PLUS POINT ANSWER

When running tests on multiple websites with different URLs supporting different countries, it is crucial to structure your tests in a way that is modular, maintainable, and adaptable to variations in input data, element locators, and languages. Here is a comprehensive approach to achieve this:

Step-by-Step Strategy

1. Environment Configuration

Define configurations for each environment (country/website) in a centralized configuration file. This includes URLs, input datasets, element selectors, and language-specific texts.

Example: `config.js`

```
const config = {
  US: {
    url: 'https://us.example.com',
    data: {
      address: '123 Main St',
      user: 'testuser',
      // other US-specific data
    },
    selectors: {
      emailInput: '#email',
      passwordInput: '#password',
      submitButton: '#submit',
      // other US-specific selectors
    },
    texts: {
      successMessage: 'Registration successful',
      // other US-specific texts
    }
  },
  FR: {
    url: 'https://fr.example.com',
    data: {
      address: '456 Rue Principale',
      user: 'utilisateurdetest',
      // other FR-specific data
    },
```

```
    selectors: {
      emailInput: '#courriel',
      passwordInput: '#motdepasse',
      submitButton: '#envoyer',
      // other FR-specific selectors
    },
    texts: {
      successMessage: 'Inscription réussie',
      // other FR-specific texts
    }
  },
  // Add other countries/environments
};

module.exports = config;
```

2. Test Data Management

Use the configuration data in your tests to handle different datasets for each country.

Example: `data.js`

```
const config = require('./config');

function getData(country) {
  return config[country].data;
}

module.exports = { getData };
```

3. Page Object Model

Create page objects that use the environment-specific selectors. This allows you to write tests that are agnostic of the underlying implementation details.

Example: `registrationPage.js`

```
class RegistrationPage {
  constructor(page, selectors) {
    this.page = page;
    this.selectors = selectors;
  }
```

```
  async enterEmail(email) {
    await this.page.fill(this.selectors.emailInput, email);
  }

  async enterPassword(password) {
    await this.page.fill(this.selectors.passwordInput, password);
  }

  async submitForm() {
    await this.page.click(this.selectors.submitButton);
  }
}

module.exports = RegistrationPage;
```

4. Test Implementation

Write tests that dynamically use the configuration and page objects based on the specified country/environment.

Example: `registration.spec.js`

```
const { test, expect } = require('@playwright/test');
const RegistrationPage = require('./registrationPage');
const config = require('./config');
const { getData } = require('./data');

function generateUniqueEmail(baseEmail) {
  const uniqueSuffix = Date.now();
  const emailParts = baseEmail.split('@');
  return `${emailParts[0]}+${uniqueSuffix}@${emailParts[1]}`;
}

test.describe('User Registration', () => {
  for (const country of Object.keys(config)) {
    test(`should register a user in ${country}`, async ({ page }) =>
{
      const envConfig = config[country];
      const testData = getData(country);
      const uniqueEmail = generateUniqueEmail('test@example.com');
```

```
      const registrationPage = new RegistrationPage(page,
envConfig.selectors);

      await page.goto(envConfig.url);
      await registrationPage.enterEmail(uniqueEmail);
      await registrationPage.enterPassword('yourpassword');
      await registrationPage.submitForm();

      // Add assertions to verify registration was successful
      await
expect(page).toHaveURL(`${envConfig.url}/registration-success`);
      await
expect(page.locator(`text=${envConfig.texts.successMessage}`)).toBeV
isible();
    });
  }
});
```

Key Points

- **Centralised Configuration**: All environment-specific data (URLs, input datasets, selectors, texts) is centralised, making it easy to manage and update.
- **Dynamic Test Data**: Tests dynamically pull in the necessary data and selectors based on the specified environment, ensuring flexibility and scalability.
- **Page Object Model**: Encapsulating page interactions in page objects ensures that tests are modular and maintainable.
- **Parameterized Tests**: Using loops or parameterized tests to run the same test logic across multiple environments/countries.

This approach ensures that your test suite is robust, maintainable, and capable of handling variations across different environments. It also makes it easy to add support for new countries or environments by simply updating the configuration files and selectors.