Postman Testing & Debugging Cheatsheet

A comprehensive cheat sheet covering Postman's testing and debugging features, including writing tests, debugging requests, and using the Postman console effectively.

Test Script Basics

Writing Tests

Tests in Postman are written in JavaScript and executed after a response is received. They use the `pm` object to access response data and the `tests` object to define assertions.

Example:

```
pm.test("Status code is 200", function () {
    pm.response.to.have.status(200);
});
```

The `pm.test()` function takes a test name (string) and a function that contains the assertions.

Assertions are written using the `pm.expect()` function, which provides a chainable syntax for expressing expectations about the response.

Example:

```
pm.test("Response time is less than 200ms", function () {

pm.expect(pm.response.responseTime).to.be.below(200);
});
```

Accessing Response Data

`pm.response.json()` — Parses the response body as JSON.

Example:

```
const jsonData = pm.response.json();
pm.expect(jsonData.name).to.eql("John Doe");
```

`pm.response.text()` — Returns the response body as a string.

Example:

```
const responseText = pm.response.text();
pm.expect(responseText).to.include("Success");
```

`pm.response.headers` — An object containing the response headers.

Example:

```
pm.expect(pm.response.headers.get('Content-Type')).to.include('application/json');
```

Common Assertions

`pm.expect(pm.response.statusCode).to.eql(200);` - Status code is 200.

`pm.expect(pm.response.body).to.include('value');` - Response body contains 'value'.

`pm.expect(pm.response.headers.get('Content-Type')).to.include('application/json');` - Content type is JSON.

`pm.expect(pm.response.responseTime).to.be.below(500);` - Response time is less than 500ms.

Debugging Requests

Postman Console

The Postman Console is a powerful tool for debugging requests and tests. It logs detailed information about request and response lifecycle.

To open the console, click the 'Console' button in the bottom left corner of the Postman window.

The console displays network information, sent headers, received headers, request body, response body, test results, and any console logs generated by your scripts.

Use `console.log()` in your pre-request scripts and test scripts to log custom messages to the console.

Example:

```
console.log("Request URL:", pm.request.url);
console.log("Response Status:", pm.response.status);
```

Request Information

`pm.request.url` — The URL of the request.

Example: `console.log(pm.request.url);`

`pm.request.headers` — An object containing the request headers.

Example: `console.log(pm.request.headers);`

`pm.request.body` — The request body (if any).

Example: `console.log(pm.request.body);`

Debugging Techniques

Use `console.log()` statements liberally to trace the execution flow of your scripts and inspect variable values.

Check the Postman Console for error messages and warnings that can help you identify problems.

Use the 'Preview' feature in the request body editor to see how Postman is interpreting your request data.

Inspect the raw response body to ensure that the server is returning the expected data.

Advanced Testing

Using Variables

Postman variables allow you to store and reuse values across requests and collections. Variables can be defined at different scopes: global, collection, environment, and local.

Use `pm.environment.get('variable_name')` to retrieve an environment variable.

Use `pm.environment.set('variable_name', 'value')` to set an environment variable.

Example:

```
pm.test("Check variable value", function () {

pm.expect(pm.environment.get('api_key')).to.not.be.empty;
});
```

Chaining Requests

Setting a variable	`pm.environment.set('user_id', pm.response.json().id);`
Using a variable in another request	`GET /users/{{user_id}}`
Getting a variable	`const userId = pm.environment.get('user_id');`

Data-Driven Testing

Postman supports data-driven testing by allowing you to import data from a CSV or JSON file and use it to parameterize your requests and tests.

Access data values using `pm.iterationData.get('column_name')`.

Example:

```
pm.test("Check email", function () {

pm.expect(pm.iterationData.get('email')).to.include('@');
});
```

Use the Collection Runner to execute the collection multiple times, each time with a different set of data.

Mock Servers & Contract Testing

Mock Servers

Postman Mock Servers simulate API endpoints, allowing you to develop and test your application without relying on a live API.

Create a mock server in Postman by defining the endpoint URL, request method, headers, and response body.

Use mock servers to test different scenarios, such as successful responses, error responses, and edge cases.

Verify interactions with the mock server using the Postman Console.

Contract Testing

Define API contracts	Create a Postman Collection that defines the expected request and response schemas for your API endpoints.
Validate responses	Write tests in Postman to validate that the actual responses from the API match the expected schemas.
Automate contract testing	Use the Postman Collection Runner or Newman to automate the execution of your contract tests.

Schema Validation

Use libraries like `tv4` or `ajv` within your Postman tests to validate the response body against a JSON schema.

Example:

```
const schema = {
  "type": "object",
  "properties": {
    "name": {"type": "string"},
    "age": {"type": "integer"}
  },
  "required": ["name", "age"]
};

pm.test("Validate schema", function () {
    const jsonData = pm.response.json();
    pm.expect(tv4.validate(jsonData, schema)).to.be.true;
});
```