JSONPlaceholder API Testing - Simple Guide

Version: 1.0

Date: August 2, 2025

Team: QA Team

📋 What's Inside

🎯 What is this project?

The main idea

We need to test the **JSONPlaceholder API**. This is a fake REST API that people use for learning and testing.

What we test

We test these parts:

- **/posts** - Blog posts (create, read, update, delete)
- **/users** - User accounts
- **/comments** - Comments on posts
- **/albums** - Photo albums
- **/photos** - Pictures
- **/todos** - Task lists

Why API testing?

API testing is better than UI testing because:

What	API Testing	UI Testing	Why API wins
Speed	Fast (0.5 seconds)	Slow (5 seconds)	Get results faster
Stable	Works every time	Sometimes breaks	More reliable
Easy to fix	Simple to update	Hard to change	Less work later
Early testing	Start right away	Wait for UI	Find bugs sooner

Result: We test the API first, then UI later.

📁 About JSONPlaceholder

What it is

JSONPlaceholder is a free testing API that gives us fake data to practice with.

- **Website:** https://jsonplaceholder.typicode.com
- **Type:** REST API (uses HTTP requests)
- **Data format:** JSON
- **Login needed:** No, it's free for everyone

How it works

It's like a simple blog website with:

- **Users** who write posts
- **Posts** with text content
- **Comments** from other users
- **Albums** to organize photos
- **Photos** as pictures
- **Todos** for task management

Why we chose this

1. **Good for learning** - Shows real API testing
2. **Build tools** - Create testing framework we can use again
3. **Follow best practices** - Learn professional methods
4. **Industry standard** - APIs are very important today

🎯 What we want to do

Main goals

1. **Check all functions work** - Every API call should work correctly
2. **Data is correct** - Information doesn't get lost or changed
3. **Fast enough** - API responds quickly (under 2 seconds)
4. **Good error messages** - Clear errors when something goes wrong
5. **Parts work together** - Different API calls work with each other

Extra goals

1. **Build test tools** - Create framework for future projects
2. **Write good docs** - Explain everything clearly
3. **Share knowledge** - Teach team best ways to test
4. **Keep improving** - Learn from mistakes and get better

How we know we succeeded

- ✅ **Test all API endpoints** (100%)
- ✅ **Most tests pass** (95% or more)
- ✅ **Fast responses** (under 2 seconds)
- ✅ **No big problems** (zero critical bugs)
- ✅ **Automatic testing** (runs by itself)

🧪 How we will test

Our method

Risk-based testing - we focus on the most important parts first:

1. **High risk** - Core features (create, read, update, delete)
2. **Medium risk** - Data checks and connections
3. **Low risk** - Error handling and edge cases

Test pyramid

We use different amounts of each test type:

BDD approach

BDD = Behavior Driven Development

We write tests in plain English so everyone can understand:

```
Feature: Managing blog posts
  As a blog user
  I want to create posts
  So I can share my ideas

  Scenario: Create a new post
    Given I have post information
    When I send a create request
    Then the post should be saved
    And I should get the post details back
```

Why this helps:

- Business people can read our tests
- Tests explain what the system should do
- Everyone understands what we're testing
- Easy to maintain and update

📊 Test levels

1. Single API tests

What: Test one API call at a time

- **Focus:** One endpoint works by itself
- **Example:** GET /posts/1, POST /posts
- **Check:** All basic operations (create, read, update, delete)

2. Integration tests

What: Test how APIs work together

- **Focus:** Data flows between different calls
- **Example:** Create user → Create post → Add comment
- **Check:** Related operations work together

3. Full system tests

What: Test complete workflows

- **Focus:** Real user scenarios from start to finish
- **Example:** Complete blog management workflow
- **Check:** Everything works like in real life

4. Business tests

What: Check business requirements

- **Focus:** User stories are satisfied
- **Example:** "User can manage their blog content"
- **Check:** Business goals are met

🔧 Test types

Testing functions

1. CRUD Testing 🗂️

Priority: Very important

CRUD = Create, Read, Update, Delete

- **Create** - POST requests with good and bad data
- **Read** - GET requests with different options

- **Update** - PUT/PATCH requests with changes
- **Delete** - DELETE requests with cleanup check

2. Data checking ✅

Priority: Important

- **Input validation** - Required fields, correct data types
- **Business rules** - Domain-specific checks
- **Boundary testing** - Min/max values, edge cases
- **Response format** - JSON structure is correct

3. Error handling ❌

Priority: Important

- **4xx errors** - Bad requests, not found, unauthorized
- **5xx errors** - Server problems, timeouts
- **Bad inputs** - Wrong data format, wrong types
- **Edge cases** - Empty requests, too much data

Testing performance

1. Speed testing ⚡

Priority: Medium

- **Response time** - How fast each API responds
- **Throughput** - How many requests per second
- **Load testing** - Many users at the same time
- **Stress testing** - Find the breaking point

2. Security testing 🔒

Priority: Medium

- **Input safety** - Stop dangerous code injection
- **Data validation** - Block malicious requests
- **Error info** - Don't show sensitive data
- **Rate limits** - Prevent abuse

3. Reliability testing 🔄

Priority: Medium

- **Consistency** - Same input gives same output
- **Data integrity** - Information stays correct
- **Recovery** - Handle errors gracefully
- **Availability** - Service stays online

🎨 Test design methods

1. Group similar inputs

Use for: Input data validation

Good post data:

- **Title:** 1-255 characters ✅
- **Body:** 1-5000 characters ✅
- **User ID:** 1-10 (existing users) ✅

Bad post data:

- **Title:** Empty ❌
- **Body:** Too long (over 5000 characters) ❌
- **User ID:** 0 or negative number ❌

2. Test limits

Use for: Numbers and text length limits

User ID testing:

- **Below minimum:** 0 ❌
- **Minimum valid:** 1 ✅
- **Maximum valid:** 10 ✅
- **Above maximum:** 11 ❌

Text length testing:

- **Empty:** "" ❌

- **One character:** "a" ✅
- **Maximum:** 255 characters ✅
- **Too long:** 256 characters ❌

3. Decision tables

Use for: Complex business rules

User exists?	Data valid?	What happens?
Yes	Yes	✅ 201 Created
Yes	No	❌ 400 Bad Request
No	Yes	❌ 422 Cannot process
No	No	❌ 400 Bad Request

4. Test workflows

Use for: Multi-step processes

Blog management workflow:

1. User creates account
2. User writes post
3. Others add comments
4. User manages comments
5. User archives old posts

🧱 When to start and stop testing

Before we start testing

We need these things ready:

- ✅ **API is running** and we can reach it
- ✅ **Test environment works** and is stable
- ✅ **Test data ready** and we know how to create it
- ✅ **Test tools set up** and working
- ✅ **API documentation** available and reviewed
- ✅ **Test cases written** and checked

When we can stop testing

Testing is complete when:

- ✅ **All planned tests done**
- ✅ **95% or more tests pass**
- ✅ **All big problems fixed**
- ✅ **Speed requirements met**
- ✅ **All endpoints tested** (100% coverage)
- ✅ **Regression testing done**
- ✅ **Reports created** and reviewed

When to pause testing

Stop testing if:

- ❌ **API down for more than 2 hours**
- ❌ **Big bugs block most tests**
- ❌ **Test environment broken**
- ❌ **Major API changes** need test updates

When to continue testing

Resume testing when:

- ✅ **Problems are fixed**
- ✅ **Environment is stable again**
- ✅ **Tests updated** for any changes
- ✅ **Impact analysis complete**

⚠️ What can go wrong

Big risks

1. Data problems 🔴

Risk: Data gets lost or corrupted

- **How likely:** Medium
- **Impact:** High

- **Prevention:** Test data carefully
- **Focus on:** Create, read, update, delete operations

2. Slow performance 🔴

Risk: API becomes too slow

- **How likely:** Medium
- **Impact:** Medium
- **Prevention:** Performance testing and monitoring
- **Focus on:** Load testing, response times

3. Bad error messages 🔴

Risk: Users get confusing errors

- **How likely:** Low
- **Impact:** High
- **Prevention:** Test error scenarios
- **Focus on:** Wrong inputs, edge cases

Medium risks

4. APIs don't work together 🟡

Risk: Related functions break each other

- **How likely:** Medium
- **Impact:** Medium
- **Prevention:** Integration testing
- **Focus on:** Cross-function operations

5. Security problems 🟡

Risk: Malicious input causes damage

- **How likely:** Low
- **Impact:** Medium
- **Prevention:** Security testing
- **Focus on:** Input validation, injection attacks

Small risks

6. Documentation wrong 🟢

Risk: Misunderstanding how API works

- **How likely:** Low
- **Impact:** Low
- **Prevention:** Test-driven documentation
- **Focus on:** API behavior validation

Risk plan

Risk level	Testing effort	Automation priority	Check frequency
🔴 **High**	60% of time	High priority	Every day
🟡 **Medium**	30% of time	Medium priority	Every week
🟢 **Low**	10% of time	Low priority	Every month

🌐 Test environments

Where we test

1. Production (real JSONPlaceholder website)

- **Purpose:** Test the real API
- **Access:** Free for everyone
- **Data:** Read-only (fake writes)
- **Use:** Main testing

2. Local development (on our computers)

- **Purpose:** Build and debug framework
- **Access:** Our own machines
- **Data:** Controlled test data
- **Use:** Development work

3. CI/CD pipeline (automatic testing)

- **Purpose:** Run tests automatically

- **Access:** Cloud containers
- **Data:** Generated test data
- **Use:** Continuous checking

Settings for each environment

Production:

```yaml
website: https://jsonplaceholder.typicode.com
timeout: 30 seconds
retry: 3 times
```

Development:

```yaml
website: http://localhost:3000
timeout: 60 seconds
retry: 5 times
debug: on
```

Automatic:

```yaml
website: https://jsonplaceholder.typicode.com
timeout: 15 seconds
retry: 1 time
parallel: yes
```

📊 Test data plan

Types of test data

1. Fixed reference data 📄

Source: Pre-made data files

- **Users:** 10 sample users with complete info
- **Posts:** Different post types and lengths

- **Location:** JSON files in `src/fixtures/` folder

2. Generated data 🎲

Source: Faker.js library

- **Good:** Unique data for each test
- **Covers:** Boundary values, edge cases
- **Flexible:** Can create any type of data

3. Edge case data 🎯

Source: Manually created special cases

- **Empty values:** "", null, undefined
- **Extreme lengths:** Very long/short text
- **Invalid formats:** Wrong emails, URLs

Rules for managing data

✅ Do this

- Create fresh data for each test
- Clean up test data after tests finish
- Use realistic data like real users would create
- Check data validity in helper functions

❌ Don't do this

- Put test data directly in test code
- Depend on existing data that might change
- Leave test data behind after tests
- Use real production data in tests

Test data examples

Good test data:

```javascript
```

```
const goodPost = {
  userId: 1,
  title: "How to test APIs effectively",
  body: "A complete guide to API testing best practices..."
};

// Edge case testing
const edgeCases = {
  emptyTitle: { userId: 1, title: "", body: "Content" },
  tooLong: { userId: 1, title: "x".repeat(300), body: "Content" },
  badUser: { userId: 999, title: "Title", body: "Content" }
};

// Special characters
const specialCases = {
  nullValues: { userId: null, title: null, body: null },
  scriptTags: { title: "<script>alert('dangerous')</script>" },
  emojis: { title: "🎯 Testing with emojis 🎉" }
};
```

🛠️ Tools we use

Main tools

1. Cucumber.js 🥒

What it does: Writes tests in plain English

- **Good:** Anyone can read the tests
- **Use:** Feature files and test steps
- **Works with:** TypeScript

2. TypeScript 📘

What it does: Catches errors before running tests

- **Good:** Finds mistakes early
- **Use:** All test code
- **Works with:** All editors and IDEs

3. Axios 🌐

What it does: Sends HTTP requests to APIs

- **Good:** Easy to use, lots of features
- **Use:** All API calls
- **Works with:** Custom wrappers

4. Faker.js 🎭

What it does: Creates realistic fake data

- **Good:** Makes different data each time
- **Use:** Generate test data
- **Works with:** Data helper functions

Helper tools

5. Winston 📝

What it does: Logs what happens during tests

- **Features:** Different log levels
- **Use:** Track test execution
- **Works with:** Custom formatters

6. Docker 🐳

What it does: Creates consistent test environments

- **Features:** Same setup everywhere
- **Use:** Local development and CI/CD
- **Works with:** Multi-stage builds

7. GitHub Actions ⚡

What it does: Runs tests automatically

- **Features:** Runs tests in parallel
- **Use:** Automatic test execution
- **Works with:** Multiple test stages

Why we chose these tools

Need	Our choice	Other option	Why we chose it
BDD framework	Cucumber.js	Jest, Mocha	Easier for non-technical people to read
Type safety	TypeScript	JavaScript	Catches errors before runtime
HTTP client	Axios	Fetch, Supertest	More features and easier to use
Test data	Faker.js	Custom code	Creates realistic varied data
Containers	Docker	Direct setup	Same environment everywhere

🐛 Managing bugs

Bug severity levels

Critical 🔴

Impact: System doesn't work or major features broken

Examples:

- API completely down
- Data gets lost or corrupted
- Security holes
- **Fix time:** 24 hours

High 🟠

Impact: Major features don't work properly

Examples:

- Core create/read/update/delete fails
- Very slow performance
- Error handling broken
- **Fix time:** 3 days

Medium 🟡

Impact: Minor feature problems

Examples:

- Non-essential features broken

- Small performance issues
- Response format problems
- **Fix time:** 1 week

Low 🟢

Impact: Small annoyances

Examples:

- Documentation errors
- Small delays
- Enhancement ideas
- **Fix time:** 2 weeks

Bug priority levels

P1 - Urgent 🚨

- Blocks testing
- Critical business functions
- Security issues

P2 - High ⚡

- Major feature impact
- Delays test schedule
- Data problems

P3 - Medium 📋

- Minor feature impact
- Workaround exists
- Performance issues

P4 - Low 🎨

- Enhancement request
- Documentation issue
- Nice-to-have feature

Bug lifecycle

```
New → Assigned → Working on it → Fixed → Tested → Closed
 ↓        ↓                       ↓
Rejected  Reopened ←-------------------------------↓
```

Bug report example

```markdown
```

Bug ID: BUG-001
Title: POST /posts returns error 500 for valid data
Severity: High
Priority: P2
Environment: Production API
Can reproduce: Yes

Steps to reproduce:
1. Send POST request to /posts
2. Use valid data: {userId: 1, title: "Test", body: "Content"}
3. Look at response

Expected result:
- Status: 201 Created
- Response has created post with ID

Actual result:
- Status: 500 Internal Server Error
- Response: {"error": "Database connection failed"}

Test data used:
```
{
  "userId": 1,
  "title": "Test Post",
  "body": "Test content for testing"
}
```

Screenshots/Logs:
[Attach relevant logs or API response screenshots]

Impact:
- Blocks post creation
- Affects 5 test scenarios
- Could be production problem

📈 Reports

Types of reports

1. Daily reports 🗓️

For: Development team
Contains:

- Test summary
- Pass/fail numbers
- New bugs found
- Performance numbers

2. Weekly reports 📋

For: Project managers
Contains:

- Progress vs plan
- Risk updates
- Milestone achievements
- Resource usage

3. Final report 🏁

For: All stakeholders
Contains:

- Final test results
- Quality assessment
- Recommendations
- Lessons learned

What we measure

Quality numbers

- **Test pass rate:** Target >95%
- **Bugs found per day:** How many we discover
- **Bugs fixed per day:** How many get resolved
- **Test coverage:** 100% endpoint coverage

Performance numbers

- **Test execution time:** <5 minutes for full suite

- **Average response time:** <2 seconds
- **Requests per second:** >100
- **Uptime:** 99.9%

Process numbers

- **Test cases written per day:** Productivity
- **Automation percentage:** How much is automated
- **Environment downtime:** Hours lost
- **Maintenance effort:** Hours spent updating tests

Reporting tools

1. HTML reports 📊

Features:

- Interactive dashboards
- Drill-down capabilities
- Visual charts
- Can export

2. JSON reports 📄

Features:

- Machine-readable
- CI/CD integration
- Historical storage
- Custom analysis

3. Log analysis 📝

Features:

- Detailed logs
- Error tracking
- Performance analysis
- Debug info

🗓️ When we start and finish

Project timeline (7 weeks total)

Weeks 1-2: Planning & Design

- Create test plan
- Design test cases
- Set up framework
- Prepare environment

Weeks 3-4: Build Tests

- Write automated tests
- Prepare test data
- Set up CI/CD pipeline
- Try test execution

Weeks 5-6: Run Tests

- Smoke testing
- Function testing
- Integration testing
- Performance testing

Week 7: Finish & Report

- Final test run
- Create reports
- Check bugs are fixed
- Close project

Week by week details

Week 1: Foundation

- **Monday:** Get test plan approved
- **Tuesday-Wednesday:** Design test cases
- **Thursday-Friday:** Plan framework

Week 2: Preparation

- **Monday-Tuesday:** Set up test automation
- **Wednesday-Thursday:** Configure environment
- **Friday:** Implement test data plan

Week 3: Development

- **Monday-Tuesday:** Write core tests
- **Wednesday-Thursday:** Write integration tests
- **Friday:** Set up performance tests

Week 4: Integration

- **Monday-Tuesday:** Set up CI/CD pipeline
- **Wednesday-Thursday:** End-to-end testing
- **Friday:** Validate test execution

Week 5: Execution

- **Monday:** Run smoke tests
- **Tuesday-Wednesday:** Run function tests
- **Thursday-Friday:** Run integration tests

Week 6: Validation

- **Monday-Tuesday:** Run performance tests
- **Wednesday-Thursday:** Check bug fixes
- **Friday:** Run regression tests

Week 7: Closure

- **Monday-Tuesday:** Final validation
- **Wednesday:** Create reports
- **Thursday:** Review with stakeholders
- **Friday:** Close project

Important milestones

Milestone	When	What we deliver	Success means
Test plan approved	Week 1	Strategy document	Stakeholder sign-off
Framework ready	Week 2	Test automation	Smoke tests pass
Tests complete	Week 4	Full test suite	All tests can run
First test cycle	Week 5	Test results	Coverage goals met
Final report	Week 7	Completion report	Quality gates passed

👥 Who does what

Team roles

Test Manager 👔

What they do:

- Plan overall test strategy
- Assign people and schedule work
- Talk to stakeholders
- Manage risks
- Make quality decisions

What they create:

- Test strategy document
- Test plans and schedules
- Status reports
- Risk updates

Test Architect 📐

What they do:

- Design test framework
- Choose tools and integrate them
- Design technical solutions
- Define best practices
- Share knowledge with team

What they create:

- Framework design
- Technical documentation
- Tool guides
- Code standards

Test Automation Engineer 🤖

What they do:

- Write automated tests
- Build framework
- Set up CI/CD pipeline
- Maintain and update tests
- Optimize performance

What they create:

- Automated test scripts
- Framework components
- CI/CD setup
- Maintenance docs

Test Analyst 📊

What they do:

- Design and document test cases
- Run tests and create reports
- Analyze and track bugs
- Manage test data
- Analyze results

What they create:

- Test case specifications
- Test execution reports
- Bug reports

- Test data sets

Communication schedule

Role	Daily standup	Weekly reports	Monthly reviews	Emergency updates
Test Manager	✅	✅	✅	✅
Test Architect	✅	✅	✅	✅
Automation Engineer	✅	✅	⚡	✅
Test Analyst	✅	✅	⚡	✅

How to escalate problems

Level 1: Test Team (Daily problems)
↓ (2 days)
Level 2: Test Manager (Project problems)
↓ (1 week)
Level 3: Project Manager (Strategic problems)
↓ (2 weeks)
Level 4: Steering Committee (Critical problems)

✅ Approval

Document review

Role	Reviewer	Date	Status	Comments
Test Manager	[Name]	[Date]	✅ Approved	Plan fits project goals
Project Manager	[Name]	[Date]	✅ Approved	Resources confirmed
Technical Lead	[Name]	[Date]	✅ Approved	Technical approach good
Business Analyst	[Name]	[Date]	✅ Approved	Requirements covered

Final approval

Document Status: ✅ **APPROVED**

Start Date: August 2, 2025

Next Review: October 2, 2025

Approved by:

[Project Manager Signature]

[Date]

📚 References

Industry standards

- **ISTQB Foundation Level** - Test strategy guidelines
- **IEEE 829** - Test documentation standards
- **ISO/IEC 25010** - Software quality model
- **OWASP API Security** - Security testing guidelines

Internal standards

- Company test strategy template
- API testing best practices guide
- Bug management procedures
- CI/CD integration standards

External references

- JSONPlaceholder API documentation
- REST API testing best practices
- BDD and Cucumber guidelines
- TypeScript testing patterns

This document will be updated as the project grows and we learn new things.