← Playwright API Testing

Advanced API Testing →

Playwright API Testing Fundamentals

Table of Contents

REST API Concepts and Architecture

What is REST?

REST (Representational State Transfer) is an architectural style for designing networked applications. It relies on stateless, client-server communication using HTTP.

Key Principles of REST

- Server doesn't store client state
- Each request is independent

2. Client-Server Architecture

- Separation of concerns
- Client handles UI
- Server handles data storage

3. Cacheable

- Responses can be cached
- Improves performance
- Reduces server load

4. Uniform Interface

- Standard HTTP methods (GET, POST, PUT, DELETE)
- Resource-based URLs
- Standardized responses

5. Layered System

- Client can't tell if connected directly to server
- Allows intermediaries (load balancers, caches)

REST API Components

API Endpoint Structure:

```
https://api.example.com/v1/users/123?include=posts
└──────┬──────┘└┬┘└──┬──┘└┬┘└──────┬──────┘
   Base URL     Ver Route  ID   Query Params
```

Common API Patterns

CRUD Operations:

- **C**reate - POST
- **R**ead - GET
- **U**pdate - PUT/PATCH
- **D**elete - DELETE

Resource URLs:

```
 GET    /api/users           # Get all users
GET    /api/users/123       # Get specific user
POST   /api/users           # Create new user
PUT    /api/users/123       # Update user (full)
PATCH  /api/users/123       # Update user (partial)
DELETE /api/users/123       # Delete user
```

Request Components

1. HTTP Method

```
GET, POST, PUT, PATCH, DELETE, HEAD, OPTIONS
```

2. URL/Endpoint

```
https://api.example.com/v1/users
```

3. Headers

```
User-Agent: Playwright/1.40
```

4. Body (for POST, PUT, PATCH)

```
{
  "name": "John Doe",
  "email": "john@example.com",
  "age": 30
}
```

5. Query Parameters

```
?page=2&limit=10&sort=created_at&order=desc
```

Response Components

1. Status Code

```
200 OK, 201 Created, 400 Bad Request, 404 Not Found, 500 Internal Server E
```

2. Headers

```
Content-Type: application/json
Content-Length: 1234
Set-Cookie: session_id=abc123
```

3. Body

```
{
  "data": {
    "id": 123,
    "name": "John Doe",
    "email": "john@example.com"
  },
  "meta": {
```

API Response Patterns

Success Response:

```
{
  "success": true,
  "data": {
    "id": 1,
    "name": "John Doe"
  },
  "message": "User retrieved successfully"
}
```

Error Response:

```
{
  "success": false,
  "error": {
    "code": "USER_NOT_FOUND",
    "message": "User with ID 999 does not exist"
  },
  "statusCode": 404
}
```

List Response with Pagination:

```
{
  "data": [
    { "id": 1, "name": "User 1" },
    { "id": 2, "name": "User 2" }
  ],
  "pagination": {
    "page": 1,
    "pageSize": 10,
    "totalPages": 5,
    "totalRecords": 50
  }
```

HTTP Methods and Status Codes

HTTP Methods

GET - Retrieve Data

```
GET /api/users/123

Response:
{
  "id": 123,
  "name": "John Doe",
  "email": "john@example.com"
}
```

- **Safe**: Doesn't modify data
- **Idempotent**: Multiple calls return same result
- **Cacheable**: Can be cached

POST - Create Data

```
POST /api/users
Content-Type: application/json

{
  "name": "Jane Smith",
  "email": "jane@example.com"
}

Response: 201 Created
{
  "id": 124,
  "name": "Jane Smith",
  "email": "jane@example.com",
  "createdAt": "2024-01-27T10:30:00Z"
}
```

- **Not cacheable**

PUT - Full Update

```
 PUT /api/users/123
Content-Type: application/json

{
  "name": "John Updated",
  "email": "john.updated@example.com",
  "age": 31
}

Response: 200 OK
{
  "id": 123,
  "name": "John Updated",
  "email": "john.updated@example.com",
  "age": 31
}
```

- **Not safe**: Modifies data
- **Idempotent**: Multiple calls have same effect
- Requires full resource representation

PATCH - Partial Update

```
 PATCH /api/users/123
Content-Type: application/json

{
  "email": "newemail@example.com"
}

Response: 200 OK
{
  "id": 123,
  "name": "John Doe",
  "email": "newemail@example.com",
  "age": 30
```

- **Not safe**: Modifies data
- **Not idempotent** (usually)
- Only modified fields sent

DELETE - Remove Data

```
DELETE /api/users/123

Response: 204 No Content
or
Response: 200 OK
{
  "message": "User deleted successfully"
}
```

- **Not safe**: Modifies data
- **Idempotent**: Deleting same resource multiple times is safe

HEAD - Get Headers Only

```
HEAD /api/users/123

Response: 200 OK
(Headers only, no body)
```

- Check if resource exists
- Get metadata
- No response body

OPTIONS - Get Allowed Methods

```
OPTIONS /api/users

Response: 200 OK
Allow: GET, POST, PUT, DELETE
```

HTTP Status Codes

1xx - Informational

- `100 Continue` - Continue with request
- `101 Switching Protocols` - Protocol change

2xx - Success

- `200 OK` - Request succeeded
- `201 Created` - Resource created
- `202 Accepted` - Request accepted, processing
- `204 No Content` - Success, no response body

3xx - Redirection

- `301 Moved Permanently` - Resource moved permanently
- `302 Found` - Temporary redirect
- `304 Not Modified` - Use cached version

4xx - Client Errors

- `400 Bad Request` - Invalid request
- `401 Unauthorized` - Authentication required
- `403 Forbidden` - Authenticated but not authorized
- `404 Not Found` - Resource doesn't exist
- `405 Method Not Allowed` - HTTP method not supported
- `409 Conflict` - Request conflicts with current state
- `422 Unprocessable Entity` - Validation failed
- `429 Too Many Requests` - Rate limit exceeded

5xx - Server Errors

- `500 Internal Server Error` - Generic server error

- `504 Gateway Timeout` - Upstream server timeout

Status Code Categories

Range	Category	When to Use
2xx	Success	Operation completed successfully
3xx	Redirection	Resource moved or cached
4xx	Client Error	Problem with request (bad data, auth, etc.)
5xx	Server Error	Server failed to process valid request

Playwright API Testing Overview

Why Playwright for API Testing?

Benefits:

- Single framework for UI and API tests
- Built-in request context
- Automatic retries
- Fixtures support
- Easy integration with UI tests
- TypeScript support

Playwright APIRequestContext

Features:

- Request/response interception
- Authentication support
- Cookie handling
- File uploads

Setting Up API Testing

Basic setup:

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

test('basic API test', async ({ request }) => {
  const response = await request.get('https://api.example.com/users');

  expect(response.ok()).toBeTruthy();
  expect(response.status()).toBe(200);

  const data = await response.json();
  console.log('Users:', data);
});
```

Creating API Request Context

Using built-in request fixture:

```
test('using request fixture', async ({ request }) => {
  const response = await request.get('https://api.example.com/users/1');
  const user = await response.json();

  expect(user.id).toBe(1);
});
```

Creating custom request context:

```
import { test, request as playwrightRequest } from '@playwright/test';

test('custom request context', async () => {
```

```
      'Accept': 'application/json',
      'Content-Type': 'application/json',
    },
  });

  const response = await apiContext.get('/users/1');
  expect(response.status()).toBe(200);

  await apiContext.dispose();
});
```

Making API Requests

GET Request:

```
test('GET request', async ({ request }) => {
  const response = await request.get('https://api.example.com/users');

  expect(response.status()).toBe(200);

  const users = await response.json();
  expect(Array.isArray(users)).toBeTruthy();
  expect(users.length).toBeGreaterThan(0);
});
```

GET with Query Parameters:

```
test('GET with query params', async ({ request }) => {
  const response = await request.get('https://api.example.com/users', {
    params: {
      page: 1,
      limit: 10,
      sort: 'created_at',
      order: 'desc',
    },
  });

  expect(response.status()).toBe(200);

  const data = await response.json();
  expect(data.length).toBeLessThanOrEqual(10);
```

POST Request:

```
test('POST request - create user', async ({ request }) => {
  const newUser = {
    name: 'Jane Doe',
    email: 'jane@example.com',
    age: 28,
  };

  const response = await request.post('https://api.example.com/users', {
    data: newUser,
  });

  expect(response.status()).toBe(201);

  const createdUser = await response.json();
  expect(createdUser.name).toBe(newUser.name);
  expect(createdUser.email).toBe(newUser.email);
  expect(createdUser.id).toBeDefined();
});
```

PUT Request:

```
test('PUT request - update user', async ({ request }) => {
  const updatedUser = {
    name: 'John Updated',
    email: 'john.updated@example.com',
    age: 31,
  };

  const response = await request.put('https://api.example.com/users/123',
    data: updatedUser,
  });

  expect(response.status()).toBe(200);

  const user = await response.json();
  expect(user.name).toBe(updatedUser.name);
  expect(user.email).toBe(updatedUser.email);
});
```

PATCH Request:

```
  };

  const response = await request.patch('https://api.example.com/users/123'
    data: updates,
  });

  expect(response.status()).toBe(200);

  const user = await response.json();
  expect(user.email).toBe(updates.email);
});
```

DELETE Request:

```
test('DELETE request', async ({ request }) => {
  const response = await request.delete('https://api.example.com/users/123

  expect(response.status()).toBe(204);

  // Verify deletion
  const getResponse = await request.get('https://api.example.com/users/123
  expect(getResponse.status()).toBe(404);
});
```

Custom Headers

Add headers to requests:

```
test('request with custom headers', async ({ request }) => {
  const response = await request.get('https://api.example.com/users', {
    headers: {
      'X-Custom-Header': 'custom-value',
      'Accept': 'application/json',
      'User-Agent': 'Playwright-Test',
    },
  });

  expect(response.status()).toBe(200);
});
```

```
test('request with auth token', async ({ request }) => {
  const token = 'eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...';

  const response = await request.get('https://api.example.com/protected',
    headers: {
      'Authorization': `Bearer ${token}`,
    },
  });

  expect(response.status()).toBe(200);
});
```

Request Context Configuration

Configure base URL and default headers:

```
import { test as base } from '@playwright/test';

const test = base.extend({
  apiContext: async ({ playwright }, use) => {
    const context = await playwright.request.newContext({
      baseURL: 'https://api.example.com',
      extraHTTPHeaders: {
        'Accept': 'application/json',
        'Content-Type': 'application/json',
        'Authorization': 'Bearer token123',
      },
    });

    await use(context);
    await context.dispose();
  },
});

test('use configured context', async ({ apiContext }) => {
  // baseURL is automatically prepended
  const response = await apiContext.get('/users');
  expect(response.status()).toBe(200);
});
```

Response Validation

Status Code Validation:

```
test('validate status codes', async ({ request }) => {
  // Success
  const response1 = await request.get('https://api.example.com/users');
  expect(response1.ok()).toBeTruthy();
  expect(response1.status()).toBe(200);

  // Created
  const response2 = await request.post('https://api.example.com/users', {
    data: { name: 'Test' },
  });
  expect(response2.status()).toBe(201);

  // Not Found
  const response3 = await request.get('https://api.example.com/users/99999
  expect(response3.status()).toBe(404);
});
```

Response Headers Validation:

```
test('validate response headers', async ({ request }) => {
  const response = await request.get('https://api.example.com/users');

  // Get headers
  const headers = response.headers();

  // Validate specific headers
  expect(headers['content-type']).toContain('application/json');
  expect(headers['cache-control']).toBeDefined();

  // Check header value
  const contentType = response.headers()['content-type'];
  expect(contentType).toBe('application/json; charset=utf-8');
});
```

Response Body Validation:

```
  expect(response.ok()).toBeTruthy();

  const user = await response.json();

  // Validate structure
  expect(user).toHaveProperty('id');
  expect(user).toHaveProperty('name');
  expect(user).toHaveProperty('email');

  // Validate values
  expect(user.id).toBe(1);
  expect(typeof user.name).toBe('string');
  expect(user.email).toMatch(/^[\w-\.]+@([\w-]+\.)+[\w-]{2,4}$/);
});
```

Array Response Validation:

```
test('validate array response', async ({ request }) => {
  const response = await request.get('https://api.example.com/users');

  const users = await response.json();

  // Validate array
  expect(Array.isArray(users)).toBeTruthy();
  expect(users.length).toBeGreaterThan(0);

  // Validate first item
  expect(users[0]).toHaveProperty('id');
  expect(users[0]).toHaveProperty('name');

  // Validate all items have required fields
  users.forEach(user => {
    expect(user.id).toBeDefined();
    expect(user.name).toBeDefined();
    expect(user.email).toBeDefined();
  });
});
```

Schema Validation

Using JSON Schema:

```
const userSchema = {
  type: 'object',
  properties: {
    id: { type: 'number' },
    name: { type: 'string' },
    email: { type: 'string', format: 'email' },
    age: { type: 'number', minimum: 0 },
    active: { type: 'boolean' },
  },
  required: ['id', 'name', 'email'],
  additionalProperties: true,
};

test('validate response schema', async ({ request }) => {
  const response = await request.get('https://api.example.com/users/1');
  const user = await response.json();

  const validate = ajv.compile(userSchema);
  const valid = validate(user);

  if (!valid) {
    console.error('Schema validation errors:', validate.errors);
  }

  expect(valid).toBeTruthy();
});
```

Array schema validation:

```
const usersArraySchema = {
  type: 'array',
  items: {
    type: 'object',
    properties: {
      id: { type: 'number' },
      name: { type: 'string' },
      email: { type: 'string' },
    },
    required: ['id', 'name', 'email'],
  },
  minItems: 1,
};
```

```
  const users = await response.json();

  const validate = ajv.compile(usersArraySchema);
  expect(validate(users)).toBeTruthy();
});
```

Response Time Validation

Measure response time:

```
test('validate response time', async ({ request }) => {
  const startTime = Date.now();

  const response = await request.get('https://api.example.com/users');

  const endTime = Date.now();
  const responseTime = endTime - startTime;

  console.log(`Response time: ${responseTime}ms`);

  // Assert response time SLA
  expect(responseTime).toBeLessThan(2000); // Under 2 seconds
  expect(response.status()).toBe(200);
});
```

Using response timing:

```
test('detailed timing info', async ({ request }) => {
  const response = await request.get('https://api.example.com/users');

  // Get timing information
  const timing = response.timing();

  console.log('Start time:', timing.startTime);
  console.log('Domain lookup:', timing.domainLookupEnd - timing.domainLook
  console.log('Connect:', timing.connectEnd - timing.connectStart);
  console.log('Request:', timing.requestStart);
  console.log('Response:', timing.responseStart);
  console.log('Response End:', timing.responseEnd);
});
```

```
test('validate data types', async ({ request }) => {
  const response = await request.get('https://api.example.com/users/1');
  const user = await response.json();

  // Type assertions
  expect(typeof user.id).toBe('number');
  expect(typeof user.name).toBe('string');
  expect(typeof user.email).toBe('string');
  expect(typeof user.active).toBe('boolean');

  // Value range validation
  expect(user.age).toBeGreaterThanOrEqual(0);
  expect(user.age).toBeLessThan(150);

  // Pattern validation
  expect(user.email).toMatch(/^[\w-\.]+@([\w-]+\.)+[\w-]{2,4}$/);
  expect(user.phone).toMatch(/^\+?[\d\s-()]+$/);
});
```

Error Response Validation

Validate error responses:

```
test('validate error response', async ({ request }) => {
  const response = await request.get('https://api.example.com/users/99999'

  expect(response.status()).toBe(404);

  const error = await response.json();

  // Validate error structure
  expect(error).toHaveProperty('error');
  expect(error).toHaveProperty('message');
  expect(error).toHaveProperty('statusCode');

  // Validate error content
  expect(error.statusCode).toBe(404);
  expect(error.error).toBe('Not Found');
  expect(error.message).toContain('User not found');
});
```

```
async function validateSuccessResponse(
  response: APIResponse,
  expectedStatus: number = 200
) {
  expect(response.status()).toBe(expectedStatus);
  expect(response.ok()).toBeTruthy();

  const contentType = response.headers()['content-type'];
  expect(contentType).toContain('application/json');

  const data = await response.json();
  expect(data).toBeDefined();

  return data;
}

test('use validation helper', async ({ request }) => {
  const response = await request.get('https://api.example.com/users/1');
  const user = await validateSuccessResponse(response, 200);

  expect(user.id).toBe(1);
});
```

Debugging API Tests

Console Logging

Log request/response details:

```
test('log API details', async ({ request }) => {
  const url = 'https://api.example.com/users';

  console.log(`Making GET request to: ${url}`);

  const response = await request.get(url);

  console.log('Response Status:', response.status());
  console.log('Response Status Text:', response.statusText());
  console.log('Response Headers:', response.headers());
```

```
});
```

Log timing information:

```
test('log request timing', async ({ request }) => {
  const startTime = Date.now();

  const response = await request.get('https://api.example.com/users');

  const duration = Date.now() - startTime;

  console.log(`Request completed in ${duration}ms`);
  console.log('Status:', response.status());
});
```

Response Body Inspection

Pretty print JSON:

```
test('inspect response body', async ({ request }) => {
  const response = await request.get('https://api.example.com/users/1');

  const user = await response.json();

  // Pretty print
  console.log('User data:');
  console.log(JSON.stringify(user, null, 2));

  // Log specific fields
  console.log('User ID:', user.id);
  console.log('User Name:', user.name);
  console.log('User Email:', user.email);
});
```

Request/Response Logging Utility

Create logging helper:

```
  response: APIResponse,
  requestBody?: any
) {
  console.log('\n=== API Call ===');
  console.log(`${method} ${url}`);

  if (requestBody) {
    console.log('Request Body:', JSON.stringify(requestBody, null, 2));
  }

  console.log('\nResponse Status:', response.status(), response.statusText
  console.log('Response Headers:', response.headers());

  const contentType = response.headers()['content-type'];
  if (contentType?.includes('application/json')) {
    const body = await response.json();
    console.log('Response Body:', JSON.stringify(body, null, 2));
  } else {
    const text = await response.text();
    console.log('Response Body:', text);
  }

  console.log('================\n');
}

test('use logging utility', async ({ request }) => {
  const newUser = { name: 'Test User', email: 'test@example.com' };
  const response = await request.post('https://api.example.com/users', {
    data: newUser,
  });

  await logAPICall('POST', 'https://api.example.com/users', response, newU
});
```

Debugging Failed Tests

Capture failure details:

```
test('debug failed API call', async ({ request }) => {
  try {
    const response = await request.get('https://api.example.com/users/inva
```

```
    console.error('Test failed!');
    console.error('Error:', error);

    // Re-throw for test failure
    throw error;
  }
});
```

After each hook for debugging:

```
test.afterEach(async ({}, testInfo) => {
  if (testInfo.status !== testInfo.expectedStatus) {
    console.log('\n=== Test Failed ===');
    console.log('Test:', testInfo.title);
    console.log('Status:', testInfo.status);
    console.log('Error:', testInfo.error);
    console.log('==================\n');
  }
});

test('failing test', async ({ request }) => {
  const response = await request.get('https://api.example.com/nonexistent'
  expect(response.status()).toBe(200); // This will fail
});
```

Network Debugging

Log all API calls in test:

```
test('track all API calls', async ({ request }) => {
  const apiCalls: Array<{
    method: string;
    url: string;
    status: number;
  }> = [];

  // Note: This is conceptual - actual implementation may vary
  const originalGet = request.get.bind(request);
  request.get = async (url: string, options?: any) => {
    const response = await originalGet(url, options);
    apiCalls.push({
```

```
    });
    return response;
  };

  // Your test code
  await request.get('https://api.example.com/users');
  await request.get('https://api.example.com/posts');

  console.log('API Calls Made:', apiCalls);
});
```

Compare Expected vs Actual

Detailed comparison:

```
test('compare responses', async ({ request }) => {
  const expected = {
    id: 1,
    name: 'John Doe',
    email: 'john@example.com',
    active: true,
  };

  const response = await request.get('https://api.example.com/users/1');
  const actual = await response.json();

  console.log('Expected:', JSON.stringify(expected, null, 2));
  console.log('Actual:', JSON.stringify(actual, null, 2));

  // Compare each field
  Object.keys(expected).forEach(key => {
    const expectedValue = expected[key];
    const actualValue = actual[key];

    if (expectedValue !== actualValue) {
      console.log(`❌ Mismatch in ${key}:`);
      console.log(`   Expected: ${expectedValue}`);
      console.log(`   Actual: ${actualValue}`);
    } else {
      console.log(`✓ ${key} matches`);
    }
  });
```

Practice Exercises

Exercise 1: Complete CRUD API Tests

Task: Implement full CRUD test suite

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

const BASE_URL = 'https://jsonplaceholder.typicode.com';

test.describe('Users API CRUD Operations', () => {
  test('GET all users', async ({ request }) => {
    // 1. Make GET request to /users
    // 2. Validate status code 200
    // 3. Validate response is an array
    // 4. Validate array has at least 1 user
    // 5. Validate first user has required fields

    // Your implementation
  });

  test('GET single user', async ({ request }) => {
    // 1. Get user with ID 1
    // 2. Validate status 200
    // 3. Validate user has: id, name, email, address, phone
    // 4. Validate email format

    // Your implementation
  });

  test('POST create user', async ({ request }) => {
    // 1. Create new user with name, email, username
    // 2. Validate status 201
    // 3. Validate response contains id
    // 4. Validate returned data matches sent data

    // Your implementation
  });

  test('PUT update user', async ({ request }) => {
```

```
    // Your implementation
  });

  test('PATCH partial update', async ({ request }) => {
    // 1. Update only email for user 1
    // 2. Validate status 200
    // 3. Validate email changed, other fields intact

    // Your implementation
  });

  test('DELETE user', async ({ request }) => {
    // 1. Delete user 1
    // 2. Validate status 200 or 204
    // 3. Optional: Verify user is deleted (may not work with dummy API)

    // Your implementation
  });
});
```

Exercise 2: Response Validation Framework

Task: Build comprehensive validation utility

```
 // Create a validation utility class

interface ValidationResult {
  valid: boolean;
  errors: string[];
}

class APIValidator {
  async validateStatus(
    response: APIResponse,
    expectedStatus: number
  ): Promise<ValidationResult> {
    // Implementation
  }

  async validateSchema(
    response: APIResponse,
    schema: any
```

```
  async validateResponseTime(
    response: APIResponse,
    maxTime: number
  ): Promise<ValidationResult> {
    // Implementation
  }

  async validateHeaders(
    response: APIResponse,
    expectedHeaders: Record<string, string>
  ): Promise<ValidationResult> {
    // Implementation
  }
}

// Use in tests
test('use validation framework', async ({ request }) => {
  const validator = new APIValidator();
  const response = await request.get('https://api.example.com/users');

  const statusResult = await validator.validateStatus(response, 200);
  expect(statusResult.valid).toBeTruthy();

  // More validations...
});
```

Exercise 3: Pagination Testing

Task: Test paginated API responses

```
test.describe('Pagination Tests', () => {
  test('first page', async ({ request }) => {
    // 1. Get first page (page=1, limit=10)
    // 2. Validate 10 or fewer items returned
    // 3. Validate pagination metadata

    // Your implementation
  });

  test('navigate pages', async ({ request }) => {
    // 1. Get page 1, store first item
    // 2. Get page 2
```

```
  });

  test('different page sizes', async ({ request }) => {
    // Test with limit=5, limit=10, limit=20
    // Validate correct number of items returned

    // Your implementation
  });

  test('last page', async ({ request }) => {
    // 1. Get total count
    // 2. Calculate last page
    // 3. Get last page
    // 4. Validate it's the last

    // Your implementation
  });
});
```

Exercise 4: Query Parameters Testing

Task: Test various query parameter combinations

```
test.describe('Query Parameters', () => {
  test('filter by field', async ({ request }) => {
    // Filter users by name, email, or other field
    // Your implementation
  });

  test('sort ascending', async ({ request }) => {
    // Sort by id ascending
    // Validate order
    // Your implementation
  });

  test('sort descending', async ({ request }) => {
    // Sort by id descending
    // Validate order
    // Your implementation
  });

  test('multiple filters', async ({ request }) => {
    // Combine multiple query parameters
```

Exercise 5: Error Handling Tests

Task: Test error scenarios

```
test.describe('Error Scenarios', () => {
  test('404 not found', async ({ request }) => {
    // Request non-existent resource
    // Validate 404 status
    // Validate error message

    // Your implementation
  });

  test('400 bad request', async ({ request }) => {
    // Send invalid data
    // Validate 400 status
    // Validate error details

    // Your implementation
  });

  test('422 validation error', async ({ request }) => {
    // Send data that fails validation
    // Validate 422 status
    // Validate validation errors

    // Your implementation
  });
});
```

Exercise 6: Response Time Performance

Task: Create performance test suite

```
test.describe('Performance Tests', () => {
  test('average response time', async ({ request }) => {
    const times: number[] = [];

    // Make 10 requests
```

```
    // Your implementation
  });

  test('concurrent requests', async ({ request }) => {
    // Make 5 concurrent requests
    // Measure total time
    // Compare with sequential time

    // Your implementation
  });
});
```

Exercise 7: Data-Driven API Tests

Task: Parameterize API tests

```
 const testUsers = [
  { id: 1, expectedName: 'Leanne Graham' },
  { id: 2, expectedName: 'Ervin Howell' },
  { id: 3, expectedName: 'Clementine Bauch' },
];

test.describe('Data-Driven Tests', () => {
  for (const user of testUsers) {
    test(`verify user ${user.id}`, async ({ request }) => {
      // Get user by id
      // Validate name matches expected

      // Your implementation
    });
  }
});
```

Summary

In Day 6, you learned:

- Resource-based URLs
- API endpoint structure
- Request/response components

✅ HTTP Methods

- GET, POST, PUT, PATCH, DELETE
- Idempotency and safety
- When to use each method

✅ HTTP Status Codes

- 2xx (Success)
- 3xx (Redirection)
- 4xx (Client Errors)
- 5xx (Server Errors)

✅ Playwright API Testing

- APIRequestContext
- Making HTTP requests
- Custom headers
- Request configuration

✅ Response Validation

- Status code validation
- Header validation
- Body validation
- Schema validation
- Response time validation
- Data type validation

✅ Debugging

- Error handling
- Comparison utilities

Key Takeaways

- **Playwright** provides built-in API testing
- **Single framework** for UI and API tests
- **Comprehensive validation** ensures quality
- **Debugging tools** help troubleshoot issues
- **Type safety** with TypeScript

Best Practices

1. Validate status codes first
2. Check response structure before values
3. Use schema validation for complex responses
4. Log failures with details
5. Test error scenarios
6. Measure response times
7. Use helper functions for common validations

Next Steps

- Practice CRUD operations
- Build validation frameworks
- Test error scenarios
- Combine UI and API tests
- Learn authentication patterns

End of Day 6 Documentation