RESTful API Lifecycle Management

BY JOHN VESTER

CONTENTS

INTRODUCTION

Application Programming Interface (API) design has been in existence since the early days of computing – shortly after programmers realized that a clearly defined set of methods or functions were beneficial in facilitating programmatic communication. While the specifications vary between various APIs, the end goal is to provide value to the programmer through utilization of the services gained from using an API.

Like many other elements of software engineering, a managed lifecycle is beneficial in facilitating API development. API Lifecycle Management requires the highest degree of management due to the impact of external API consumers — which may be unknown to the API developer. This is because developers using that API must rely on decisions that were made outside of their insight or control.

The number of different APIs are vast, ranging from proprietary routines to those based upon established standards. This document will be focused on RESTful API Lifecycle Management.

WHAT IS AN API?

According to TechTerms.com, an application programming interface (API) is "a set of commands, functions, protocols, and objects that programmers can use to create software or interact with an external system. It provides developers with standard commands for performing common operations so they do not have to write the code from scratch."

While utilized for decades within various segments of Information Technology (IT), the concept of API usage gained momentum through web-based services. These services initially grew through SOAP-based services, paving the way for the RESTful counterpart – which has contributed to a majority of the API growth over the last five to seven years.

DIFFERENTIATING BETWEEN SOAP AND REST

From a web-based services perspective, SOAP (Simple Object Access Protocol) and REST (REpresentational State Transfer) are the two primary options that exist for developers. It is important to understand how to differentiate between SOAP and REST.

REST	SOAP
An architecture style using standard HTTP to provide a simple manner of connectivity. A standardized or enforced contract does not exist.	A protocol utilizing a service's interface to expose business logic in a strictly enforced WSDL contract.
Exposes named resources, building upon POST, GET, PUT, DELETE, and PATCH operations.	Exposes functions and processes using XML-based protocol.
Security is handled by the underlying infrastructure.	Supports WS-Security, which provides the ability to protect both data from a privacy and integrity perspective.
Caching can be utilized to yield performance improvements.	Caching is not an option for SOAP method calls.
Limited to HTTP and cannot perform two-phase commits across distributed systems.	Supports WS-Atomic Transaction and allows for the ability to perform two-phase commits.
Allows use of multiple data formats (JSON, XML, text, user-defined).	Supports only XML format.
Smaller learning curve based upon the simplicity of the architecture style.	Learning curve is higher, but is justified by the advantages of using the standardized protocol.

IBM

IBM **API** Connect

API Connect integrates IBM StrongLoop and IBM API Management with a built-in gateway, allowing you to create, run, manage, and secure APIs and Microservices.

Unparalleled, integrated user experience.

ibm.biz/apiconnect

Understanding the benefits and differences between REST and SOAP is crucial when making architecture/design decisions regarding API development. Keep in mind that it is possible to support both REST and SOAP with your API offering. This is often a preferred approach, depending on the customer's needs.

INTERFACE MODEL

RESTful services utilize a uniform interface that decouples the architecture and is broken down into the following four constraints.

IDENTIFICATION OF RESOURCES

Resources are named using uniform resource identifiers (URI). The resources differ from the results that are returned to the client. Consider the following GET request:

```
http://dzone.com/products
```

This fictional request would contain a list of products offered by the fakelibrary.org domain, perhaps using JSON:

```
[
  {
      "id" : 1,
      "name" : "Product One"
  },
  {
      "id" : 2,
      "name" : "Product Two"
  },
  {
      "id" : 3,
      "name" : "Product Three"
  }
]
```

Using REST, the following GET example could be used to return a specific resource from the list of products:

```
http://dzone.com/products/2
```

This URI would return the product where the ID is equal to two:

```
{
  "id" : 2,
  "name" : "Product Two"
}
```

MANIPULATION OF RESOURCES THROUGH REPRESENTATIONS

With a representation of the resource on the client, modifications and deletions can occur – provided the calling program has proper authority. Using the example above, the following JSON data could be constructed:

```
{
  "id" : 2,
  "name" : "Product Two Updated"
}
```

And passed as the body for a PUT request to the following URI:

```
http://dzone.com/products/2
```

If the PUT is successful, the name for product with an ID = 2 would change from "Product Two" to "Product Two Updated."

SELF-DESCRIPTIVE MESSAGES

As part of the REST message, an internet media type (formerly known as a MIME type) is specified so that the proper parser can be invoked. A common internet media type is "application/json."

HYPERMEDIA AS THE ENGINE OF APPLICATION STATE (HATEOAS)

A RESTful client, upon accessing a URI path, has the ability to discover all the available actions and resources required – avoiding the need to perform any hard-coding of information.

INTERFACE CONTRACT

A RESTful client, upon accessing a URI path, has the ability to discover all the available actions and resources required – avoiding the need to perform any hard-coding of information.

- **Request:** Handles the inbound processing that has been sent to the RESTful server.
- **Response:** Encapsulates the information provided back to the client from the server.
- **Path:** The unique identifier of the resource being requested.
- **Parameters:** Elements included in the requests to filter or specify key-value pairs used during the request.

API SECURITY

SECURITY MODEL

RESTful applications rely on the underlying security for the API ecosystem rather than including security within the REST architecture style. In addition to securing RESTful API calls with the HTTPS protocol, session-based authentication should be utilized. Currently, most RESTful applications leverage the OAuth 2.0 and Open ID Connect (OIDC) protocols.

SAML

Security Assessment Markup Language (SAML) was originally designed by universities to grant access to libraries for students at other universities. Built upon XML and SOAP is the original federated identity system. SAML was introduced in the early 2000s during a time when the Internet browser was the primary client.

OAUTH 2

Created in 2006, OAuth 2 is an open standard for authentication protocol that provides authorization workflow over HTTP and authorizes devices, servers, applications, and APIs with access tokens instead of credentials. OAuth gained popularity from usage by Facebook, Google, Microsoft, and Twitter, who allow usage of their accounts to be shared with third-party applications or websites.

OPEN ID CONNECT (OIDC)

Open ID Connect (OIDC) extends OAuth 2 and includes user

information (an identity layer) as part of the request. Considered a modern version of SAML, OIDC allows for a range of clients – including web-based, mobile, and those using JavaScript.

JSON WEB TOKEN (JWT)

JSON Web Token (JWT) is an open standard for creating access tokens that assert some number of claims. Written in JSON, the tokens are designed to be compact – focused for use in a web-browser, single-sign on (SSO) context. While not an identity provider or service provider, JWT is used to pass authenticated user identities between identity and service providers.

RAML

RESTful API Modeling Language (RAML) is a language intended to describe RESTful APIs. RAML is written in the YAML human-readable data serialization language. The RAML effort was first proposed in 2013 and garnered support from technology leaders like MuleSoft, AngularJS, Intuit, Box, PayPal, Programmable Web and API Web Science, Kin Lane, SOA Software, and Cisco. The goal of RAML is to provide all the necessary information to describe RESTful APIs, thus providing a simpler way to design APIs.

A sample RAML file of the Notes Example API (courtesy of MuleSoft) is shown below.

```
#%RAML 0.8
title: Notes Example API
version: v2
mediaType: application/json
documentation:
  - title: Overview
    content: This is an example of a simple API for a
    “notes” service
/notes:
  description: A collection of notes
  get:
    description: List all notes, optionally filtered by
    a query string
    queryParameters:
      q:
        description: An optional search query to filter
        the results
        example: shopping
    responses:
      200:
        body:
          example: |
            [ { “id”: 1, “title”: “Buy some milk”,
            “status”: “done” },
              { “id”: 2, “title”: “Return sweater”,
            “status”: “overdue”, “dueInDays”: -2 },
              { “id”: 3, “title”: “Renew license”,
            “status”: “not done”, “dueInDays”: 1 },
              { “id”: 4, “title”: “Join gym”, “status”:
              “not done”, “dueInDays”: 3 } ]
    post:
        description: Create a new note in the collection
        body:
          example: |
          { “title”: “Return sweater”, “dueInDays”: -2 }
    headers:
        X-Tracking-Example:
          description: You can specify request headers
          like this
```

code continued top right

```
          enum: [ accounting, payroll, finance ]
          required: false # require it by changing this to
          true
          example: accounting
      responses:
        201:
          headers:
            X-Powered-By:
              description: You can describe response
              headers like this
              example: RAML
          body:
            example: |
              {
                “id”: 2,
                “title”: “Return sweater”,
                “status”: “overdue”,
                “dueInDays”: -2
              }
  /{id}:
    description: A specific note, identified by its id
    uriParameters:
      id:
        description: The `id` of the specific note
        type: number
        example: 2
    get:
      description: Retrieve the specified note
      responses:
        200:
          body:
            example: |
              {
                “id”: 2,
                “title”: “Return sweater”,
                “status”: “overdue”,
                “dueInDays”: -2
```

RAML itself provides a full API design lifecycle, broken into five categories.

DESIGN

By use of the easy-to-read YAML format, API design can become more visual than prior API development approaches. Utilizing a dedicated RAML tool (API Workbench, API Designer) or IDE plug-in (Sublime, Visual Studio) facilitates faster development, eliminating code duplication and providing functionality to prototype and perfect APIs being developed.

With the building blocks in place for an API inside the RAML file, mock data can be added to allow for prototyping and testing before any actual program code is written. As a result, designers can sit with stakeholders and product owners to validate the API early in the development process.

BUILD

With the design of the RAML file in place, the actual programming of the API logic can begin. At this point, the RAML file becomes a specification and popular languages like NodeJS, Java, .NET, Mule, and IOT Noble can simplify the build process.

Below is an example based in Java and the RAML for JAX-RS framework:

```
@Path(“/notes”)
public interface NotesExampleResource
{
   @POST
   @Consumes(“application/json”)
   Response createNote(Note note, @Context UriInfo
   uriInfo);

   @GET
   @Produces(“application/json”)
   Notes getNotes(@QueryParam(“q”) String query,
                          @Context UriInfo uriInfo);

  ...
}
```

Using the RAML for JAX-RS framework, it is possible for Java interfaces to generate a RAML file, as well, which provides another option for leveraging the RAML specification.

TEST

With the design and build phases in place, the next logical step in the API Development Lifecycle is the testing stage. These unit tests are critical to making sure that the API being developed maintains any backward compatibility while also meeting all the current requirements.

Tools like Abao, Vigia, and Postman allow RAML specifications to be imported, leading to setup scripts and tests being created to validate the API. Additionally, testing services (like API Fortress, API Science, and SmartBear) provide assistance toward testing latency, responses, payloads, and errors.

DOCUMENT

API documentation has been a challenge, with tools like Swagger and Miredot often falling short at providing complete information and leading us to rely on developers to specify cryptic annotations and language-specific documentation like JavaDocs.

With the RAML specification keeping documentation as a core priority, the documentation is kept in sync with the code itself. This refreshing benefit is due to the fact that the RAML specification serves as an interface (or contract) for the API itself – in sync with the underlying business logic providing results for the service.

Tools like API Console, RAML to HTML, and RAML2HTML for PHP provide quick and easy ways to expose standardized documentation – which can be kept private on a corporate intranet or available for public consumption.

SHARE

With all the building blocks in place in the API Development Lifecycle, the final segment focuses on sharing the API. The RAML specification introduces several ways in which APIs can be integrated.

- **SDK Generation:** Languages like Java, .NET, PHP, Ruby, NodeJS, iOS, Windows, and Go provide push-button functionality to build Software Development Kits (SDKs) automatically using the RAML file.
- **Third Party Tooling:** Oracle and MuleSoft include RAML functionality into their toolsets to provide the ability to connect to any API utilizing RAML just by pasting in the specification.
- **API Notebook:** Provides an environment for developers to test APIs, manipulate results from API calls, and connect to multiple APIs using the JavaScript language.

RAML 0.8 V 1.0

RAML specification 0.8 continues to be the current standard, but version 1.0 began gaining momentum in September 2016. Version 1.0 includes the following updates.

- **Data types:** Provides a unified and efficient way to model API data with support for sub-schemas.
- **Examples:** Multiple examples and allowing annotations to facilitate injection of semantics.
- **Annotations:** Incorporating the proven pattern to allow for extensibility.
- **Libraries:** Improved modularity to promote API artifact reuse.
- **Overlays/extensions:** Allowing use of separate files to increase extensibility.
- **Improved security schemas:** Additional OAuth support and key-based schemas.

API VERSIONING

Versioning RESTful APIs has been a topic of great debate, mostly around the way versioning is implemented. The three main options for versioning are URI, HTTP Header, and Message Schema Identifier.

While there is no right or wrong answer, the recommendation is to set a standard and stick with that decision to reduce confusion from consumers of your API.

URI

The URI-based versioning includes the version number in the URI for the RESTful API. As an example, the 3.0 version of the products API would appear as follows:

```
http://dzone.com/v3.0/products
```

This approach is currently the most popular because it is very clear to see which API version is being utilized. Critics of the approach indicate that the URI to the resource should not change just because the version of the API is changing.

HTTP HEADER

The HTTP Header approach focuses on keeping the URI clean and adds version information in the header. The 3.0 version of the products API would maintain a generic URI:

```
http://dzone.com/products
```

However, the HTTP Header would include the following information:

```
HTTP GET:
https://dzone.com/products
api-version: 3.0
```

While the URIs are always the same, critics of this approach point out that this approach is not a semantic way to describe the resource.

MESSAGE SCHEMA IDENTIFIER (CONTENT TYPE)

Like the HTTP Header option, the Message Schema Identifier (or Content Type) versioning strategy creates a custom internet content type within the header. So, the same generic URI is used:

```
http://dzone.com/products
```

The header is updated to reflect expecting a custom content type:

```
Accept: application/vnd.fakelibrary.app.products-
v3.0+json
```

Again, the URIs are always the same, but critics of this approach point out that the version reference is hidden and that custom internet content types can appear messy and are difficult to test.

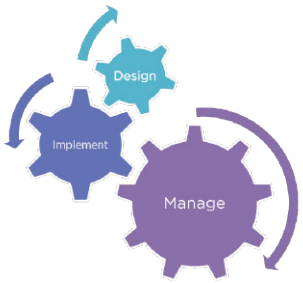
NO VERSIONING

While not an option for public APIs, those who are developing APIs internally and have influence and control over all the consumers of the API may consider not implementing versioning at all. In this instance, the challenges associated with versioning and maintaining multiple versions can be avoided.

Critics of this approach are likely to point out that this approach is only one public integration away from versioning needing to be addressed. Thus, even private APIs should be designed and treated as publically available resources – which would include the need for versioning.

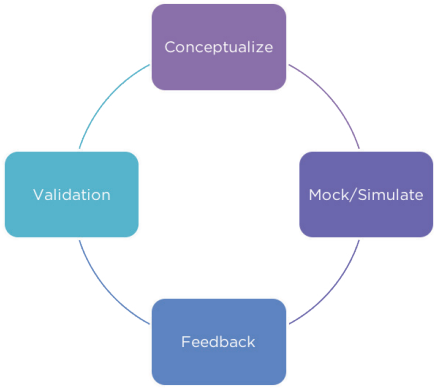
API LIFECYCLE

The API Lifecycle itself builds upon the existing concepts that have been discussed. At the highest level, three core aspects exist – Design, Implement, and Manage – each containing their own respective lifecycles.



DESIGN

The Design lifecycle maintains similarities to the RAML Development Lifecycle (noted above). This is by design, since the RAML specification was founded as a result of successful API design.



Conceptualize: Includes the initial design and requirements-gathering tasks surrounding the API. Prior to the RAML specification, some degree of building was required to return a set of results for the Mock/Simulation phase.

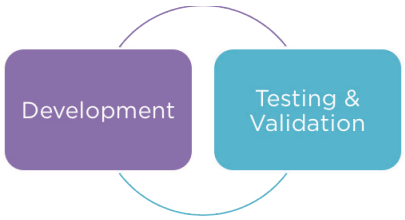
Mock/Simulate: Provides results from the API in a mocked or simulated manner. The actual API itself has not been built, but calling the API simulates data that is expected – paving the way for the feedback phase.

Feedback: Brings the stakeholder or product owner into the discussion so that they can review the results (while only mocked at this point) and compare them with the expectations that were set during the conceptualize phase.

Validation: Upon receiving feedback, the API design is validated and considered ready for the Implement aspect of the API Lifecycle.

IMPLEMENT

The implement aspect of the API Lifecycle focuses on the development and testing/validation of the actual program code and processes required for the API itself. This simple flow is depicted below:

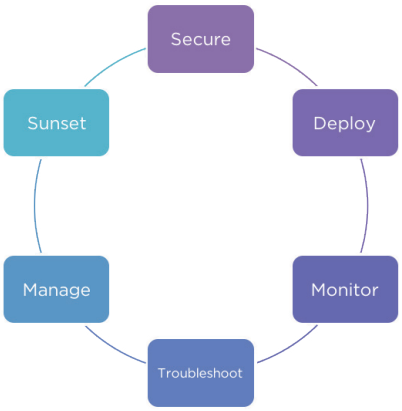


Development: The actual programmatic development, including unit and integration tests, required to meet the needs of the API.

Testing & Validation: Handles the Quality Assurance (QA) effort of validating that the acceptance criteria is being met by the API service being tested.

MANAGE

With the API designed, developed, and validated, the Manage aspect handles the remaining tasks associated with making the API available to consumers. Six elements are part of this final flow:



Secure: Handles aspects related to securing the API. This includes adding options for thresholds and varying service levels and setting access levels to the API. It is at this point that Information Security teams get involved by reviewing the API and/or performing penetration tests against the pre-production version of the service.

Deploy: With the API created, validated, and secured, the deployment of the API is handled using Continuous Delivery/ Continuous Integration (Jenkins, Bamboo/Pipelines, GitLab, Travis CI) tools as the next part of the management lifecycle.

Monitor: At this point, DevOps or Network Operations participates in the flow by monitoring usage of the API.

Troubleshoot: When issues arise with a deployed API, the logs from the runtime are utilized to help diagnose the cause. If a tracing framework exists within the design, the ability exists to trace a given message/request/transaction through the lifecycle to help identify the situation.

Manage: Makes sure the API has the necessary capacity to meet current and future needs. This can include increasing the number of instances running and the overall sizing of a given run-time environment hosting the service.

Sunset: When the API is no longer required or needed, this final step of the lifecycle handles properly sunsetting the API. In regulated environments, additional tasks may be required to provide insight into APIs that were relied on at one time but are no longer available.

CONCLUSION

RESTful API Lifecycle Management consists of three core aspects: Design, Implement, and Manage. These three aspects span the full life of an API from conception, to validation, to implementation, to finally deprecation. The lifecycle is built upon the proven RESTful API design and wraps the simplicity around concepts that will assure a stable and secure implementation with the ability to scale as required.

The introduction of RAML has helped standardize elements in the Design phase, but is architected to align well within the entire RESTful API Lifecycle Management structure. Usage of RAML places organizations in a better position to build, deliver, and document APIs – all using standard nomenclature.

ABOUT THE AUTHOR

JOHN VESTER is an Information Technology professional with 25+ years expertise in application development, project management, system administration, and team management. Currently focusing on enterprise architecture/application design/Continuous Delivery utilizing object-oriented programming languages and frameworks. Prior expertise building Java-based APIs against React and Angular client frameworks. CRM design, customization and integration with Salesforce. Additional experience using both C# (.NET Framework) and J2EE (including Spring MVC, JBoss Seam, Struts Tiles, JBoss Hibernate, Spring JDBC).

BROUGHT TO YOU IN PARTNERSHIP WITH IBM

DZONE, INC.
150 PRESTON EXECUTIVE DR.
CARY, NC 27513

888.678.0399
919.678.0300

REFCARDZ FEEDBACK WELCOME
refcardz@dzone.com

SPONSORSHIP OPPORTUNITIES
sales@dzone.com