RESTful Web API Design

Date: 03/27/2023

1. What is REST?

- **Definition:** Representational State Transfer (REST) is an architectural style for distributed systems.
- **Origin:** Proposed by Roy Fielding in 2000.
- **Characteristics:** Uses HTTP protocol, platform-independent, open standards.
- **Benefits:** Decouples client and service implementations, promotes scalability and flexibility.

2. Organize the API Design Around Resources

- **Resource-Centered Design:** Focus on business entities (e.g., customers, orders).
- **Use nouns for URIs rather than verbs** (e.g., /orders vs. /create-order).
- **URI Structure:** Utilize a hierarchical naming convention (e.g., /customers, /customers/5).
- **Avoid Complexity:** Simplify by using navigable links in response bodies.

3. Define API Operations Using HTTP Methods

- Standard HTTP Methods:
  - GET: Retrieve resource data.
  - POST: Create new resources or perform processing.
  - PUT: Create or update resources.
  - PATCH: Partially update resources.
  - DELETE: Remove resources.

4. Conform to HTTP Semantics

- **Media Types:** JSON and XML commonly used.
- **HTTP Status Codes:**
  - 200 OK: Successful GET or processing.
  - 201 Created: Resource successfully created.
  - 204 No Content: No response body, used for empty responses.
  - 400 Bad Request: Invalid data from the client.
  - 404 Not Found: Resource does not exist.
- **Asynchronous Operations:**
  - Use HTTP status code 202 Accepted for long-running requests.
  - Provide endpoints for monitoring request status.

RESTful Maturity Model

- Level 0: One URI with all operations as POST.
- Level 1: Separate URIs for resources.
- Level 2: Use HTTP methods for resource operations.
- Level 3: Implement hypermedia (HATEOAS).

Use Hypermedia (HATEOAS)

- **HATEOAS** (Hypertext As The Engine Of Application State)
- **Principle:** Clients can navigate the API using links in the response body.
- **Benefit:** Allows dynamic interaction with resources.
- **Key Aspects of HATEOAS:**
    - **Hypermedia-Driven:** The server includes hyperlinks in its responses. These links guide the client on how to navigate the API and interact with resources.
    - **Dynamic Interaction:** The client does not need prior knowledge of the API's URI structure. Instead, it uses the links provided in each response to discover available actions and navigate the application.
    - **State Management:** The client application moves through states based on the links and actions available in each response, similar to how a web browser navigates a website.

Example of HATEOAS

```
{
  "orderID": 3,
  "productID": 2,
  "quantity": 4,
  "orderValue": 16.60,
  "links": [
    { "rel": "customer", "href": "https://api.example.com/customers/3", "action": "GET" },
    { "rel": "self", "href": "https://api.example.com/orders/3", "action": "PUT" },
    { "rel": "self", "href": "https://api.example.com/orders/3", "action": "DELETE" }
  ]
}
```

Example of HATEOAS

Here, the "**links**" array provides URLs for:

- Retrieving customer information associated with the order (`GET` action).
- Updating (`PUT`) or deleting (`DELETE`) the current order (`self` reference).

Benefits of HATEOAS:

- **Decoupling:** The client is not tightly coupled with the server’s structure, making it resilient to changes in the API.
- **Discoverability**: Clients can automatically discover and utilize available functionality.
- **Flexibility**: It allows for API evolution without breaking existing clients.

HATEOAS enhances RESTful APIs by making them more robust, dynamic, and easier to maintain.

```
{

  "orderID": 3,

  "productID": 2,

  "quantity": 4,

  "orderValue": 16.60,

  "links": [

    { "rel": "customer", "href": "https://api.example.com/customers/3", "action": "GET" },

    { "rel": "self", "href": "https://api.example.com/orders/3", "action": "PUT" },

    { "rel": "self", "href": "https://api.example.com/orders/3", "action": "DELETE" }

  ]

}
```

Versioning a RESTful Web API

- **Approaches:**
  - **No Versioning:** Simple, best for internal APIs.
  - **URI Versioning:** Add version numbers to URIs (e.g., /v2/customers).
  - **Query String Versioning:** Specify version as a query parameter (e.g., /customers/1?version=2).
  - **Header Versioning:** Use custom headers (e.g., api-version).
  - **Media Type Versioning:** Specify version in the Accept header (e.g., application/vnd.example.v1+json).

Performance Considerations

- **Pagination and Filtering:**
    - Use query parameters (e.g., ?limit=25&offset=50) to manage large datasets.
- **Data Denormalization:**
    - Combine related data into larger resources to reduce "chatty" APIs.
- **Partial Responses:**
    - Support range requests and partial responses for large resources like files.

Conclusion

- Designing a RESTful API involves structuring resources intuitively, using HTTP methods appropriately, and ensuring evolution without breaking clients.
- Following REST principles leads to scalable, maintainable, and user-friendly APIs.

Links

- https://learn.microsoft.com/en-us/azure/architecture/best-practices/api-design
- https://restfulapi.net/