← Playwright Advanced Scenarios

API Testing Fundamentals →

Playwright API Testing

Table of Contents

REST API Concepts and Architecture

What is REST API?

REST (Representational State Transfer) is an architectural style for designing networked applications. It uses HTTP requests to access and manipulate data.

Key Characteristics:

- **Stateless**: Each request contains all information needed
- **Client-Server**: Separation of concerns
- **Cacheable**: Responses can be cached

RESTful Principles

1. Resource-Based:

- Everything is a resource (users, products, orders)
- Each resource has a unique URI
- Resources are nouns, not verbs

```
Good URIs:
  GET /api/users
  GET /api/users/123
  GET /api/products
  POST /api/orders

Bad URIs:
  GET /api/getUsers
  POST /api/createUser
  GET /api/deleteProduct/123
```

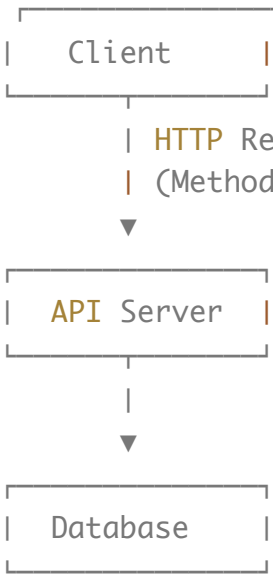
2. HTTP Methods (Verbs):

- `GET` - Retrieve resource(s)
- `POST` - Create new resource
- `PUT` - Update entire resource
- `PATCH` - Update partial resource
- `DELETE` - Delete resource

3. Status Codes:

- `2xx` - Success
- `3xx` - Redirection
- `4xx` - Client errors
- `5xx` - Server errors

4. JSON Format: Most modern REST APIs use JSON for data exchange.



Request Structure

HTTP Request Components:

```
POST /api/users HTTP/1.1
Host: api.example.com
Content-Type: application/json
Authorization: Bearer token123
Accept: application/json

{
  "name": "John Doe",
  "email": "john@example.com",
  "role": "user"
}
```

Components:

1. **Method**: POST
2. **URL**: /api/users
3. **Headers**: Metadata about request
4. **Body**: Data being sent (for POST/PUT/PATCH)

```
HTTP/1.1 201 Created
Content-Type: application/json
Date: Mon, 27 Jan 2026 10:00:00 GMT

{
  "id": 123,
  "name": "John Doe",
  "email": "john@example.com",
  "role": "user",
  "createdAt": "2026-01-27T10:00:00Z"
}
```

Components:

1. **Status Code**: 201 Created
2. **Headers**: Metadata about response
3. **Body**: Response data

Common API Patterns

CRUD Operations:

```
Create  → POST   /api/users
Read    → GET    /api/users/{id}
Update  → PUT    /api/users/{id}
Delete  → DELETE /api/users/{id}
List    → GET    /api/users
```

Collection vs Resource:

```
Collection: /api/users          (multiple resources)
Resource:   /api/users/123      (single resource)
```

Nested Resources:

Query Parameters:

```
 GET /api/users?page=1&limit=10              (pagination)
GET /api/products?category=electronics      (filtering)
GET /api/users?sort=name&order=asc          (sorting)
GET /api/products?search=laptop             (searching)
```

HTTP Methods and Status Codes

HTTP Methods in Detail

GET - Retrieve Data

```
 Purpose: Fetch resource(s)
Request Body: No
Response Body: Yes
Idempotent: Yes (same result on repeated calls)
Safe: Yes (no side effects)

Examples:
GET /api/users              → Get all users
GET /api/users/123          → Get user with ID 123
GET /api/products?page=2    → Get products page 2
```

POST - Create Resource

```
 Purpose: Create new resource
Request Body: Yes
Response Body: Yes (created resource)
Idempotent: No
Safe: No

Example:
```

```
  "email": "alice@example.com"
}

Response:
201 Created
{
  "id": 124,
  "name": "Alice",
  "email": "alice@example.com",
  "createdAt": "2026-01-27T10:00:00Z"
}
```

PUT - Update Entire Resource

```
 Purpose: Replace entire resource
Request Body: Yes (complete resource)
Response Body: Yes (updated resource)
Idempotent: Yes
Safe: No

Example:
PUT /api/users/123
{
  "name": "Alice Smith",
  "email": "alice.smith@example.com",
  "role": "admin"
}
```

PATCH - Partial Update

```
 Purpose: Update specific fields
Request Body: Yes (only fields to update)
Response Body: Yes (updated resource)
Idempotent: Yes
Safe: No

Example:
PATCH /api/users/123
{
  "email": "newemail@example.com"
}
```

```
Purpose: Delete resource
Request Body: No
Response Body: Optional
Idempotent: Yes
Safe: No

Example:
DELETE /api/users/123

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

HTTP Status Codes

1xx - Informational

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

2xx - Success

- `200 OK` - Request succeeded
- `201 Created` - Resource created
- `202 Accepted` - Request accepted, processing not complete
- `204 No Content` - Success, no response body

3xx - Redirection

- `301 Moved Permanently` - Resource moved permanently
- `302 Found` - Temporary redirect
- `304 Not Modified` - Cached version is still valid

4xx - Client Errors

- `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 errors
- `429 Too Many Requests` - Rate limit exceeded

5xx - Server Errors

- `500 Internal Server Error` - Generic server error
- `501 Not Implemented` - Server doesn't support functionality
- `502 Bad Gateway` - Invalid response from upstream
- `503 Service Unavailable` - Server temporarily unavailable
- `504 Gateway Timeout` - Upstream server timeout

Common Headers

Request Headers:

```
Content-Type: application/json          (request body format)
Accept: application/json                (desired response format)
Authorization: Bearer token123          (authentication)
User-Agent: Playwright/1.40.0           (client identifier)
Cookie: session=abc123                  (session info)
```

Response Headers:

```
Content-Type: application/json          (response body format)
Content-Length: 1234                    (body size in bytes)
Cache-Control: no-cache                 (caching policy)
Set-Cookie: session=xyz789              (set cookie)
Location: /api/users/123                (redirect or created resource)
```

---

context)

Introduction to Playwright API Testing

Playwright provides a built-in **APIRequestContext** for making HTTP requests. This allows you to test APIs directly without a browser.

Benefits:

- Fast execution (no browser overhead)
- Easy setup and configuration
- Built-in assertions
- Integration with UI tests
- Parallel execution
- Request/response logging

Basic API Request

Import and setup:

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

test('basic GET request', async ({ request }) => {
  // Make GET request
  const response = await request.get('https://jsonplaceholder.typicode.com

  // Verify status code
  expect(response.status()).toBe(200);

  // Get response body
  const users = await response.json();

  // Verify response
  expect(Array.isArray(users)).toBe(true);
  expect(users.length).toBeGreaterThan(0);

  console.log('First user:', users[0]);
});
```

```
test('GET single user', async ({ request }) => {
  const response = await request.get('https://jsonplaceholder.typicode.com

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

  const user = await response.json();

  expect(user).toHaveProperty('id', 1);
  expect(user).toHaveProperty('name');
  expect(user).toHaveProperty('email');

  console.log('User:', user);
});
```

GET with query parameters:

```
test('GET with query params', async ({ request }) => {
  const response = await request.get('https://jsonplaceholder.typicode.com
    params: {
      userId: 1,
      _limit: 5,
    },
  });

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

  const posts = await response.json();
  expect(posts.length).toBeLessThanOrEqual(5);

  // Verify all posts belong to userId 1
  posts.forEach((post: any) => {
    expect(post.userId).toBe(1);
  });
});
```

GET with headers:

```
      'Authorization': 'Bearer your-token-here',
      'Accept': 'application/json',
      'X-Custom-Header': 'value',
    },
  });

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

POST Requests

Create resource with POST:

```
test('POST create user', async ({ request }) => {
  const newUser = {
    name: 'John Doe',
    email: 'john@example.com',
    username: 'johndoe',
  };

  const response = await request.post('https://jsonplaceholder.typicode.co
    data: newUser,
  });

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

  const createdUser = await response.json();

  expect(createdUser).toHaveProperty('id');
  expect(createdUser.name).toBe(newUser.name);
  expect(createdUser.email).toBe(newUser.email);

  console.log('Created user:', createdUser);
});
```

POST with JSON data:

```
test('POST with complex data', async ({ request }) => {
  const postData = {
    title: 'Test Post',
```

```
    metadata: {
      category: 'testing',
      priority: 'high',
    },
  };

  const response = await request.post('https://jsonplaceholder.typicode.co
    data: postData,
    headers: {
      'Content-Type': 'application/json',
    },
  });

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

  const result = await response.json();
  expect(result.title).toBe(postData.title);
});
```

POST with form data:

```
test('POST form data', async ({ request }) => {
  const formData = new FormData();
  formData.append('name', 'John Doe');
  formData.append('email', 'john@example.com');
  formData.append('age', '30');

  const response = await request.post('https://api.example.com/users', {
    multipart: {
      name: 'John Doe',
      email: 'john@example.com',
      age: '30',
    },
  });

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

PUT Requests

Update resource with PUT:

```
    name: 'Updated Name',
    email: 'updated@example.com',
    username: 'updateduser',
  };

  const response = await request.put('https://jsonplaceholder.typicode.com
    data: updatedUser,
  });

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

  const result = await response.json();

  expect(result.id).toBe(1);
  expect(result.name).toBe(updatedUser.name);
  expect(result.email).toBe(updatedUser.email);
});
```

PATCH Requests

Partial update with PATCH:

```
test('PATCH partial update', async ({ request }) => {
  const partialUpdate = {
    email: 'newemail@example.com',
  };

  const response = await request.patch('https://jsonplaceholder.typicode.c
    data: partialUpdate,
  });

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

  const result = await response.json();

  expect(result.email).toBe(partialUpdate.email);
  expect(result).toHaveProperty('name'); // Other fields still exist
});
```

DELETE Requests

```
test('DELETE user', async ({ request }) => {
  const response = await request.delete('https://jsonplaceholder.typicode.

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

  // Verify deletion
  const getResponse = await request.get('https://jsonplaceholder.typicode.
  // Depending on API, might return 404 or empty object
});
```

API Context Configuration

Create API context with base configuration:

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

// Extend test with configured API context
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 your-token-here',
      },
    });

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

// Use configured context
test('test with configured context', async ({ apiContext }) => {
  // baseURL is already set
  const response = await apiContext.get('/users');

  expect(response.ok()).toBe(true);
});
```

Configure in playwright.config.ts:

```
  use: {
    baseURL: 'https://api.example.com',
    extraHTTPHeaders: {
      'Accept': 'application/json',
    },
  },
});
```

Timeout Configuration

Set timeout for API requests:

```
test('API with timeout', async ({ request }) => {
  // Default timeout is 30 seconds
  const response = await request.get('https://api.example.com/slow-endpoin
    timeout: 60000, // 60 seconds
  });

  expect(response.ok()).toBe(true);
});
```

Configure globally:

```
// playwright.config.ts
export default defineConfig({
  use: {
    actionTimeout: 15000, // 15 seconds for all actions
  },
});
```

Request and Response Validation

Status Code Assertions

```
test('status code assertions', async ({ request }) => {
  // Success
  const getResponse = await request.get('https://api.example.com/users');
  expect(getResponse.ok()).toBe(true);
  expect(getResponse.status()).toBe(200);

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

  // No Content
  const deleteResponse = await request.delete('https://api.example.com/use
  expect(deleteResponse.status()).toBe(204);

  // Not Found
  const notFoundResponse = await request.get('https://api.example.com/user
  expect(notFoundResponse.status()).toBe(404);

  // Server Error
  const errorResponse = await request.get('https://api.example.com/error')
  expect(errorResponse.status()).toBeGreaterThanOrEqual(500);
});
```

Response Header Validation

Check response headers:

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

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

  const headers = response.headers();

  // Check Content-Type
  expect(headers['content-type']).toContain('application/json');

  // Check custom headers
  expect(headers).toHaveProperty('x-rate-limit-limit');
  expect(headers).toHaveProperty('x-rate-limit-remaining');
```

Specific header checks:

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

  // Content-Type
  const contentType = response.headers()['content-type'];
  expect(contentType).toBe('application/json; charset=utf-8');

  // Rate limiting
  const rateLimit = response.headers()['x-rate-limit-remaining'];
  expect(parseInt(rateLimit)).toBeGreaterThan(0);

  // CORS headers
  const corsHeader = response.headers()['access-control-allow-origin'];
  expect(corsHeader).toBeDefined();
});
```

Response Body Validation

JSON structure validation:

```
test('validate JSON structure', async ({ request }) => {
  const response = await request.get('https://jsonplaceholder.typicode.com

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

  const user = await response.json();

  // Check properties exist
  expect(user).toHaveProperty('id');
  expect(user).toHaveProperty('name');
  expect(user).toHaveProperty('email');
  expect(user).toHaveProperty('address');

  // Check property types
  expect(typeof user.id).toBe('number');
  expect(typeof user.name).toBe('string');
  expect(typeof user.email).toBe('string');

  // Check nested objects
```

```
});
```

Array response validation:

```
test('validate array response', async ({ request }) => {
  const response = await request.get('https://jsonplaceholder.typicode.com

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

  const users = await response.json();

  // Check it's an array
  expect(Array.isArray(users)).toBe(true);

  // Check array length
  expect(users.length).toBeGreaterThan(0);
  expect(users.length).toBeLessThanOrEqual(10);

  // Validate each item
  users.forEach((user: any) => {
    expect(user).toHaveProperty('id');
    expect(user).toHaveProperty('name');
    expect(user).toHaveProperty('email');

    // Email format validation
    expect(user.email).toMatch(/^[^\s@]+@[^\s@]+\.[^\s@]+$/);
  });
});
```

Specific value assertions:

```
test('validate specific values', async ({ request }) => {
  const response = await request.get('https://jsonplaceholder.typicode.com

  const user = await response.json();

  // Exact values
  expect(user.id).toBe(1);
  expect(user.name).toBe('Leanne Graham');

  // String contains
  expect(user.email).toContain('@');
```

```
  // Nested object values
  expect(user.address.city).toBe('Gwenborough');
  expect(user.company.name).toBe('Romaguera-Crona');
});
```

Complex Validation

Deep object comparison:

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

  const user = await response.json();

  // Expected structure
  const expectedStructure = {
    id: expect.any(Number),
    name: expect.any(String),
    email: expect.stringMatching(/^[^\s@]+@[^\s@]+\.[^\s@]+$/),
    role: expect.stringMatching(/^(admin|user|guest)$/),
    isActive: expect.any(Boolean),
    createdAt: expect.stringMatching(/^\d{4}-\d{2}-\d{2}T/),
    profile: {
      age: expect.any(Number),
      country: expect.any(String),
    },
  };

  expect(user).toMatchObject(expectedStructure);
});
```

Custom validation functions:

```
function validateUser(user: any) {
  // Required fields
  expect(user).toHaveProperty('id');
  expect(user).toHaveProperty('name');
  expect(user).toHaveProperty('email');

  // Type validation
```

```
  // Format validation
  expect(user.email).toMatch(/^[^\s@]+@[^\s@]+\.[^\s@]+$/);

  // Value constraints
  expect(user.id).toBeGreaterThan(0);
  expect(user.name.length).toBeGreaterThan(0);

  return true;
}

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

  users.forEach((user: any) => {
    expect(validateUser(user)).toBe(true);
  });
});
```

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();

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

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

Validation error response:

```
    email: 'invalid-email', // Invalid email format
  };

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

  expect(response.status()).toBe(422);

  const error = await response.json();

  expect(error).toHaveProperty('errors');
  expect(Array.isArray(error.errors)).toBe(true);

  // Check specific validation errors
  const nameError = error.errors.find((e: any) => e.field === 'name');
  expect(nameError).toBeDefined();
  expect(nameError.message).toContain('required');

  const emailError = error.errors.find((e: any) => e.field === 'email');
  expect(emailError).toBeDefined();
  expect(emailError.message).toContain('invalid');
});
```

Debugging API Tests

Console Logging

Basic logging:

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

  console.log('Status:', response.status());
  console.log('Headers:', response.headers());

  const body = await response.json();
  console.log('Body:', JSON.stringify(body, null, 2));
```

Request logging:

```
test('log request details', async ({ request }) => {
  console.log('Making request to:', 'https://api.example.com/users');
  console.log('Method:', 'POST');

  const requestData = {
    name: 'Test User',
    email: 'test@example.com',
  };

  console.log('Request data:', JSON.stringify(requestData, null, 2));

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

  console.log('Response status:', response.status());
  console.log('Response body:', await response.json());
});
```

Response Time Measurement

Measure API performance:

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

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

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

  console.log(`API response time: ${duration}ms`);

  expect(response.status()).toBe(200);
  expect(duration).toBeLessThan(2000); // Should respond within 2 seconds
});
```

Track multiple requests:

```
  // Request 1
  let start = Date.now();
  await request.get('https://api.example.com/users');
  timings.push({ endpoint: '/users', duration: Date.now() - start });

  // Request 2
  start = Date.now();
  await request.get('https://api.example.com/products');
  timings.push({ endpoint: '/products', duration: Date.now() - start });

  // Request 3
  start = Date.now();
  await request.get('https://api.example.com/orders');
  timings.push({ endpoint: '/orders', duration: Date.now() - start });

  console.log('API Performance:');
  timings.forEach(t => {
    console.log(`${t.endpoint}: ${t.duration}ms`);
  });

  // Assert all under 2 seconds
  timings.forEach(t => {
    expect(t.duration).toBeLessThan(2000);
  });
});
```

Request/Response Interceptor

Create logging utility:

```
 async function loggedRequest(
  request: any,
  method: string,
  url: string,
  options?: any
) {
  console.log(`\n=== ${method} ${url} ===`);
  console.log('Request options:', JSON.stringify(options, null, 2));

  const startTime = Date.now();

  let response;
```

```
      break;
    case 'POST':
      response = await request.post(url, options);
      break;
    case 'PUT':
      response = await request.put(url, options);
      break;
    case 'PATCH':
      response = await request.patch(url, options);
      break;
    case 'DELETE':
      response = await request.delete(url, options);
      break;
    default:
      throw new Error(`Unsupported method: ${method}`);
  }

  const duration = Date.now() - startTime;

  console.log('Response status:', response.status());
  console.log('Response time:', `${duration}ms`);
  console.log('Response headers:', response.headers());

  try {
    const body = await response.json();
    console.log('Response body:', JSON.stringify(body, null, 2));
  } catch (e) {
    console.log('Response body: (not JSON)');
  }

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

  return response;
}

test('use logged request', async ({ request }) => {
  const response = await loggedRequest(
    request,
    'GET',
    'https://api.example.com/users/1'
  );

  expect(response.ok()).toBe(true);
});
```

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

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

    const user = await response.json();
    expect(user).toHaveProperty('email');

  } catch (error) {
    console.error('Test failed!');
    console.error('Error:', error);

    // Log additional context
    console.error('Current URL:', 'https://api.example.com/users/1');

    throw error;
  }
});
```

Save response for debugging:

```
import fs from 'fs';

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

  const body = await response.text();

  // Save to file
  fs.writeFileSync('debug-response.json', body);
  console.log('Response saved to debug-response.json');

  // Continue with assertions
  const users = JSON.parse(body);
  expect(users.length).toBeGreaterThan(0);
});
```

Exercise 1: CRUD Operations

Task: Implement complete CRUD for Users API

```
test.describe('User CRUD Operations', () => {
  let createdUserId: number;

  test('Create user (POST)', async ({ request }) => {
    // Create a new user
    // Verify response status is 201
    // Verify response contains user ID
    // Store user ID for later tests
  });

  test('Read user (GET)', async ({ request }) => {
    // Use stored user ID
    // Get user details
    // Verify all fields are correct
  });

  test('Update user (PUT)', async ({ request }) => {
    // Update the user
    // Verify response status is 200
    // Verify updated fields
  });

  test('Partial update (PATCH)', async ({ request }) => {
    // Update only email
    // Verify only email changed
    // Other fields remain same
  });

  test('Delete user (DELETE)', async ({ request }) => {
    // Delete the user
    // Verify deletion successful
  });

  test('Verify user deleted (GET)', async ({ request }) => {
    // Try to get deleted user
    // Should return 404
  });
});
```

```
test.describe('Query Parameters', () => {
  test('pagination', async ({ request }) => {
    // Test ?page=1&limit=10
    // Verify correct number of results
    // Test ?page=2&limit=5
    // Verify different results
  });

  test('filtering', async ({ request }) => {
    // Test ?userId=1
    // Verify all results match filter
    // Test multiple filters
  });

  test('sorting', async ({ request }) => {
    // Test ?sort=name&order=asc
    // Verify results are sorted
    // Test desc order
  });

  test('search', async ({ request }) => {
    // Test ?search=john
    // Verify results contain search term
  });
});
```

Exercise 3: Error Handling

Task: Test error scenarios

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

  test('400 bad request', async ({ request }) => {
    // Send invalid data
    // Verify 400 status
```

```
  test('422 validation errors', async ({ request }) => {
    // Send data with validation errors
    // Verify 422 status
    // Check validation error messages
  });

  test('401 unauthorized', async ({ request }) => {
    // Request without auth
    // Verify 401 status
  });

  test('403 forbidden', async ({ request }) => {
    // Request with insufficient permissions
    // Verify 403 status
  });
});
```

Exercise 4: Response Validation

Task: Create comprehensive validation suite

```
test.describe('Response Validation', () => {
  test('validate user schema', async ({ request }) => {
    // Get user
    // Validate all required fields exist
    // Validate field types
    // Validate field formats (email, date, etc.)
  });

  test('validate array response', async ({ request }) => {
    // Get users list
    // Validate it's an array
    // Validate each item structure
    // Validate consistency across items
  });

  test('validate nested objects', async ({ request }) => {
    // Get resource with nested data
    // Validate nested object structure
    // Validate nested array structure
  });

  test('validate relationships', async ({ request }) => {
```

```
  });
});
```

Exercise 5: Performance Testing

Task: Measure and assert API performance

```
test.describe('API Performance', () => {
  test('response time test', async ({ request }) => {
    // Measure response time for single request
    // Assert under threshold (e.g., 1 second)
  });

  test('concurrent requests', async ({ request }) => {
    // Make 10 concurrent requests
    // Measure total time
    // Verify all succeed
    // Assert average response time
  });

  test('sequential requests', async ({ request }) => {
    // Make 10 sequential requests
    // Measure total time
    // Compare with concurrent
  });

  test('large dataset', async ({ request }) => {
    // Request large amount of data
    // Measure response time
    // Verify pagination works
  });
});
```

Exercise 6: Authentication Testing

Task: Test various auth scenarios

```
test.describe('Authentication', () => {
  test('login and get token', async ({ request }) => {
    // POST to /login
```

```
  test('access protected endpoint', async ({ request }) => {
    // Login to get token
    // Use token in Authorization header
    // Access protected resource
    // Verify success
  });

  test('invalid token', async ({ request }) => {
    // Use invalid token
    // Verify 401 response
  });

  test('expired token', async ({ request }) => {
    // Use expired token
    // Verify 401 response
  });

  test('refresh token', async ({ request }) => {
    // Get initial token
    // Use refresh token endpoint
    // Verify new token works
  });
});
```

Exercise 7: Data Dependencies

Task: Test chained API calls

```
test.describe('Data Dependencies', () => {
  test('create and use resource', async ({ request }) => {
    // Create user
    // Use user ID to create post
    // Use post ID to create comment
    // Verify all linked correctly
  });

  test('update cascade', async ({ request }) => {
    // Create parent resource
    // Create child resource
    // Update parent
    // Verify child updated
  });
```

```
    // Delete parent
    // Verify children deleted
  });
});
```

Summary

In Day 6, you learned:

✅ REST API Concepts

- RESTful architecture principles
- Resource-based design
- HTTP methods and their purposes
- API architecture patterns

✅ HTTP Methods

- GET - Retrieve data
- POST - Create resources
- PUT - Full updates
- PATCH - Partial updates
- DELETE - Remove resources
- Idempotency and safety

✅ HTTP Status Codes

- 2xx Success codes
- 4xx Client errors
- 5xx Server errors
- Proper code usage

✅ Playwright API Testing

- Configuring API context
- Setting base URL and headers
- Timeout configuration

✅ Request/Response Validation

- Status code assertions
- Header validation
- JSON structure validation
- Array response validation
- Error response handling
- Custom validation functions

✅ Debugging

- Console logging
- Response time measurement
- Request/response logging
- Failure diagnostics
- Saving responses for analysis

Key Takeaways

- **API testing** is faster than UI testing
- **Playwright** provides built-in API testing
- **Validation** should be comprehensive
- **Error scenarios** are important to test
- **Performance** should be measured
- **Debugging tools** help troubleshoot issues

Best Practices Learned

1. Test all CRUD operations

4. Measure response times
5. Use proper HTTP methods
6. Verify status codes
7. Log for debugging
8. Create reusable utilities

Next Steps

- Practice with real APIs
- Build comprehensive test suites
- Create validation utilities
- Implement performance tests
- Complete all exercises

End of Day 6 Documentation

← Playwright Advanced Scenarios

API Testing Fundamentals →