!');
```

Navigation

cy.visit()

- Opens a URL (or relative path if `baseUrl` is set).

```
cy.visit('/login'); // uses baseUrl from config
cy.visit('https://example.com/dashboard'); // full URL
```

cy.go()

- Navigate **backward/forward** in browser history.

```
cy.go('back');    // equivalent to browser back
cy.go('forward'); // equivalent to browser forward
cy.go(-1);        // same as back
cy.go(1);         // same as forward
```

cy.reload()

- Reloads the current page.

```
cy.reload();              // normal reload
cy.reload(true);          // force reload ignoring cache
```

Handling Popups, Alerts, and Iframes

Alerts & Confirm

```
cy.on('window:alert', (msg) => {
  expect(msg).to.equal('Are you sure?');
});

cy.on('window:confirm', () => true); // click 'OK' automatically
```

File Upload

Using `<input type="file">`

```
const filePath = 'files/testFile.pdf';

cy.get('input[type="file"]').attachFile(filePath); // requires cypress-file-upload
```

Install plugin:

```
npm install --save-dev cypress-file-upload
```

- // cypress/support/e2e.js

```
import 'cypress-file-upload';
```

`cypress/support/e2e.js` (Cypress ≥10)

- This file **replaces the old `index.js`** in `support/`.
- It is **automatically loaded before every test**.
- Its main purpose:
  - Import `commands.js` (custom commands)
  - Import any plugins that extend Cypress **in the browser context**
  - Global configurations for your tests

Takeaway:

- `e2e.js` is like a **central import hub** for **browser-side stuff**.
- `commands.js` is just **your custom commands library**.
- Node-level tasks stay in `setupNodeEvents`.

Cypress Command Queue

- **Cypress commands are asynchronous** but look synchronous because Cypress queues them internally.
- Every command you call (`cy.get()`, `cy.click()`) is **added to the queue**, then executed in order.
- Example:

```
cy.get('#username').type('Abhishek');
cy.get('#password').type('password123');
cy.get('#loginBtn').click();
```

- These commands **do not return values directly**, they pass the subject to the next command in the queue.

Using `.then()`

- Use `.then()` to **access the value of a Cypress command**.
- It gives you the **actual JavaScript value** inside the chain.

```
cy.get('#welcomeMsg').then(($el) => {
  const text = $el.text();
  console.log(text);  // Access the real value
});
```

- **Key:** `.then()` executes **after the previous command resolves**.

3️⃣ Using `cy.wrap()`

- Converts a **plain JS value** (string, object, promise) into a **Cypress chainable**.
- Allows chaining `.should()`, `.then()`, etc.

```
const name = 'Abhishek';
cy.wrap(name).should('eq', 'Abhishek');
```

- Useful when combining **JS values with Cypress commands**.

Avoiding `cy.wait()` Misuse

- `cy.wait(5000)` **pauses unconditionally** → slows tests, flaky.
- Instead, rely on **commands + assertions + retry**:

```
// Bad
cy.wait(5000);
cy.get('#submitBtn').click();

// Good
cy.get('#submitBtn').should('be.visible').click();
```

Reports pending