← API Testing Fundamentals

Cypress Fundamentals →

Advanced Playwright API Testing

Table of Contents

Authentication Mechanisms

Basic Authentication

HTTP Basic Auth uses username and password encoded in Base64.

Implementation:

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

test('basic authentication', async ({ request }) => {
  const username = 'admin';
```

```
  const authHeader = 'Basic ' + Buffer.from(`${username}:${password}`).toS

  const response = await request.get('https://httpbin.org/basic-auth/admin
    headers: {
      'Authorization': authHeader,
    },
  });

  expect(response.status()).toBe(200);
  const data = await response.json();
  expect(data.authenticated).toBe(true);
});
```

Using Playwright's built-in support:

```
test('basic auth with context', async ({ playwright }) => {
  const context = await playwright.request.newContext({
    httpCredentials: {
      username: 'admin',
      password: 'admin123',
    },
  });

  const response = await context.get('https://httpbin.org/basic-auth/admin

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

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

Configure in playwright.config.ts:

```
// playwright.config.ts
import { defineConfig } from '@playwright/test';

export default defineConfig({
  use: {
    httpCredentials: {
      username: process.env.API_USERNAME || 'admin',
      password: process.env.API_PASSWORD || 'admin123',
    },
  },
```

Bearer Token Authentication

Most common authentication method for modern APIs.

Login to get token:

```
test.describe('Bearer Token Authentication', () => {
  let authToken: string;

  test.beforeAll(async ({ request }) => {
    // Login to get token
    const loginResponse = await request.post('https://api.example.com/auth
      data: {
        email: 'user@example.com',
        password: 'password123',
      },
    });

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

    const loginData = await loginResponse.json();
    authToken = loginData.token;

    console.log('Auth token obtained:', authToken);
  });

  test('access protected endpoint', async ({ request }) => {
    const response = await request.get('https://api.example.com/users/me',
      headers: {
        'Authorization': `Bearer ${authToken}`,
      },
    });

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

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

Create authenticated API context:

```
  authenticatedRequest: APIRequestContext;
};

const test = base.extend<AuthFixtures>({
  authenticatedRequest: async ({ playwright }, use) => {
    // Login
    const context = await playwright.request.newContext({
      baseURL: 'https://api.example.com',
    });

    const loginResponse = await context.post('/auth/login', {
      data: {
        email: 'user@example.com',
        password: 'password123',
      },
    });

    const { token } = await loginResponse.json();

    // Create new context with token
    await context.dispose();

    const authContext = await playwright.request.newContext({
      baseURL: 'https://api.example.com',
      extraHTTPHeaders: {
        'Authorization': `Bearer ${token}`,
      },
    });

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

export { test, expect };
```

Using authenticated context:

```
import { test, expect } from './fixtures';

test('use authenticated context', async ({ authenticatedRequest }) => {
  // Already authenticated
  const response = await authenticatedRequest.get('/users/me');
```

```
test('get user profile', async ({ authenticatedRequest }) => {
  const response = await authenticatedRequest.get('/profile');

  const profile = await response.json();
  expect(profile).toHaveProperty('name');
  expect(profile).toHaveProperty('email');
});
```

API Key Authentication

API keys in headers or query parameters.

Header-based API key:

```
test('API key in header', async ({ request }) => {
  const apiKey = process.env.API_KEY || 'your-api-key';

  const response = await request.get('https://api.example.com/data', {
    headers: {
      'X-API-Key': apiKey,
      // or
      'Authorization': `ApiKey ${apiKey}`,
    },
  });

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

Query parameter API key:

```
test('API key in query param', async ({ request }) => {
  const apiKey = process.env.API_KEY || 'your-api-key';

  const response = await request.get('https://api.example.com/data', {
    params: {
      api_key: apiKey,
      // or
      apikey: apiKey,
    },
  });
```

Configure API context with key:

```
const test = base.extend({
  apiContext: async ({ playwright }, use) => {
    const context = await playwright.request.newContext({
      baseURL: 'https://api.example.com',
      extraHTTPHeaders: {
        'X-API-Key': process.env.API_KEY || 'test-key',
      },
    });

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

OAuth 2.0 Authentication

OAuth 2.0 flow for API testing.

Client Credentials Flow:

```
test('OAuth 2.0 client credentials', async ({ request }) => {
  const clientId = process.env.OAUTH_CLIENT_ID;
  const clientSecret = process.env.OAUTH_CLIENT_SECRET;

  // Get access token
  const tokenResponse = await request.post('https://oauth.example.com/toke
    form: {
      grant_type: 'client_credentials',
      client_id: clientId,
      client_secret: clientSecret,
      scope: 'read:users write:users',
    },
  });

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

  const tokenData = await tokenResponse.json();
  const accessToken = tokenData.access_token;
```

```
  // Use access token
  const apiResponse = await request.get('https://api.example.com/users', {
    headers: {
      'Authorization': `Bearer ${accessToken}`,
    },
  });

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

Password Grant Flow:

```
test('OAuth 2.0 password grant', async ({ request }) => {
  const tokenResponse = await request.post('https://oauth.example.com/toke
    form: {
      grant_type: 'password',
      username: 'user@example.com',
      password: 'password123',
      client_id: process.env.OAUTH_CLIENT_ID,
      client_secret: process.env.OAUTH_CLIENT_SECRET,
    },
  });

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

  const { access_token, refresh_token } = await tokenResponse.json();

  // Use access token for API calls
  const response = await request.get('https://api.example.com/profile', {
    headers: {
      'Authorization': `Bearer ${access_token}`,
    },
  });

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

Refresh Token Flow:

```
test('refresh access token', async ({ request }) => {
  const refreshToken = 'stored-refresh-token';
```

```
      grant_type: 'refresh_token',
      refresh_token: refreshToken,
      client_id: process.env.OAUTH_CLIENT_ID,
      client_secret: process.env.OAUTH_CLIENT_SECRET,
    },
  });

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

  const { access_token, refresh_token: newRefreshToken } = await tokenResp

  console.log('New access token:', access_token);
  console.log('New refresh token:', newRefreshToken);
});
```

JWT Token Handling

Working with JSON Web Tokens.

Decode JWT:

```
function decodeJWT(token: string) {
  const parts = token.split('.');
  if (parts.length !== 3) {
    throw new Error('Invalid JWT');
  }

  const payload = Buffer.from(parts[1], 'base64').toString();
  return JSON.parse(payload);
}

test('decode and validate JWT', async ({ request }) => {
  // Login to get JWT
  const loginResponse = await request.post('https://api.example.com/auth/l
    data: {
      email: 'user@example.com',
      password: 'password123',
    },
  });

  const { token } = await loginResponse.json();
```

```
  console.log('Token payload:', decoded);

  // Verify token claims
  expect(decoded).toHaveProperty('sub'); // Subject (user ID)
  expect(decoded).toHaveProperty('exp'); // Expiration
  expect(decoded).toHaveProperty('iat'); // Issued at

  // Check expiration
  const now = Math.floor(Date.now() / 1000);
  expect(decoded.exp).toBeGreaterThan(now);

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

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

Session-Based Authentication

Cookie-based sessions.

Login and save cookies:

```
test('session authentication', async ({ request }) => {
  // Login
  const loginResponse = await request.post('https://api.example.com/auth/l
    data: {
      email: 'user@example.com',
      password: 'password123',
    },
  });

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

  // Cookie is automatically stored in context

  // Subsequent requests use the session cookie
  const profileResponse = await request.get('https://api.example.com/profi
```

Extract and use session cookie:

```
test('manage session cookie', async ({ playwright }) => {
  const context = await playwright.request.newContext({
    baseURL: 'https://api.example.com',
  });

  // Login
  await context.post('/auth/login', {
    data: {
      email: 'user@example.com',
      password: 'password123',
    },
  });

  // Get cookies
  const cookies = await context.storageState();
  console.log('Cookies:', cookies.cookies);

  // Use in new context
  const newContext = await playwright.request.newContext({
    baseURL: 'https://api.example.com',
    storageState: cookies,
  });

  const response = await newContext.get('/profile');
  expect(response.status()).toBe(200);

  await context.dispose();
  await newContext.dispose();
});
```

API Chaining and Data Dependencies

Sequential API Calls

Chain API calls where response from one is used in another.

```
test('API call chaining', async ({ request }) => {
  // Step 1: Create user
  const createResponse = await request.post('https://jsonplaceholder.typic
    data: {
      name: 'John Doe',
      email: 'john@example.com',
    },
  });

  const user = await createResponse.json();
  const userId = user.id;

  console.log('Created user ID:', userId);

  // Step 2: Create post for user
  const postResponse = await request.post('https://jsonplaceholder.typicod
    data: {
      userId: userId,
      title: 'My First Post',
      body: 'This is the post content',
    },
  });

  const post = await postResponse.json();
  const postId = post.id;

  console.log('Created post ID:', postId);

  // Step 3: Add comment to post
  const commentResponse = await request.post('https://jsonplaceholder.typi
    data: {
      postId: postId,
      name: 'Commenter',
      email: 'commenter@example.com',
      body: 'Great post!',
    },
  });

  const comment = await commentResponse.json();

  console.log('Created comment ID:', comment.id);

  // Verify relationships
  expect(post.userId).toBe(userId);
  expect(comment.postId).toBe(postId);
```

Complex workflow:

```
test('complete workflow', async ({ request }) => {
  // 1. Register user
  const registerResponse = await request.post('https://api.example.com/aut
    data: {
      email: 'newuser@example.com',
      password: 'password123',
      name: 'New User',
    },
  });

  expect(registerResponse.status()).toBe(201);
  const { userId, token } = await registerResponse.json();

  // 2. Update profile
  const profileResponse = await request.put(`https://api.example.com/users
    data: {
      bio: 'Test user bio',
      location: 'New York',
    },
    headers: {
      'Authorization': `Bearer ${token}`,
    },
  });

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

  // 3. Upload avatar
  const avatarResponse = await request.post(`https://api.example.com/users
    multipart: {
      file: {
        name: 'avatar.jpg',
        mimeType: 'image/jpeg',
        buffer: Buffer.from('fake-image-data'),
      },
    },
    headers: {
      'Authorization': `Bearer ${token}`,
    },
  });

  expect(avatarResponse.status()).toBe(200);
```

```
      'Authorization': `Bearer ${token}`,
    },
  });

  const profile = await getProfileResponse.json();

  expect(profile.bio).toBe('Test user bio');
  expect(profile.location).toBe('New York');
  expect(profile.avatar).toBeDefined();
});
```

Parallel API Calls

Execute multiple API calls concurrently.

Using Promise.all:

```
test('parallel API calls', async ({ request }) => {
  // Execute 3 API calls in parallel
  const [usersResponse, postsResponse, commentsResponse] = await Promise.a
    request.get('https://jsonplaceholder.typicode.com/users'),
    request.get('https://jsonplaceholder.typicode.com/posts'),
    request.get('https://jsonplaceholder.typicode.com/comments'),
  ]);

  // Verify all succeeded
  expect(usersResponse.status()).toBe(200);
  expect(postsResponse.status()).toBe(200);
  expect(commentsResponse.status()).toBe(200);

  // Get data
  const users = await usersResponse.json();
  const posts = await postsResponse.json();
  const comments = await commentsResponse.json();

  console.log('Users count:', users.length);
  console.log('Posts count:', posts.length);
  console.log('Comments count:', comments.length);
});
```

Parallel with error handling:

```
    request.get('https://jsonplaceholder.typicode.com/invalid'), // Will f
    request.get('https://jsonplaceholder.typicode.com/posts'),
  ]);

  results.forEach((result, index) => {
    if (result.status === 'fulfilled') {
      console.log(`Request ${index} succeeded:`, result.value.status());
    } else {
      console.log(`Request ${index} failed:`, result.reason);
    }
  });

  // Get successful responses
  const successfulResponses = results
    .filter((r): r is PromiseFulfilledResult<any> => r.status === 'fulfill
    .map(r => r.value);

  expect(successfulResponses.length).toBe(2);
});
```

Data Dependency Management

Store and reuse data across tests.

Using test.describe with shared state:

```
test.describe('User workflow with dependencies', () => {
  let userId: number;
  let authToken: string;
  let postId: number;

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

    const user = await response.json();
    userId = user.id;
    authToken = user.token;
```

```
test('create post for user', async ({ request }) => {
  test.skip(!userId, 'User not created');

  const response = await request.post('https://api.example.com/posts', {
    data: {
      userId: userId,
      title: 'Test Post',
      body: 'Content',
    },
    headers: {
      'Authorization': `Bearer ${authToken}`,
    },
  });

  const post = await response.json();
  postId = post.id;

  expect(postId).toBeDefined();
});

test('get user posts', async ({ request }) => {
  test.skip(!userId, 'User not created');

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

  const posts = await response.json();

  expect(posts.length).toBeGreaterThan(0);
  expect(posts.some((p: any) => p.id === postId)).toBe(true);
});

test('delete post', async ({ request }) => {
  test.skip(!postId, 'Post not created');

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

Using fixtures for dependencies:

```
 type DependencyFixtures = {
  testUser: { id: number; token: string };
};

const test = base.extend<DependencyFixtures>({
  testUser: async ({ request }, use) => {
    // Create user
    const response = await request.post('https://api.example.com/users', {
      data: {
        name: 'Fixture User',
        email: `fixture-${Date.now()}@example.com`,
      },
    });

    const user = await response.json();

    await use({ id: user.id, token: user.token });

    // Cleanup
    await request.delete(`https://api.example.com/users/${user.id}`, {
      headers: {
        'Authorization': `Bearer ${user.token}`,
      },
    });
  },
});

test('use test user fixture', async ({ request, testUser }) => {
  // User already created
  const response = await request.get(`https://api.example.com/users/${test
    headers: {
      'Authorization': `Bearer ${testUser.token}`,
    },
  });

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

Transaction-like Testing

```
test('transaction test', async ({ request }) => {
  // Create order
  const orderResponse = await request.post('https://api.example.com/orders
    data: {
      customerId: 123,
      items: [
        { productId: 1, quantity: 2, price: 29.99 },
        { productId: 2, quantity: 1, price: 49.99 },
      ],
    },
  });

  const order = await orderResponse.json();
  const orderId = order.id;

  // Verify order created
  expect(order.total).toBe(109.97);
  expect(order.status).toBe('pending');

  // Process payment
  const paymentResponse = await request.post(`https://api.example.com/orde
    data: {
      method: 'credit_card',
      cardToken: 'tok_visa',
    },
  });

  const payment = await paymentResponse.json();
  expect(payment.status).toBe('completed');

  // Verify order status updated
  const orderCheckResponse = await request.get(`https://api.example.com/or
  const updatedOrder = await orderCheckResponse.json();

  expect(updatedOrder.status).toBe('paid');
  expect(updatedOrder.paymentId).toBe(payment.id);

  // Verify inventory updated
  for (const item of order.items) {
    const productResponse = await request.get(`https://api.example.com/pro
    const product = await productResponse.json();

    // Stock should be reduced
    console.log(`Product ${item.productId} stock:`, product.stock);
  }
```

Schema Validation

JSON Schema Validation

Use JSON Schema to validate API responses.

Install Ajv:

```
npm install -D ajv
```

Define schema:

```
import Ajv from 'ajv';
import addFormats from 'ajv-formats';

const ajv = new Ajv();
addFormats(ajv);

const userSchema = {
  type: 'object',
  properties: {
    id: { type: 'number' },
    name: { type: 'string', minLength: 1 },
    email: { type: 'string', format: 'email' },
    age: { type: 'number', minimum: 0, maximum: 150 },
    role: { type: 'string', enum: ['admin', 'user', 'guest'] },
    isActive: { type: 'boolean' },
    createdAt: { type: 'string', format: 'date-time' },
    profile: {
      type: 'object',
      properties: {
        bio: { type: 'string' },
        website: { type: 'string', format: 'uri' },
        location: { type: 'string' },
      },
      required: ['bio'],
    },
  },
```

```
test('validate user 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).toBe(true);
});
```

Array schema:

```
const usersArraySchema = {
  type: 'array',
  items: userSchema,
  minItems: 1,
  maxItems: 100,
};

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

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

Reusable validation function:

```
function validateResponse(data: any, schema: any): { valid: boolean; erro
  const validate = ajv.compile(schema);
  const valid = validate(data);

  return {
    valid,
    errors: validate.errors || [],
  };
```

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

  const result = validateResponse(user, userSchema);

  if (!result.valid) {
    console.log('Validation errors:');
    result.errors.forEach(error => {
      console.log(`- ${error.instancePath}: ${error.message}`);
    });
  }

  expect(result.valid).toBe(true);
});
```

Complex Schema Examples

Nested object schema:

```
const orderSchema = {
  type: 'object',
  properties: {
    id: { type: 'number' },
    customerId: { type: 'number' },
    status: {
      type: 'string',
      enum: ['pending', 'processing', 'shipped', 'delivered', 'cancelled']
    },
    items: {
      type: 'array',
      items: {
        type: 'object',
        properties: {
          productId: { type: 'number' },
          productName: { type: 'string' },
          quantity: { type: 'number', minimum: 1 },
          price: { type: 'number', minimum: 0 },
          total: { type: 'number', minimum: 0 },
        },
        required: ['productId', 'productName', 'quantity', 'price', 'total
      },
      minItems: 1,
    },
```

```
        street: { type: 'string' },
        city: { type: 'string' },
        state: { type: 'string' },
        zipCode: { type: 'string', pattern: '^\\d{5}(-\\d{4})?$' },
        country: { type: 'string' },
      },
      required: ['street', 'city', 'zipCode', 'country'],
    },
    total: { type: 'number', minimum: 0 },
    createdAt: { type: 'string', format: 'date-time' },
  },
  required: ['id', 'customerId', 'status', 'items', 'total', 'createdAt'],
};

test('validate order schema', async ({ request }) => {
  const response = await request.get('https://api.example.com/orders/123')
  const order = await response.json();

  const validate = ajv.compile(orderSchema);
  expect(validate(order)).toBe(true);
});
```

Conditional schema:

```
const productSchema = {
  type: 'object',
  properties: {
    id: { type: 'number' },
    name: { type: 'string' },
    type: { type: 'string', enum: ['physical', 'digital'] },
    price: { type: 'number', minimum: 0 },
    stock: { type: 'number', minimum: 0 },
  },
  required: ['id', 'name', 'type', 'price'],
  if: {
    properties: { type: { const: 'physical' } },
  },
  then: {
    required: ['stock'], // Stock required for physical products
    properties: {
      weight: { type: 'number', minimum: 0 },
      dimensions: {
        type: 'object',
```

```
          height: { type: 'number' },
        },
      },
    },
  },
  else: {
    properties: {
      downloadUrl: { type: 'string', format: 'uri' },
      fileSize: { type: 'number', minimum: 0 },
    },
    required: ['downloadUrl'],
  },
};
```

Schema Registry

Centralize schemas:

```
// schemas/index.ts
export const schemas = {
  user: {
    type: 'object',
    properties: {
      id: { type: 'number' },
      name: { type: 'string' },
      email: { type: 'string', format: 'email' },
    },
    required: ['id', 'name', 'email'],
  },

  post: {
    type: 'object',
    properties: {
      id: { type: 'number' },
      userId: { type: 'number' },
      title: { type: 'string' },
      body: { type: 'string' },
    },
    required: ['id', 'userId', 'title', 'body'],
  },

  comment: {
    type: 'object',
```

```
      name: { type: 'string' },
      email: { type: 'string', format: 'email' },
      body: { type: 'string' },
    },
    required: ['id', 'postId', 'name', 'email', 'body'],
  },
};

// Use in tests
import { schemas } from './schemas';

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

  const validate = ajv.compile(schemas.user);
  expect(validate(user)).toBe(true);
});
```

Combining UI and API Testing

Hybrid Testing Approach

Use API for setup, UI for testing, API for verification.

Example: E-commerce flow

```
test('complete purchase flow', async ({ page, request }) => {
  // API: Create test user
  const userResponse = await request.post('https://api.example.com/users',
    data: {
      email: 'test@example.com',
      password: 'password123',
      name: 'Test User',
    },
  });

  const { id: userId, token } = await userResponse.json();
```

```
  data: {
    productId: 123,
    quantity: 2,
  },
  headers: {
    'Authorization': `Bearer ${token}`,
  },
});

// UI: Login
await page.goto('https://example.com/login');
await page.fill('#email', 'test@example.com');
await page.fill('#password', 'password123');
await page.click('button[type="submit"]');

// UI: Navigate to cart
await page.click('a[href="/cart"]');

// UI: Verify cart contents
await expect(page.locator('.cart-item')).toHaveCount(1);
await expect(page.locator('.cart-item .quantity')).toHaveText('2');

// UI: Proceed to checkout
await page.click('button:has-text("Checkout")');

// UI: Fill shipping info
await page.fill('#address', '123 Main St');
await page.fill('#city', 'New York');
await page.fill('#zipCode', '10001');
await page.click('button:has-text("Continue")');

// UI: Complete payment
await page.fill('#cardNumber', '4111111111111111');
await page.fill('#cardExpiry', '12/25');
await page.fill('#cardCvv', '123');
await page.click('button:has-text("Place Order")');

// UI: Verify success
await expect(page.locator('.order-confirmation')).toBeVisible();
const orderNumber = await page.locator('.order-number').textContent();

// API: Verify order in database
const orderResponse = await request.get(`https://api.example.com/orders/
  headers: {
    'Authorization': `Bearer ${token}`,
```

```
  const order = await orderResponse.json();
  expect(order.status).toBe('completed');
  expect(order.userId).toBe(userId);
  expect(order.items[0].productId).toBe(123);
  expect(order.items[0].quantity).toBe(2);
});
```

API for Test Data Setup

Faster test setup with API:

```
test('test with API setup', async ({ page, request }) => {
  // API: Create test data
  const [user, products, orders] = await Promise.all([
    request.post('https://api.example.com/users', {
      data: { name: 'Test User', email: 'test@example.com' },
    }),
    request.post('https://api.example.com/products/bulk', {
      data: {
        products: [
          { name: 'Product 1', price: 29.99 },
          { name: 'Product 2', price: 49.99 },
        ],
      },
    }),
    request.post('https://api.example.com/orders/bulk', {
      data: {
        orders: [
          { customerId: 1, total: 100 },
          { customerId: 1, total: 200 },
        ],
      },
    }),
  ]);

  const userData = await user.json();
  const { token } = userData;

  // Set auth cookie for UI
  await page.context().addCookies([
    {
      name: 'auth_token',
      value: token,
```

```
  ]);

  // UI: Now test with pre-populated data
  await page.goto('https://example.com/dashboard');
  await expect(page.locator('.order-history .order')).toHaveCount(2);
});
```

API for Verification

Use API to verify UI actions:

```
test('verify UI action via API', async ({ page, request }) => {
  // UI: Register user
  await page.goto('https://example.com/register');
  await page.fill('#name', 'New User');
  await page.fill('#email', 'newuser@example.com');
  await page.fill('#password', 'password123');
  await page.click('button[type="submit"]');

  // UI: Verify success message
  await expect(page.locator('.success-message')).toBeVisible();

  // API: Verify user was created in database
  const response = await request.get('https://api.example.com/users', {
    params: {
      email: 'newuser@example.com',
    },
  });

  const users = await response.json();
  expect(users.length).toBe(1);
  expect(users[0].name).toBe('New User');
  expect(users[0].email).toBe('newuser@example.com');
});
```

API Mocking for UI Tests

Mock API responses in UI tests:

```
    await route.fulfill({
      status: 200,
      contentType: 'application/json',
      body: JSON.stringify({
        users: [
          { id: 1, name: 'Mock User 1', email: 'mock1@example.com' },
          { id: 2, name: 'Mock User 2', email: 'mock2@example.com' },
        ],
      }),
    });
  });

  // UI: Navigate to page that calls API
  await page.goto('https://example.com/users');

  // UI: Verify mocked data is displayed
  await expect(page.locator('.user-list .user')).toHaveCount(2);
  await expect(page.locator('.user').first()).toContainText('Mock User 1')
});
```

Data-Driven API Tests

Parameterized Tests

Test with multiple data sets:

```
 const testUsers = [
  { name: 'John Doe', email: 'john@example.com', age: 30 },
  { name: 'Jane Smith', email: 'jane@example.com', age: 25 },
  { name: 'Bob Johnson', email: 'bob@example.com', age: 35 },
];

test.describe('Data-driven user creation', () => {
  for (const userData of testUsers) {
    test(`create user: ${userData.name}`, async ({ request }) => {
      const response = await request.post('https://api.example.com/users',
        data: userData,
      });
```

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

Reading from JSON Files

External test data:

```
// testData/users.json
{
  "validUsers": [
    {
      "name": "Alice Brown",
      "email": "alice@example.com",
      "role": "admin"
    },
    {
      "name": "Charlie Davis",
      "email": "charlie@example.com",
      "role": "user"
    }
  ],
  "invalidUsers": [
    {
      "name": "",
      "email": "invalid@example.com",
      "expectedError": "Name is required"
    },
    {
      "name": "Test",
      "email": "not-an-email",
      "expectedError": "Invalid email format"
    }
  ]
}
```

Use in tests:

```
  for (const user of testData.validUsers) {
    test(`create user: ${user.name}`, async ({ request }) => {
      const response = await request.post('https://api.example.com/users',
        data: user,
      });

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

      const created = await response.json();
      expect(created.name).toBe(user.name);
      expect(created.role).toBe(user.role);
    });
  }
});

test.describe('Invalid users', () => {
  for (const user of testData.invalidUsers) {
    test(`reject invalid user: ${user.name || 'empty name'}`, async ({ req
      const response = await request.post('https://api.example.com/users',
        data: {
          name: user.name,
          email: user.email,
        },
      });

      expect(response.status()).toBe(400);

      const error = await response.json();
      expect(error.message).toContain(user.expectedError);
    });
  }
});
```

CSV Data Source

Read from CSV:

```
 import fs from 'fs';
import { parse } from 'csv-parse/sync';

interface ProductData {
  name: string;
```

```
}

const csvContent = fs.readFileSync('./testData/products.csv', 'utf-8');
const products = parse(csvContent, {
  columns: true,
  skip_empty_lines: true,
}) as ProductData[];

test.describe('Product API tests', () => {
  for (const product of products) {
    test(`create product: ${product.name}`, async ({ request }) => {
      const response = await request.post('https://api.example.com/product
        data: {
          name: product.name,
          price: parseFloat(product.price),
          category: product.category,
          stock: parseInt(product.stock),
        },
      });

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

Dynamic Test Generation

Generate tests based on API response:

```
test.describe('Dynamic user tests', () => {
  let users: any[];

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

  test('verify all users have valid email', async () => {
    const emailRegex = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;

    users.forEach(user => {
      expect(user.email).toMatch(emailRegex);
    });
```

```
    users.forEach(user => {
      expect(user).toHaveProperty('id');
      expect(user).toHaveProperty('name');
      expect(user).toHaveProperty('email');
    });
  });
});
```

Practice Exercises

Exercise 1: Authentication Flow

Task: Implement complete auth flow

```
test.describe('Authentication Flow', () => {
  test('register, login, access protected, logout', async ({ request }) =>
    // 1. Register new user
    // 2. Login with credentials
    // 3. Get access token
    // 4. Access protected endpoint with token
    // 5. Logout
    // 6. Verify token is invalid after logout

    // Your implementation
  });

  test('refresh token flow', async ({ request }) => {
    // 1. Login to get access and refresh tokens
    // 2. Use access token
    // 3. Wait for access token to expire (or simulate)
    // 4. Use refresh token to get new access token
    // 5. Verify new token works

    // Your implementation
  });

  test('OAuth 2.0 flow', async ({ request }) => {
    // Implement OAuth client credentials flow
    // Your implementation
```

Exercise 2: Complex Workflow

Task: Multi-step workflow with dependencies

```
test('E-commerce complete flow', async ({ request }) => {
  // 1. Create user account
  // 2. Login
  // 3. Browse products (GET /products)
  // 4. Add items to cart (POST /cart/items)
  // 5. Update cart quantity (PATCH /cart/items/:id)
  // 6. Apply coupon code (POST /cart/coupon)
  // 7. Create order (POST /orders)
  // 8. Process payment (POST /orders/:id/payment)
  // 9. Verify order status (GET /orders/:id)
  // 10. Get order history (GET /users/:id/orders)

  // Verify data consistency at each step

  // Your implementation
});
```

Exercise 3: Schema Validation Suite

Task: Create comprehensive schema validation

```
test.describe('Schema Validation', () => {
  test('user schema', async ({ request }) => {
    // Define complete user schema with all fields
    // Validate GET /users/:id response
    // Your implementation
  });

  test('order schema with nested items', async ({ request }) => {
    // Define order schema with nested items array
    // Each item should have product details
    // Validate complex nested structure
    // Your implementation
  });

  test('error response schema', async ({ request }) => {
```

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

Exercise 4: Parallel API Testing

Task: Test concurrent operations

```
test.describe('Concurrent Operations', () => {
  test('create multiple users in parallel', async ({ request }) => {
    // Create 10 users concurrently
    // Verify all succeeded
    // Check for any conflicts

    // Your implementation
  });

  test('race condition test', async ({ request }) => {
    // Create a product with initial stock
    // Attempt to purchase same product from multiple requests
    // Verify stock is correctly decremented
    // Ensure no overselling

    // Your implementation
  });

  test('rate limiting', async ({ request }) => {
    // Make many requests quickly
    // Detect when rate limit is hit (429 status)
    // Verify rate limit headers

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

Exercise 5: UI + API Hybrid

Task: Combine UI and API testing

```
  // UI: Login
  // UI: Navigate to checkout
  // UI: Complete purchase

  // API: Verify order was created correctly
  // API: Verify inventory was updated
  // API: Verify payment was processed

  // Your implementation
});
```

Exercise 6: Data-Driven Validation

Task: Test with multiple data sets

```
 // Create testData/api-tests.json with:
// - Valid inputs (expect success)
// - Invalid inputs (expect validation errors)
// - Edge cases (boundary values)
// - Special characters in fields

test.describe('Data-Driven Validation', () => {
  // Read from JSON file
  // Create tests for each scenario
  // Verify expected outcomes

  // Your implementation
});
```

Exercise 7: API Performance Suite

Task: Performance and load testing

```
test.describe('API Performance', () => {
 test('response time test', async ({ request }) => {
   // Measure response time for various endpoints
   // Assert all under thresholds:
   // - GET endpoints < 500ms
   // - POST endpoints < 1000ms
```

```
  });

  test('bulk operation performance', async ({ request }) => {
    // Test bulk endpoints
    // Measure time to create 100 records
    // Verify performance is acceptable

    // Your implementation
  });

  test('pagination performance', async ({ request }) => {
    // Test different page sizes
    // Measure response time for each
    // Verify larger pages don't timeout

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

Summary

In Day 7, you mastered:

✅ Authentication Mechanisms

- Basic Authentication
- Bearer Token (JWT)
- API Key authentication
- OAuth 2.0 flows
- Session-based auth
- Token refresh patterns

✅ API Chaining

- Sequential API calls
- Parallel execution

- Workflow automation

✅ Schema Validation

- JSON Schema basics
- Complex schema patterns
- Nested objects validation
- Array validation
- Schema registry
- Custom validators

✅ UI + API Hybrid

- API for test setup
- UI for user workflows
- API for verification
- API mocking in UI tests
- Complete E2E flows

✅ Data-Driven Testing

- Parameterized tests
- JSON data files
- CSV data sources
- Dynamic test generation
- Test data management

Key Takeaways

- **Authentication** is critical for API testing
- **Chaining APIs** enables complex scenarios
- **Schema validation** ensures data quality
- **Hybrid approach** combines best of both worlds

Best Practices Learned

1. Centralize authentication logic
2. Use fixtures for test data
3. Validate schemas rigorously
4. Combine UI and API strategically
5. Externalize test data
6. Handle async operations properly
7. Clean up test data

Advanced Patterns

- JWT token management
- OAuth 2.0 implementation
- Complex workflows
- Schema registries
- Hybrid testing strategies
- Performance testing
- Concurrent operations

Next Steps

- Master authentication patterns
- Build complex workflows
- Create schema validators
- Implement hybrid tests
- Complete all exercises
- Apply to real projects

End of Day 7 Documentation