9 REST API Design Principles You Must Know Before Writing Code

---

1. Use Proper HTTP Methods
Match HTTP methods to the action being performed:
- **GET**: Retrieve resources.
- **POST**: Create resources.
- **PUT**: Update/replace resources.
- **PATCH**: Partially update resources.
- **DELETE**: Remove resources.

Examples:
- `GET /users` (Retrieve user list)
- `POST /users` (Create a new user)
- `PUT /users/123` (Update entire user data for ID 123)
- `PATCH /users/123` (Partially update user data for ID 123)
- `DELETE /users/123` (Delete user with ID 123)

---

2. Use Meaningful and Consistent Resource Names
- Use **nouns**, not verbs, for endpoints.
- Keep names **consistent and hierarchical**.
- Use **plural names** for collections.

Examples:
- `GET /books` (Correct) vs. `GET /getBooks` (Incorrect)
- `POST /orders` (Create a new order)
- `GET /users/456/orders` (Retrieve orders for user ID 456)

---

3. Statelessness
- The server should **not store any state** about the client session.
- Each request should contain **all required information** (e.g., authentication tokens).

Examples:
- `GET /profile` with `Authorization: Bearer <token>` header
- `POST /checkout` with full order details included in the request body

---

4. Implement Proper Status Codes

Use standard HTTP status codes:
- **200 OK**: Success.
- **201 Created**: Resource successfully created.
- **204 No Content**: Success with no response body.
- **400 Bad Request**: Client-side error.
- **401 Unauthorized**: Authentication failure.
- **404 Not Found**: Resource not found.
- **500 Internal Server Error**: Server-side error.

Examples:
- `GET /products/9999` → `404 Not Found` (If product ID 9999 does not exist)
- `POST /users` with missing fields → `400 Bad Request`

---

5. Support Filtering, Sorting, and Pagination
Allow clients to manipulate large collections efficiently:
- **Filtering:** `GET /products?category=electronics`
- **Sorting:** `GET /products?sort=price`
- **Pagination:** `GET /products?page=2&limit=10`

Examples:
- `GET /users?role=admin&status=active`
- `GET /orders?sort=date_desc&page=1&limit=20`

---

6. Version Your API
Use versioning to avoid breaking changes:
- **In URL:** `/v1/users`
- **In headers:** `Accept: application/vnd.example.v1+json`

Examples:
- `GET /v1/orders`
- `GET /v2/orders` (Newer version with updated response structure)

---

7. Use JSON Format for Data
- JSON is widely accepted and easy to parse.
- Follow a consistent field naming convention (e.g., camelCase or snake_case).

Examples:
- Request:

  ```json
  {
    "firstName": "John",
    "lastName": "Doe",
    "email": "john.doe@example.com"
  }
  ```
- Response:
  ```json
  {
    "id": 123,
    "firstName": "John",
    "lastName": "Doe",
    "email": "john.doe@example.com"
  }
  ```

---

8. Implement Authentication and Authorization
- Use industry-standard methods like **OAuth 2.0, JWT, or API keys**.
- Example: Add an `Authorization: Bearer <token>` header for secure access.

Examples:
- API key authentication:
  ```
  GET /users?api_key=abcdef123456
  ```
- JWT authentication:
  ```
  Authorization: Bearer eyJhbGciOiJIUzI1...
  ```

---

9. Document the API
Provide clear documentation with:
- **Endpoint definitions**
- **Parameters**
- **Request/response examples**
- **Error codes**

Use tools like **Swagger/OpenAPI, Postman, or Redoc**.

Examples:
- `/docs` endpoint serving OpenAPI documentation.
- Markdown-based API documentation hosted in a repository.

Bonus: Handle Errors Gracefully
Return meaningful error messages:
```json
{
  "error": "User not found",
  "code": 404,
  "message": "The requested user ID does not exist."
}
```

Examples:
- `GET /orders/12345` for a deleted order → `410 Gone`
- `POST /login` with incorrect credentials → `401 Unauthorized`

By following these principles, your REST API will be **scalable, maintainable, and user-friendly**.