УДК 004.9

О.О. Решетнік

REST API DESIGN PATTERNS AND MATURITY MODEL

Вінницький національний технічний університет

Анотація

Використання патернів REST API покращує продуктивність, зручність та безпеку API. Запропоновано огляд основних патернів проектування. Також запропоновано розширену модель для оцінки зрілості API як розширення базової моделі Річардсона.

Ключові слова: REST API, патерни проектування API, модель оцінки API

Abstract

REST API design patterns and their application to improve usability, performance, and security of the API described. The article provides review of REST API design patterns. The article also proposes an extended maturity model built on top of Richardson's model.

Keywords: REST API, API design patterns, API maturity model

Introduction

RESTful API design patterns are a set of guidelines and best practices to follow when designing a RESTful API. They are designed to improve the usability, performance, and security of the API, and to provide a consistent and intuitive experience for API clients. In this article, we will explore the most common REST API design patterns and discuss how they can be applied to improve the design of your API. We will also provide some resources for further reading, to help you dive deeper into the world of REST API design.

REST API design patterns

Roy Fielding's dissertation, "Architectural Styles and the Design of Network-based Software Architectures" is a seminal work in the field of software architecture, and specifically in the design of network-based software architectures [1]. In the dissertation, Fielding proposes the Representational State Transfer (REST) architectural style, which has since become one of the most widely used architectural styles for web-based systems. The dissertation also explores other architectural styles and their benefits and drawbacks, and provides a framework for evaluating and comparing different styles. Overall, the dissertation is a comprehensive and influential work that has had a significant impact on the field of software architecture.

REST is not a strict protocol, but rather an architectural style that provides a set of guidelines and best practices for designing web-based systems. While these guidelines can be incredibly helpful in creating a well-designed and functional API, they are not strict rules, and REST APIs can vary greatly in their implementation and design. This freedom can sometimes lead to issues, such as inconsistent or incompatible APIs, security vulnerabilities, or poor performance. It's important to carefully consider the requirements of your API and its clients, and to follow best practices and design patterns to ensure that your API is secure, performant, and easy to use.

If you're looking to design or improve a REST API, it's important to be familiar with common design patterns. These patterns can help you create an API that is both user-friendly and secure. By using these guidelines, you can ensure that your API is consistent, intuitive, and easy to use. Check out the list of design patterns above and the resources provided to learn more about how to create an effective RESTful API [2-10].

1. Resource-Oriented Design: This pattern focuses on designing API endpoints based on resources, where each resource is represented by a unique URI.
2. HATEOAS (Hypermedia as the Engine of Application State): This pattern focuses on providing additional information about the API in the response body to allow clients to dynamically discover the available actions.
3. Filtering, Sorting, and Pagination: This pattern is used to support operations like filtering, sorting, and pagination in API results.
4. Versioning: This pattern allows multiple versions of the API to be supported, allowing clients to use a specific version that is compatible with their needs.

5. Error Handling: This pattern outlines how errors should be communicated to the client, including the use of HTTP status codes and error messages.

6. Caching: This pattern is used to optimize API performance by caching responses for a certain amount of time to reduce the number of requests made to the API.

7. Authentication and Authorization: This pattern outlines the methods for securely authenticating clients and granting them access to protected resources.

8. Compound Documents: This pattern allows related resources to be returned in a single response to reduce the number of round trips between client and server.

9. Validation: This pattern outlines how to validate incoming requests and return appropriate error messages in the case of invalid data.

10. Partial Updates: This pattern outlines how to handle partial updates of resources, allowing clients to make changes to a specific field rather than replacing the entire resource.

11. Batch Processing: This pattern outlines how to process multiple requests in a single call, reducing the number of round trips between client and server.

12. Sideloading: This pattern allows related resources to be included in the main response to reduce the number of round trips between client and server.

13. Code on Demand: This pattern allows clients to download executable code to be run on their end to enhance the functionality of the API.

14. Server-Driven Negotiation: This pattern outlines how the server can dynamically select a representation of a resource that is most appropriate for the client, based on client-specified preferences.

15. Idempotence: This pattern outlines how to make API calls that have the same outcome, regardless of the number of times they are executed.

16. Client-Generated IDs: This pattern outlines how to handle the creation of resources where the client generates the identifier for the new resource.

17. Return Types: This pattern outlines how to handle the return types of API calls, including whether to return full resources or just a summary, and whether to return a single resource or a collection of resources.

18. Asynchronous Processing: This pattern outlines how to handle long-running tasks asynchronously, allowing the client to continue processing other requests while the task is being executed.

19. Resource Naming: This pattern outlines how to name resources and resource collections in a consistent and meaningful way to improve discoverability and usability.

20. Documentation: This pattern outlines the importance of providing clear and comprehensive documentation for the API, including descriptions of resources, methods, and expected responses.

21. Resource Expansion: This pattern outlines how to include related resources in a single response, reducing the number of round trips between the client and server.

22. Resource Aggregation: This pattern outlines how to combine multiple resources into a single response to simplify the response for the client.

23. Query Parameters: This pattern outlines how to use query parameters to filter, sort, and paginate results from an API.

24. Query Language: This pattern outlines how to use a query language, such as GraphQL or SPARQL, to allow clients to specify exactly what information they need from the API.

25. Resource Descriptions: This pattern outlines how to include descriptions and metadata for each resource in the API response, to improve discoverability and usability.

These patterns, when applied appropriately, can help to improve the usability, performance, and security of a REST API. However, it's worth noting that REST APIs can vary greatly in their implementation and design, and these patterns are only guidelines and not strict rules. The specific patterns used will depend on the requirements of the API and its clients.

REST API maturity model extension

The Richardson Maturity Model is a model for assessing the maturity level of a RESTful API, based on the degree to which it adheres to the principles of the REST architectural style [11]. The model consists of four levels, each representing a higher level of maturity and adherence to REST principles:

Level 0: The API uses HTTP as a transport mechanism, but does not adhere to any RESTful principles.

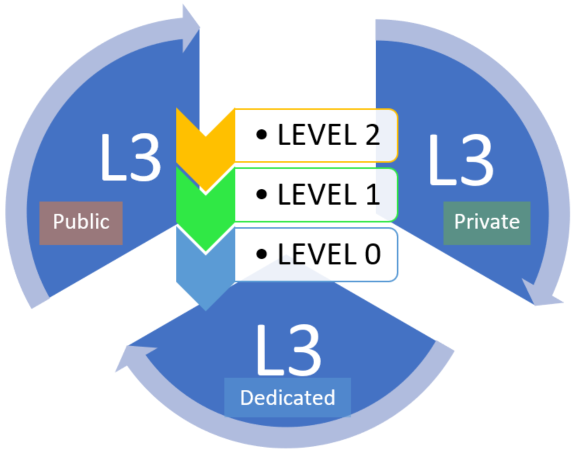
Level 1: The API uses HTTP methods to indicate the intended action for each request, but does not adhere to other RESTful principles.

Level 2: The API uses HTTP methods and adheres to the principle of resource orientation, where resources are identified by URIs and are manipulated through a uniform interface.

Level 3: The API uses HTTP methods, adheres to the principle of resource orientation, and supports HATEOAS (Hypermedia as the Engine of Application State), which allows clients to dynamically discover available actions.

The Richardson Maturity Model can be a useful tool for assessing the maturity and adherence to REST principles of a RESTful API, and for identifying areas for improvement.

We propose an alternative model for REST API maturity assessment that defines design patterns required for each level. This model, on the picture 1, goes beyond the Richardson Maturity Model's four levels by breaking down level 3 into several sections depending on API specialization: public, private or p2p dedicated. By following this model, you can ensure that your API meets the necessary design patterns for each level, and is therefore more mature and adheres to REST principles. It's important to note that while this model provides a useful framework for API design, each implementation will vary based on the features and requirements of the specific system with consideration to the consuming style: public, private or dedicated.



Picture 1 – An extended REST API maturity model

Conclusion

In conclusion, REST is a popular and widely used architectural style for web-based systems. While there are many design patterns and best practices to follow when designing a RESTful API, it's important to remember that each implementation will vary based on the features and requirements of the specific system. However, by following best practices and design patterns, you can create a well-designed and functional API that is secure, performant, and easy to use.

LIST OF REFERENCES

1. Fielding R.T. Architectural Styles and the Design of Network-based Software Architectures: Doctoral dissertation, University of California: Irvine, 2000. – 162 pages. – ISBN:978-0-599-87118-2
2. API Design Review FAQ. URL: https://google.aip.dev/100
3. Learn REST: A RESTful Tutorial. URL: https://www.restapitutorial.com/
4. Lokesh G. What is REST. URL: https://restfulapi.net/
5. Analytics REST API Guide: https://docs.wso2.com/display/DAS300/Analytics+REST+API+Guide
7. Richardson L., Amudsen M., Ruby S. RESTful Web APIs: Services for a Changing World – O'Reilly Media, Incorporated, 2013. – 406 pages. – ISBN 978-144-935-806-8
8. Masse M. REST API Design Rulebook – O'Reilly Media, Incorporated, 2011. – 112 pages. – ISBN 978-144-931-050-9
9. Newman S. Building Microservices: Designing Fine-Grained Systems – O'Reilly Media, Incorporated, 2015. – 280 pages. – ISBN 978-149-195-035-7
10. Geewax JJ. API Design Patterns – Manning, 2021. – 480 pages. – ISBN 978-161-729-585-0
11. Fowler M. Richardson Maturity Model. URL: https://martinfowler.com/articles/richardsonMaturityModel.html

Решетнік Олександр Олександрович – студент групи 1ПІ-22М, факультет інформаційних технологій та комп'ютерної інженерії, Вінницький національний технічний університет, Вінниця, e-mail: degratnik@gmail.com

Reshetnik Oleksandr O. – student of 1PI-22M group, Department of Information Technologies and Computer Engineering, Vinnytsia National Technical University, Vinnytsia, email: degratnik@gmail.com