**Task 1: Initialize Project (Day 1 - 2 hours)**

**Goal:** Create a new TypeScript project with Playwright

**Actions:**

* Create new project folder
* Initialize npm project
* Install Playwright as dev dependency
* Install TypeScript as dev dependency
* Install @types/node for TypeScript support
* Run Playwright install command to download browsers

**Deliverable:**

* ✅ package.json exists with Playwright and TypeScript
* ✅ Browsers installed (Chromium, Firefox, WebKit)

**Task 2: Setup TypeScript Configuration (Day 1 - 1 hour)**

**Goal:** Configure TypeScript compiler settings

**Actions:**

* Create tsconfig.json file in root
* Set target to ES2020
* Enable strict mode
* Configure module system as commonjs
* Set output directory
* Include types for Node and Playwright
* Enable JSON module resolution

**Deliverable:**

* ✅ tsconfig.json file created
* ✅ TypeScript can compile without errors

**Task 3: Create Folder Structure (Day 1 - 30 minutes)**

**Goal:** Setup organized project directory structure

**Actions:**

* Create src folder for source code
* Inside src: create pages, api, utils, config, fixtures, data folders
* Create tests folder for test files
* Inside tests: create ui and api subfolders
* Create reports folder for test reports
* Create .gitignore file

**Deliverable:**

* ✅ Clean folder structure
* ✅ Separation of concerns (pages, tests, utils, etc.)

**Task 4: Write Your First UI Test (Day 3 - 2 hours)**

**Goal:** Create a simple test to verify setup

**Actions:**

* Create a simple test file in tests/ui folder
* Write a test that navigates to a public website (e.g., example.com)
* Add basic assertions (title check, URL check)
* Run the test using npx playwright test
* View the HTML report
* Verify screenshot and video are captured

**Deliverable:**

* ✅ First test passes successfully
* ✅ HTML report shows test results
* ✅ Screenshot/video captured on failure

**Task 5: Create Base Page Class and playwright.config.ts** **(Day 4 - 3 hours)**

**Goal:** Build reusable base class for all page objects and `playwright.config.ts`

**Actions:**

* Create BasePage class in src/pages folder
* Accept page object in constructor
* Create common methods: navigate, getTitle, waitForElement
* Create method to take screenshot
* Create method for generic click action
* Create method for generic type/fill action
* Add TypeScript types for all methods
* Create `playwright.config.ts` and config `baseURL`, and use install Chrome

**Deliverable:**

* ✅ BasePage class created
* ✅ Reusable methods available
* ✅ All pages can extend this class

**Task 6: Create Page Object and Refactor Test (Day 4-5, 4 hours)**

**Goal:** Implement Page Object Model for a real page and update your test to use it.

**Actions:**

1. **Choose a page** from your application (e.g., Login page).
2. **Create LoginPage class** extending BasePage.
   * Define all locators as **private properties**.
   * Create **methods for page actions**: enterUsername, enterPassword, clickLogin.
   * Add a method to **verify page loaded**.
   * Use **descriptive method names** representing user actions.
   * Add **TypeScript interfaces** for method parameters.
3. **Refactor an existing test** to use the page object:
   * Import the page object into the test file.
   * Instantiate the page object in the test.
   * Replace direct Playwright commands with **page object methods**.
   * Run the test and verify it still passes.
   * Compare readability **before and after**, and document observed benefits.

**Deliverables:**

* ✅ First page object created.
* ✅ Locators separated from test logic.
* ✅ Clean, readable methods.
* ✅ Test uses page object.
* ✅ Test is more readable and maintainable.
* ✅ Test logic separated from page interactions.

**Task 7: Create Page Component Pattern (Day 5 - 3 hours)**

**Goal:** Build reusable UI components (header, footer, navigation)

**Actions:**

* Identify common components across pages (header, sidebar, modal)
* Create component classes in src/pages/components folder
* Create HeaderComponent class with header-specific actions
* Create methods for common header actions
* Compose components into page objects
* Update page objects to use components

**Deliverable:**

* ✅ Reusable component classes
* ✅ DRY principle applied
* ✅ Components used in multiple pages

**Task 8: Understand and Create Playwright Fixtures (Day 7-8, 5 hours)**

**Goal:**

* Learn built-in fixtures and their purposes.
* Build and use custom fixtures for page objects to improve test setup and readability.

**Actions:**

1. **Learn Built-in Fixtures**
   * Read Playwright documentation on fixtures.
   * Understand key fixtures: page, context, request.
   * Experiment with different fixture scopes (test, worker).
   * Explore fixture lifecycle (setup/teardown).
2. **Create Custom Page Fixtures**
   * Create a fixtures file in src/fixtures folder.
   * Extend the base test from @playwright/test.
   * Create a fixture that returns a LoginPage instance.
   * Create fixtures for other page objects as needed.
   * Use custom fixtures in tests and verify proper initialization.
   * Add teardown logic if required.
3. **Practice and Document Learnings**
   * Create tests using multiple built-in and custom fixtures.
   * Document the benefits, proper usage, and lifecycle of fixtures.

**Deliverables:**

* ✅ Understanding of fixture concept and lifecycle.
* ✅ Knowledge of when to use built-in vs custom fixtures.
* ✅ Custom page fixtures working and integrated in tests.
* ✅ Cleaner, more maintainable test setup.

**Task 9: Implement Multi-tab/Window Testing**

**Goal:** Test multi-window scenarios

**Actions:**

* Handle new window/tab events
* Switch between windows/tabs
* Interact with multiple windows simultaneously
* Close specific windows
* Verify data across windows
* Test popup windows
* Handle window focus issues

**Deliverable:**

* ✅ Multi-window testing works
* ✅ Window management handled
* ✅ Complex scenarios covered

**Task 10: Setup Test Data Files**

**Goal:** Externalize test data from tests

**Actions:**

* Create JSON files in src/data folder
* Create users.json with test user data
* Create products.json with test product data
* Create utility function to read JSON files
* Create TypeScript interfaces for data structures
* Import and use data in tests
* Verify tests work with external data

**Deliverable:**

* ✅ Test data externalized
* ✅ JSON files readable in tests
* ✅ Data reusable across tests

**Task 11: Install and Setup Faker.js**

**Goal:** Generate dynamic test data

**Actions:**

* Install @faker-js/faker package
* Create data generator utility class
* Create method to generate random user data
* Create method to generate random product data
* Create method to generate random email
* Use generated data in tests
* Compare static vs dynamic data benefits

**Deliverable:**

* ✅ Faker.js integrated
* ✅ Data generator utility created
* ✅ Tests use dynamic data

**Task 12: Create Test Data Builders (Day 10 - 3 hours)**

**Goal:** Implement Builder Pattern for test data

**Actions:**

* Create UserBuilder class
* Implement fluent interface methods (withName, withEmail, etc.)
* Add build() method to return constructed object
* Add default values for optional fields
* Create builders for other entities
* Use builders in tests
* Document builder pattern benefits

**Deliverable:**

* ✅ Builder classes created
* ✅ Fluent, readable data creation
* ✅ Tests use builders

**Task 13: Create Wait Utility Class (Day 10 - 2 hours)**

Goal: Centralize wait/synchronization logic

Actions:

* Create WaitUtils class in src/utils
* Create method: waitForElementVisible
* Create method: waitForElementClickable
* Create method: waitForTextToAppear
* Create method: waitForNetworkIdle
* Create method: waitForCondition (custom condition)
* Use wait utilities in page objects
* Test different wait scenarios

Deliverable:

* ✅ Wait utility class created
* ✅ Reusable wait methods
* ✅ Reduced flakiness in tests

**Task 14: Basic Playwright Configuration**

**Goal:** Setup playwright.config.ts with basic settings

**Actions:**

* Create playwright.config.ts in root
* Configure test directory path
* Set base URL for application
* Configure timeout settings (test timeout, action timeout)
* Setup chromium, firefox, webkit browsers
* Enable screenshot on failure
* Enable video on failure
* Configure HTML reporter
* Set parallel worker count

**Deliverable:**

* ✅ playwright.config.ts created
* ✅ Can run basic Playwright test
* ✅ HTML report generates

**Task 15: Configure Test-Level Retries**

**Goal:** Auto-retry failed tests

**Actions:**

* Update playwright.config.ts
* Set retries: 2 for failed tests
* Configure retries only for specific environments (CI)
* Run failing test and observe retry behavior
* Check HTML report for retry information
* Document retry strategy

**Deliverable:**

* ✅ Test retries configured
* ✅ Flaky tests auto-retry
* ✅ Retry count visible in reports

**Task 16: Environment Configuration Setup**

**Goal:** Handle multiple environments (dev, staging, prod)

**Actions:**

* Install dotenv package
* Create .env.example file with sample variables
* Create .env file (add to .gitignore)
* Create environment config files for dev, staging, prod
* Create a config utility class to read environment variables
* Setup environment selection mechanism
* Document environment variables needed

**Deliverable:**

* ✅ Can switch between environments
* ✅ Environment variables loaded correctly
* ✅ Config utility accessible throughout project

**Task 17: Implement Self-Healing Test Capabilities**

* Goal
  + Reduce test flakiness and maintenance effort by enabling tests to automatically recover from broken or changed locators using smart fallback strategies—without full reliance on external AI tools.
* Deliverable
  + ✅ SmartLocator utility class implemented in `src/utils/SmartLocator.ts`
  + ✅ Page objects updated to use smart locators for fragile elements (e.g., subscription form, login fields)
  + ✅ Tests pass even when primary locators break, thanks to fallbacks
  + ✅ Clear console logs when a fallback is used (enables monitoring of UI changes)