

ÍNDICE

[Iker Ríos Navarro - Práctica de Postman — REST Basics - 2º DAW - 23/09/2025](#)

[ÍNDICE](#)

[Requisitos previos](#)

[Preparación del entorno en Postman \(5 pasos\)](#)

[Conceptos clave \(mini-resumen\)](#)

[Parte A — Lectura de recursos \(GET\)](#)

[A1. Listar recursos \(GET collection\)](#)

[A2. Obtener un recurso concreto \(GET item con path param\)](#)

[A3. Filtrado por query params](#)

[Parte B — Crear, actualizar y borrar \(POST, PUT, PATCH, DELETE\)](#)

[B1. Crear \(POST\)](#)

[B2. Actualizar completo \(PUT\)](#)

[B3. Actualizar parcial \(PATCH\)](#)

[B4. Borrar \(DELETE\)](#)

[Parte C — Headers, Auth y “sin estado”](#)

[C1. Header personalizado \(User-Agent\)](#)

[C2. Autorización tipo Bearer](#)

[Parte D — Errores y latencia](#)

[D1. Códigos de error](#)

[D2. Retrasos en la respuesta](#)

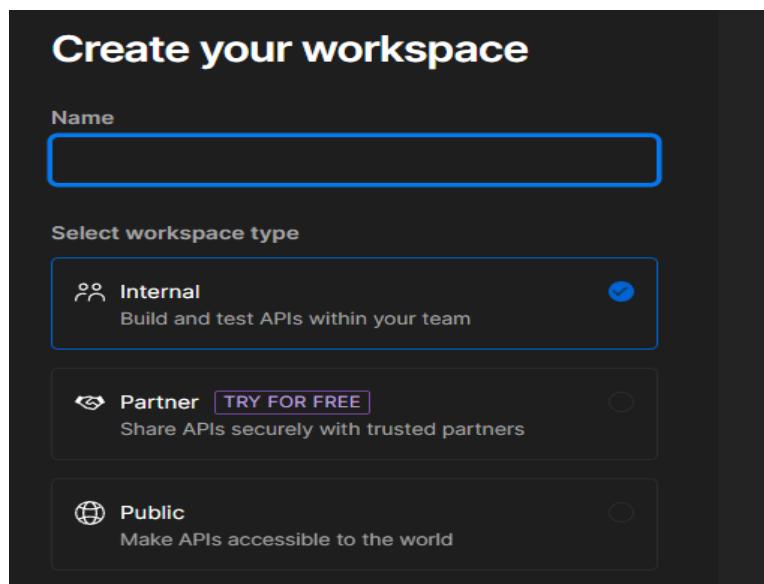
[Preguntas entregables](#)

Requisitos previos

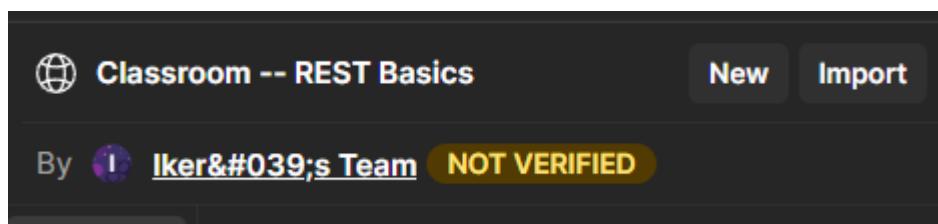
- Tener Postman instalado.
- Conexión a Internet.
- Usaremos endpoints públicos de prueba:
 - JSONPlaceholder (datos falsos): <https://jsonplaceholder.typicode.com>
 - httpbin (pruebas de HTTP): <https://httpbin.org>
 - reqres (opcional autenticación): https://reqres.in

Preparación del entorno en Postman (5 pasos)

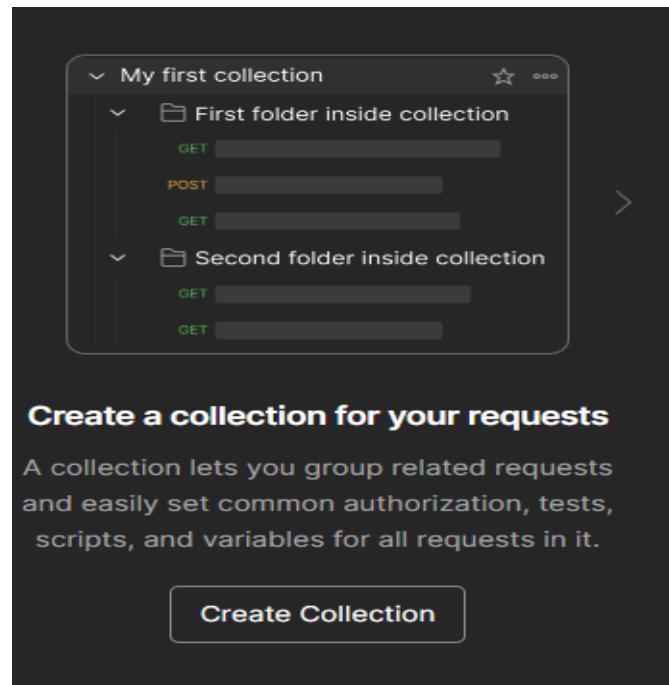
1. Crear un WorkSpace: Classroom – Rest Basics.



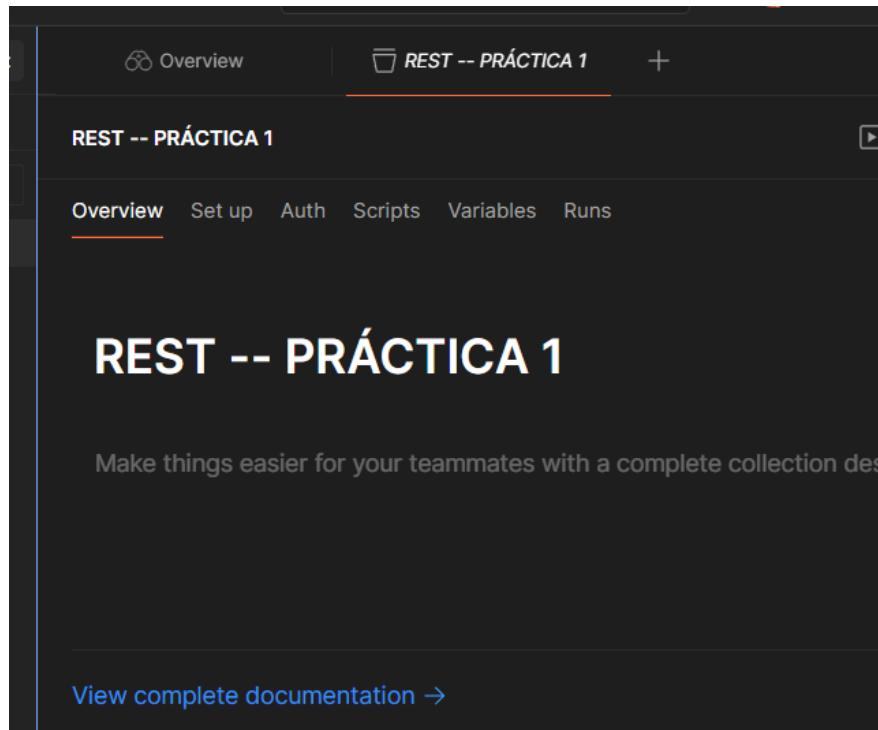
Una vez creado nos saldrá así:



2. Crear una Colección: REST – Práctica 1.

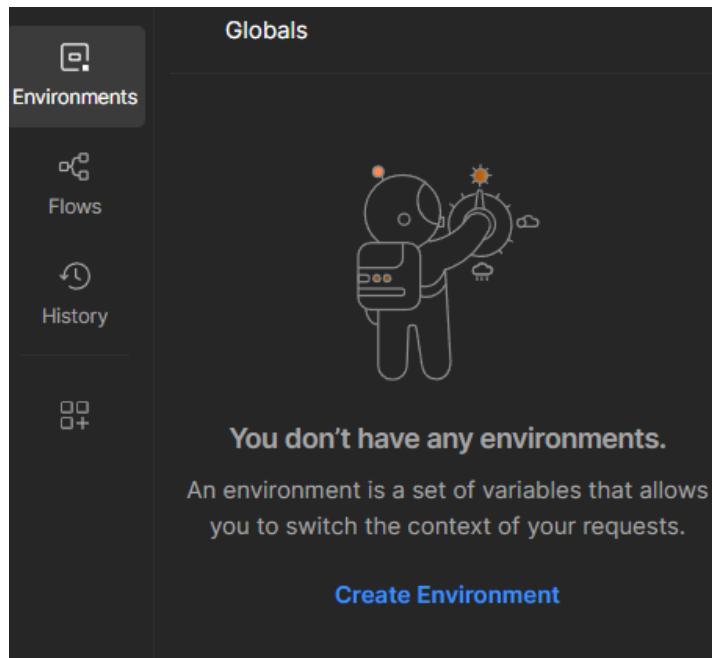


Una vez creado nos saldrá así:



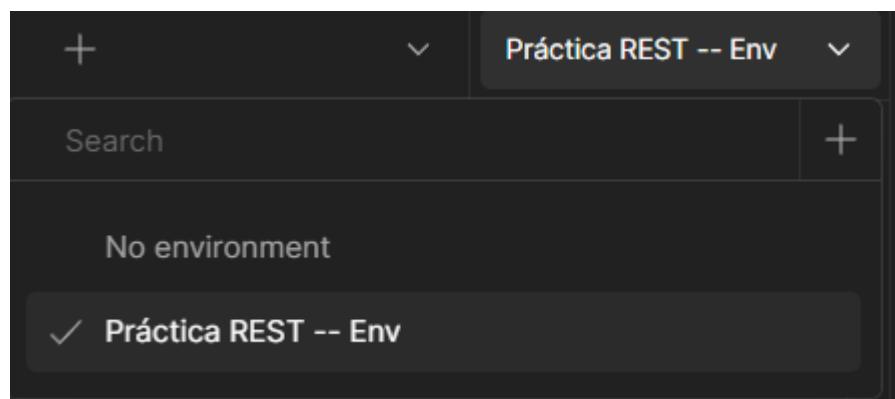
3. Crear un Entorno: Práctica REST – Env. Añade variables:

- baseUrl = https://jsonplaceholder.typicode.com
- binUrl = https://httpbin.org
- authUrl = https://reqres.in (opcional, para el apartado de login)



Práctica REST -- Env	
Variable	Value
baseUrl	https://jsonplaceholder.typicode.com
binUrl	https://httpbin.org
authUrl	https://reqres.in
Add variable	

4. Seleccionar el entorno en la parte superior de Postman:



5. (Opcional) En la Colección, en la pestaña Variables, define: contentType = application/json.

Variable	Value
contentType	application/json
Add variable	

Conceptos clave (mini-resumen)

API: Conjunto de servicios que permiten a un cliente interactuar con un servidor.

REST: Estilo de arquitectura que trabaja con recursos usando métodos HTTP y representaciones (normalmente JSON).

API RESTful: API que aplica correctamente los principios REST (URLs de recursos, verbos HTTP adecuados, sin estado, caché cuando toca, etc.).

Parte A — Lectura de recursos (GET)

A1. Listar recursos (GET collection)

Crea una petición en la colección:

- **GET {{baseUrl}}/posts**

Qué comprobar:

- Código de estado 200.

200 OK

- Header Content-Type: application/json.

Content-Type

①

application/json; charset=utf-8

- El cuerpo es un array de posts.

```
{  
    "userId": 1,  
    "id": 1,  
    "title": "sunt aut facere repellat provident occaecati excepturi optio reprehenderit",  
    "body": "quia et suscipit\\nsuscipit recusandae consequuntur expedita et cum\\nreprehenderit  
        molestiae ut ut quas totam\\nnostrum rerum est autem sunt rem eveniet architecto"  
},  
{  
    "userId": 1,  
    "id": 2,  
    "title": "qui est esse",  
    "body": "est rerum tempore vitae\\nsequi sint nihil reprehenderit dolor beatae ea dolores  
        neque\\nfugiat blanditiis voluptate porro vel nihil molestiae ut reiciendis\\nqui aperiam non  
        debitis possimus qui neque nisi nulla"  
},  
{
```

Test:

```
pm.test("Status 200", () => pm.response.to.have.status(200));  
pm.test("Content-Type JSON", () => pm.response.to.have.header("Content-Type"));  
pm.test("Respuesta es array", () => Array.isArray(pm.response.json()));
```

The screenshot shows the Postman interface with the 'Tests' tab selected. It displays three test cases in the 'Post-response' section:

```
1 pm.test("Status 200", () => pm.response.to.have.status(200));  
2  
3 pm.test("Content-Type JSON", () => pm.response.to.have.header("Content-Type"));  
4  
5 pm.test("Respuesta es array", () => Array.isArray(pm.response.json()));
```

The screenshot shows the Postman results panel with three test results displayed:

- PASSED Status 200
- PASSED Content-Type JSON
- PASSED Respuesta es array

A2. Obtener un recurso concreto (GET item con path param)

GET {{baseUrl}}/posts/1

Qué comprobar:

- Status 200.
- El cuerpo contiene id: 1.
- Guarda userId en una variable de entorno currentUserId.

Test:

```
const data = pm.response.json();
pm.test("id = 1", () => pm.expect(data.id).to.eql(1));
pm.environment.set("currentUserId", data.userId);
```

The screenshot shows the Postman test runner interface. At the top, it displays a 'GET' request with the URL {{baseUrl}}/posts/1. Below the request details, there are tabs for Overview, Params, Authorization, Headers (6), Body, Scripts (which is currently selected), and Settings. In the Scripts tab, there are two sections: 'Pre-request' and 'Post-response'. The 'Post-response' section contains the following code:

```
1 const data = pm.response.json();
2 pm.test("id = 1", () => pm.expect(data.id).to.eql(1));
3 pm.environment.set("currentUserId", data.userId);
```

At the bottom of the interface, there are tabs for Body, Cookies, Headers (25), and Test Results (1/1). The 'Test Results' tab is selected, showing a single result: 'PASSED' with the message 'id = 1'.

A3. Filtrado por query params

GET {{baseUrl}}/comments?postId=1

Qué comprobar:

- Status 200.
- Todos los comments tienen postId = 1.

Test:

```
const items = pm.response.json();
pm.test("Todos con postId=1", () => pm.expect(items.every(x => x.postId
===== 1)).to.be.true);
```

The screenshot shows a Postman test interface. At the top, it says "GET {{baseUrl}} /comments?postId=1". Below that, there are tabs for Overview, Params, Authorization, Headers (6), Body, Scripts (underlined), and Settings. The Scripts tab contains the following code:

```
const items = pm.response.json();
pm.test("Todos con postId=1", () => pm.expect(items.every(x => x.postId
===== 1)).to.be.true);
```

Under the "Test Results" tab, it shows "1/1" and "PASSED" with the message "Todos con postId=1". The status code is listed as "200 OK".

Parte B — Crear, actualizar y borrar (POST, PUT, PATCH, DELETE)

B1. Crear (POST)

*POST {{baseUrl}}/posts
Headers: Content-Type: {{contentType}}
Body (raw JSON):*
{
"title": "Mi primer post",
"body": "Esto es una prueba con Postman",
"userId": 99}

Qué comprobar:

- Status 201 (o 200 en algunos mocks).
- Respuesta JSON con un id nuevo.
- Guardar id en newPostId.

Test:

```
pm.test("Creacin ok (200/201)", () => pm.expect([200,201]).to.include(pm.response.code));
const created = pm.response.json();
pm.environment.set("newPostId", created.id);
pm.test("Tiene id", () => pm.expect(created).to.have.property("id"));
```

The screenshot shows the Postman interface with a successful POST request. The request URL is {{baseUrl}}/posts. The Body tab is selected, showing the raw JSON payload. The test results section shows two green 'PASSED' status boxes: 'Creacin ok (200/201)' and 'Tiene id'.

B2. Actualizar completo (PUT)

PUT {{baseUrl}}/posts/1

Body:

```
{  
  "id": {{newPostId}},  
  "title": "Título actualizado (PUT)",  
  "body": "Contenido actualizado por PUT",  
  "userId": 99  
}
```

Test:

```
const data = pm.response.json();  
pm.test("Status 200", () => pm.response.to.have.status(200));  
pm.test("PUT aplicó cambios", () => pm.expect(data.title).to.include("PUT"))  
);
```

The screenshot shows a Postman test collection interface. At the top, it says "PUT" and the URL "{{baseUrl}}/posts/1". Below this, there are tabs for Overview, Params, Authorization, Headers (8), Body, Scripts (green dot), and Settings. The Scripts tab is active. Under the "Pre-request" section, there is a code block with four numbered lines. Lines 1-3 are part of a pm.test block, and line 4 is a closing brace. The "Post-response" section is empty. At the bottom, there are tabs for Body, Cookies, Headers (24), and Test Results (2/2), with "Test Results" being the active tab. It shows two test cases: "Status 200" and "PUT aplicó cambios", both of which are marked as "PASSED". A status bar at the bottom right shows "200".

B3. Actualizar parcial (PATCH)

PATCH {{baseUrl}}/posts/1

Body:

```
{  
    "title": "Título modificado (PATCH)"  
}
```

Test:

```
pm.test("Status 200", () => pm.response.to.have.status(200));  
pm.test("PATCH aplic cambios", () => pm.response.json().title.includes("PATCH"));
```

The screenshot shows the Postman interface with a dark theme. At the top, it says "HTTP REST -- PRÁCTICA 1 / New Request". Below that, the method is set to "PATCH" and the URL is "{{baseUrl}}/posts/1". The header bar has tabs for Overview, Params, Authorization, Headers (8), Body, Scripts (which is currently selected), and Settings. In the "Pre-request" section, there are two test scripts:

```
1 pm.test("Status 200", () => pm.response.to.have.status(200));  
2 pm.test("PATCH aplic cambios", () => pm.response.json().title.includes("PATCH"));  
3
```

In the "Test Results" section, it shows 2/2 results:

- PASSED Status 200
- PASSED PATCH aplic cambios

The status bar at the bottom right shows "200 OK".

B4. Borrar (DELETE)

DELETE {{baseUrl}}/posts/1

Test:

```
pm.test("Borrado ok (200/204)", () => pm.expect([200,204]).to.include(pm.response.code));
```

Idea clave (REST):

- GET, PUT, DELETE son idempotentes.
- POST no es idempotente.

The screenshot shows the Postman interface with the following details:

- HTTP REST -- PRÁCTICA 1 / New Request**
- Method:** DELETE
- URL:** {{baseUrl}}/posts/1
- Scripts:** Pre-request script (highlighted):

```
1 pm.test("Borrado ok (200/204)", () => pm.expect([200,204]).to.include(pm.response.code));
```
- Test Results:** PASSED | Borrado ok (200/204) | 200 OK

Parte C — Headers, Auth y “sin estado”

C1. Header personalizado (User-Agent)

Petición GET con un header HTTP añadido por el cliente.

El servidor refleja los headers recibidos y podemos verificar el cambio.

Finalidad: Practicar la personalización de headers en las peticiones.

1. Abre una nueva petición: Método: GET.
2. URL: <https://httpbin.org/headers>
3. Ve a la pestaña Headers.
4. Añade: Key: User-Agent, Value: MiNavegador-REST/1.0
5. Envía la petición.

Respuesta esperada:

```
{  
  "headers": {  
    "Accept": "*/*",  
    "Host": "httpbin.org",  
    "User-Agent": "MiNavegador-REST/1.0"  
  }  
}
```

Test extra:

```
const j = pm.response.json();  
pm.test("User-Agent modificado", () => {  
  pm.expect(j.headers["User-Agent"]).to.eql("MiNavegador-REST/1.0");  
});
```

The screenshot shows the Postman interface with a successful API call. The request details are as follows:

- Método:** GET
- URL:** <https://httpbin.org/headers>
- Headers:** (7) - This tab is selected, showing the added header "User-Agent: MiNavegador-REST/1.0".
- Body:** ({} JSON) - The response body is displayed as a JSON object.
- Response Status:** 200 OK

The response body is a JSON object with the following content:

```
1  {  
2   "headers": {  
3     "Accept": "*/*",  
4     "Accept-Encoding": "gzip, deflate, br",  
5     "Host": "httpbin.org",  
6     "Postman-Token": "93fe1668-6101-4ec5-b848-df195cae6911",  
7     "User-Agent": "MiNavegador-REST/1.0",  
8     "X-Amzn-Trace-Id": "Root=1-68d40ae5-434d2b443af37573021eaeeaa"  
9   }  
10 }
```

C2. Autorización tipo Bearer

Petición GET con header de autenticación Bearer.

Enviamos el token en Authorization y el servidor valida si es correcto.

Finalidad: Practicar autenticación sin estado (stateless), en donde cada petición es independiente.

1. Crea variable de entorno token = alumno-123.

2. Crea:

GET {{binUrl}}/bearer

Authorization Bearer Token Token: {{token}}

Qué comprobar:

- Status 200 y authenticated: true.
- Sin token debe devolver 401.

Test:

```
pm.test("200 con token", () => pm.expect([200,401]).to.include(pm.response.code));
```

Principio REST: cada petición lleva lo necesario (sin estado).

The screenshot shows the Postman interface for a GET request to {{binUrl}}/bearer. The request details include:

- Method: GET
- URL: {{binUrl}}/bearer
- Headers (8): Authorization (set to Bearer {{token}})
- Body: None
- Scripts: A pre-request script is present:

```
1 pm.test("200 con token", () => pm.expect([200,401]).to.include(pm.response.code));
```

The test results show a successful 200 OK response with the following body:

```
{ "authenticated": true, "token": "alumno-123" }
```

Details at the bottom indicate the response was received in 11.63 seconds with a size of 283 B.

Parte D — Errores y latencia

D1. Códigos de error

Peticiones diseñadas para provocar errores de forma controlada.

El servidor devuelve el código HTTP correspondiente (404, 400).

Finalidad: Validar el manejo de errores en las pruebas automatizadas.

```
GET {{binUrl}}/status/404  
GET {{binUrl}}/status/400
```

Test:

```
pm.test("Es 404", () => pm.response.to.have.status(404));
```

The screenshot shows two separate Postman requests side-by-side. Both requests are set to 'GET' and point to the URL {{binUrl}}/status/{404|400}. The 'Headers' tab contains a 'Content-Type' header set to 'application/json'. The 'Scripts' tab in the left request contains the assertion: 'pm.test("Es 404", () => pm.response.to.have.status(404));'. The 'Scripts' tab in the right request also contains the same assertion. Both requests result in a '404 NOT FOUND' response, which is highlighted in red at the bottom of each interface. The 'Test Results' section for the left request shows a single test named 'Es 404' with a status of 'PASSED'. The right request's results are not fully visible but appear to show a similar outcome.

D2. Retrasos en la respuesta

Petición que simula una respuesta lenta.
El servidor espera antes de responder (ej. 3 segundos).

Finalidad: Comprobar configuración de timeout y comportamiento del cliente frente a la latencia.

GET {{binUrl}}/delay/3

Ajusta **Request timeout** si fuese necesario.

The screenshot shows the Postman interface with the following details:

- Method:** GET
- URL:** {{binUrl}}/delay/3
- Script Tab:** Scripts (selected)
- Test Results Tab:** Test Results (selected)
- Status:** 200 OK
- Duration:** 38.77 s
- Icon:** A cartoon character icon representing a user or developer.

Preguntas entregables

Breve respuesta escrita (máx. 10 líneas):

- Diferencia entre path y query params.

Path lo usamos para buscar algo en específico y único, mientras que query lo usamos para filtrar y por decirlo así realizar una búsqueda algo más general.

- ¿Por qué POST no es idempotente y PUT sí?

POST: crea cosas nuevas, si lo mandas varias veces, el resultado cambia → **no idempotente**.

PUT: actualiza algo concreto, si lo mandas varias veces, el resultado final es siempre el mismo → **idempotente**.

- ¿Qué significa que una API sea sin estado?

Cada petición que le mandas es independiente, la API no recuerda nada de lo que hiciste antes.