API Management

An Architect's Guide to Developing and Managing APIs for Your Organization

First Edition

Brajesh De

Apress®

API Management

An Architect's Guide to Developing and Managing APIs for Your Organization

First Edition

Brajesh De

Apress®

API Management: An Architect's Guide to Developing and Managing APIs for Your Organization

Brajesh De
Bangalore, Karnataka, India

ISBN-13 (pbk): 978-1-4842-1306-3 ISBN-13 (electronic): 978-1-4842-1305-6
DOI 10.1007/978-1-4842-1305-6

Library of Congress Control Number: 2017935977

Copyright © 2017 by Brajesh De

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

Trademarked names, logos, and images may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, logo, or image we use the names, logos, and images only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The use in this publication of trade names, trademarks, service marks, and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

Managing Director: Welmoed Spahr
Editorial Director: Todd Green
Acquisitions Editor: Celestin Suresh John
Development Editor: Matthew Moodie
Technical Reviewer: Chandresh Pancholi
Coordinating Editor: Prachi Mehta
Copy Editor: Kim Burton-Weisman
Compositor: SPi Global
Indexer: SPi Global
Artist: SPi Global
Cover image designed by Freepik

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail rights@apress.com, or visit http://www.apress.com/rights-permissions.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at http://www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/978-1-4842-1306-3. For more detailed information, please visit http://www.apress.com/source-code.

Printed on acid-free paper

Dedicated to my family for their constant encouragement and support

Contents at a Glance

About the Author xv
About the Technical Reviewer xvii
Acknowledgments xix
Chapter 1: Introduction to APIs 1
Chapter 2: API Management 15
Chapter 3: Designing a RESTful API Interface 29
Chapter 4: API Documentation 59
Chapter 5: API Patterns 81
Chapter 6: API Version Management 105
Chapter 7: API Security 111
Chapter 8: API Monetization 143
Chapter 9: API Testing Strategy 153
Chapter 10: API Analytics 165
Chapter 11: API Developer Portal 171
Chapter 12: API Governance 179
Index 189

Contents

About the Author xv

About the Technical Reviewer xvii

Acknowledgments xix

■ **Chapter 1: Introduction to APIs** 1

The Evolution of APIs 3

APIs Are Different from Web Sites 5

Defining an API and Its Characteristics 5

Types of APIs 6

Examples of Popular APIs 8

The Difference Between a Web Service and a Web API 10

How Are APIs Different from SOA? 11

The API Value Chain 13

Business Models for APIs 14

■ **Chapter 2: API Management** 15

Secure, Reliable, and Flexible Communication 17

The API Gateway 18

API Auditing, Logging and Analytics 23

API Analytics 24

Developer Enablement for APIs 25

Developer Portal 25

API Lifecycle Management 27
API Creation 27
API Publication 27
Version Management 27
Change Notification 28
Issue Management 28
■ **Chapter 3: Designing a RESTful API Interface 29**
REST Principles 29
Uniform Interface 30
Client-Server 30
Stateless 30
Cache 30
Layered Systems 31
Code on Demand 31
Designing a RESTful API 31
Identification of Resources 31
Manipulation of Resources through Representation 33
Self-Descriptive Messages 33
Hypermedia as the Engine of Application State (HATEOAS) 33
Resource Identifier Design Using URIs 34
Resource Naming Conventions 34
Modelling Resources and Subresources 34
Best Practices for Identifying REST API Resources 35
URI Path Design 35
URI Format 36
Naming Conventions for URI Paths 37
HTTP Verbs for RESTful APIs 37
GET 38
POST 39

PUT 39
DELETE 40
PATCH 41
OPTIONS 41
HEAD 42
Idempotent and Safe Methods 42
HTTP Status Code 42
Resource Representation Design 45
Hypermedia Controls and Metadata 46
Accept (Client Request Header) 47
Accept-Charset (Client Request Header) 47
Authorization (Client Request Header) 48
Host (Client Request Header) 48
Location (Server Response Header) 48
ETag (Server Response Header) 49
Cache-Control (General Header) 49
Content-Type (General Header) 49
Header Naming Conventions 49
Versioning 50
Querying, Filtering, and Pagination 50
Limiting via Query-String Parameters 51
Filtering 51
The Richardson Maturity Model 52
Level 0: Swamp of POX (Plain Old XML) 53
Level 1: Resources 54
Level 2: HTTP Verbs 55
Level 3: Hypermedia Controls 56

■ Chapter 4: API Documentation 59
The Importance of API Documentation 59
Audience for API Documentation 60
Model for API Documentation 60
Title 61
Endpoint 62
Method 62
URL Parameters 62
Message Payload 62
Header Parameters 63
Response Code 64
Error Codes and Responses 64
Sample Calls 65
Tutorials and Walk-throughs 65
Service-Level Agreements 66
API Documentation Standards: Swagger, RAML, and API Blueprint 66
Swagger 66
RAML 69
API Blueprint 75
Comparing Swagger, RAML, and API Blueprint 77
Other API Documentation Frameworks 80
■ Chapter 5: API Patterns 81
Best Practices for Building a Pragmatic RESTful API 81
API Management Patterns 86
API Facade Pattern 86
API Throttling 92
Caching 93
Logging and Monitoring 94
API Analytics 95

API Security Patterns.. 95

Common Forms of Attack 95

API Risk Mitigation Best Practices..96

API Deployment Patterns... 100

Cloud Deployment 100

On-Premise Deployment...102

API Adoption Patterns.. 102

APIs for Internal Application Integration 103

APIs for Business Partner Integration..103

APIs for External Digital Consumers 103

APIs for Mobile 104

APIs for IoT 104

■ **Chapter 6: API Version Management.. 105**

API Versioning vs. Software Versioning 105

The Need to Version APIs.. 106

API Versioning Principles... 106

The API Version Should Not Break any Existing Clients 106

Keep the Frequency of Major API Versions to a Minimum 106

Make Backward-Compatible Changes and Avoid Making New API Versions........106

API Versioning Should Not Be Directly Tied to Software Versioning 107

Approaches to API Version Management... 107

Versions Using URLs...107

Versions Using an HTTP Header 108

Versions Using Query Parameters 108

Versions Using a Host Name..109

Handling Requests for Deprecated Versions 109

API Version Lifecycle Management 109

Chapter 7: API Security 111

The Need for API Security 111

API Security Threats 112

API Authentication and Authorization 113

API Keys 113

Username and Password 114

X.509 Client Certificates and Mutual Authentication 115

OAuth 115

OpenID Connect 123

Protecting Against Cyber Threats 133

Injection Threats 134

Insecure Direct Object Reference 136

Sensitive Data Exposure 136

Cross-Site Scripting (XSS) 137

Cross-Site Resource Forgery (CSRF or XSRF) 138

Bot Attacks 139

Considerations for Designing an API Security Framework 140

API Security Threat Model 140

API Security Recommendations 141

Chapter 8: API Monetization 143

Which Digital Assets Can Be Monetized? 143

How to Increase Revenue Using APIs? 143

Increase Customer Channels 143

Increase Customer Retention 144

Upsell Premium and Value-Added Services 144

Increase Affiliate Channels 145

Increase Distribution Channels 145

API Monetization Models 145
Free Model 146
Fee-Based Model (a.k.a. Developer Pays Model) 147
Revenue-Sharing Model 148
Monetization Concepts 149
API Product 149
API Package 150
Rate Plan 150
Billing Documents 151
Monetization Reports 151
Chapter 9: API Testing Strategy 153
The Importance of API Testing 153
Challenges in API Testing 153
API Testing Considerations 154
API Interface Specification Testing 155
API Documentation Testing 156
API Security Testing 156
Testing API Gateway Configuration 157
API Performance Testing 158
Preparing for the Load Test 158
Setting up for the Load Test 160
API Performance Test Metrics 161
Selecting The Right API Testing Tool 162
Must-Have Features 162
Nice-to-Have Features 163
Common API Testing Tools 164

■ Chapter 10: API Analytics 165
The Importance of API Analytics 165
API Analytics Stakeholders 166
API Metrics and Reports 168
Custom Analytics Reports 169
■ Chapter 11: API Developer Portal 171
The API Lifecycle 171
Publishing and Sharing APIs 171
The Importance of the API Developer Portal 172
Supporting App Developers 172
Invitations 173
Social Forums 173
Federated Developer Communities 174
Types of Portal Users 174
API Developer Portal Features 175
The Relationship Between a Developer Portal and an API Gateway 177
■ Chapter 12: API Governance 179
The Scope of API Governance 179
The Aim of API Governance 182
API Governance Model 182
Index 189

About the Author

Brajesh De is a seasoned technology expert with over 18 years of experience in technology consulting, architecture, design and implementation of highly distributed and scalable application integration solutions using REST API, SOA and JEE technologies. He is an Accenture certified Senior Technology Architect. With specialization in API Management, he currently leads the API Management capability for Accenture's India Development Center. Prior to joining Accenture, he has worked as a Principal Architect with Apigee, architecting API Management solutions for large enterprises in Telco domain. He has also worked with Dell, where he was responsible for SOA governance rollout and building integration solutions for Dell's internal applications using SOA technologies. Before Dell he was working as a Senior Technical Architect with Wipro Technologies where he has been instrumental in building complex integration solution for their tier one clients.

Brajesh is also an experienced trainer, providing corporate training in advanced API and SOA technologies. He holds a B. Tech degree in Electrical Engineering from IIT-BHU, Varanasi. He was awarded the IIT BHU gold medal for securing the first position and first division in B.Tech. Electrical Engineering Examination, 1998.

About the Technical Reviewer

Chandresh Pancholi is SDE-3 at nnnow.com (Arvind Internet group). Prior to that, he worked with Flipkart Internet Pvt. Ltd. as a senior software developer. He has worked on multiple back-end frameworks, such as Spring, Dropwizard, Flask, Golang, and Spring Boot. Chandresh graduated from LNMIIT, Jaipur and received a master's degree from BITS, Pilani. He is also a keen contributor to Apache open source foundations projects.

Acknowledgments

First and foremost, I would like to thank my wife, Roopa, for her constant love, support and sacrifice throughout the lengthy process of authoring this book. She has been my source of encouragement and inspiration from start to finish. Her timely reminders helped me to pen down each chapter at a steady pace. My son Bornik deserves special thanks for his subtle yet valuable inputs, which helped me to plan the contents of each chapter. Without his patience and sacrifice, getting time to write this book would have been an uphill task. Last but not the least, no words can express the love and blessings of my parents, Bamapada and Minakshi; without them, I could not have authored this book.

I would also like to thank Celestin Suresh John, Prachi Mehta, Baby Gopalakrishnan, Mercy Thomas and all the editors of this book for their support, review comments and input that helped to constantly improve the quality of each chapter.

CHAPTER 1

Introduction to APIs

API stands for application programming interface. An API helps expose a business service or an enterprise asset to the developers building an application. Applications can be installed and accessed from a variety of devices, such as smartphones, tablets, kiosks, gaming consoles, connected cars, and so forth. Google Maps APIs for locating a place on a map, Facebook APIs for gaming or sharing content, and the Amazon APIs for product information are some examples of APIs. Developers use these APIs to build cool and innovative apps that can provide an enriching user experience. For example, developers can use APIs from different travel companies to build an app that compares and displays each travel companies' price for the same hotel. A user can then make an informed decision and book the hotel through the company that is providing the best offer. This saves the user from doing the comparison on his own—thus improving his overall experience. APIs thus help provide an improved user experience.

An API is a software-to-software interface that defines the contract for applications to talk to each other over a network without user interaction. When you book a hotel room online from a travel portal with your credit card, the travel portal/application sends your booking information to the hotel's reservation system to block the room. It also sends the credit card information to a payment application. The payment application interacts with a remote banking application to validate the credit card details and process the payment. If the processing is successful, the hotel room is reserved for you. The interaction of the travel portal with the hotel's reservation system and the payment application both use APIs. As a user, you see only one interaction to collect the booking and credit card information. But behind the scenes, the applications work together using APIs. An API does this by "exposing" some of the business functions to the outside world in a limited fashion. That makes it possible to share the business services, assets, and data in a way that can be easily consumed by other applications, without sharing the code base. APIs can be thought of as *windows to the code base*. They clearly define exactly how a program will interact with the rest of the software application—saving time and resources, and avoiding any potential legal entanglements along the way. The API contract defines how the service will be provided by the provider and consumed by a consumer. The contract can include things like the definition and terms of service, SLAs like uptime/availability, licensing agreements for the usage of the service, pricing and support model etc.

© Brajesh De 2017
B. De, *API Management*, DOI 10.1007/978-1-4842-1305-6_1

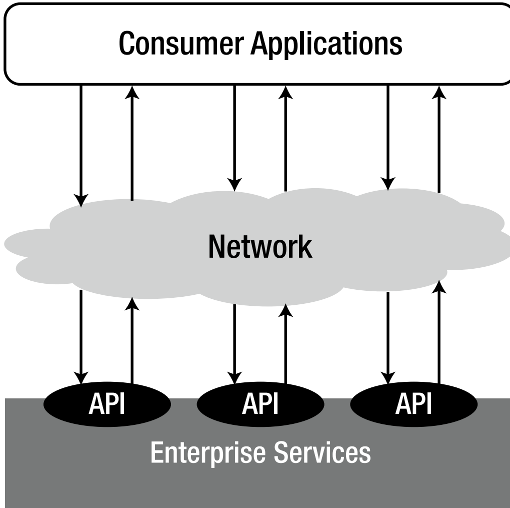

Figure 1-1. *An API provides an interface for consumer applications to interact with enterprise services over a network*

The contract defines the protocol, the input and output formats, and the underlying data types to be used for the software components to interact. It defines the functionality that is independent of the underlying implementation technologies of the component. The underlying implementation may change, but the contract definition should remain constant. The contract helps increase the confidence and thus the use of a component. An API with a well-defined contract provides all the building blocks needed to easily create a software application.

The term API in this book refers to web APIs, a.k.a. REST APIs; such APIs are implemented using REST principles, the details of which are covered in subsequent chapters of this book.

This chapter covers the following topics:

- The evolution of APIs
- The difference between web APIs and web sites
- The characteristics of an API
- The types of APIs (using some popular examples)
- The difference between web APIs, a web services, and service-oriented architecture
- An API value chain
- Various business models for APIs

The Evolution of APIs

The term *API* may mean different things to different people, depending on the context. There are APIs for operating systems, applications, and the Web. For example, Windows provides APIs that are used by system hardware and applications. When you copy text or a picture from Microsoft PowerPoint to Word, the APIs are at work. Most operating environments provide an API so that programmers can write applications consistent with the operating environment. Today when you talk about APIs, you are probably referring to web APIs built using REST technologies. Hence, web APIs are synonymous to REST APIs. Web APIs allow you to expose your assets and services in a form that can be easily consumed by another application remotely over HTTP(s). The following describes the evolution of the modern-day web API:

2000: Roy Thomas Fielding's dissertation, "Architectural Styles and Design of Network Based Software Architectures," is published.

February 2000: APIs are first demonstrated by SalesForce during the launch of its XML APIs at the IDG Demo 2000.

November 2000: eBay launches the eBay API, along with the eBay Developers Program. It is made available to a number of licensed eBay partners and developers.

July 2002: Amazon Web Services is launched. It allows third parties to search and display Amazon.com products in an XML format.

February 2004: Marks the beginning of the social media era, with Flickr launching its popular photo sharing site.

August 2004: Flickr launches its API, which help it to become the most preferred image platform. The Flickr API allows users to easily embed their Flickr photos into their blogs and social network streams.

June 2005: The Google Maps API launches, allowing developers to integrate Google Maps into their web sites. Today, over a million web sites use the Google Maps API, making it one of the most heavily used web application development APIs.

August 2006: Facebook launches its Developer API platform, allowing developers access to Facebook friends, photos, events, and profile information.

September 2006: Twitter introduces its APIs to the world in response to the growing usage of people scraping its web site or creating rogue APIs.

By the year 2006, web APIs are demonstrating the power of the Internet. They are being used to share content and made available to social networks. But they are still not considered fit for mainstream businesses. This year also marks the beginning of the c*loud computing* era.

> **March 2006**: Amazon S3 is launched. It provides a simple interface to retrieve and store any amount of data at anytime from anywhere on the Web.

> **September 2006**: Amazon launches EC2, also known as the Elastic Compute Cloud platform. It provides resizable compute capacity in the cloud, allowing developers to launch different sizes of virtual servers within Amazon data centers.

With cloud computing, web APIs witness their real power. APIs can now be used to deploy global infrastructure. APIs move from being used only for social fun and interaction to actually run real businesses. The emergence of mobile devices, smartphones, and app stores becomes the next big game changer.

> **March 2009**: Foursquare is launched to provide a local search-and-discovery service mobile app. It provides a personalized local search experience for its users. By taking into account the places a user goes, the things they have told the app that they like, and the other users whose advice they trust, Foursquare aims to provide highly personalized recommendations on the best places to go near the user's current location. By March 2013, the Foursquare API has more than 40,000 registered developers building a new generation of apps using of Foursquare's location-aware services.

> **June 2009**: Apple launches the iPhone 3G. iPod Touch and iPhone owners can download apps through iTunes desktop software or the App Store onto their devices. The APIs emerge as the driving force for the growth of the app economy.

> **October 2010**: Instagram launched its photo-sharing iPhone app.

By 2012—after the introduction of powerful smartphones, iPads, tablets, and the growth of Android and Windows Mobile, the need for APIs to provide resources to build apps has grown exponentially. Mobile is the last piece in the digital strategy puzzle, which includes ecommerce, social media, and the cloud. The growth of Android smartphones and iPhones complemented the growth of digital technology, and APIs grew beyond powering ecommerce, social media, and the cloud to delivering valuable resources to the average person via smartphones. APIs make valuable resources modular, portable, and distributed. They have become the perfect channel for building apps for mobile devices, tablets, and handheld devices. Today, the success of the digital strategy for any company depends on the use of SMAC (social, mobile, analytics, and cloud) technologies—all powered by APIs.

APIs Are Different from Web Sites

Web sites publish information that can be consumed by a user, but web sites do not have contracts. The layout, content, and the look and feel of a web site can change without prior notice to users. There is no contract around a web site's structure and content. When a web site changes its content, visitors see the update; perhaps it has a new look and feel. When a web site is dramatically redesigned, the only impact is users getting accustomed to the new layout. Users might initially find it difficult to find their favorite information at a particular place or in a particular form, but most get used to the changes over time.

An API, on the other hand, has a well-defined contract. Other applications depend on this contract to use it. Unlike humans, programs are not flexible. So if the contract of the API changes, there is a ripple effect on the apps built using the contract. The effect could be potentially large. This does not mean that an API cannot change. Changes necessary to meet evolving business needs are inevitable. Changes could be in the business logic, or the back-end infrastructure, or to the interface defining the API contract. Changes to the implementation or to the infrastructure do not necessarily require changes to the API interface. Such changes can happen frequently. However, any change to the API interface will impact the applications using them, and hence, should be versioned and backward compatible.

Defining an API and Its Characteristics

In technical terms, an API *defines the contract of a software component in terms of the protocol, data format, and the endpoint for two computer applications to communicate with each other over a network*. In simple terms, APIs are a set of requirements that govern how two applications can talk to each other.

An API provides a framework for building services that can be consumed over HTTP by a wide range of clients running on different platforms, such as iPhone, tablets, smartphones, browsers, kiosks, connected cars, and so forth. These applications can be web applications or apps running on devices.

An API provider should provide the following information about the API:

- The functionality provided.
- The location where the API can be accessed. An HTTP URL is normally used to specify the location.
- The input and output parameters for the API, such as parameter names, message format, and data types.
- The service-level agreement (SLA) that the API provider adheres to—such as response time, throughput, availability, and so forth.
- The technical requirements about the rate limits that control the number of requests that an app or user can make within a given period.

- Any legal or business constraints on using the API. This can include commercial licensing terms, branding requirements, fees and payments for use, and so on.
- Documentation to aid the understanding of the API.

Optionally, the API provider may provide the following to aid developers in building and monitoring their apps:

- A portal on which developers can register themselves and their apps before they start using the APIs.
- Example programs and tutorials for using the APIs.
- A developer community forum and blogs to support developers and help them collaborate.
- Tools to expose and test the APIs.
- Health and usage information on the APIs used by developer apps.

Types of APIs

Broadly classifying, APIs can be divided into two types: *public APIs* and *private APIs* (see Figure 1-2). Going by name, public APIs are open to all for use. Private APIs, on the other hand, are accessible only by a restricted group. Private APIs may be for B2B partner integrations or for internal use. Those used for partner integration are also known as *partner APIs*. Those for internal use are referred to as *internal APIs*. An internal API can ease and streamline internal application integrations. It can also be used by internal developers for building mobile apps for an organization's own use.

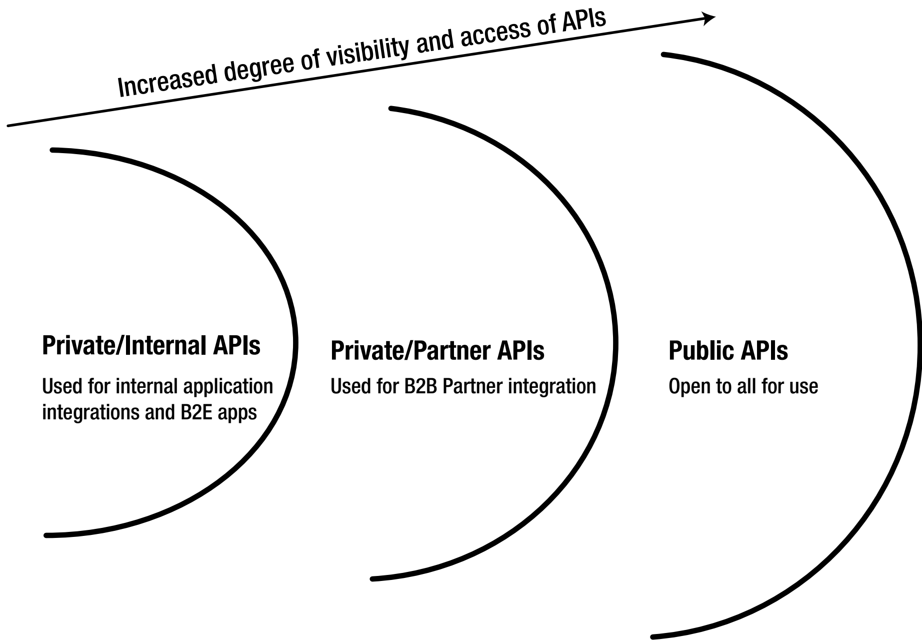

Figure 1-2. *Types of APIs*

The interface of a public API is designed to be accessible by a wider developer community for building mobile and web apps. Public APIs can be accessed by internal app developers within an organization, as well as the outside developer community that wants to build apps using them. By being open to a wider audience of app developers, public APIs can help an organization to add value to its core business through innovation. Open developers use their imagination to build cool apps using public APIs. Public APIs also help increase the use of company assets and add business value without direct investment in app development. Public APIs can help generate new business ideas and decrease development costs. The success of a public API depends on its ability to attract developers and help them create truly great apps. A well-designed, well documented, clean, and intuitive interface helps developers quickly understand the functionality of an API and how to use it.

However, public APIs can significantly add a lot of management overhead. For example, when a lot of third-party apps are actively using an API, it is challenging to upgrade the interface without impacting the apps that are in production.

Increased security risks are another major challenge for public APIs. Since public APIs expose the back-end systems of an organization through the enterprise firewall that can be accessed by all, they are open doors for hackers to intrude into the system. Hence, when an enterprise uses public APIs, they need to build in additional layers of security to protect their systems from hacker attacks via these APIs.

Private APIs are behind the closed doors of your organization. They are mostly intended for internal app integration or B2B integration with partners, or for developing mobile and web apps for internal consumption. Every enterprise developing a public API probably first developed a private API. Be it Facebook, or Twitter, or Google, or any enterprise—their public APIs, web sites, and mobile apps are all powered by their private APIs behind the scenes. The visible public APIs are only the tip of the iceberg. Private APIs form the large underwater mass of the iceberg. Most of these APIs are private and internal to companies, used exclusively by their own developers or by partners with contractual agreements. These APIs are not exposed to the external developer community but are actually driving the entire API economy. Sometimes the internal use of a company's private APIs for business transformation can derive more business benefits than public APIs. Hence, the importance of building private APIs should never be underestimated.

How do you make an API private? One simple way is to host it on a public network but not publicize its existence and documentation to the developer community. This approach can work initially, but can lead to problems in the future. Developers have a habit of trying out uncanny things and could accidentally discover your unpublicized, private API—and then start using it for app development. If the app becomes popular and then the API publisher decides to modify or retire their private API, it can lead to public outcry. A better approach is to provide security and access control to your APIs and restrict their use to a limited set of known developers and partners. Approaches to secure your APIs are discussed later in this book.

Examples of Popular APIs

The history of web APIs dates back to 2005. Since then, the growth in the number of APIs is exponential. ProgrammableWeb maintains a repository of public APIs and has more than 13,000 APIs under different categories. The number of private APIs is estimated to be more than 10 to 15 times greater than this. Some of the most popular APIs are by Facebook, Google, Twitter, Flickr, and Instagram—to name a few. The following list provides an overview of some popular APIs.

- **Facebook APIs** provide a platform for building applications that can be used by a member of the Facebook community. Developers can build more engaging and interesting applications using the social connections and profile information provided by these APIs. Facebook APIs can be used by other third-party applications to publish activities to the newsfeed and profile pages of Facebook—subject to an individual user's privacy settings. The API uses the RESTful protocol and the responses are in JSON format. The Facebook API home page is at `https://developers.facebook.com`.

- **Google APIs** allow communication with Google services, such as Search, Translate, Gmail, Maps, social, and advertising. These APIs can be used by developers to build apps that extend the functionality of existing services. The Google+ APIs for user registration and login are used to include a "Sign in with Google" button in Android apps. This helps to improve the user experience, because manually typing login credentials on a small screen is time-consuming. Since a user is usually signed into her Google account on her mobile device, signing in/signing up for a new Google service is usually only a matter of a few button clicks. The Google Maps APIs can embed Google maps using a JavaScript or a Flash interface in a variety of applications. For example, Uber uses Google Maps APIs for its app. Developers can build collaborative apps for document editing or picture/video editing through Google's Drive API. Custom Search APIs can provide a search within a web site. The Google API home page is at `https://developers.google.com`.
- **Yelp APIs** provide rich content about local businesses around the world. These APIs can enhance an app with a Yelp rating, reviews, photos, and much more. The API uses the RESTful protocol and the responses are in JSON format. These APIs are protected using a secure authentication protocol. The Yelp API home page is at `https://www.yelp.com/developers/`.
- **AccuWeather APIs** provide subscribers with access to location-based weather data via a simple RESTful web interface. These APIs provide current weather conditions, forecasts, severe weather alerts, and much more. The AccuWeather API home page is at `https://api.accuweather.com/developers/`.
- The **Flickr API** is used to build applications for sharing, editing, and managing photos on Flickr. It consists of a set of callable methods and some API endpoints. The API uses a RESTful protocol and the responses are in XML and JSON format. The API homepage is at `https://www.flickr.com/services/api`.
- **Instagram APIs** allow you to get photos from Instagram and display them on your own web site or app. The Instagram API console is on the home page at `https://www.instagram.com/developer/`.
- **Twitter** provides three types of APIs: REST APIs, search APIs, and streaming APIs. The REST APIs provide programmatic access to read and write core data about individual Twitter users, their timelines, and status updates. The search APIs help retrieve tweets with specific filters. The streaming APIs continuously deliver new responses to REST API queries over a long-lived HTTP connection. It helps receive updates on the latest tweets matching a search query. The Twitter API homepage is at `https://dev.twitter.com`.

- The **YouTube API**, which is part of the Google API offering, lets developers integrate YouTube videos and functionality into web sites or applications. YouTube APIs include the YouTube Analytics API, the YouTube Data API, the YouTube Live Streaming API, the YouTube Player API, and others. The YouTube API homepage is at `https://developers.google.com/youtube/`.
- **Amazon** provides APIs for In-App Purchasing, Mobile Ads, and Mobile Accessories. It also offers a host of other engaging services, such as push notifications to send targeted messages to devices running the app, to sync game data across devices and platforms to improve the player experience, and retention and login with Amazon to provide a personalized user experience. Building an app using these APIs can help monetize the app. More information about the Amazon APIs and its developer program can be found at `https://developer.amazon.com/`.
- **AT&T** provides a wide range of APIs that expose their internal assets and services. These APIs can be used to build apps that can send messages (SMS/MMS), locate users, do text-to-speech and speech-to-text conversion, monetize apps through embedded advertisements, use M2X capabilities, and much more. For a detailed list of available APIs, visit the AT&T developer home page at `http://developer.att.com/apis`.

The Difference Between a Web Service and a Web API

Wikipedia defines a *web service* as "a method of communication between two electronic devices over a network". It is a software function provided at a network address over the Web, with the service always on—as in the concept of utility computing. The W3C defines a *web service* generally as "a software system designed to support interoperable machine-to-machine interaction over a network".

Going by these definitions, a web API can be considered as a subset of a web service. The W3C Web Services Architecture Working Group states that a *web service architecture* requires specific implementation of a web service. In this, a web service "has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the web service in a manner prescribed by its description using SOAP (Simple Object Access Protocol) messages, typically conveyed using HTTP with an XML serialization in conjunction with other web-related standards".

SOAP web services typically use HTTP as a transport protocol, although this is not mandatory. SOAP can be over JMS/FTP/SMTP or any layer 7 protocol. The SOAP message structure consists of an SOAP envelope, inside of which are the SOAP headers and the SOAP body. The SOAP body contains the actual information we want to send. It is based on the standard XML format, designed especially to transport and store structured data. SOAP is a mature standard and is heavily used in many systems, but it does not use many of the functionalities built into HTTP.

A web API is a special kind of web service, where the emphasis has been moving to a simpler **RE**representational **S**tate transfer (REST)-based communications. RESTful APIs do not need XML-based web service protocols like SOAP and WSDL to support their interfaces. REST is another architectural pattern (resource-oriented), an alternative to SOAP. Unlike SOAP, RESTful applications use the HTTP built-in headers (with a variety of media types) to carry meta-information and use the GET, POST, PUT, and DELETE verbs to perform CRUD operations. REST is resource-oriented and uses clean URLs (or RESTful URLs). The body of can be JSON or XML, the former being preferred more due to its simple structure. Later in this book, we look into the principles of RESTful APIs.

So far web services have been synonymous to SOAP web services. With the advent of REST, web APIs have been commonly referred to as RESTful web services. SOAP is preferred for service interactions within enterprises. REST, on the other hand, is the choice for services that are exposed, such as public APIs using HTTP(s).

In terms of performance, SOAP-based web services are heavyweight, requiring additional processing of extra SOAP elements in the payload. REST-based web services are simpler with lightweight request and responses in JSON format, which provides a performance advantage and reduced network traffic. RESTful services have better cache support and are preferred for mobile and web apps. Since JSON is lighter, apps run faster and more smoothly.

How Are APIs Different from SOA?

Many often ask what the difference between APIs and SOA is. Most enterprises are already using SOA. Are APIs still needed? If yes, why? Then what is the real difference between the two? There is a lot of confusion about whether APIs are different, or similar to SOA. Let’s look at their characteristics to understand it better.

SOA stands for *service-oriented architecture*. Its core concept is the notion of service. A service can be defined as “a logical representation of a repeatable activity that has a specific outcome.” Service-oriented architecture defines the architecture and principles for designing services for an application to increase its reuse. Services are well contained and have a well-defined interface that defines the contract between the service provider and the consumer.

From a technical perspective, APIs also share the same characteristics. But they are more open, developer centric, easily consumable, and support human-readable formats, such as JSON. APIs are designed with consumer needs in mind. What makes APIs different from SOA is the objective behind them: SOA helps in the agility and pace of the delivery of a service, whereas APIs help in the pace of innovation for building apps. SOA emerged as a means to shield service consumers from back-end changes. With the growing needs of omnichannel front-end application channels, there is also a need to protect these services. APIs can provide a layer to shield the services from the rapidly changing demands of front-end apps. With APIs and SOA together, you can create a calm eye in the middle of the hurricane of change.

Services are the means by which providers codify the base capabilities of their domains. APIs are the way in which those capabilities are repackaged, productized, and shared in an easy-to-use format. In that fashion, APIs and services are complementary rather than contradictory, and applied together, dramatically increase the overall effectiveness of enterprise innovation.

At a technology level, SOA is related to XML and SOAP, whereas APIs are related to REST and JSON. SOA services are described using WSDL, whereas APIs are described using Swagger or RAML. SOA services are normally published in an UDDI registry that is internal to the organization. APIs are published by an API provider in a portal that is normally used by developers for onboarding and finding information about the APIs.

Keeping the technical differences aside, the real difference between SOA and APIs center on scope and governance. SOA is more focused on building reusable enterprise services that enable integration within the enterprise. It provides controlled access to the services for trusted and well-known partners; whereas APIs open services for developers to access them on the public internet using REST principles. APIs are managed as a product that app developers can consume. RESTful design, a JSON data format, and a simple versioning approach complemented with well-documented and human-readable interface, makes it easier for developers to adopt and consume APIs.

API technology focuses on the consumption of the back-end services created using SOA principles. Hence, APIs can be thought of as an evolution of SOA, imbibing a lot of the same concepts and principles of creating and exposing reusable services. The main difference between them is that APIs are focused more on making consumption easier, whereas SOA is focused on control and has an extensive and well-defined description language (see Figure 1-3).

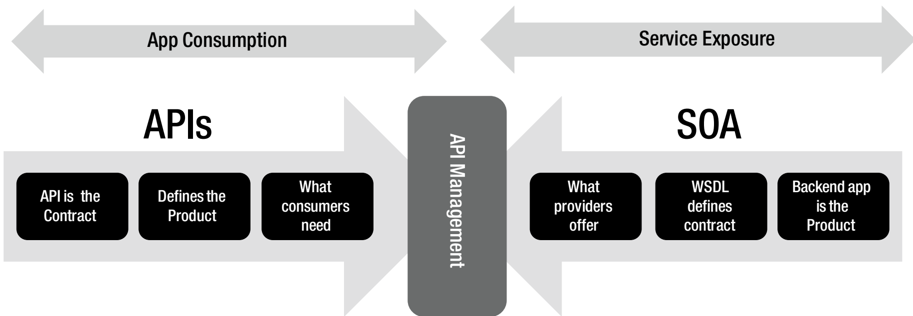

Figure 1-3. *APIs vs. SOA*

APIs provide an agile, flexible, and robust approach to building mobile apps. SOA cannot provide the agility and flexibility required to meet growing customer demands. SOA does not match the preferred design for today's mobile apps. API management has become a necessary component to build, manage, and scale apps for the digital economy. With the help of an API tier to connect your systems of record to your systems of engagement, you can extend your SOA capabilities to match the data requirements of a digital economy.

From a governance perspective, SOA is managed through a governance model that is more formal, heavyweight, and prescriptive in nature. Data schemas and interface specifications have been a strong focus of SOAP services. Any change in the data type for the SOA services has to go through rigorous governance approval. This makes SOA slow. API initiatives, on the other hand, are more agile and focused on developer adoption and usage. The success of an API is measured by the agility that it offers to application delivery.

The API Value Chain

APIs provide a means to expose business assets to the end user. To understand the API value chain, you need to understand what is happening when an API is being advanced by a business and identify the actors involved at each step (see Figure 1-4).

Figure 1-4. *API value chain*

The *business asset* marks the beginning of the API value chain. The business identifies the asset and its value and decides to make it available for others to use. The business asset can be any data or business functionality. It can range from product catalogs, to customer information, to Twitter feeds, to postal tracking information, to payment and banking services. The value derived through the use of the asset depends on multiple factors. The following questions might help understand the value of the asset:

- What business asset is being exposed as an API and what is the value to its owner?
- What benefits would the provider get by creating a channel for using the assets via API?
- Who are the potential users of the asset and how would the end users get access to the assets?
- What benefits would the end user get by the using the asset? Of what potential value could these assets be to the others?
- How easily can the end user access and use it?

The value of the asset determines the success of the API. Exposing the assets to others should also benefit the owner.

Once an asset has been identified, the next step is to create an API to expose the business assets. The **API provider**'s job is to design the API so that it can be used easily by the intended audience. In most cases, the asset owners are themselves the API provider. In this case, the benefits of the API flow directly to the asset owner. But in some cases, the owner may have an agreement with another organization to create APIs to expose it assets. In such cases, the rewards get distributed between the asset owner and the API provider.

The **app developers** then assess the APIs and create apps using them. Developers can be an individual entity or a group belonging to an organization. If they belong to an organization, they are sometimes referred to as *company developers*.

The **apps** created by the developers can be mobile apps or web apps. These apps should be made available to the end user in order to add value to the business. An app store is the most popular channel for distribution. But there may also be other channels for distribution and marketing. Apps can be either freely downloadable or paid.

The **end users** are the final actor in the API value chain. They are the users of the app. They can use the app on their mobile devices, smartphones, tablets, iPhones, or desktops, or from other connected devices, such as connected cars, kiosks, and so forth.

The success of the API strategy depends on the various links in the API value chain. It depends on the involvement and commitment of the key stakeholders in the value chain. It is important to get them all involved for the success of your API. There has to be a proper handshake among all the stakeholders. The API provider needs to understand the value of the business asset and decide on the best interface to expose it. The developer has to understand the business asset and its interface, and create an app that meets the needs of the end user and adds value for them. All the stakeholders should understand the core business needs and the value for creating the API. The app built using the API should be easy to use, and its purpose and value should be easily understood by the average person. Only then can the API strategy be successful.

Business Models for APIs

APIs form the foundation of digital business. The business model to adopt depends on the asset being exposed as an API. The asset can be the *data*, the *business logic*, or the *presentation*. Some of the business drivers for building APIs include (but are not limited to) the following:

- Growing new business capabilities and opportunities
- Opening new marketing channels and lines of business
- Improving customer reach and loyalty
- Innovating at the edge of business
- Accelerating time to market
- Advancing operational efficiency and control
- Driving traffic and accelerate internal projects

As APIs help to drive business agility, growth and open new channels for revenue, there are many business models for API exposure. The model to choose from depends on the business goals of the API provider. Depending on the goals, a provider may choose to adopt an available API business model. The business model can be free, developer pays, and developer gets paid or indirect. Details on the various monetization models are discussed later in the book.

CHAPTER 2

API Management

Customers today want to have access to enterprise data and services through a variety of digital devices and channels. To meet customer expectations, enterprises need to open their assets in an agile, flexible, secure, and scalable manner. APIs form the window into an enterprise's data and services. They allow applications to easily communicate with each other using a lightweight protocol like HTTP. Developers use APIs to write applications that interact with the back-end system. Once an API has been created, it needs to be managed using an *API management platform*. An API management platform helps an organization publish APIs to internal, partner, and external developers to unlock the unique potential of their assets. It provides the core capabilities to ensure a successful API program through developer engagement, business insights, analytics, security, and protection. An API management platform helps business accelerate outreach across digital channels, drive partner adoption, monetize digital assets, and provide analytics to optimize investments in digital transformation (see Figure 2-1).

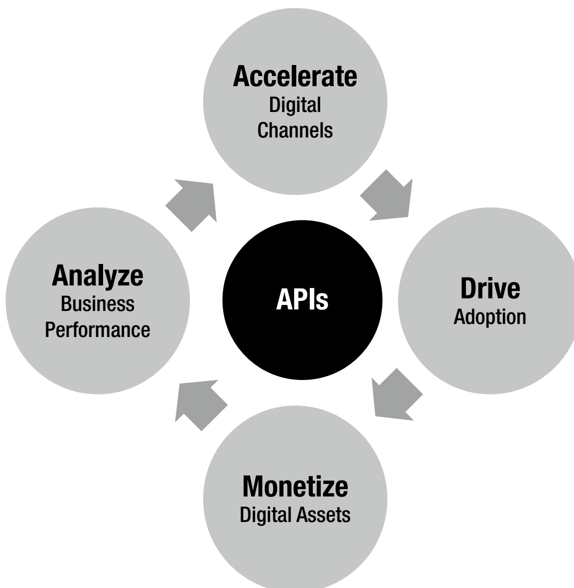

Figure 2-1. *API management offerings*

© Brajesh De 2017
B. De, *API Management*, DOI 10.1007/978-1-4842-1305-6_2

An API management platform enables you to create, analyze, and manage APIs in a secure and scalable environment (see Figure 2-2). An API management platform should provide the following capabilities:

- Developer Enablement for APIs
- Secure, Reliable and Flexible Communications
- API lifecycle Management
- API Auditing, Logging and Analytics

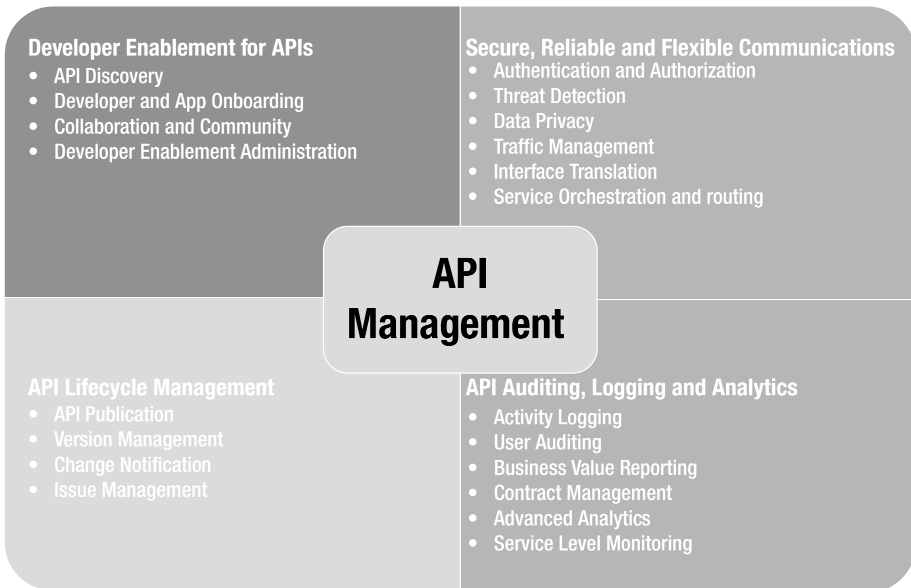

Figure 2-2. *API management capabilities*

API management capabilities can be delivered by any API management vendor in a public cloud as a hosted service or can be deployed on-premise in a private cloud. A hybrid approach can also be followed, with some components of the API management platform being offered as a hosted solution and others deployed on-premise for increased security and control.

An API management platform provides these capabilities as three major types of services (and as illustrated in Figure 2-3):

- **API gateway services** allow you to create and manage APIs from existing data and services. They allow you to add security, traffic management, interface translation, orchestration, and routing capabilities into your API.

- **Analytics services** monitor traffic from individual apps and provide business with insight and operational metrics, API and app performance, and developer engagement metrics.
- **Developer portals** provide capabilities for developer and app registration and onboarding, API documentation, community management, and API monetization.

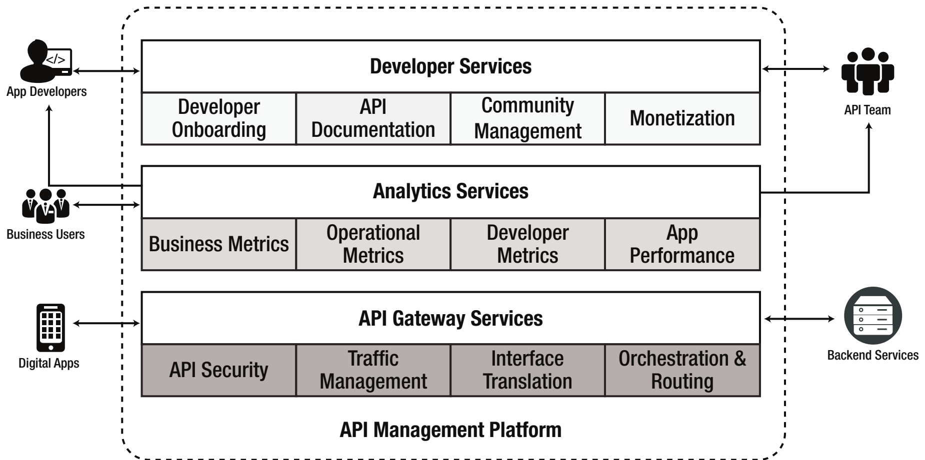

Figure 2-3. *API management platform services*

This chapter introduces you to the different capabilities required for an API management platform and shows how the different services provided by the platform help enable these capabilities. In the process, it also introduces the various concepts and technologies for API management.

Secure, Reliable, and Flexible Communication

APIs help digital apps to communicate with back-end services. Communication forms the core of APIs. Communication can use REST, SOAP, Plain Old XML (POX), or any other protocol of choice. REST is by far the most preferred communication protocol for APIs due to its inherent characteristics, which are described later in this book. An API management platform must provide a framework that allows secure, reliable, and flexible channels of communication. The *API gateway* within the API management platform provides the services that form the core capabilities required for API communications.

The API Gateway

An API gateway forms the heart of any API management solution that enables secure, flexible, and reliable communication between the back-end services and digital apps (see Figure 2-4). It helps to expose, secure, and manage back-end data and services as RESTful APIs. It provides a framework to create a facade in front of the back-end services. This facade intercepts the API requests to enforce security, validate data, transform messages, throttle traffic, and finally route it to the back-end service. The static response may be cached to improve the performance. The API gateway can optionally orchestrate requests between multiple back-end services and also connect to databases to service the request. All of these functionalities can be implemented in a gateway, mostly through configurations and scripting extensions.

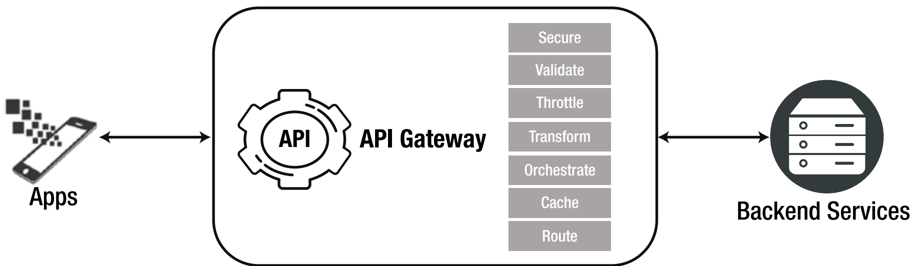

Figure 2-4. *API Gateway capabilities*

The main features of an API gateway include—but are not limited to—the following.

API Security

APIs provide access to valuable and protected data and assets. Therefore, security for APIs is of utmost importance to protect the underlying assets from unauthenticated and unauthorized access. Due to the programmatic nature of APIs and their accessibility over the public cloud, they are also prone to a different kind of threat attack. The API management platform should therefore address the following aspects of API security.

- **Authentication**: Authentication is the process of uniquely determining and validating the identity of a client. An *app* acts like a client making an API call. It is a piece of software that consumes an API to get access to enterprise assets, data, and services. It can run on the Internet, a computer, smartphones, tablets, or any other electronic device. Apps are usually made available by their developers through a distribution platform, such as Apple's App Store, or Google Play, or the Windows Phone Store. Every app is identified by its *name* and a unique UUID known as the *app key*. The app key often serves as an identity for the app making a call to the API. It is normally issued and managed via the API management platform of the API provider. An app key is also known as an *API key*, an *app ID*, or a *client ID*. The API management platform must have the ability to issue, track, and revoke the app key. Authentication services may also require integration with identity management systems that control user access to applications and other services.
- **Authorization**: Authorization controls the level of access that is provided to an app making an API call. It controls which API resources and methods that an app can invoke. When an app makes an API call, it normally passes an OAuth *access token* in the HTTP headers. This token is generated as part of the OAuth handshake and is associated with *scopes* that determine the APIs that can be accessed using the token. An access token can be associated with one or multiple scopes. Each access token may have an expiry duration that controls the duration for which the token is valid. If the token is expired, a new access token would be required to be generated. An app can do this automatically by presenting a *refresh token*. The refresh token may be exchanged to get a new access token with a renewed validity period. The use of a refresh token by an app to regenerate the access token helps to improve the overall user experience.
- **Identity mediation**: APIs normally use OAuth protocols for implementing security. However, the back-end services may be secured using SAML or any other WS-Security headers. Hence, the API management platform must have the capability to integrate with back-end IDM platforms and do identity mediation. OAuth to SAML is a very common identity mediation requirement.

- **Data privacy**: APIs expose data that may be sensitive; such data should be visible only to its intended recipient. Any sensitive data in transit should be encrypted. If such data gets logged anywhere, it must be masked. The API management platform must therefore possess data privacy capabilities. Data privacy can be achieved through encryption and data masking. Sensitive data should be encrypted with digital certificates in transit. The API management platform should have support for SSL/TLS. For some use cases, additional encryption of specific elements within the payload may also be required. Masking sensitive data at rest within audits and log files is yet another data privacy requirement that an API management platform should provide.
- **Key and certificate management:** The API management platform should also provide the capability to manage keys and certificates required for data privacy.
- **DoS protection**: APIs open valuable data and assets outside the firewalls of the enterprise. This increases the attack surface and makes them more prone to attacks. Hackers may try to bring down back-end systems by pumping unexpectedly high traffic through the APIs. Denial-of-service (DoS) attacks are very common on APIs. Hence, the API management platform should be able to detect and stop such attacks.
- **Threat detection**: For public APIs, the likelihood of bad actors making attacks using malicious content is high. Content-based attacks can be in the form of malformed XML or JSON, malicious scripts, or SQL within the payload. Such attacks can also happen to private and enterprise APIs. The API management platform should be able to identify malformed request formats or malicious content within the payload and then protect against such attacks. Error visualization capability can also help detect any hacker attempting to find an exploitable weakness in APIs.

API Traffic Management

Depending on the nature of data and services provided by the API, traffic management offers a different business value to different classes of customers. Each customer class may be willing to pay differently for access. For example, some app developers prefer to try out APIs for free. The API provider may provision such users to make a small number of API calls in a day/week/month. Paying customers, however, want access to a higher or an unlimited number of API calls. Again, the API provider may allow customers a different level of access depending on their location or the time of the day; for example, internal enterprise users may get unlimited access to a high-performing API, whereas public Internet users may have limited access. More API calls may be allowed during off-peak hours but there is a limited number allowed during peak business hours.
The API provider may have different requirements to throttle and manage the API traffic. The API platform should provide the following capabilities for traffic management.

- **Consumption quota**: Defines the number of API calls that an app is allowed to make to the back end over a given time interval. Calls exceeding the quota limit may be throttled or halted. The quota allowed for an app depends on the business policy and monetization model of the API. A common purpose for a quota is to divide developers into categories, each of which has a different quota and thus a different relationship with the API. For example, free developers who sign up might be allowed to make a small number of calls. But paid developers (after their verification) might be allowed to make a higher number of calls.
- **Spike arrest:** Identifies an unexpected rise in the API traffic. It helps to protect back-end systems that are not designed to handle a high load. API traffic volume exceeding the spike arrest limit may be dropped by the API management platform to protect back-end systems in the event of DoS attacks.
- **Usage throttling:** Provides a mechanism to slow down subsequent API calls. This can help to improve the overall performance and reduce impacts during peak hours. It helps to ensure that the API infrastructure is not slowed down by high volumes of requests from a certain group of customers or apps.
- **Traffic prioritization:** Helps the API management platform determine which class of customers should be given higher priority. API calls from high-priority customers should be processed first. Not all API management platforms support this capability. Hence, an alternative approach or design may be required to implement traffic prioritization.

Interface Translation

When an enterprise creates an API to expose its data and services, it needs to ensure that the API interface is intuitive enough for developers to easily use. APIs should be created with an API-First approach, which promotes API creation with a consumer focus. Hence, the interface for the API will most likely be different from that of the back-end services that it exposes. The API gateway should therefore be able to transform the API interface to a form that the back end can understand. To support interface translation, the API gateway should support the following:

- **Format translation:** The back-end system might expect data in SOAP, or XML, or CSV or any other proprietary format. Such data format cannot be easily consumed by the API consumer. Hence, the API gateway should have the capability to easily transform from one format to other. Most API management platforms provide the capability to transform data from XML to JSON (and vice versa) with a one-to-one mapping of the data elements. Mapping from JSON to any other data format may be supported through customization.

- **Protocol translation:** Most back-end systems that host services provide a SOAP interface for consumers. However, SOAP is not a protocol that is suitable for APIs to build apps for digital devices. API management platforms must be able to do a protocol transformation from SOAP to REST to provide a lightweight interface for consumers. Support for other protocol transformations—like HTTP(s) to JMS/FTP/JDBC—may be a nice to have feature in the API management platform.
- **Service and data mapping:** An API management platform should provide a graphical representation of the different back-end service component that maps to provide an API service. It should incorporate service mapping tools that enable the discovery and description of existing service delivery assets so that they can be wired into your API design.

Caching

Caching is a mechanism to optimize performance by responding to requests with static responses stored in-memory. An API proxy can store back-end responses that do not change frequently in memory. As apps make requests on the same URI, the cached response can be used to respond instead of forwarding those requests to the back-end server. Thus caching can help to improve an API's performance through reduced latency and network traffic.

Similarly, some static data required for request processing may also be stored in-memory. Instead of referring to the main data source each time, such data can be retrieved from the cache for processing the request. An expiry date/time can be set for the cached data or the data can be invalidated based on defined business rules. If the data is expired, new data would be retrieved from the original data source and the cache would be refreshed with the updated data.

Service Routing

APIs need to route requests from consumers to the right back-end service providing the business functionality. There may be one more backend systems providing the backend functionality. Hence, the API management platform should be able to identify and route the request to the correct instance of the back-end. The API management platform should support the following routing capabilities:

- **URL mapping**: The path of the incoming URL may be different from that of the back-end service. A URL mapping capability allows the platform to change the path in the incoming URL to that of the back-end service. This URL mapping happens at runtime so that the requested resource is retrieved by the consumer via service dispatching.

- **Service dispatching:** This allows the API management platform to select and invoke the right back-end service. In some cases, multiple services may have to be invoked to perform some sort of orchestration and return an aggregated response to the consumer.
- **Connection pooling**: The API management platform should be able to maintain a pool of connections to the back-end service. Connection pooling improves overall performance. Also, it may be required for traffic management purposes to ensure that only a fixed maximum number of active connections are opened at any point in time to the back-end service.
- **Load balancing**: Load balancing helps to distribute API traffic to the back-end services. Various load balancing algorithms may be supported. Based on the selected algorithm, the requests must be routed to the appropriate resource that is hosting the service. Load balancing capabilities also improve the overall performance of an API.

Service Orchestration

In many scenarios, the API gateway may need to invoke multiple back-end services in a particular sequence or in parallel and then send an aggregated response to the client. This is known as *service orchestration*. The service orchestration capability helps to create a coarse-grained service by combining the results of multiple back-end services invocation. This helps to improve overall performance of the client by reducing latency introduced due to multiple API calls. Service orchestration capability may require the API gateway to maintain states in-between the API calls. However, the API gateway should be kept as light and stateless as possible. Hence, it is recommended that the API gateway only be involved in the orchestration of read-only services that are non-transactional in nature.

API Auditing, Logging and Analytics

Businesses need to have insight into the API program to justify and make the right investments to build the right APIs. They need to understand how an APIs is used, know who is using it, and see the value generated from it. With proper insight, business can then make decisions on how to enhance the business value either by changing the API or by enriching it. An API gateway should provide the capability to measure, monitor, and report API usage analytics. Good business-friendly dashboards for API analytics measure and improve business value. A monetization report on API usage measure business value; hence, it is yet another desirable feature on an API management platform.

API Analytics

Analytics provide you with information to make future decisions about your API. When you see an increase in API traffic, you need to know whether this indicates the success of your API program or whether it is being used in a malicious way, resulting in inflated traffic. How do you determine the adoption of your API? Is there an increased interest in your APIs within the developer community? Is there an increase in the number of apps built using your APIs? How has the performance of the APIs been in terms of response time and throughput? What are the different kinds of devices being used to access the APIs? How have the APIs been adopted across the globe? As an API provider and consumer, you need to know the answer to these questions and many others. The more you know, the better you are able to determine what's going on. You need metrics to decide which features should be added to your API program. API analytics is the answer to all queries.

The API management platform should be able provide the following capabilities required for analytics.

Activity Logging

Activity logging provides basic logging of API access, consumption, performance, and any exceptions. The platform should capture and provide information on who is using an API, what types of apps and devices the API are being called from, and which geographical region is the source of the API traffic. It should log the IP address of the clients, as well as the date and time when a request was received and the response was sent. The gateway within the API management platform should log which API and method is being invoked by the client. Various metainformation, such as URI, HTTP verb, API proxy, developer app, and other information can be logged into the gateway for every API call. The platform can process this information at a later time to provide meaningful reports for API analysis. API performance metrics and response/error codes should also be logged as part of activity logging.

User Auditing

User auditing can help the API administrator review historical information to analyze who accesses an API, when it is accessed, how it is used, and how many calls are made from the various consumers of the API.

Business Value Reports

Business value reports gauge the monetary value associated with the API program. Monetization reports of API usage provide information on the revenue generated from the API. The API gateway should be able to provide API usage monetization reports. Some APIs may be directly monetized, but many have an indirect model for monetization. Hence, additional value-based reporting should also be possible within an API management platform to measure customer engagements. Engagements can be measured by the number of unique users, the number of developers registered, the number of active developers, the number of apps built using the APIs, the number of active apps, and many other items.

Advanced Analytics

The API management platform should be able to extract and log custom variables from within the message payload for advanced analytics reporting. It should provide API administrators and product managers the capability to create pluggable and custom reports from the captured information.

Service-level Monitoring

The API management platform should provide performance statistics that track the latency within the platform and the latency for back-end calls. This helps the API administrator find the source of any performance issues reported on any API. The platform should have the capability to provide reports on errors raised during the processing of the API traffic within the platform, or ones that are received from the back end. Classifying the errors by type, frequency, and severity gives API administrators a valuable aid for troubleshooting.

Developer Enablement for APIs

An API program cannot be successful without the active involvement of a developer community. Application developers use APIs to build mobile apps or to build a custom integration between two or more applications. Hence, developers need to know which APIs are available, what their functionalities are, and how they can be used. Developers should have a playground to experience and test APIs to effectively use them in their applications. An API management platform should provide services that enable developers to build apps using the APIs. A developer portal can provide these services.

Developer Portal

A *developer portal* is a customized web site that allows an API provider to provide services to the developer community. It is essentially a content-management system that documents the APIs—their functionalities, interfaces, getting-started guides, terms of use, and much more. Developers can sign up through the portal and register their applications to use the APIs. The can interact with other developers in the community through blogs and threaded forums. The portal can also be used to configure and control the monetization of the APIs. Monetization gives developers self-service access to billing and reports, catalogs and plans, and monetization-specific settings.

An API management platform developer portal should include the capabilities described in the following sections.

API Catalog and Documentation

As an API provider, you need a platform to publicize and document your APIs. Developer enablement services should allow an API provider to publish a discoverable catalog of APIs. An API catalog is also sometimes referred to as an *API registry*. Developers should

be able to search the catalog based on various metadata and tags. The catalog should document the API functionality, its interface, how-to guides, terms of use, reference documents, and so forth. Information about the API versions available should also be included in the documentation.

Developer Support

Properly designed REST APIs are normally very intuitive for developers to understand. App developers can easily start using them for app development. Still, the API provider should provide resources that developers can use to build innovative apps. Good API documentation and accelerators in the form of test and development kits can help speed up the adoption of APIs. API documentation should not only describe the API interface, but must also provide how-to guides for interacting with the APIs. The developer portal can provide embedded test consoles that developers can use to play with an API and get a feel for it. Sample code that demonstrates the use of APIs can act as a quick start guide and be very helpful to app developers. App developers often look for device-specific libraries to interact with the services exposed by the APIs, such as downloadable SDKs within the developer portal.

Developer Onboarding

To start consuming the APIs, developers must register with the API provider to get access credentials. Developers can either sign up independently or as part of a company. The signup process should be simple and easy. Developers should be able to go through a self-registration process and view the APIs available from the API provider. Developers can then select an API product and register their apps to use it. After successful registration and approval, an API key is generated along with a secret to uniquely identify the app. The API key is also referred to as an app key or a client ID. The approval process may be automatic or manual, based on the terms and conditions and the monetization model setup. In a manual approval, a member of the API management team approves the registration request. The API key is generated only after successful approval of the app. In some cases, developers may form part of a company. In such scenarios, a key management capability is important so that API consumers can add, modify, or revoke the API keys within their organization.

Community Management

App developers often like to know the views of other developers in the community. They may want to collaborate and share their API usage learnings and experiences with one another. Blogs and forums form a major part of collaboration and community management. Developers may share their experiences with API usage via blog posts; such posts may need to be moderated by the API provider before they become visible to everyone. An API provider may also create a blog to share updates and future plans with the API consumer community. Advice and best practices on API usage may also be shared on blogs and discussion forums. A developer should also be able to report any issues with an API or its usage to the API provider’s support team. The developer portal may have a link to raise support tickets. Integrated blogs and forums can help build a truly dynamic community to enhance the use of the provider’s APIs.

API Lifecycle Management

API lifecycle management provides the capability to control how an API is developed and released to consumers. Published APIs can be can be used by consumers to build apps. They can report problems or raise a request for a new API feature. An API management platform should provide the following capabilities required for API lifecycle management.

API Creation

An API acts as a facade to interact with the back-end services. The API team should be able to design the REST interface for the API and create an *API proxy* to interact with the back-end services. An API proxy acts as a facade to securely expose the back-end services to its consumers. Policies attached in the flow paths of the API proxy should be able to implement security, traffic management, message translation, encryption, filtering, caching, orchestration, and routing. Once the development is complete, the API team must be able to deploy and test the API through a console. An embedded console to test APIs can be very handy and can help reduce development time. The API management platform should provide tools that enable the creation of the APIs and subsequently deploy and test them on an environment before they are published for production.

API Publication

Once an API has been created, it must be published to an environment before it can be discovered and consumed. The API management platform must therefore provide tools that can be used to migrate the APIs from lower environments and deploy to production. Once it is deployed to production, the API specifications and other details should be published in the developer portal for consumers to discover and use in their apps. In case of any incorrect deployment, the platform must provide the ability to roll back to a previously deployed version of the API.

Version Management

APIs evolve over time with newer business requirements. Hence, managing multiple versions of an API to support existing consumers is an important capability that must be provided by the API management platform. Version management should also provide the ability to deprecate and retire older versions smoothly. When an API version is marked as deprecated, the existing consumers should be notified though deprecation warnings. Deprecated APIs may continue to serve traffic from existing consumers. However, new consumers should not be able to sign up to use deprecated APIs. With proper notice and period, deprecated APIs should be retired and removed from the platform so as to avoid any maintenance overheads. The API management platform should therefore provide the capability to manage the retirement of an API.

Change Notification

Changes to an API may adversely affect its consumers. Hence, consumers must be notified of any planned changes to the API. Developers using the APIs should be made aware of any changes to the API. The API management platform must therefore provide a mechanism to notify API consumers of any API upgrades or outages. Notification can be made via email, SMS, or social media. Release notifications can provide updates about new releases and features added to the API. API consumers should be notified about planned or unplanned downtimes. An API developer portal can be used to send release and availability notifications to subscribed users.

Issue Management

The API management platform should provide API consumers with the facility to log issues found in the APIs. App developers consuming APIs must be able to report any issues or shortcomings related to their APIs. They should be able to raise support tickets and seek help regarding API usage. The issues can be reported through the developer portal. The API management platform should provide the capability to integrate defects reporting and issue management capabilities in existing systems within the enterprise.

CHAPTER 3

Designing a RESTful API Interface

REST is an architectural style. It is not any strict standard but provides certain guidelines and constraints to be followed. Roy Fielding originally described these constraints in his doctoral dissertation and coined the name *Representational State Transfer*.

REST relies on stateless, cacheable, and client-server communication protocols such as HTTP. By following the principles of REST and applying it to stateless protocols such as HTTP, developers can build API interfaces that can be used from any device or operating system. Well-designed REST APIs attract developers to build apps that use them. An API interface should be easy to understand and intuitive to the developers. Creating a well-crafted, aesthetically designed REST API is a must-have for the success of any enterprise API program. This chapter looks at the different constraints advocated by REST and how they can be used to design a truly RESTful API interface.

REST Principles

REST is a set of design principles for building scalable web services. Roy Fielding described the following six constraints in his PhD dissertation for building a RESTful architecture:

- Uniform interface
- Client-server
- Stateless
- Cache
- Layered system
- Code on demand

Let's look at each of these constraints in more detail.

© Brajesh De 2017
B. De, *API Management*, DOI 10.1007/978-1-4842-1305-6_3

Uniform Interface

A uniform interface helps to define the communication contract between client and the server. It helps to decouple the architecture. Client and server applications can be developed independently as long as they abide by the interface. The interface defines the mechanism and format for interaction—where and how the client can access a server resource. A resource URI identifies resources. Each resource has its own unique URI. However, the physical resources are themselves separate from their representation; for example, the server does not send information about the back-end database storing the product information. Instead, it sends an XML or JSON representation of a product or a collection of products to the client.

Client-Server

The client-server constraint builds a loosely coupled and scalable web architecture. As long as the client and the server follow a uniform interface, they can be developed independently, using any language or technology. The client need not be worried about the database used for the server to store data and assets. Similarly, the server need not be worried about the client implementation technologies or the user interface or user state. It helps to achieve separation of concerns and build simpler and scalable architecture.

Stateless

Statelessness is one of the key principles of a RESTful service. It dictates that a web server is not required to remember the state of the client application. All relevant contextual information should be sent by the client application in the request to the server for all its interactions. The state information can be included as part of the URI as a variable or it can be included as a query parameter, header parameter, or in the body. Once the request is processed by the server, the updated state of the resource is sent back in the response via headers and the body. If the state must span multiple requests, the responsibility of resending the state information lies with the client. This helps to reduce the burden of the server to maintain, update, and communicate the state information of each of its client, thus helping to increase the server scalability. Additionally, even load balancers do not have to worry about the session affinity for stateless systems.

Cache

Caching is yet another REST constraint that increases the scalability and overall performance of the server application. The cache may reside anywhere in the network path between the client and server. It can reside in the server, or an external location like the CDN, or inside the client application itself. By following the caching constraint, the server can specify if a particular response can be cached or not. If the response is cacheable, the server may specify the lifetime of the cached response. Based on the lifetime, the client can decide if it wants to use a cached response or make a separate request to get the live data. Caching the response data can reduce the client-perceived latency and increase the overall availability and reliability of the application.Providing a

cached response from the API layer can also reduce the load on the back-end systems, which may not have been originally designed for high loads. Well-managed caching can partially or completely eliminate some client–server interactions, further improving scalability and performance.

Layered Systems

The layered system principle enables a network intermediary to be installed between the client app and the actual back-end server. The layered system can be a proxy or a gateway that acts as a facade for the back-end system. It can be used to implement security, caching, rate limiting, load balancing, and so forth. The client never gets to know if it is connected directly to the source of the service or to an intermediary. The caching and load balancing implemented on the intermediary node can improve the scalability of the system.

Code on Demand

The code-on-demand constraint enables a web server to transfer executable programs to a client. This constraint tends to establish a technology coupling between the client and the web server. The client must be able to understand and execute the code it downloads on demand from the server. This is the only optional constraint for the REST architectural style. Examples of code-on-demand are Java applets, scripts, plug-ins, and Flash.

Designing a RESTful API

Now that you understand the fundamentals of REST principles, let's look at the various considerations for designing a REST API interface.

A uniform interface is one of the fundamental principles of the RESTful architectural style. Web components interoperate consistently within the uniform interface's four constraints, which Fielding identified as follows:

- Identification of resources
- Manipulation of resources through representation
- Self-descriptive messages
- Hypermedia as the engine of application state (HATEOS)

Identification of Resources

Before we can identify a resource, we need to understand what a *resource* is. A *resource* is any web-based concept that can be referenced by a unique identifier and manipulated via the uniform interface. While designing a REST API for a travel portal, your resources could be customer, reservation, ticket, hotel, flight, bus, car, and so forth. A resource can be a single entity or a collection of entities. According to Roy Fielding's dissertation: "The key abstraction of information in REST is a resource. Any information

that can be named can be a resource: a document or image, a temporal service (e.g., today's weather in Los Angeles), a collection of other resources, a non-virtual object (e.g., a person), and so on."

A resource is identified by a URI (Uniform Resource Identifier). A URI provides the name and the network address of a resource. All the information that a server provides can be identified as a resource. For example, the URI `http://www.foo.com/v1/customers` identifies a resource by name— `"customers"`. To manipulate a resource, the client connects to the server address specified in the URI (in this case `www.foo.com`) using a method like `GET` and access it using the relative path (`/v1/customers`). If the request is successfully executed, the response is a collection of customers. Again, resources can be related to each other; for example, a customer may have multiple reservations for different dates and hotels in different places. So a reservation is related to the customer as a subresource; for example, `http://www.foo.com/v1/customers/12345/reservations`.

The resources themselves are conceptually separate from the representations that are returned to the client. For example, the resource may be residing in some database, but when the server responds to a request for a resource, it does not send the database itself; rather it responds with some representation of the resource that represents a record in the database. For example, the record of a resource instance may be represented in XML, JSON, or HTML format, when it is returned to the client. The following is an example of a *customer* resource representation in JSON format with a *reservation* subresource:

```
{
     "firstName": "Mark",
     "lastName": "Johnson",
     "CustId": "John123",
     "age": 26,
     "address":
     {
         "streetAddress": "28 2nd Street",
         "city": "New York",
         "state": "NY",
         "postalCode": "10021"
     },
     "reservations":
     [
         {
           "type": "official",
           "number": "212-555-4321",
           "date": "03-12-2016"
         },
         {
           "type": "personal",
           "number": "646-555-9765",
           "date": "02-06-2015"
         }
     ]
}
```

Manipulation of Resources through Representation

Clients modify a representation of a resource. The same exact resource may be represented in different ways for different clients. For example, for a UI client, it might be represented in HTML format; whereas for application clients, it might be represented in either JSON or XML format. The representation is a way for clients to interact with the resource, but it is not the resource itself.

Self-Descriptive Messages

Each message (request/response) must be self-descriptive. That mean that the message may contain additional information to tell the recipient how to process it. Information such as format (JSON/XML), size, payload itself, and other metadata information included in the message can be used by the recipient for processing. An HTTP message provides headers to organize the various types of metadata into uniform fields. For example, *Content-Type* can he used to specify the format of the message; *Content-Length* can be used to specify the size of the payload. Many such HTTP headers can be included in the message to describe to the recipient on how they should process the message.

Hypermedia as the Engine of Application State (HATEOAS)

A resources' state information may include links to other resources. These links provide information on what to do next and how to traverse through other related resources in a meaningful manner; for example, after getting information about the *account*, you may want to deposit, withdraw, or transfer money. So the response of a RESTful service providing the account information may include links for the next action that the customer may want to do, as follows:

```
GET /account/12345 HTTP/1.1
HTTP/1.1 200 OK
{"account_number":"12345",
 "balance":"100.0",
 "currency":"USD",
 "links": [ {
        "rel": "deposit",
        "href": "http://localhost:8080/account/12345/deposit"
    },
            {
        "rel": "withdraw",
        "href": "http://localhost:8080/account/12345/withdraw"
    },
            {
        "rel": "transfer",
        "href": "http://localhost:8080/account/12345/transfer"
    }]
}
```

The presence or absence of a link in a resource representation is an important part of resource's current state.

While designing a REST API interface, you should keep all of these constraints in mind. The next few sections look at how to build a REST API interface by following these constraints.

Resource Identifier Design Using URIs

In a RESTful API, designing the resource is one of the most important tasks for its success. A well-designed resource makes the API intuitive, simple to understand, and easy to use. Let's look at some of the best practices for designing RESTful APIs.

Resource Naming Conventions

Every resource should have a meaningful name to identify itself. Name a resource using a noun as opposed to a verb or an action. The URI for the resource should refer to a thing rather than an action. Also CRUD function names should not be used in the URI or resource names; for example, while designing resource for a customer's entity, the resource URI should be named `/customers` instead of `/getCustomers`.

Modelling Resources and Subresources

According to Roy Fielding's dissertation a resource is "*any concept that might be the target of an author's hypertext reference must fit within the definition of a resource*." It can be single instance of an object or a collection of objects. Even business processes and capabilities can fit the definition of a resource according to Roy Fielding. Resources form the core of REST API design. The starting point of modelling resources is to analyze the current business domain and identify all the relevant objects in it that can be named. The focus for identifying resources and modelling them should be from the consumer's point of view. It is important to select the right resources and model them at the right level of granularity.

For example, a resource can be a collection of customers in an online store or it can be a single customer. You can identify a collection of 'customers' using `/customers`, while a single instance of a customer can be identified using `/customers/{customerId}`. Each customer may further have multiple orders. The URI to refer to the subcollection of 'orders' is modelled as `/customers/{customerId}/orders`. A single instance of the order may be identified by `/customers/{customerId}/orders/{orderId}`. By following a logical grouping or resources and their hierarchy, you can model the resource URI path to access a collection of resources or an individual resource.

Best Practices for Identifying REST API Resources

The following are some of the best practices for identifying resources for RESTful API design.

- Resources should not be too fine grained because they lead to chatty communication between the consumer and the provider. Chatty communication degrades overall performance of the app that is using the API; hence, it should be avoided.
- Resources should not be too course grained because this leads to APIs that are too difficult to use and maintain.
- Resources should be designed such that they do not lead to migration of control flow business logic to the API consumer side; for example, if updates to the customer information requires multiple fields to be updated in a specific sequence that depends on some logic, then the API to update the customer information should be designed so that the client is not responsible for executing the required flow logic. The responsibility of executing the logic should lie with the resource server hosting the resource. Shifting the logic to the consumer side has the risk of putting the resource data in an inconsistent state, especially in the event of failure. Fine-grained APIs that perform CRUD operations may put the business logic on the client side, creating tight coupling between the API consumer and the provider. Any change in business logic at the provider end would require corresponding changes on the API consumer side. They may not be possible in many cases, where consumers do not want to make frequent changes to the applications on their side.
- Resource selection should be independent of the underlying domain implementation details. Hence, even a business process can be modelled as a resource if the process involves the operation of multiple low-level resources. For example, the process of setting up a customer in a bank may be modelled as a resource. So there can be a resource created for a customer account setup—such as `/accountSetup`–that needs to call operations on related resources for entities such as customer and account. By modeling a business process as a resource, the API consumer does not need to apply the business logic in the code.

URI Path Design

Every collection and resource in an API has its own URL. It is recommended to design URLs using an alternate combination of collection/resource path segments, relative to the API entry point. Table 3-1 explains the concept better, with guidelines on how to define the top-level resource and related subresources.

Table 3-1. *Top-Level Resources and Related Subresources*

URL	Description
/api	The entry point for the API. Also sometimes referred as 'basePath'. *Eg:/api*
/api/{ResColName}	Resource name of a top-level collection *Eg:/api/customers*
/api/{ResColName}/{ResId}	The ResId inside collection of resources *Eg:/api/customers/Customer1234*
/api/{ResColName}/{ResId}/{SubResColName}	Sub resource collection under resource ResId *Eg:/api/customers/Customer1234/orders*
/api/{ResColName}/{ResId}/{SubResColName}/{SubResId}	SubResId inside sub resource collection *Eg:/api/customers/Customer1234/orders/order-123*

There may be arbitrary levels of nesting for subresources. However, it is recommended to limit the depth to two or three, if possible, because longer URLs are more difficult to work with.

A URI design that follows a predictable pattern with a hierarchical approach to traverse through the resources eases developer adoption; for example, `/stores/{storeId}/products/{productId}`. This helps developers to guess the URI for a given resource; and hence, it can make direct calls without going through links.

URI Format

Let's now look at the recommended format of a URI and learn how this format can be effectively used for designing an API. As per RFC 23964: "a Uniform Resource Identifier (URI) is a compact string of characters for identifying an abstract or physical resource." This identifier can be realized in one of two ways: as a Uniform Resource Locator (URL) or a Uniform Resource Name (URN).

URLs (e.g., `http://www.foo.com/users/mike`) are used to identify the online location of an individual resource; whereas URNs (e.g., `urn:user:mike`) are intended to be persistent, location-independent identifiers. The URN functions like a person's name; whereas a URL is like that person's street address. In other words, the URN defines an item's identity (the user's name is Mike) and the URL provides a method for finding it (Mike can be found at `www.foo.com/users/`).

The syntax of an URI is a hierarchical sequence of components as follows:

```
scheme:[//authority][/]path[?query][#fragment]:
```

- **Scheme name**: Identifies the protocol (e.g., FTP, HTTP, HTTPS, IRC:)
- **Authority**: Refers to the actual DNS resolution of the server. It consists of the hostname or IP address of the server, optionally along with the port number. The credentials to access the server can also be included as part of the authority as follows: *[user:password@]host[:port].*
- **Path**: Pertains to a sequence of segments separated by a forward slash (/).

- **Query**: Contains additional identification information that is non-hierarchical in nature and often separated by a question mark (?).
- **Fragment**: Provides direction to a secondary resource within the primary one identified by the authority and path, and separated from the rest by a hash (#).

Naming Conventions for URI Paths

Keep URIs short and simple because this helps you write, remember, and spell it easily. The following are some of the recommended naming conventions for URI paths.

- Name a collection resource with a *plural noun*; for example, `http://www.foo.com/api/customers`
- Name a singular resource with a *singular noun*; for example, `http://www.foo.com/api/customers/customer1234`
- Name a controller resource using a *verb*; for example, `http://www.foo.com/api/customers/customer1234/register`
- Avoid using CRUD operation names in URIs. For example, do not use URIs such as `http://www.foo.com/api/getcustomers`.
- Use lowercase letters for naming URIs. Avoid mixed and uppercase letters in URIs. Mixed case is harder to type and read.
- Use hyphens instead of a space or an underline. They are more aesthetic and easier to read. Spaces in URLs get transformed into URL encoded `%20`s, further degrading readability. For example, use URIs such as `http://www.foo.com/api/about-us`.
- Avoid using characters that require URL encoding, such as spaces.

HTTP Verbs for RESTful APIs

Once the resources have been identified, these are next set of questions to ask:

- What would a consumer like to do with the resource?
- What aspects of the resource would be of interest to a consumer?

The answers to these questions identify the HTTP verbs to be used for each of the identified resources.

HTTP verbs form an important part of a RESTful API design. They identify the actions to be performed on a resource. A consumer's actions with a resource can be mapped to an HTTP verb in most cases; for example, creating a product can be done using the HTTP verb POST. The primary and most commonly used HTTP verb are POST, GET, PUT, and DELETE. These verbs perform the CRUD operations on the resource as follows:

- POST verb creates a new instance of the resource
- GET is used to read
- PUT is used to update
- DELETE is used to delete

There are other verbs—such as HEAD, OPTIONS, TRACE, and CONNECT—in the HTTP 1.1 spec. Let's look at the detailed usage of these verbs in the design of a REST API interface in the next few sections of this chapter.

GET

The GET verb is used by the client to retrieve information about the requested resource entity identified by the request URI. Requests using GET should only retrieve data and should never modify the data in any way. The GET request is considered *safe*. GET is a read-only method and does not make any changes to the resource data. Hence, it can be used without risk of data modification or corruption. Also, calling the GET method on a resource once has the same effect as calling it multiple times. Hence, the GET verb is *idempotent* and *safe*.

If the request has been executed successfully, the server returns the requested data normally in XML or JSON, depending on the format requested by the client. The HTTP '`Accept`' header is used by the client to specify the expected format of the response. The request may contain additional HTTP headers that can control the data returned by the server in response to the GET request. For example, if the request message includes headers such as *`If-Modified-Since`*, *`If-Unmodified-Since`*, *`If-Range`*, *`If-Match`*, or *`If-None-Match`*, it is processed as a conditional GET method. The server responds with the entity only if the conditions described by the header field(s) are satisfied. The conditional GET method is used to reduce unwanted network usage. These conditional headers are inspected by the server to determine if the client is already in possession of some of the data it is requesting. Data is returned only if the condition is satisfied; otherwise, no data is transferred in the response. Thus, conditional GET headers help reduce network traffic.

On successful execution of the GET request, the server responds with HTTP response code of *200 OK*. In the event of an error, the server usually responds with the *404 Not Found* or *400 Bad Request* status code.

The following are examples of GET request for a resource:

```
GET https://www.foo.com/customers
GET https://www.foo.com/customers/{customerId}
```

POST

The POST verb is normally used to create a new resource. In particular, it is used to create a subresource, which is subordinate to the parent resource identified by the request URI. To create a new resource, send a POST request to the URI of the parent resource and the server takes care of creating the new resource as a subresource of the parent, based on the information provided in the payload. Each new resource created is assigned a name or an ID to uniquely identify it. This identifier may be used to retrieve the resource information using a GET request at a later time.

On successful execution of the POST request, the origin server should respond with a *201 Created status* code. The response payload should contain the details of the resource created in a format expected by the client. The response should also contain a `'Location'` header to specify the location of the newly created resource. If the resource cannot be created, the server may respond with a *204 No Content* status code.

POST is neither *safe* nor *idempotent*. It is therefore recommended for non-idempotent resource requests. Making two identical POST requests usually results in two resources containing the same entity.

The following is an example of a POST request to create a 'customer' resource:

```
POST http://www.foo.com/customers HTTP/1.1
{
  "customers": {
    "customerId": "12345",
    "customerName": "Brajesh De",
    "Address":{
      "AddressLine1":"206 Lane 1",
      "AddressLine2":"22 Cross",
      "City":"Bangalore",
      "State":"Karnataka"
      }
  }
}
```

PUT

The PUT method is generally used to update an existing resource entity identified by the request URI. If the resource identified by the request URI exists, then the message payload should be considered as the changed version of the existing resource entity. If the resource does not exist, and the URI is capable of being defined as a new resource, the server can create a new resource with the information provided in the message payload. On successful execution of the PUT request, if a new resource is created, the server must respond with a *201 Created* status code. If an existing resource is modified, the server must respond with either the *200 OK* or the *204 No Content* status codes to indicate successful execution of the request. In the event of errors in modifying or creating a PUT request, the server should respond with an HTTP error response status code and an error message that indicates the nature of the problem.

PUT is *idempotent* but not *safe*. This means invoking the PUT method multiple times with the same request payload has the same effect on the resource—it continues to exist in the same state. But since the PUT method updates the resource entity, this method is not safe.

The following is an example of a PUT request.

```
PUT http://www.foo.com/customers/12345 HTTP/1.1
{
  "customers": {
    "customerId": "12345",
    "customerName": "Brajesh De",
    "Address":{
      "AddressLine1":"206 Lane 1",
      "AddressLine2":"22 Cross",
      "City":"Bangalore",
      "State":"Karnataka"
      }
  }
}
```

The Difference Between PUT and POST

It is recommended to use POST for creating new resources and PUT for updating an already existing resource. Use POST if the *server* is responsible for creating the resource name or ID and hence is the URI of the new resource. PUT may be used for creating a new resource only when the *client* is responsible for deciding the new URI (via its resource name or ID) for the resource. A POST verb should be used if the client doesn't or shouldn't know the resulting URI of the new resource before creation. If the resource is already created, PUT should be used to update the resource.

DELETE

The DELETE verb is used to delete the resource represented by the request URI.

On successful execution, the server responds with 200 OK or 204 No Content status codes. If the 200 OK status code is returned, it may also contain the representation of the deleted resource. Since additional bandwidth requirements for the response payload may impact the overall performance, it is recommended to respond with HTTP 204 No Content on successful deletion of the resource.

The DELETE verb is idempotent and not safe. The resource is removed or is marked as deleted in the database on successful execution of the DELETE request.

Repeatedly calling DELETE on a resource ends up the same: the resource is gone. However, there is a caveat about DELETE idempotence. Calling DELETE on a resource a second time will often return a 404 (NOT FOUND) since it was already removed and hence can no longer be found. This makes DELETE operations no longer idempotent. However, this is an appropriate compromise if resources are removed from the database instead of being simply marked as deleted.

The following is an example of a DELETE request:

```
DELETE http://www.foo.com/customers/12345 HTTP/1.1
```

PATCH

The PATCH method was added to HTTP specs in March 2010. This method is similar to the PUT method and can be used to update an existing resource definition. The difference between PUT and PATCH is that PATCH can be used to do a partial update of an existing resource definition; whereas PUT does a complete update. With the PATCH method, only certain attributes of the resource can be specified for update.

The following is an example of a PATCH request:

```
PATCH http://www.foo.com/customers/12345 HTTP/1.1
{
  "customers": {
    "Address":{
      "AddressLine1":"205 Lane 2"
      }
  }
}
```

OPTIONS

The OPTIONS verb allows the client to determine the options and/or requirements for interacting with a resource or a server. The OPTIONS verb determines the HTTP methods and headers allowed for interacting with a resource. It indicates to the client the capabilities of a server without actually performing any of the CRUD operations. The client can specify a URL for the OPTIONS method to refer to a specific resource. An asterisk (*) should be used if the client is interested in knowing or testing the capabilities of the entire server. Responses of this method cannot be cached.

This is an optional method that is not always supported by all service implementations. Many popular sites do not support this method; for example, GitHub responds with a 500, Google Maps with 405 Method Not Allowed. If this method is supported, the response should be 200 OK and have an `'Allow'` header containing a list of HTTP methods that may be used on this resource.

The OPTIONS method can be used by the client to provide support for cross-origin resource scripting (CORS) implementation. Chapter 7 looks at how to implement CORS for building secure web APIs.

The following is an example of an OPTIONS request:

```
OPTIONS * HTTP/1.1
```

HEAD

The HEAD method is identical to GET. The difference is that with HEAD method, the server responds only with a response line and headers. The response to the HEAD method does not contain the entity-body. The metainformation contained in the HTTP headers in response to a HEAD request is identical to the information sent in response to a GET request. This gets only the metainformation about the resource entity, without actually transferring the resource entity-body in the response payload. It reduces network bandwidth usage. This method is often used for testing recent modifications, the validity of hypertext links, and accessibility.

Idempotent and Safe Methods

Some HTTP methods can be called multiple times without any change in the result or the state of the resource. This brings in the concept of a method being idempotent and/or safe. An *idempotent* HTTP method can be called many times without getting a different outcome. It does not matter if the method is called one time or 100 times—the result is going to be the same. A point to note is that idempotency refers to the result of the method execution and not to the resource itself. For example, calling a GET method on a particular resource always gives the same result unless the resource has been changed in some other way. An HTTP method is considered safe if it does not modify the state of the resource. For example, calling a GET or HEAD method on a resource URL never modifies the resource itself; hence, it is considered safe.

Table 3-2 summarizes whether an HTTP method is idempotent and/or safe.

Table 3-2. *Idempotent and/or Safe HTTP Methods*

HTTP Verb Name	Idempotent	Safe
GET	Yes	Yes
POST	No	No
PUT	Yes	No
DELETE	Yes	No
HEAD	Yes	Yes
OPTION	Yes	Yes
PATCH	No	No

HTTP Status Code

The HTTP response communicates the status of the request processing. The response contains certain metadata and optional payloads. The `Status-Line` part of the HTTP response message is used to inform clients of their request processing results in the following format:

```
Status-Line = <HTTP-Version> SP <Status-Code> SP <Reason-Phrase> CRLF
```

HTTP defines 40 status codes to communicate the execution results of a client's request. The status code is divided into the following five categories.

- **1xx Informational**: Communicates transfer protocol level information.
- **2xx Success**: Communicates that the request from the client was successfully received, understood, and accepted.
- **3xx: Redirection**: Communicates that additional action needs to be taken by the user agent like browser in order to fulfil the request.
- **4xx Client Error**: Indicates errors caused by the client.
- **5xx Server Error**: Indicates that server is aware that an error occurred while processing the request and cannot process it further.

Normally, 2xx and 3xx status codes are treated as success codes. Any 4xx or 5xx status code is treated as an error code.

Table 3-3 lists the most commonly used success codes.

Table 3-3. *The Most Commonly Used Success Codes*

Status Code	Reason-Phrase	Meaning
200	OK	Indicates that the request has been processed successfully.
201	Created	Indicates that the request has been processed and a new resource has been created successfully.
202	Accepted	Indicates that the request has been received by the server and is being processed asynchronously.
204	No Content	Indicates that the response body has been purposely left blank.
301	Moved Permanently	Indicates that a new permanent URI has been assigned to the client's requested resource.
303	See Others	Indicates that the response to the request can be found in a different URI.
304	Not Modified	Indicates that the resource has not been modified for the conditional GET request of the client.
307	Use Proxy	Indicates that the request should be accessed through a proxy URI specified in the Location field.

Table 3-4 lists the most commonly used error codes.

Table 3-4. *The Most Commonly Used Error Codes*

Status Code	Reason Phrase	Meaning
400	Bad Request	Indicates that the request had some malformed syntax error due to which it could not be understood by the server. Probable reason is missing mandatory parameters or syntax error.
401	Unauthorized	Indicates that the request could not be authorized, possibly due to missing or incorrect authentication token information.
403	Forbidden	Indicates that the request was understood by the server but it could not be processed due to some policy violation or the client does not have access to the requested resource.
404	Not Found	Indicates that the server did not find anything matching the request URI.
405	Method Not Allowed	Indicates that the method specified in the request line is not allowed for the resource identified by the request URI.
408	Request Timeout	Indicates that the server did not receive a complete request within the time it was prepared to wait.
409	Conflict	Indicates that the request could not be processed due to a conflict with the current state of the resource.
414	Request URI Too Long	Indicates that the request URI length is longer than the allowed limit for the server.
415	Unsupported Media Type	Indicates that the request format is not supported by the server.
429	Too Many Requests	Indicates that the client sent too many requests within the time limit than it is allowed to.
500	Internal Server Error	Indicates that the request could not be processed due to an unexpected error in the server.
501	Not Implemented	Indicates that the server does not support the functionality required to fulfill the request.

(*continued*)

Table 3-4. (*continued*)

Status Code	Reason Phrase	Meaning
502	Bad Gateway	Indicates that the server, while acting as a gateway or proxy, received an invalid response from the back-end server.
503	Service Unavailable	Indicates that the server is currently unable to process the request due to temporary overloading or maintenance of the server. Trying the request at a later time might result in success.
504	Gateway Timeout	Indicates that the server, while active as a gateway or proxy, did not receive a timely response from the back-end server.

Resource Representation Design

A REST API resource entity representation is used to convey the state of the resource. The message body of the request/response is used to convey the state of the resource entity. The client sends the resource entity to the server in the request message payload of a POST, PUT, or PATCH message. The server sends the resource entity state in the response message payload for a GET, POST, PUT, or optionally, DELETE request.

A text-based format is normally used to represent the resource state. JSON and XML are the most commonly used text formats for representing the state of the resource entity. JSON is lightweight and provides a simple way to represent a resource. Due to the seamless integration of JSON with the browser's native runtime environment, JSON is the preferred choice for data representation in the design of a REST API. XML, on the other hand, is verbose, hard to parse, hard to read, and its data model is not compatible with many programing languages. This makes JSON a preferred choice over XML for representing the resource entity for a REST API. Many popular API providers have already moved away from XML to the JSON format. However, if the API consumer base consists of a large number of enterprise customers, you still have to support the XML data format for your APIs.

As a general guideline, it is advisable to support JSON data format by default and provide additional support for the XML format, if required. With support for both JSON and XML formats, how does the client specify the preferred format for the response? There are the following options:

- Use the *'Accept'* header.
- Append `.json` or `.xml` extensions to the endpoint URL.
- Include a query parameter in the URL to specify the response format.

Of the three options, use of *'Accept'* header to specify the response message format is most preferred. The following are some of the basic best practices for the JSON format representation of the resource entity.

- JSON should be in a well-formed format, with the variable names and their values enclosed in double quotes.
- JSON names should use mixed lowercase and uppercase letters. Special characters should be avoided whenever possible. JSON names like `fooName` is preferred over `foo-Name` because it allows the use of the cleaner dot notation for property access in JavaScript.
- The *`'Content-Type'`* header in the message should be set to `application/json` when a JSON format payload is included in the message.

Hypermedia Controls and Metadata

HTTP headers in the request/response convey metadata about the messages and about the resource entity contained in the message. HTTP specification defines a set of standard headers that can be used for various purposes. The specification also allows extension mechanisms to include custom HTTP headers. HTTP headers are classified under four types.

- **Entity headers**: This type of header provides metainformation about the entity body or resource in the message. Information such as the allowed HTTP methods, the media type, size, and location of the resource entity or cache expiration date-time, and so forth, are some of the examples of `Entity Header` types.
- **General headers**: This type of header provides information that can be applicable for both request and response messages. Caching directive, connection information, message origination date-time, and any message transformation applied on the whole message, are some examples of `General Header` types.
- **Client request headers**: This type of header is included *only* in the request message sent by the client or browser to the server. Authorization information, user agent information, information about the character set, encoding, or language that the client can accept, are some examples of information provided by `Client Request` headers.
- **Server response headers:** This type of header is included *only* by the server in the response sent to the client. Information about the age of the response generated by origin server, ETag information for caching purposes, the duration for which the server is unavailable for the requesting client, are some examples of `Server Response` headers.

This section looks at the most commonly used HTTP headers and how they can be used to design a better RESTful interface.

Accept (Client Request Header)

The *`Accept`* header is used in the request message to specify the media types that are acceptable by the client for the response. It is a mechanism for the client application or browser to indicate to the server which MIME types it is expecting.

The client can specify a range of media types using an asterisk (*) or multiple media types using comma-separated values. Media ranges can be overridden by specific media ranges or specific media types. If more than one media range applies to a given type, the most specific reference has precedence.

For example,

```
Accept: text/*, application/xhtml+xml, application/xml;q=0.9, */*
```

has the following precedence:

1. `application/xml;q=0.9`
2. `application/xhtml+xml`
3. `text/*`
4. `*/*`

The client can specify its relative preference for a media type using an optional *q* parameter. The following is an example:

```
Accept: audio/*; q=0.3, audio/basic
```

These examples indicate that `audio/basic` is preferred, but any audio type is also acceptable if it is the best available after a 70% markdown in quality.

If no `Accept` header field is specified, then it is assumed that the client accepts all media types. If an `Accept` header field is present but the server cannot send a response that is acceptable according to the Accept field value, then the server should respond with a HTTP status code of *406 Not Acceptable.*

Accept-Charset (Client Request Header)

The *`Accept-Charset`* request header is used by the client to specify the character sets that it understands and therefore can be included by the server in the response. As with the `Accept` header, the client can specify multiple charsets in a comma-separated list. A *q* value on a scale of 0 to 1 can also be included to specify the acceptable quality level for non-preferred character sets.

If the client does not include an `Accept-Charset` header in the request, it is assumed that any character set is acceptable. If a `Accept-Charset` header is present but the server cannot send a response that is acceptable according to the `Accept-Charset` header, then the server should send an error response with the *406 Not Acceptable* HTTP status code, though the sending of an unacceptable response is also allowed as per the HTTP specs.

The following is an example of Accept-Charset header:

```
Accept-Charset: iso-8859-5, unicode-1-1;q=0.8
```

Authorization (Client Request Header)

The `Authorization` header is used by the client to include authentication information needed to access a server resource. If the server needs authentication and the `Authorization` header is not present in the request or is having an incorrect value, the server should send an error response with a 401 Unauthorized HTTP status code. The server should also include the `WWW-Authenticate` header in the response, which indicates the authentication scheme(s) required. The authentication schemes can be basic or digest access.

The following is an example of *Authorization* header:

```
Authorization: BASIC Z3Vlc3Q6Z3Vlc3QxMjM=
```

Host (Client Request Header)

The *`Host`* request header specifies the server address and the port of the resource requested. A `Host` without any port information implies the default port. The default port is 80 for HTTP and 443 for HTTPS.

The following is an example of *Host* header:

```
Host: http://www.foo.com
```

Location (Server Response Header)

The *`Location`* response header is used by the server to redirect the recipient to a URI other than the request URI for completion. This header is returned by the server in the following two scenarios.

- When a new resource is created after the successful execution of a POST or a PUT request. In this scenario, the `Location` header contains the location information of the newly created resource and the HTTP response status code should be *201 Created*.
- When the resource has moved temporarily or permanently, or is the result of a request execution is available at a different location. In this scenario, the `Location` header contains the redirected URI and the HTTP response status code should be 3xx. The `Location` information is then used by the browser to load a different web page, as specified in the header, thus helping in automatic redirection.

The following is an example of *Location* header:

```
Location: http://www.foo.com/http/index.htm
```

ETag (Server Response Header)

The *ETag* (entity tag) response header provides a mechanism for the server to send information about the current state of the entity. It is an alphanumeric string that uniquely identifies a specific version of the resource. If the resource has changed, the ETag value changes. Hence, the ETag value can be compared to determine if the cached resource entity on the client side matches that on the server.

It is a mechanism used for web cache validation that allows a client to make conditional requests. It makes caches more efficient and saves bandwidth because the server does not need to send the full response if the content has not changed.

The following is an example of *ETag* header:

```
ETag: "686897696a7c876b7e"
```

Cache-Control (General Header)

The *Cache-Control* general header field is specifies instructions on caching response information by the client and/or any intermediary along the request/response chain. Directives contained in this header provide information about the cache-ability of the response. It specifies if the response can be cached or not. If yes, can it be cached in public or private cache? It also specifies if the cache can be archived and stored. This header also contains information about the maximum duration for which the response can be cached.

The following is an example of *Cache-Control* header:

```
cache-control: private, max-age=300, no-cache
```

Content-Type (General Header)

The *Content-Type* header specifies the media type of the payload included in the message.

The following is an example of *Content-type* header:

```
Content-Type: text/html; charset=ISO-8859-4
```

Header Naming Conventions

Earlier sections looked at the best practices for naming resources and URIs. For good API design, even the HTTP headers should be named according to a convention. This section looks at some of the recommended best practices for naming headers.

HTTP specifications provide names for all standard HTTP headers and their syntax. It also provides extension mechanisms to include custom headers, if required. The following conventions are recommended for naming custom HTTP headers.

- Historically, `X-` has been used as a prefix for naming non-standard custom headers. RFC 6648 has deprecated the use of this convention because it causes more problems than it solves. Hence, do not prefix custom header names with `X-` or similar constructs.
- Name custom headers meaningfully and with the assumption that all custom headers may become standardized, public, commonly deployed, or usable across multiple implementations.
- Use hyphens in header names if required; for example, `My-Header-Name`.
- Do not use spaces in header names.

Versioning

Versioning is one of the most important considerations for web API design. Regardless of the approach followed, REST APIs should always be versioned. It helps to develop APIs in an iterative approach.

There are multiple approaches for versioning an API. The following are some questions to ask when thinking about API versioning.

- Which versioning approach should be used?
- When should a new version of the API be created?
- How and where to indicate the version of the API?
- How many versions should be maintained?
- How long should the older versions of the API be maintained?
- What are the deprecation mechanisms for older versions?

This and many other considerations and approaches for API versioning are discussed later in this book.

Querying, Filtering, and Pagination

Enterprises use REST APIs to expose their data and services. The resource collection returned by REST API may be huge. Transmitting the entire payload over the network is heavy on the bandwidth. Additionally, processing an entire collection on the client side would be processor intensive. Since a UI can display only a limited amount of data, this becomes important from UI processing standpoint as well; for example, 20 results per page. Hence, the need arises to be able to query, filter, and paginate the response. The API should provide a mechanism for the consumer to specify the query parameters and filter criteria. They should also be able to specify a range of data to be returned in the response. The range can be in terms of the number of elements, a date and time range, or in terms of offset and a limit.

It is important to note that it is not mandatory to provide support for querying, filtering, and pagination for all REST APIs. This is a resource-specific requirement and by default is not required to be supported on all resources. Consider designing the API to support filtration and pagination only if the number of entities in the resource collection that can be returned by default is high. The API documentation should specify if these complex functionalities are available for any specific service.

Limiting via Query-String Parameters

Filtering and pagination for an API is best implemented by designing the API interface with `offset` and `limit` query-string parameters. The `offset` parameter indicates the beginning item number in a collection and the `limit` specifies the maximum number of items to return.

The following is an example:

```
GET http://www.foo.com/products?offset=0&limit=25
```

In this example, the `offset` value 0 and `limit` value 25 indicate to return the first 25 items in the list. If the number of items fetched from the back end is more than 25, only the first 25 are returned. To retrieve the next set of items, the client has to make another call with a changed value for `offset` (=25) and `limit` (=25). If the number of items in the list is less than 25, all the items are returned in the response. This approach helps implement pagination support in the API.

It is important to understand that `offset` and `limit` are query-string parameters and are not dictated by any standards or specifications. Hence, different API providers may implement the same concept by using different parameter names. `start`, `count`, `page`, and `rpp` (records per page) are other examples of query-string parameters that can be used to implement pagination. An API designer can name them anything to suite the business context.

Filtering

Filtering is an approach to restrict the results returned in the response by specifying additional search criteria. These search criteria must be met on the data returned in the result. The filtering can become complex if the API has to support a complicated set of search criteria. The filtering criteria is based on the resource attribute. The complexity increases if filtering involves a complex combination of comparison operators. However, filtration can be achieved by supporting simple criteria, such as *`starts-with`* or *`contains`*, and so forth.

The filtering criteria can be specified by using the `filter` query-string containing a delimiter-separated list of name/value pairs. The delimiters that have conventionally worked are the vertical bar (`|`) to separate individual filter phrases and a double colon (`::`) to separate the names and values. This approach supports a wide range of use cases for filtering and also makes the filter criteria user-readable. The following is an example:

```
GET http://www.foo.com/customers?filter="name::matt|city::delhi"
```

Note that the property names in the name/value pairs match the name of the properties returned by the service in the payload. Wild cards can also be included in the filter values by using the asterisk (*).

Filtering can be implemented for an API by using one of the following approaches.

- Map the filter criteria to the back-end database SQL queries and implement filters at the database layer. This would retrieve the data matching the criteria from the data store; the same can be passed to the client with minimal messaging.
- Implement filter criteria in the service implementation layer. The service accepts the filter criteria as inputs and applies them on the data fetched from the data store. This may be required when the search criteria is complex or requires some business logic to be executed on the data set returned from the data store.
- Implement filter criteria on the API's intermediary layer. In the event that there is no change to the database or service implementation layer, the filtering is done on the intermediary API node that is generally introduced for creating and exposing REST APIs. Implementation of the filter on the intermediary API node might be complex due to the limited programming support provided by these tools.

When deciding on which of these approaches to adopt, it is recommended to implement filtering as close to the resource data store as possible.

The Richardson Maturity Model

The Richardson Maturity Model defines the levels to assess the maturity of a REST API service. It defines the following four levels (0–3) based on services support for URI, HTTP verbs, and hypermedia.

- Level 0: Swamp of POX
- Level 1: Resources
- Level 2: HTTP verbs
- Level 3: Hypermedia controls

Figure 3-1 shows the three core technologies with which Richardson evaluates service maturity. Each layer builds on top of concepts and technologies of the layer below. The higher up the stack an application sits, and the more it employs the technologies in each layer, the more mature it is.

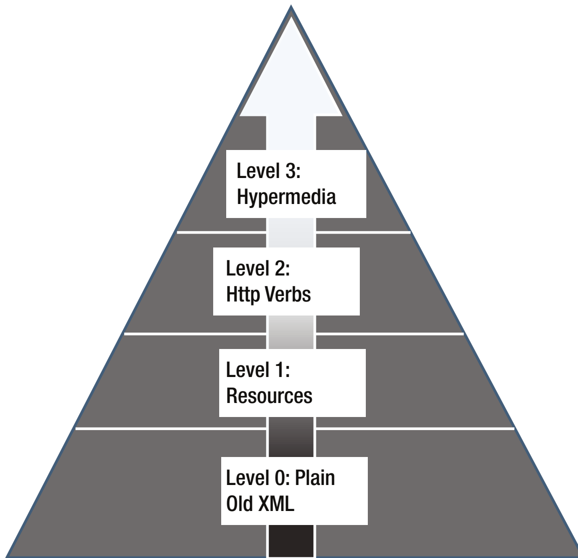

Figure 3-1. *Richardson's Maturity Model for REST APIs*

Let's look at each of these levels in detail.

Level 0: Swamp of POX (Plain Old XML)

This is the most basic level of maturity. At this level, the service is characterized as having a single URI that acts as the entry point. HTTP is used as the transport system for remote interactions. The payload content can be described in XML, JSON, YAML, key-value pairs, or any format of your choice. Normally, the POST method is used for sending the request to the server. SOAP and XML RPC are examples of services at Level 0 maturity. Figure 3-2 below shows a client making a request to an appointment service to get the availability of slots for a given date and doctor. The search parameters are sent in plain old XML format using POST request.

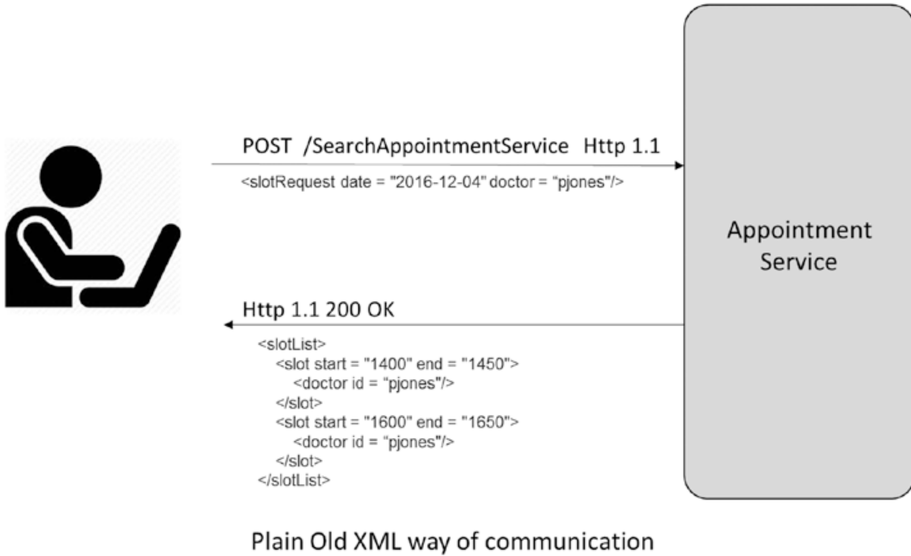

Figure 3-2. *Level 0- Plain Old XML way of communication*

Level 1: Resources

The first step toward RESTful maturity is the introduction of resources. At this level, instead of having a single URI as an endpoint for all services, you start interacting with individual resources through separate URIs. So instead of going through an endpoint like `http://www.foo.com/searchAppointmentService`, you start using resource URIs like `http://www.foo.com/api/doctors/{doctorId}`. Here `doctors` is a resource and you get access to an individual doctor's information by using `{doctorId}`. At this level, you still use POST as the only HTTP method for all of your communication. Figure 3-3 below shows a client making a request to an appointment service to get the availability of slots for a given date and doctor. The URL used to get the slot availability of the doctor is resource oriented.

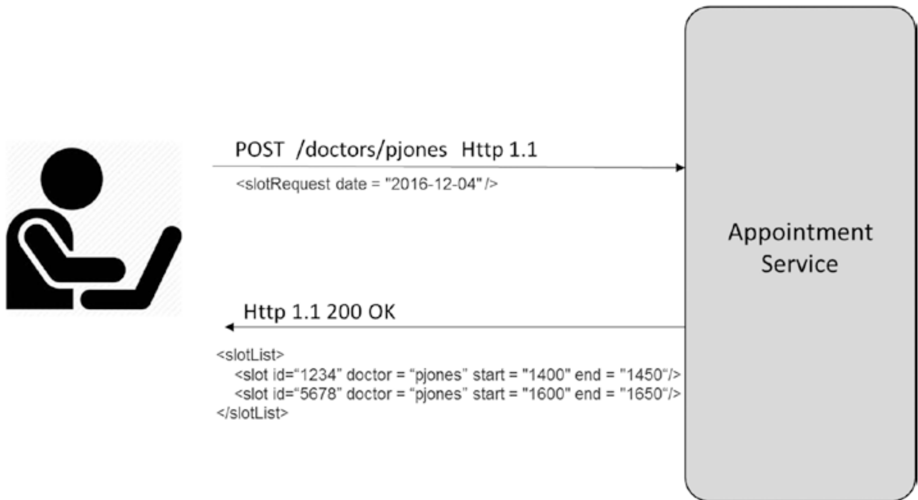

Figure 3-3. *Level 1- Using resources for communication*

Level 2: HTTP Verbs

At Level 0 and 1, the applications use the POST method for all communication. Level 2 maturity moves toward using the HTTP verbs more closely to how they are used in HTTP itself. To fetch the slot availability of a particular doctor, it should be using the HTTP verb GET at this level. As you've seen, the GET verb is safe because it is read-only and does not make any significant changes to the state of the resource. Hence, you can use the GET verb any number of times, in any order, and still get the same result every time, unless the resource has been modified using a different method. If you have to create a new appointment, you can use the POST method. If you want to update an existing appointment, you may use the PUT method.

In addition to the use of HTTP verbs, Level 2 also introduces the use of HTTP response codes to indicate the status of an operation on a resource. If a resource was successfully created, the service returns with HTTP response code 201. If the operation on a resource was successful, the 200 status code is used in the response. If the operation on a resource resulted in an error, an appropriate 4xx or 5xx response code should be used in the response. Figure 3-4 below shows a client making a request to an appointment service to get the availability of slots for a given date and doctor. 'GET' Http verb is used to access the resource oriented URL to get the appointment slots of the doctor. Http response code 200 OK is returned to indicate successful response.

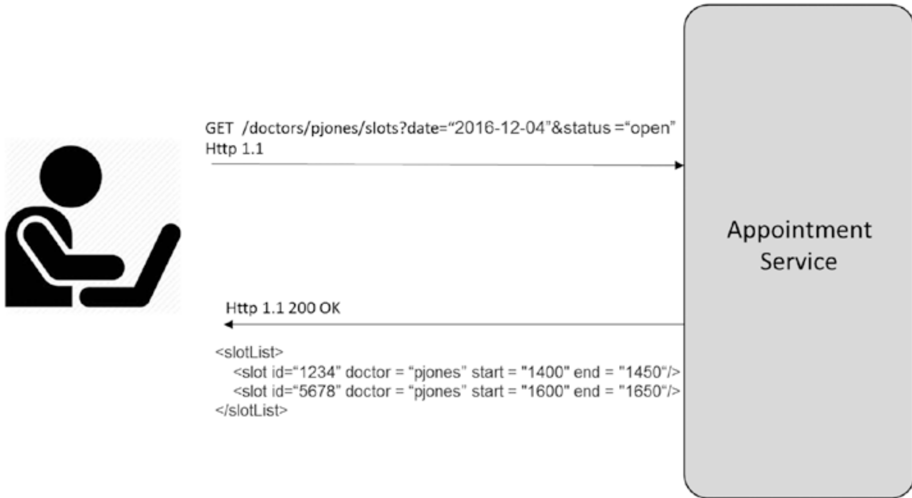

Figure 3-4. *Level 2- Using resources and verb for communication*

Level 3: Hypermedia Controls

This is the final level for REST maturity and it is where HATEOS enters the picture. It addresses the question of what to do next. After receiving the response for a service invocation, what are the next logical steps for the client? At a given node, what are the possible branches for traversal in a tree? This helps the client to be more intelligent and decide or prompt the user for the necessary possible actions.

At Level 3 maturity, the response of a REST service may contain a list of URIs. These URIs are the resources that the client wants to act upon as the next course of action. So rather than the client having to know where to post the next request, the hypermedia controls in the response tells how to do it. Figure 3-5 below shows a client making a request to an appointment service to get the availability of slots for a given date and doctor. The response returned for the GET request contains hyperlinks for the next possible actions that the client can do to book a slot.

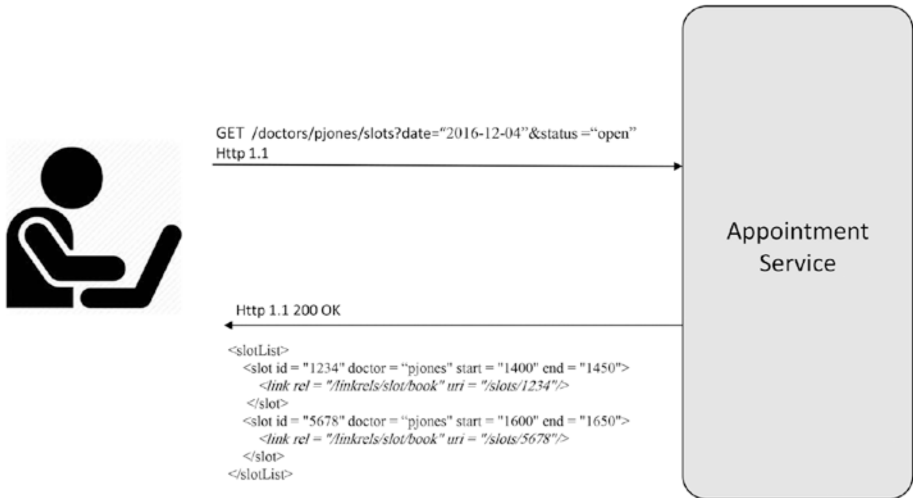

Figure 3-5. *Using Resource, Verb and HATEOAS for communication*

An obvious advantage of hypermedia controls is that it allows the server to change its URI scheme without breaking clients. It also helps client developers expose the protocol. The link gives client developers a hint on what the next possible options are. It may not provide all the information, but it at least gives developers a starting point to think about more information for the API and to look for a similar URI in the API documentation. Currently, there are no absolute standards on how to represent hypermedia controls. It is up to the service implementation team to decide how to implement HATEOS in their service.

As per Martin Fowler's article on Richardson Maturity Model, RMM provides a good way to think of the different elements of a RESTful service, but it is not a definition of levels of REST itself. Roy Fielding has made it clear that Level 3 RMM is a precondition of REST.

CHAPTER 4

API Documentation

Documenting a REST API is important for its successful adoption. APIs expose data and services that consumers want to use. An API should be designed with an interface that the consumer can understand. API documentation is key to the app developers comprehending the API. The documentation should help the developer to learn about the API functionality and enable them to start using it easily. This chapter looks at the aspects of documenting an API and some of the tools and technologies available for API documentation, including RAML, Swagger, API Blueprint, and others.

The Importance of API Documentation

As an API provider or developer, you may master your API. You have inside knowledge about its functionality, what it is supposed to do, how it is to be used, its security, limitations, error scenarios, and so forth. As an API provider, you have gradually learned everything about the API through various discussions, documentation, and references. However, this is not the case for the consumers of your API. The app developer community or API consumers look at the API's interface and wonder what the API does, how it should be used, what to expect when an error occurs, what security credentials to use, how and where to get the security credentials to use the API, and so forth. Hence, what is easy and simple to the API developer may not be intuitive to the API consumer. Good API documentation can help bridge the gap and make the API successful. API documentation communicates a vast amount of information about the API.

As enterprises move along in the digital transformation journey, there has been exponential growth in public and private APIs. In this competitive world, it is very likely that the data and services exposed by your API may also be exposed by another API provider. If the API is being monetized, it becomes more important to make it successful for your business. Good user-friendly API documentation is a key to its successful adoption. An API document is like an entrance into your API and provides a warm welcome to the API's consumers.

The API documentation should

- Get users started quickly
- Include useful and relevant information
- Provide sample code

© Brajesh De 2017
B. De, *API Management*, DOI 10.1007/978-1-4842-1305-6_4

- Document a list of REST endpoints
- Document the message payload
- Provide Response status code and error messages

Audience for API Documentation

API documentation is used by various groups of people for various reasons. It is like a user manual for a product. Like a user manual, API documentation should have a quick-start guide, which quickly makes the first API call and lets consumers have a feel of it. At the next level, it should document the API's features, the resources and the APIs to access them, and finally, the error conditions for troubleshooting. Hence, the API documentation can be used primarily by the following types of audiences.

- **CTO**: Evaluates similar and competing APIs from a business, technology, and monetization perspective.
- **App or integration architect**: Explores the API to match the requirements for building an app or an integration solution.
- **App developer**: Wants to get started using the API with a quick-start guide and a detailed tutorial. Sample SDKs and API calls in the API documentation is of immense use to an app developer.
- **IT support specialist**: Supports the app and is interested in the error and troubleshooting information for debugging any issues with the app.

Model for API Documentation

A good API document communicates all information about the usage of the API— for both humans and machines. The API document should provide all necessary information to app developers or API consumers in a human-readable format. The documentation should help them assess its suitability for use in their client app. It should provide information about its licensing policy and usage requirements-input and output parameters, message format, error messages, and more. Similarly, the API interface should be documented such that its interface can be parsed by a machine to generate client stubs and server-side skeleton code that can be further developed. To make API documentation effective, it should include the following aspects about the API:

- Title
- Endpoint
- Method
- URL parameters
- Message payload

- Header parameters
- Response code
- Error code
- A sample request and response
- Tutorials and walkthrough
- Service-level agreement

Figure 4-1 shows an example of API documentation using Swagger.

```
swagger: '2.0'
info:
  version: 1.0.0
  title: Echo Service
  description: |
    #### Echos back every URL, method, parameter and header
    Feel free to make a path or an operation and use **Try Operation** to test it. The echo server will
    render back everything.
schemes:
  - http
host: mazimi-prod.apigee.net
basePath: /api/echo/v1
paths:
  /:
    get:
      responses:
        200:
          description: Echo GET
    post:
      responses:
        200:
          description: Echo POST
      parameters:
        - name: name
          in: formData
          description: name
          type: string
        - name: year
          in: formData
          description: year
          type: string
  /test-path/{id}:
    parameters:
      - name: id
        in: path
        description: ID
        type: string
        required: true
    get:
      responses:
        200:
          description: Echo test-path
```

Title of API
Endpoint of API
Method of API
Message Payload Parameters
URL Parameter
Response Code

Figure 4-1. *API documentation using Swagger*

Title

The *title* should provide the name of the API, which can be used for its identification.

Endpoint

The *endpoint* is the entry point for the API. It defines the URL that clients need to use to invoke the API.

Method

The *method* defines the HTTP verbs used to access the API. GET, POST, PUT, and DELETE are the most common HTTP verbs used in a REST API. The client should specify the methods along with the URI to access the API. If an API supports multiple methods to be used for an URI, it should be specified in the API document as separate entities, as shown in Figure 4-1.

URL Parameters

The *URL parameters* define the parameter names and their format, which are used in the API call as a query string. The documentation should clearly state the purpose of each parameter, as well as which parameters are mandatory and which are optional. Any requirements for URL encode should be documented.

Message Payload

The *message payload* should specify the structure and format of the request and response message. JSON and XML are the most common formats used for a REST API. Other formats can be used as well. The message structure should specify the schema of the message payload. Any data constraints in the request payload should be documented. It is a good practice to include a table that provides the name, data type, description, and remarks, if any. Figure 4-2 shows a snippet of a Swagger format specification of an API, with the message format for a request and response payload.

```yaml
paths:
  /pets:
    post:
      tags:
        - pet
      summary: Add a new pet to the store
      description: ""
      operationId: addPet
      consumes:
        - application/json
        - application/xml
      produces:
        - application/json
        - application/xml
      parameters:
        - in: body
          name: body
          description: Pet object that needs to be added to the store
          required: false
          schema:
            $ref: "#/definitions/Pet"
```

Request Message Format

Response Message Format

***Figure 4-2.** A snippet of a Swagger format specification with the message format for a request and response payload*

Header Parameters

The *header parameters* should specify the standard and custom HTTP headers included in the request and response headers. At a minimum, all mandatory headers should be specified here. Any specific format for the header values must be included.

Figure 4-3 shows a snippet of an API documentation in Swagger format with the header parameters defined.

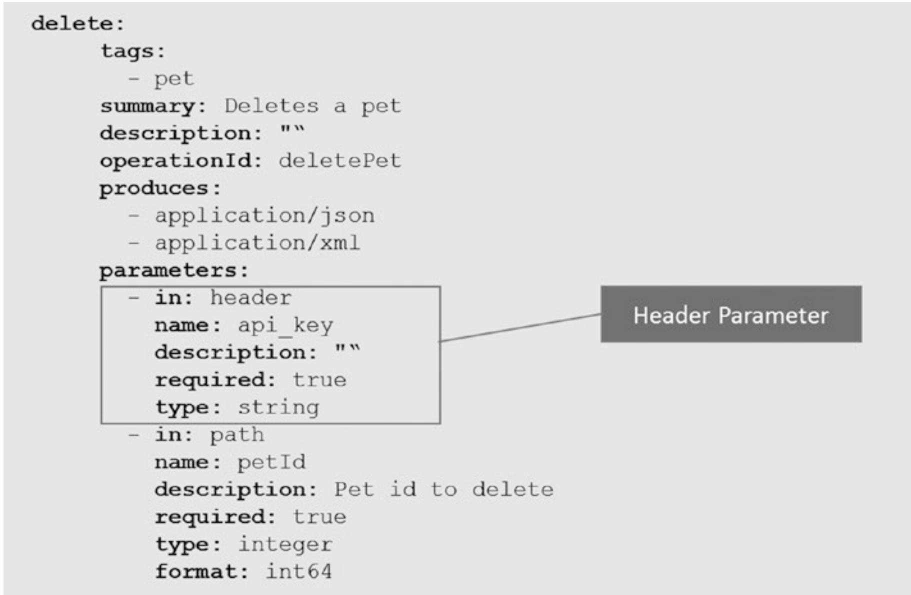

Figure 4-3. *A snippet of a Swagger format specification with the header parameters defined*

Response Code

The HTTP response codes that the client can expect from the API under various conditions should be included in documentation. It is important to document which response codes are considered successful and which are considered errors. All possible response codes, what each of them means, and their root causes should be specified. This helps the API consumer more easily troubleshoot issues.

Error Codes and Responses

Normally 4xx and 5xx HTTP response status codes are considered errors. HTTP specifications define the purpose of these status codes. Not all HTTP response status codes may have been implemented for an API. The API documentation should include the HTTP response status codes that the API consumer can expect in different error scenarios. Along with the HTTP response status code, the sample error response payload should also be specified. This helps the consumer application parse error messages. The error response payload may include specific business error codes and descriptive error messages that offer information about the exact cause of the error. All error codes and error messages should be defined in the API documentation.

Figure 4-4 shows a snippet of a Swagger format specification in an API document, with the response code and error codes defined.

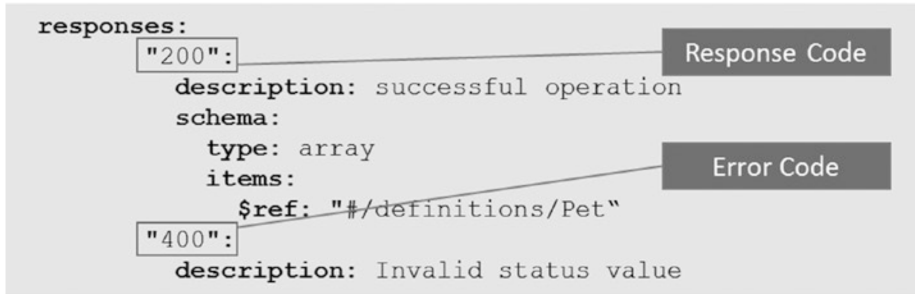

Figure 4-4. *A snippet of a Swagger format specification with the response code and error codes defined*

Sample Calls

As part of the documentation, include sample HTTP calls with all parameters and expected sample responses. This gives the developer a visual sense of what the message structure should look like. Include samples for all the various message formats supported, such as XML, JSON. Sample calls can be included in a wiki format or as interactive smart docs. Figure 4-5 is an example of a sample GET call for an API.

```
GET http://myapi-prod.mycomp.com/api/echo/v1/test-path/{id} HTTP/1.1
Host: myapi-prod.mycomp.com
Accept: */*
Accept-Encoding: gzip,deflate,sdch
Accept-Language: en-US,en;q=0.8,fa;q=0.6,sv;q=0.4
Cache-Control: no-cache
Connection: keep-alive
Origin: http://editor.swagger.io
Referer: http://editor.swagger.io/
User-Agent: Mozilla/5.0 (Windows NT 10.0; WOW64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/54.0.2840.99 Safari/537.36
```

Figure 4-5. *A sample GET call*

Tutorials and Walk-throughs

An example is always better than tons of documentation. Hence, a tutorial with example code on how the API can be called from an app is always very helpful to the developer community. Sample SDKs for making API calls in some of the most popular languages—such as Java, Node.js, C#, PHP, Ruby, and Python—helps developers quickly adopt the API. Including SDKs for different digital platforms—such as Android, iOS, and Windows—is highly recommended. Including code for all languages and platforms may not be always feasible; hence, you should evaluate the most popular languages and focus on including tutorials and sample code for them.

Service-Level Agreements

A service-level agreement (SLA) defines the API's non-functional requirements. This can include the expected throughput, response time, rate limits for various tiers (if applicable), maintenance or downtime information, and so forth.

API Documentation Standards: Swagger, RAML, and API Blueprint

The daunting task of API documentation is keeping the documentation in sync with the actual implementation. If you take a bottom-up approach and create the API documentation manually after the implementation, you risk the documentation falling out of sync if there are enhancements to the API interface in the next version, especially if the process does not enforce regeneration or validation of the API document. Similarly, with a top-down approach, you may start with the API documentation and manually create the skeleton of the API interface according to the defined interface. But later, you still run the risk of the API documentation getting out of sync with the actual implementation when enhancements are required. Hence, defining the API Interface and keeping the documentation in sync is a big challenge. This challenge can only be addressed if there are tools that autogenerate API documentation from the API Interface in a bottom-up approach, or tools that generate the API skeleton and client code from the API interface document in a top-down approach. Standards and tools based on these approaches are definitely needed to aid in API documentation.

There are many competing tools for API documentation. Some of them are in a fairly matured state, while others are still evolving. The next few sections look at the Swagger, RAML, and API Blueprint frameworks so that you can see how they are used to document an API interface. The tools that they provide are also discussed.

Swagger

Swagger is one of the most popular API documentation frameworks. It provides a standard, language-agnostic way of defining a REST API interface. This approach allows the client to understand the capabilities of the REST service without any prior access to the service implementation code or network inspection. The goal of Swagger is to keep the client, API documentation, and API server implementation in sync. The Swagger framework comes with the following:

- **OpenAPI Specifications** for documenting RESTful APIs
- **Swagger UI** to graphically display the API interface
- Tools like **Swagger Codegen** to generate clients for different languages

On January 1, 2016, the Swagger specification was renamed the OpenAPI Specification when it was donated to the Open API Initiative.

The Swagger specification defines the format to describe the REST API. The Swagger specification describes the following information about the REST API.

- URL endpoint
- HTTP verbs supported for the URI
- Description
- Query parameters to be passed in the URL
- Header parameters in the input request message
- Payload format and data type for the request and response messages
- HTTP response status codes
- Security requirements
- Vendor extensions

An API's Swagger specifications are documented in JSON or YAML format. Swagger-enabled APIs expose JSON files that adhere to the specifications. The API specification can be a manually generated static file or automatically generated from the application. The API specification needs to be documented in a certain way as per the structure defined by the Swagger specifications. We look at the high-level structure of a Swagger definition file for an API later in this chapter.

The **Swagger UI** provides a framework to dynamically generate beautiful interactive documentation from a Swagger-compliant API. It is an independent collection of HTML, CSS, and JavaScript that can be hosted on any server. It can be used by consumers to explore and interact with the API and understand its behavior. API users can use the Swagger-UI to test the API and learn about how the API responds to various parameters and options. The Swagger-UI can be used as-is or customized to meet an organization's needs. The Swagger-UI provides support for HTTP verbs, such as GET, POST, PUT, DELETE, PATCH, and OPTIONS for API invocation. `Authorization` and custom HTTP headers can be added to the Swagger-UI. It also provides support for localization and translation.

Swagger Codegen provides tools for generating code for the client and server from the Swagger-defined API spec. It is a command-line tool that can generate code for different languages and frameworks. Some of the frameworks that are supported for server stub generation are Node.js, PHP Silex, Python Flask, Scala Scalatra, Java JAX-RS, and Java Spring MVC. Client-side stubs can be generated for languages such as Scala, Java, JavaScript, Ruby, and PHP—to name a few. Refer to the documentation at `https://github.com/swagger-api/swagger-codegen` for more information on this tool.

The **Swagger Editor** can be used to edit the Swagger API specification in YAML format. The editor is opened in a web browser to edit the specification and preview the API documentation in real time.

Generating Swagger Specifications

Both top-down and bottom-up API development processes can be adopted to create a Swagger-enabled API.

- **Top-down approach**: Creates a Swagger definition using Swagger Editor and uses integrated Swagger Codegen tools to generate server implementation.
- **Bottom-up approach**: Uses Swagger Editor to manually generate Swagger definitions, or autogenerates Swagger definitions created using Swagger supported tools, such as JAX-RS or Node.js.

The Swagger File Structure

The Swagger-based API specification is documented in JSON or YAML format. The Swagger specifications include the following in the API interface.

- `swagger`: Describes Swagger specification version
- `info`: Shows the API's metadata
- `host`: The hostname or IP serving the API
- `basePath`: The base path on which the API is served relative to the host
- `schemes`: The protocol that the API uses
- `consumes`: The MIME type for the API's input data type
- `produces`: The MIME type for the API's output data type
- `paths`: The API's available paths and operation
- `definitions`: Holds the data types produced and consumed by the operation
- `parameters`: Describes a single operation parameter
- `responses`: Describes the schema of a single response of an API operation
- `securityDefinitions`: Describes the security mechanisms without enforcing them on the operation
- `securitySchemes`: Specifies the security definitions that are enforced for the API's operation
- `tags`: Specifies any additional information on the API
- `externalDocs`: Specifies links to any additional external documentations related to the API

For more information on each of these Swagger objects, please refer to the Swagger specification at https://github.com/OAI/OpenAPI-Specification/blob/master/versions/2.0.md.

Table 4-1 lists the various major Swagger tools available at the time of this writing. OpenAPI community is constantly generating new tools to expand the reach of Swagger for different platforms in API development.

Table 4-1. *Swagger Tools*

Tools	Description
Major Tools	
Swagger Core	A set of Java libraries for generating and consuming Swagger definitions built around JAX-RS.
Swagger Codegen	Tools that generate client libraries and server stubs based on the Swagger definition.
Swagger UI	A browser-based UI for exploring Swagger-defined APIs.
Swagger Editor	A browser-based editor for authoring Swagger definitions in YAML or JSON format.
Other Tools	
Swagger JS	A JavaScript client for use with Swagger-enabled APIs.
Swagger Node	Tools for designing and developing Swagger-compliant APIs entirely in Node.js.
Swagger Socket	This protocol allows any existing REST resources to be executed on top of the WebSocket protocol.
Swagger Parser	A standalone library for parsing Swagger definitions from Java.

RAML

RAML stands for *RESTful API Markup Language*. It is a Markdown-based language for modeling APIs. It makes it easy to manage the entire lifecycle of an API: design, build, test, document, and share. RAML is both machine-readable and human friendly. RAML is designed to support an API-First top-down development approach. It provides the format for the contract between the API provider and the API consumer.

Why RAML?

RAML designs API interfaces that are developer- and user-friendly. Using RAML, API interfaces can be designed, tested, and shared with users to get feedback without writing a single line of code. APIs can be described in a human-readable text format. Tools like API Workbench and API Designer provide visual design. The RAML construct lets you reuse libraries, code, and design patterns in the API design, which saves lot of work.

RAML code generation tools create server-side code from the spec for different languages, such as Node.js, Java, JAX-RS, .NET, Python, Mule, IoT, and others. A RAML specification can also generate test cases for the API using many of the open source and commercially available API testing tools. This takes advantage of the test- driven development approach and generates test cases than can be integrated with the continuous improvement process.

API documentation can be easily generated dynamically on the fly from RAML with tools such as API Console, RAML2HTML, API Notebook, and others. This keeps the API documentation in sync with the implementation. API definitions in RAML can integrate with other systems. Many open source and commercial tools available today can generate SDKs for different languages from RAML definitions.

Professional services such as APIMatic.io and REST United offer to generate up to two SDKs at no cost. Oracle and MuleSoft provide built-in functionality in their API management products to import the RAML definition, which automatically pulls in the API resources, methods, and other properties, thus avoiding any manual setup of API calls in these tools. With API Notebook, developers can use RAML definitions to create interactive API walk-throughs and sample use cases using simple JavaScript.

More information on API Notebook is at `https://api-notebook.anypoint.mulesoft.com`.

RAML Structure

This section looks at the high-level structure of a RAML-based API specification document. The detailed schema and syntax can be found in the RAML specifications document at `https://github.com/raml-org/raml-spec/blob/master/versions/raml-10/raml-10.md`.

A RAML API specification can be structured and organized into the following sections.

- **Security scheme information**: Describes the basic information about the API, such as *title* to identify the API, its *version*, the *baseURI* to specify its network location for invocation, supported *protocols*, and *default media type* and *security* requirements. Any user documentation that can serve as user guide or reference documentation can also be optionally included in this section.

- **Data type**: The data type is used to describe any data that is passed as a parameter for the API. A parameter can be in the URI as query parameters, in the header as header parameters, or in the request/response body. The data can be described using built-in types or by a new custom type definition created using a combination of the built-in data types. The data types can be defined using XML or JSON schemas, or RAML types. These can coexist as well. For more information on the data types definitions allowed in the RAML spec, please refer to `https://github.com/raml-org/raml-spec/blob/master/versions/raml-10/raml-10.md#raml-data-types`.

- **Resources**: Specifies how to access the API's resources and subresources using URIs relative to the `baseURI`. The resources definition should start with a slash (/). A resource defined at the root level is called the *parent resource*. Each parent-level resource is identified using its own URI relative to the `baseURI`. Each parent resource may optionally have one or more child resources defined under it. This approach builds and defines a nested hierarchy of resources. The relative URI of a resource may consist of multiple URI path fragments separated by slashes; for example, `/cart/items`. This approach can be used only if an individual path item is not a resource. If the path items are themselves separate resources, then they should be defined as nested subresources. These is no limit to the level of resource nesting that is allowed. Template URIs containing URI parameters can be used when the resource identifier is a variable; for example, `/products/{productId}`. For more information on defining resources and nested resources using RAML, refer to https://github.com/raml-org/raml-spec/blob/master/versions/raml-10/raml-10.md#resources-and-nested-resources.
- **Method:** Describes the allowed HTTP verbs and the request parameters that can be used to manipulate a resource. The HTTP verbs that can be specified are GET, POST, PUT, PATCH, DELETE, HEAD, and OPTIONS. Each method can have an optional friendly name and description to describe its functionality. Any optional or mandatory HTTP headers required for a method should be specified under this section. The structure and any constraints or patterns of the header parameters that the API consumer should be aware of needs to be specified here. Similarly, any optional and mandatory query parameters to be passed as query string should also be specified. The request body for POST and PUT methods can be optionally described in this section. For more information on using method in a RAML definition, please refer to https://github.com/raml-org/raml-spec/blob/master/versions/raml-10/raml-10.md#methods.
- **Response:** Contains the schema and description of the response object received from the service for a method invocation. The response has two main sections: header and body. The header describes the possible HTTP status codes expected in the response header under various conditions. The body is optional and describes the media type and the structure of the message payload included in the response body. The structure can be defined using types defined in the data type section. For more information on the response definition, refer to https://github.com/raml-org/raml-spec/blob/master/versions/raml-10/raml-10.md#responses.

- **Resource types and traits:** Resource types and traits define reusage patterns and resources across the RAML definition. You may want to define the pattern for a HTTP header, or a query parameter, or a message payload, and then reuse it at different places within the RAML definition. A *resource type* is a partial definition of a resource that can specify security schemes, methods, and other properties. A resource that uses a resource type inherits its properties. A resource type can use another resource type. *Traits* are similar to resource types. The difference is that a trait is a partial method definition. It can be used to define method parameters, such as headers, query strings, and responses. Resources and resource types can also use and inherit from one or more traits. For more information on resource types and traits, refer to `https://github.com/raml-org/raml-spec/blob/master/versions/raml-10/raml-10.md#resource-types-and-traits`.
- **Security:** The API security scheme definition is specified in this section of the RAML definition. This section defines the OAuth, Basic Authentication, or Digest Authentication mechanism for API security. Any other forms of authentication can also be specified using `x-<other>` headers. Any headers or query parameters that pass through can also be specified in this section under the '`passthrough`' attribute. If any API method requires a special security mechanism, it can be specified by the '`securedBy`' attribute for that method. This overrides whichever security scheme has been applied to the API as a whole. If a method does not require any security scheme, it can be specified by the '`securedBy`' attribute for that method with a `null` value. Multiple security schemes for a method can also be specified using the `securedBy` element with an array list of security schemes.

RAML Tools and Projects

RAML is supported by the developer community with a long list of tools and projects that address different API needs. These tools address different aspects of the API development lifecycle—from designing the API spec to sharing it with the broader community. New tools are also evolving to address newer requirements and languages. Some of the languages supported at the time of this writing are Java, JavaScript, .NET, Ruby, Node.js, Python, Go, and Haskell. This section briefly previews the most commonly used RAML tools.

- **API Workbench** is a tool by MuleSoft that provides a full-featured integrated development environment (IDE) that design, build, test, document, and share RESTful APIs. Using API Workbench, you can create RESTful APIs using a simple *design-first* approach based on RAML specifications. It supports both RAML 0.8 and

RAML 1.0 versions of the specifications. This tool is based on Atom code editor developed by GitHub. The following are some of the main features of API Workbench.

- An IDE that supports autocomplete, advanced search, live debugging, and symbol based navigation
- Dynamic generation of API Mocking Service and API Console
- Wizard-driven creation of API definitions based on RAML specifications
- Automatic validation of RAML-based API definitions
- Built-in support for integration with Git for source control and versioning
- An integrated scripting engine and tooling for API testing and documentation

- **API Designer** is a tool from MuleSoft that allows a user to see real-time post-processing of their API definition. It provides three panels with areas to organize RAML files and folders, displays the contents of the document, and offers an interactive text editor. The text editor provides features such as autocomplete, export, and contextual tag lists. It also saves the API definition. For more information, please refer to `https://github.com/mulesoft/api-designer`.
- **Restlet Studio** provides a lightweight integrated development environment that can help accelerate API design. Built using Angular.js, it provides a web-based UI for API design. The following are some of the main features for Restlet Studio:
    - Visual web-based editor to define and edit API definitions for endpoint, resources, methods, and so forth.
    - The ability to group resources and representations into sections with scrollable a navigation panel helps extend support for even the most complex API definition
    - A built-in language translator switches between Swagger and RAML definitions
    - Generates server skeleton code using a built-in code generator service based on APISpark and Swagger
    - Generates client SDKs using a built-in code generator service based on APISpark and Swagger.
    - For more information on Restlet Studio, please refer to `http://restlet.com/products/restlet-studio/#`.

- **API Notebook** is another RAML tool by MuleSoft. It helps with live testing and exploring APIs. An API RAML definition can be imported into Notebook to create a client for the API, send requests, and view responses. Notebook's autocomplete feature explores the API. Once a RAML definition for the API has been imported, the method definitions will appear in the tool-tip hints. The path segments of the API resource, separated by slashes (`/`), become nested JavaScript objects; for example, `/my/myresource` becomes `{clientName}.my.myresource`.

- **RAML for JAX-RS** provides a set of tools that generate a Java + JAX-RS-based application from a RAML API definition. It also provides roundtrip support by doing the reverse to generate a RAML API definition from an existing Java + JAX-RS definition.

- **Abao** provides a REST API testing tool for APIs defined using RAML. It tests the RAML definition for the API against the back-end implementation. This tool can be integrated with continuous integration (CI) tools such as Jenkins to test API documentation and keep it up-to-date. It uses the Mocha framework to test the validity of the API response. The following are some of its features:

    - Validates the API endpoint definition
    - Validates that each URL parameter defined in the RAML API spec is supported in the back-end service
    - Validates that each HTTP request and response header parameter defined in the RAML API spec is supported in the back-end service
    - Validates that the JSON schema for the request and response payload meets RAML specifications

- **RAML Tools for .NET** provides a Visual Studio extension for RAML-based APIs. It allows you to easily integrate and consume APIs defined using RAML and to create a new ASP .NET REST API implementation from a RAML definition using a design-first approach.

There are many other tools available as RAML projects that address the various needs of the developer community building and consuming REST APIs from a RAML definition. Based on the functionality, these tools can be primarily categorized by design, prototype, build, frameworks, test, document, share, parser, and converters. Most of the design tools provide a visual interface or plugins that can be used with other visual editors to design the RAML definition for the API. The prototype tools can be used to mockup response for APIs defined using RAML. They can test the API interface and create stubs as a replacement for the actual implementation for testing purposes. The build tools and frameworks generate the client SDK and server skeletons based on the RAML definition of various languages. This promotes the design-first approach for API development.

These tools test API documentation and implementation. They can generate test cases for APIs based on the RAML definition for the API. They validate the RAML definition against the actual implementation and thus keep the two in sync. The documentation tools create API documents in various formats; graphical API consoles, HTML, wikis, PDFs, and other formats can be shared with API consumers. The parser tools are libraries for different languages, which parse the RAML definition of the API. The converters convert the RAML to other API specification formats, such as Swagger.

Differences in RAML Specification Versions

At the time of writing, RAML has two versions: RAML 0.8 and RAML 1.0, which is in a release candidate (RC) state. RAML 1.0 provides more extensibility, code reuse, and flexibility features than the previous version. It introduces new features, such as libraries, overlays, improved security schemas, data typing, annotations, and enhanced examples.

- **Libraries** let you to include predefined sets of data types, resource types, security schemas, traits, and reusable assets.
- **Overlays** give the flexibility to include new information, such as descriptions, examples and annotations, which are defined in a different RAML file; for example:

  ```
  #%RAML 1.0
  types: !include myTypes.raml
  ```

- **Security schemas** have been enhanced to provide better support for OAuth 1 and 2, a new API key, and new custom security schemes.
- **Data types** can be used in place of schemas and examples. The data type definition can be converted to XML or JSON format on the fly using some of the RAML tools. API designers can thus define only the data type for their input and output parameters and RAML takes care of the REST.

API Blueprint

API Blueprint is a document-oriented language for describing REST API using Markdown syntax. This specification, brought in by Apiary.io, uses Markdown syntax to describe the complete specification of an API or its parts.

API Blueprint Document Structure

An API Blueprint document is structured into logical *sections*. For example, headers, URL parameters, and request/response can each be described in logically grouped sections. Each section is defined by predefined keywords. Depending on the section, the keyword is written either as a Markdown header entity or a list item entity. The following are the reserved keywords for defining the header and list entities in an API Blueprint document:

- Header keywords
  - Group
  - Data structure
  - HTTP methods
  - URI templates
  - Combination of HTTP methods and URI templates
- List keywords
  - Request
  - Response
  - Body
  - Schema
  - Model
  - Header and headers
  - Parameter and parameters
  - Values
  - Attribute and attributes
  - Relation

At a high level, the API Blueprint description for a REST API is organized in the following structure:

- **Metadata** describes the version of the API Blueprint specification used for documenting the API interface. It also contains the API name and a brief description.
- **Resource and resource group** describe the resources and the group of related resources used by the API. For example, in an online shopping experience, a customer may have one or more orders. So *orders* is defined as a resource group; within this group there can be a resource that returns a collection of orders.

    - **Actions** describe the operations that can be performed on a resource. It is specified using one of the HTTP verbs within square brackets.

- **URI templates** specify the variable parameters in the URI; for example, an order may be identified using an order ID. So to get the details of a specific order, you specify it using `/orders/{order_id}`.

    - **URI parameters** describe the variables being passed in a request URI or as a query parameter; for example, `/path/to/resources/{varone}?path=test{&vartwo,varthree}`.

For more information on each of these keywords, please refer to the API Blueprint Specification document in GitHub at https://github.com/apiaryio/api-blueprint/blob/master/API%20Blueprint%20Specification.md#def-api-blueprint-language.

API Blueprint Tools

The API Blueprint spec is supported by a good number of tools. This section discusses some of the most commonly used API Blueprint tools.

- **Apiary.io** provides a comprehensive tool that supports collaborative design, creation of API mockups, automated testing, autogeneration of interactive API documentations, API traffic inspection, and more.

- **Dredd** is an HTTP API testing tool. It is a command-line tool that can be used to test the API documentation written in API Blueprint against the back-end implementation. This tool can be integrated with CI tools to ensure that API documentations are always up-to-date.

- **Drakov** provides a Node.js implementation of a mock server for APIs written using API Blueprint.

There are many other open source tools that support the API Blueprint format for SDK generations. They have various language formats, testing API interfaces, and plugins for API test clients; they can also convert API definitions from other formats to API Blueprint and vice versa. Since API Blueprint and its tools are open source, they can be freely integrated with all kinds of products to extend support for API Blueprint.

Comparing Swagger, RAML, and API Blueprint

Tables 4-2, 4-3, and 4-4 compare Swagger, RAML, and API Blueprint.

Table 4-2. *Overview Comparison*

Criteria	Swagger 2.0	RAML	API Blueprint
Format	JSON, YAML	YAML	Markdown
Availability on Web	GitHub	GitHub	GitHub
Primary sponsor	Reverb	MuleSoft	Apiary
Is there a workgroup?	Yes	Yes	No
When was it first committed?	July 2013	September 2013	April 2013
Design approach	Top-down and bottom-up	Top-down	Top-down
Current version	2.0	1.0	A4

Table 4-3. *Tool Support*

Criteria	Swagger 2.0	RAML	API Blueprint
Authoring tool	Swagger.io	API Designer	Apiary.io
Ad-hoc testing	Swagger UI	API Console	Apiary.io
Documentation	Supported	Supported	Supported
Mocking	Extended support provided by third party	Extended support provided by third party	Extended support provided third party
Server code	Supported by third party	Supported by third party	Supported by third party
Client code	Supports multiple languages	Supports multiple languages	Supports a few languages
Generate from code	Supported by Java (third party)	Supported by third party	Supported by third party
Validation	Supported	Supported	Supported
Parsing	Java.js	Java.js	C++(Node.js, C#)

Table 4-4. *REST Modeling Capabilities*

Criteria	Swagger 2.0	RAML	API Blueprint
Resources	Supports resource definition	Supports resource definition	Supports resource definition
Nested resources	Supports nested resource definition	Supports nested resource definition	Supports nested resource definition
Representation metadata	Supports the JSON schema	Supports inline and external definitions in any format	Supports only inline definitions in any format
Composition/ inheritance	Inheritance supported by sub types	Supports inheritance of traits and resource types	Supports resource model inheritance
API version metadata	Supported via `apiVersion` tag	Supported via `version` tag	No explicit tag to specify the API version
Authentication	Has tags defined to support Basic, API Key, and OAuth2	Supports Basic, Digest, and OAuth2	Supported via custom header definitions
Methods/action	Supported	Supported	Supported
Query parameters	Supported	Supported	Supported
Path/URL parameters	Supported	Supported	Supported
Header parameters	Supported	Supported	Supported
Documentation	Supported	Supported	Supported

Other API Documentation Frameworks

Swagger, RAML, and API Blueprint are the most popular API documentation standards. Most API management vendors include support (in various forms) in their tools for one or more of these languages. However, these specs are still evolving. In parallel, there have been efforts from various corners to create specs that address competitors' shortcomings. As a result, many competing specs are available today from various vendors. These are in various levels of maturity. The following is a list of some of the other API standards used for modeling and documenting APIs:

- WADL
- ioDocs from Mashery
- Doxygen
- ASP.NET API Explorer
- Apigee Console To-Go
- Enunciate
- MireDot
- Dexy
- Docco
- TurnAPI

CHAPTER 5

API Patterns

APIs should be designed for longevity. Any change to an API carries the risk of breaking the client's application code. Frequent changes to an API frustrate the developers and the consumers using it. Building APIs from robust and proven patterns fosters a happy developer community and saves the company a lot of money. This chapter looks at some of the API design principles and patterns that have stood the test of time and make developers happy.

Best Practices for Building a Pragmatic RESTful API

APIs are the face of your enterprise. They provide users with access to enterprise data, services, and assets. Hence, while security should be ingrained in it, the API interface should be simple and elegant to attract developers. It should be intuitive and developer-friendly to make adoption easy and pleasant. Adherence to web standards is equally important. APIs should be designed with user experience in mind. Many of these principles were covered in earlier chapters. The following summarizes some of the approaches for designing a pragmatic RESTful API interface.

- **Design APIs with RESTful URLs**. Design an API based on the logical grouping of identified resources. The API URL should point to either a collection of resources/subresources or an individual entity within the collection. For example, `/customers` should refer to a collection of customers, while `/customers/{customerId}` should refer to an individual customer entity with in the collection. The URL should be intuitive enough to identify the resources and navigate through them easily.
- **Use HTTP verbs for CRUD action on resources.** Use the HTTP verbs to perform CRUD action on the resources. Use POST to create a new resource, GET to read, PUT to update, and DELETE to delete a resource. Additionally, you may consider providing support for the PATCH verb in the API resource for partial updates. OPTIONS verb can be used to determine the metainformation about the resource, such as the methods supported, HTTP headers allowed, and so forth.

© Brajesh De 2017
B. De, *API Management*, DOI 10.1007/978-1-4842-1305-6_5

- **Use operation in the URL when HTTP verb cannot map to the action.** Often, an action on a resource cannot be directly mapped to an HTTP verb. For example, actions such as `register`, `activate`, and so forth, cannot be directly mapped to an HTTP verb. These operations may be applicable on a resource collection or a single resource entity, or to a group of resources of different types. In such cases, it makes sense to have this operation in the URL and treat as a subresource. For example, the resource URI `/customers/customer123/activate` can be used to activate the account of customer with ID `customer123`.

- **Use SSL/TLS for all communications with REST APIs**. RESTful APIs expose enterprise data and assets. These can be accessed from within the company or from outside the firewall over the Internet from anywhere. This poses a security threat to the data transferred over the network. Hence, to protect the data against any eavesdropping or any impersonation in case security credentials are compromised, it pays off to use SSL/TLS for all API communication. Using SSL communication also simplifies authentication efforts. Mutual authentication with SSL/TLS can help the server to validate the identity of the client in addition to the client validating the server.

- **Do not redirect from non-SSL API endpoints to SSL endpoints**. This is a practice to always avoid when designing REST APIs. Malicious clients may gain access to actual secured and encrypted API resources through such redirections. It is recommended to respond with a proper error message if the non-SSL endpoint is not supported in the API.

- **API versions**. Versioning iterates and improves APIs by providing a smooth transition path. It supports multiple versions of the APIs simultaneously and provides time for clients to upgrade to new version and provider to retire the old version. There are multiple approaches to versioning the API. The most common of them is to include the version information in the URI base path. Version information can also be included in custom HTTP header. A hybrid approach of including the major version in the URI and minor version in the HTTP header can also be adopted. Information about API versioning approaches are covered in Chapter 6.

- **Design the API interface to support filtering on the result set**. The response to a GET request for an API resource may sometimes be quite large. Displaying this large response in the consumer app may be quite challenging considering the limited form factor and processing power on the device. Also transmitting a large payload over the network would also impact the bandwidth and the overall performance. Hence, the client app using the API would obtain a lean and filtered response for a GET request. This can be achieved only if the API supports filtering on the result set. Filtering criteria may be specified as unique query parameters for each field that supports filtering. For example, when querying for a customer's orders, you may want to limit it by the order date, such as orders placed in the previous month, six months, or year. This can be specified using a GET request with a `orderDate` such as `GET /customers/customer123/orders?orderDate>'YYYY-MM-DD'` query parameter.
- **Design the API to support pagination**. Pagination is yet another feature that is useful in handling large responses from an API. Even the filtered response from the back-end service for an API may contain hundreds of records. In such a scenario, it makes sense to display only ten of them on a page in the consumer app and provide a link to the next page with the next set of records. Supporting pagination for an API response can address this need to the app developer. This includes pagination parameters in the request as query parameters. `'limit'` and `'offset'` are the most commonly used query parameters to specify the pagination requirements. For example, `orderDate` is like `GET /customers/customer123/orders?orderDate>'YYYY-MM-DD&limit=5&offset=0`. `'limit'` indicates the number of records to be included in a page and `'offset'` denotes the page number. Also the API response should include the pagination metadata in the response. This can be included in human-readable format as envelope within the response or in a machine-readable format using the `Link` header. Following is an example of pagination metadata included as an envelope within the response:

```
"_metadata":
  {
      "offset": 2,
      "limit": 5,
      "page_count": 25,
      "total_count": 127,
      "Links": [
        {"self": "/orders?offset=2&limit=5"},
        {"first": /orders?offset=0&limit=5"},
        {"previous": "/orders?offset=1&limit=5"},
```

```
        {"next": "/orders?offset=3&limit=5"},
        {"last": "/orders?offset=25&limit=5"},
      ]
  },
  "orders": [
    {
      "id": 1,
      "item-name": "Widget #1"

    },
    . . . . .
    . . . . .
    ]
}
```

Machine-readable metadata can be included by using the `Link` header, as follows:

```
Link: </orders?offset=2&limt=5>;rel=self,</orders?offset=0&limit=5>;rel=fir
st,</orders?offset=4&limit=5>;rel=previous,</orders?offset=3&limit=20>;rel=n
ext,</orders?offset=25&limit=5>;rel=last
```

The total count of records can be included in the response using custom headers such as `X-Total-Count`.

- **Return resource representation in response to creating and updating.** The response for POST, PUT, and PATCH operations results in creating and/or updating a resource. The API response payload for these methods should include metainformation such as `'created_at'` or `'updated_at'` along with the created or updated representation of the resource. Successful execution of a POST request should return a HTTP *201 Created* status code along with the `'Location'` header containing the URL of the newly created resource. Successful update requests should return with HTTP status code 200 OK.
- **Use HTTP headers to specify the media type for the message payload.** When sending a request or a response, include the `'Content-Type'` header to specify the content type for the message payload. This helps the message recipient to easily identify the parser to be used for processing the message. For example, the `'Content-Type'` header with the value *`application/json`* indicates that the payload is in JSON format and the recipient should use a JSON parser to process the message. Similarly, use the `'Accept'` header in the request to indicate the format of the response expected by the consumer from the provider.

- **Use HTTP headers to support caching.** HTTP provides built-in features to support caching. HTTP provides a number of useful headers to efficiently communicate caching information. The `'ETag'` header contains a hashed value of the resource information. Instead of including the entire resource representation in the message payload in XML or JSON format, the `'ETag'` header with the hash value can be used to communicate information about the resource. Similarly, headers can be used to communicate resource modification date/time, expiry time of the cache, validation rules, and the cacheability of a resource. These headers should be used effectively for handling cache.
- **Secure APIs using authentication information in HTTP header**. APIs exposing data and assets should always be secured. Authentication credentials should be included in the HTTP `Authorization` header. Since REST services are stateless, cookies should not be used. Session information if any should be passed in custom HTTP headers. Basic Authentication should be used when the API needs to identify the end user. OAuth-based authentication can be used when a third-party application needs to access the API on the behalf of another user. In case of authentication failure, the API should respond with *401 Unauthorized*. All communications containing sensitive information or any security credentials should be encrypted using TLS.
- **Handle Errors using HTTP Status code and appropriate error messages**. In case of errors, APIs should respond with useful error messages in a consumable format. Appropriate HTTP error response codes in 4XX or 5XX series should be used. 4XX response codes should be used if the error occurred due to fault of the client. 5XX response codes should be used in case of server errors in processing the request. The error response payload should at a minimum communicate the following:
    - **Error message code**: A unique alphanumeric code to uniquely identify the error.
    - **Error message**: A brief summary of the error.
    - **Error description or reason**: A description of or reason for the error.

This is an example of an error response payload:

```
{
  "code" : Err_POL0001,
  "message" : "Address missing",
  "reason" : "No Address specified in the 'Address' field"
}
```

API Management Patterns

Enterprise services provide access to assets and legacy systems. SOAP and REST APIs are the two most common implementation technologies used for building services. An API management platform is used to transform and manage these services to make them more flexible, scalable, and secure. Various implementation patterns have emerged to help address different challenges. This section looks at some of the most common API management patterns.

API Facade Pattern

The API facade pattern helps the API team create developer-friendly API designs and connect to complex enterprise legacy record systems.

Back-end and internal system of records are often quite complex. They could be built using variety of technologies and sometimes even legacy ones. These are difficult to change due to the strong dependencies built over time and the complexity involved. A lot of investment was made to make them robust and stable. Hence, it is difficult to replace them as well. Therefore flexibility and agility needed for the digital business becomes difficult to achieve. Creating an API for a single system of record may still be possible, but the real problem is in creating an API for a group of complementary systems that needs to be used to make an API really valuable to the developer. In this situation, API facade patterns come in handy for creating a simple API interface for a set of complex back-end systems that are hard to change for digital transformation. This pattern provides a layer between the back-end systems and the consumer apps. This layer not only build a simple API interface but can also implement other functionalities such as security, data transformation, version management, orchestration, error handling, routing and much more. Apps access the API exposed through this facade. The facade handles the complexities to interact with the back-end systems (see Figure 5-1).

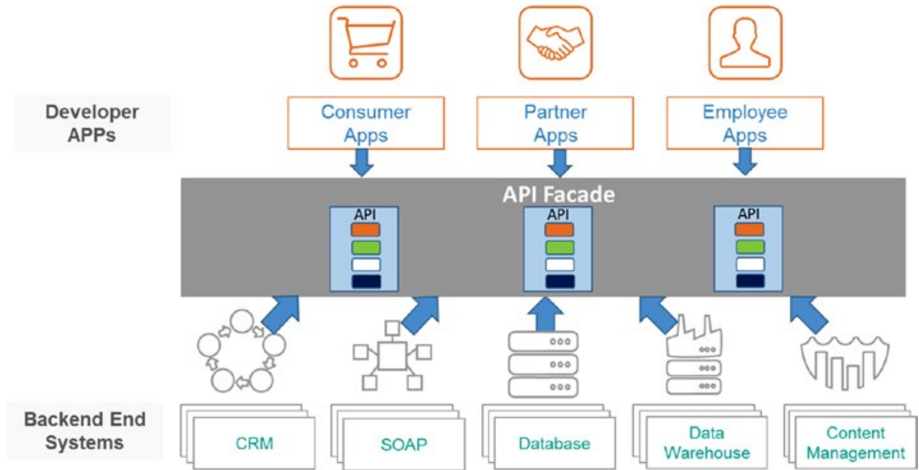

Figure 5-1. *API facade pattern*

The API Facade can be used in a variety of ways as described in the next few sections.

API Composition

Take for example that an app needs to interact with three different services for one of its transactions (see Figure 5-2). In this case, the client app has to be built so that it makes multiple calls directly to services, negotiates any security challenges, and does data format changes as required. With this approach, the client app is responsible for all the orchestration, data transformation and normalization, security, service connectivity, and retry mechanisms. This is indeed an overhead on the client app, considering the limited processing power available on the mobile devices. It is helpful for the developer building the app, if all these tasks can be off-loaded somewhere.

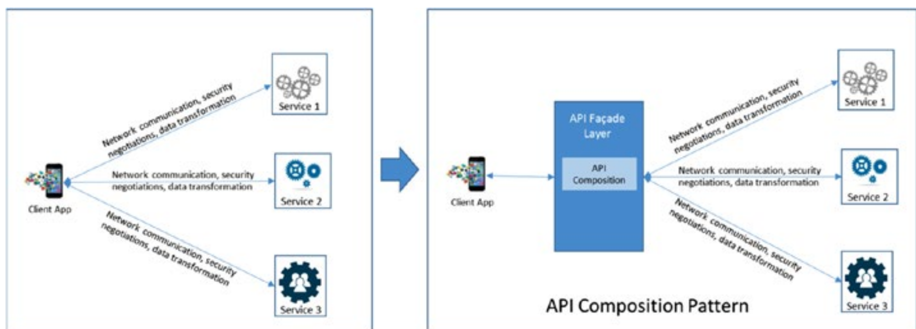

Figure 5-2. *API composition pattern*

Implementing the API composition pattern in the API facade layer can be a solution to this problem. With API composition, the developers can concentrate on the UI and business functionality. It makes the communication less chatty with reduced network calls between the app and the back-end services. Security negotiations with the back-end service are handled by the API composition at the facade. The client app or device only needs to authenticate once at the API facade layer. API composition also shields the client from changes to the back-end systems. Different service provider can be plugged in without having to change the app. The new API composition can help validate and throttle requests before it reaches the back-end. Any data format change or intermediate message processing can be done using this pattern. The API composition pattern can also help improve the overall performance by bringing in some parallelism in making calls to the back-end system.

Using an *API gateway* to implement the API facade pattern for composition is a common practice. An API gateway is a server that acts as a single point of entry into the system. It encapsulates the internal system architecture and provides an API that is customized for the client. It handles the responsibilities of request routing, orchestration, protocol translation, and finally, composing an interface as required by the client. The client communicates with the API gateway, which then fans out the request to all the back-end APIs. It invokes multiple back-end microservices and aggregates their responses. It also does the translation between web protocols such as HTTP(s), WebSockets, and other web-unfriendly protocols used within the enterprises. The API gateway provides an interface that is customized for the client's needs by following the composition pattern. It reduces the client overhead for making multiple calls to different services and aggregating them, thus simplifying the client code.

Session Management

API services should be designed to be stateless. But sometimes state management becomes necessary for designing an app with better user experience. Shopping cart, hotel booking are some examples where session management is necessary. Sessions maintain the client context on the server. In the API world, managing session information in the client apps running on devices is difficult. Devices are already constrained for memory and processing capacity. Hence, session management is an additional overhead, which can slowdown the overall performance. Managing the state information in the back-end server is expensive too. An API facade can use HATEOS principles to facilitate state management. Using these principles, the resource state information can be returned in the response payload as a URI from the facade. This URI can be used by the client in subsequent interactions to communicate the state of the resource. For example, in the shopping cart API, the GET request to fetch product information by a user may look like the following:

```
GET https://www.foo.com/products/sku/2345?user=USR123&cart=CT1234
```

The response for this request can be as follows:

```
{
"Product":{
    "item-name":"Canon EOS 5D Mark III",
     "description":"DSLR camera",
      "price": "2500 USD",
      "sku": "2345",
    "link":{
       "AddPrdURL":
        https://www.foo.com/cart/CT1234/product/sku/2345?user=USR123
       }
    }
}
```

Note that the response contains a URL that has the cart id (`CT1234`) and the user id (`USR123`), which acts as the session information. The app can use this URL for the next call to add the product to the user's cart. The session information can also be communicated as custom HTTP response headers.

Two-Phase Transaction Management

In a two-phase transaction, the transaction coordinator *prepares* the participating resources for a transaction in the first step. If the first step is successful, the *commit* is issued to the participating resources in the second step. The two phases for a two-phase commit transaction is shown in Figure 5-3.

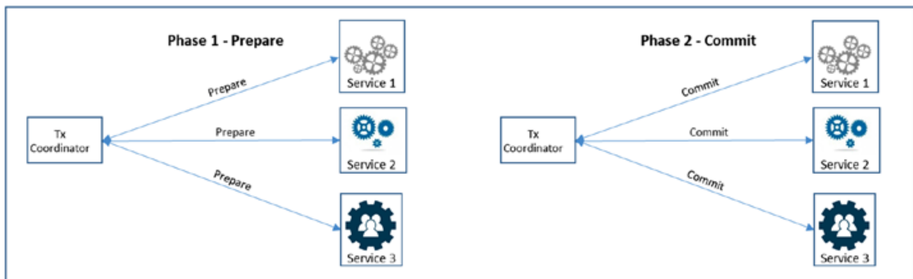

Figure 5-3. *Two-phase transaction management of APIs*

Exposing each transaction phase as an API and expecting the client app to coordinate the transaction, and roll over in case of failure, is an over kill. Managing all transaction from the client app is going to result in a chatty conversation. The complex processing logic in the app for transaction coordination and management definitely yields a poor app performance. The solution is to handle the conversation from an API facade. The logic to prepare, commit, and roll back two-phase transaction management is implemented in the facade. The facade exposes only one API that is invoked by the client. For example, a hotel

booking service can expose only one endpoint to access it (`/hotelbooking`). This endpoint may in turn invoke two separate endpoints: one to reserve the hotel (`/reserve`) in the prepare phase and the second to make the payment (`/payment`) to confirm the reservation in the commit phase. This way the client need not directly access both `/reserve` and `/payment` services and nor does it have to manage the two-phase transaction. Figure 5-4 shows how the two-phase transaction is handled by APIs.

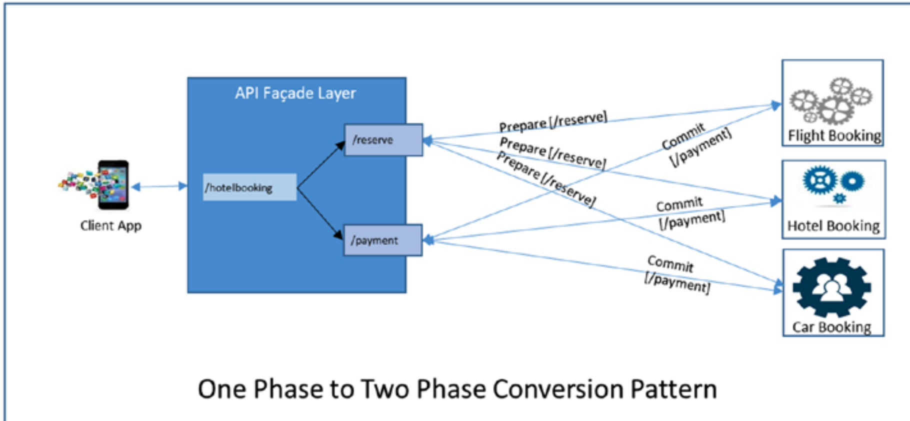

Figure 5-4. *Two-phase conversion pattern*

Synchronous to Asynchronous Mediation

In many scenarios, the application client needs to access a back-end service that is long running and may not provide an immediate response. The mobile app cannot wait for the entire duration till the response is received. A typical example is sending a message. Suppose that you are building a mobile app that sends an SMS to a given number. After the message is sent, the mobile network takes its own time to deliver the message to the recipient depending on various factors. The message delivery status may be available almost immediately or after sometime depending on the network traffic and other factors. The back-end service is asynchronous. However, the mobile app expects a synchronous response. So how do you implement this? An API facade can provide a solution to this. Implementing a callback pattern on the API facade is the first step to this solution. The high-level steps for the solution are as follows.

1. The client app makes a call to the API facade.
2. The API facade makes a call to the back end with a callback URL pointing back to the facade layer.

3. The API facade sends response back to the client with URL to check on the response status.

4. After sometime the target system sends the updates (Eg. delivery status) to the API Facade at the callback URL. API facade layer forwards the notification to the notification URL of the mobile app.

Figure 5-5 shows the steps on how to implement a synchronous to asynchronous mediation using an API facade.

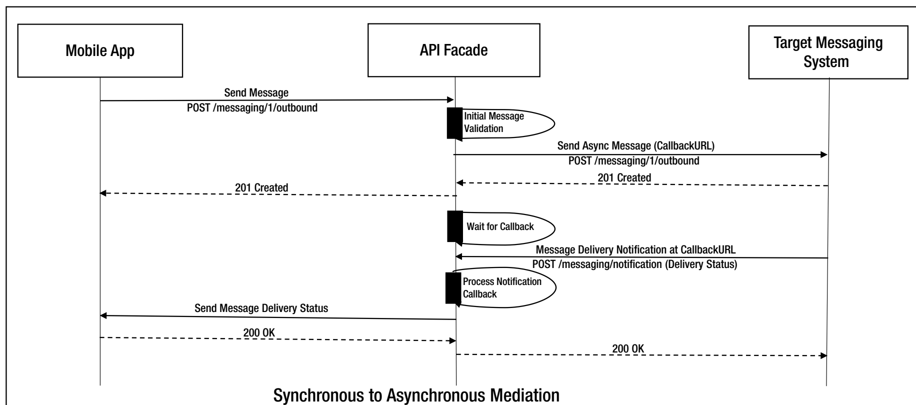

Figure 5-5. *Synchronous to asynchronous mediation pattern*

Routing

In a complex service composition scenario, the routing rules may not be fixed. The back end to which the request should be routed may have to be dynamically determined based on parameters in the incoming request. This is also known as content-based routing. The parameters for routing may be present in the request header or the message payload. In the API facade, these parameters are extracted and inspected to determine the back-end endpoint to which the request should be routed. A common example where this pattern can be applied is when routing a request to a back-end based on the originator of the request. Based on the customer category (Platinum/Gold/Silver) you may want to route the request to different back-end services that contains business logic for each category of customer (see Figure 5-6).

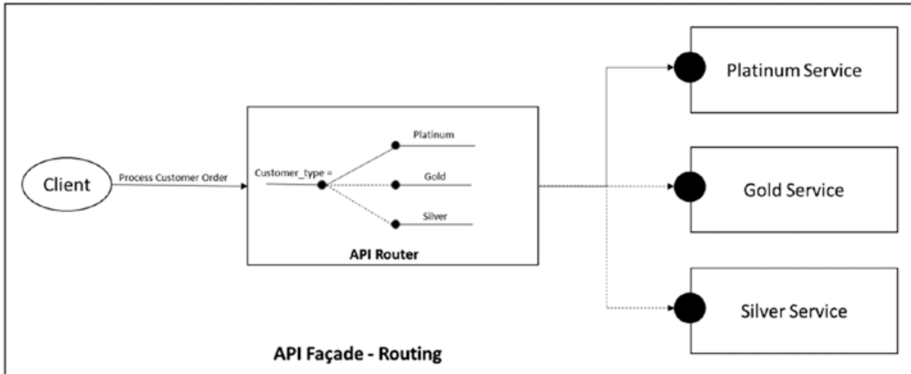

Figure 5-6. *API routing pattern*

API Throttling

When an enterprise opens their API to the external world, it is expected to see an increase in the API traffic. Developers use these APIs to build new innovative apps. As more apps are built and adopted by users, the overall traffic is bound to increase. Also since the APIs are now open to the public, there may be some unexpected and unwanted load coming from some malicious apps, which may try to bring down the system. The current back-end systems may not have been designed to scale up and withstand this increased load. To maintain the performance and overall stability, it is important to maintain the overall traffic within the capacity limits of the back-end system by throttling the API. The following are the common approaches to throttling.

- **Spike Arrest**: With Spike Arrest, you can detect sudden unexpected changes to the traffic pattern. Applying a Spike Arrest policy smooths out the traffic by uniformly distributing the traffic across each smaller interval. For example, if the set spike arrest limit is 60 per minute, then only one request is allowed every second. If in any second there is more than one request, they would all be throttled. Similarly, if the spike arrest is 200 per second, then only one request is allowed per 5 milliseconds. If there is more than one request in any 5-millisecond interval, subsequent requests is throttles. The value of the spike arrest should be calculated based on the capacity of the back-end services. The limits should be configured for shorter intervals such as sec or minutes. This feature protects the back-end services against sudden traffic burst coming from some malicious users or apps.

- **Rate Limit or Quota:** With a Rate Limiting approach (also sometime referred to as Quota), the requests are throttled based on the originating app or user, region of origination, time of the day and various other factors over a period of time. The request within the specified limit is routed successfully to the target system. Those beyond the limit are rejected. For example, if the quota is defined as 1,000 requests per day, all requests after the 1,000th request are rejected. It doesn't matter when these 1,000 requests are made. They could have been made in the first minute, or in the final minutes, or evenly paced. Additional requests are allowed only after the quota is reset at the end of the time interval. The rate-limit values depend on the API product sold to the user. It controls the number of calls allowed for APIs in that product. For trail product, the limits might be less. For high-value products, the limits allowed could be more. Unlike spike arrest, rate limit allows calls to go through till the limit is reached. Hence, the rate-limit values should be carefully derived by looking at the overall capacity of the back-end systems and the expected load. The rate limit values are normally specified for a longer duration such as minute, hour, day, or month.
- **Concurrent back-end connections:** Sometimes legacy back-end systems might have the strict restrictions on the number of connections that can be made. By implementing throttling using concurrent connections, you can limit the number of simultaneous connections that can be made from the API to the back-end services at any given point of time. Based on the value specified, the API gateway container controls the number of connections made to the back-end and rejects requests once the connection limit is reached. The limits to be set should be determined based on the capacity of the back end services.

Caching

Caching pattern can be used within an API gateway to cache backend responses or any information required for processing the request. When a client makes the same request, the cached response is returned to the client instead of forwarding the request to the back-end. This improves the overall API performance and improves the stability of the system by reducing the load on the back-end servers. Each cached response is normally stored against a unique key. The key is derived based on the parameters in the request. Hence, if the app or client makes requests using the same URI, the cached data is sent in the response, if not expired. If the cache is expired, the request is forwarded to the back-end system to fetch the latest data, which can then be cached in the API gateway to serve subsequent requests. Caching the response data is useful when the data is updated only periodically. The cache expiry time should be set based on the update interval of the back-end data. Static data such as list of stores or hotels is a good example of where caching can be beneficial as these dint change frequently. Dynamically

changing data should not be cached. Also if the data changes very frequently, the caching strategy should be examined carefully, else it can result in incorrect response to the client. The caching strategy should consider the cache expiry time, cache key, the cache skip conditions, size of the cache object as some of the top factors.

Logging and Monitoring

Logging is one of the best ways to identify and track problems. It is no different in the world of APIs. In fact, given the distributed nature of APIs, the importance of logging and monitoring increases significantly. To help identify problems during the processing of API requests, critical information should be logged. The information for logging should be collected at all stages of message processing and logged at the end of message processing or in the event of an error. Logging can be done to syslog or to a local file system. The ability to log to a local file system is generally available on a private cloud setup of API management platforms. While using a public cloud instance of the API management platform, it is recommended to log information to a syslog server. If syslog server is not available, public log management services such as Splunk, Loggly, Sumo Logic, and so forth, may be used.

Once you know how to log, it is important to determine what information should be logged. The information logged should provide sufficient data to detect, find, and analyze the issue. Since APIs are used for distributed communication, log information should help locate the source of the issue. It should also provide information about the date/time of the issue, description of the issue with error codes and messages and a correlation ID to relate it to events in other applications of the system.

It is a good practice to log certain metainformation from the request and response even in success scenarios (see Figure 5-7). It can be used for auditing purposes. All logging should be done using asynchronous mechanisms to avoid impact to the actual API performance. Using a separate thread to send the log information to a messaging queue is a common approach followed by most API management vendors to send logs to their destinations.

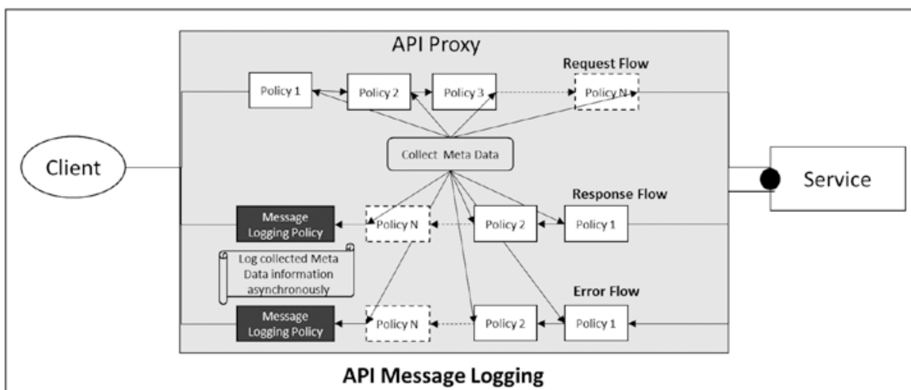

Figure 5-7. *API message logging pattern*

API Analytics

Implementing APIs for digital transformation is not enough. You need visibility into your API program to measure the success and make strategic investments. API analytics provide insight into the API program through information about the API traffic pattern and performance metrics. An API analytics dashboard can tell you which APIs are used most frequently and how traffic varies over time. You can also get behavioral information about the target services in terms of response time, errors rates, size of the payload, and so forth. Information about the developer adoption of an API and the geographic distribution of API traffic can also be gathered from API analytics. Additionally, you can collect custom data from the message payloads and derive useful analytics data for making informed business decisions. Analytics data is normally stored in databases and later processed, aggregated, and analyzed. Hence, like the logging information, analytics data should also be collected at different points in the message flow and processed asynchronously to move it to the back-end database for dashboard reporting.

API Security Patterns

When APIs provide access to enterprise data and assets to a wide audience, they are also opening a larger variety of threats and security challenges for the company. The number of malicious assault and denial-of-service (DOS) attacks are increasing as APIs make back-end systems more accessible. Since APIs can be accessed programmatically, the vulnerability is even greater. Hence, over time, new security patterns have emerged to secure the access to APIs and protect back-end systems against various attacks. The challenge is in providing an easy access to legitimate and authorized users while making it difficult for unauthorized users to access APIs. Hence, getting API security right can be a challenge. This section looks at the different approaches that have emerged as patterns for securing APIs against various types of attacks from potential hackers.

Common Forms of Attack

Hackers can attack to get access to the system, steal valuable information, or even bring down the system that impacts your business. The following are the most common forms of attack on APIs.

- **DoS attacks**: Malicious users flood your system with high-volume API traffic that the back-end systems cannot handle, bringing it to a halt.
- **Scripting attacks**: In this kind of attack, attackers inject malicious code into the system to get access and possibly tamper back-end data and assets. The malicious code can be an SQL statement, XPath or XQuery statement, or some script that tries to exploit design flows in the system to get access to back-end data.

- **Eavesdropping**: In this kind of attack, the hacker gets access to an API request or response while the data is in transit over a non-secure API communication channel. He can then manipulate the message and send it to the ultimate recipient.
- **Session attack**: In this kind of attack, the hackers gain access to the session ID used by a user or app. This information is then used for personification and access to the user's account and resources. In this common form of attack, an app makes an API call and passes the credentials or session information in the header, that can provide access to the underlying assets. The risk is worse in scenarios that use a multiparty authentication scheme, such as OAuth, to grant permissions to a third party to access to access their private data.
- **Cross-site scripting (XSS)**: This is a special form of scripting attack that takes advantage of known vulnerabilities in a web site or web application. An attacker injects a malicious link or code that is executed on the victim's web browser. This form of attack bypasses the same-origin policy that requires everything on a web page to come from the same source. When a same-origin policy is not enforced, the attacker can inject a script or modify the web page to achieve their purpose. An XSS attack delivers tainted content to the API from a trusted source that has permissions to the system. Hence, the API must protect itself by validating the `'Origin'` header in the request payload to check for the origin before allowing access to back-end resources.

API Risk Mitigation Best Practices

There are different approaches and patterns that have emerged to protect APIs from various forms of security threats and provide comprehensive security. The approaches for securing APIs should control access to APIs as well as monitor and limit API usage. Controlling access to an API should authenticate and authorize users or apps making API calls. It should also scan incoming messages for well-formedness and any potential threats in it. A monitoring approach should detect any sudden changes in the API traffic pattern and block the user from making calls. A comprehensive API security approach should look at all the links in API value chain: starting from the users and apps that consume the API, to the API team that builds the API, all the way to the API provider that exposes the data and services in the back-end systems. Since APIs provide omnichannel access, the API security approaches should also be omnichannel security. The security architecture should be flexible and responsive enough to prevent, detect, and react to all forms of API threats in near real time.

Next, let's discuss some of the best practice approaches for building a security into an API management solution.

Authentication and Authorization

Identifying and authenticating API consumers is critical in mitigating security threats. Apps consume APIs and consumers use the app. Hence, it is essential to differentiate between the app and the consumer/user and control the operations that they can perform. Every app is associated with a unique API key. Hence, API key validation on an API management layer can help identify the app and thus control access to the APIs. Once an app has been identified and validated, the user using the app should be verified to validate the end user permissions to access an API resource. This can be done through OAuth scope validation in the API management layer or by integrating with another identity and access management system, such as LDAP, Tivoli Access Manager, Microsoft Active Directory, and so forth. This kind of integration perform single sign-on and provide a seamless experience to the user. While authenticating the user, the API provider should also take into account the context in which the app or the API is being used. Validating context information such as geolocation, device capability, and time, as part of the security framework can help build a strong security for APIs.

API keys identify the app. It is the responsibility of the app to store them securely and protect them from misuse. The app should encrypt the key and store it in a secure vault to prevent any misuse. HMAC-based encryption can be used for encrypting API keys. Also keys should be transmitted in encrypted form over the network using SSL for any authentication between the app and the API gateway. The API key is the identifier of an app and not the end user. Hence, it should not be used as substitute for end user authentication or authorization.

OAuth should be used as a mechanism to provide authorization to a third-party application for access to an end user resource on behalf of them. OAuth helps with granting authorization without the need to share user credentials. OAuth 2.0 uses SSL for all of its communications. Hence, all user and app information in the OAuth dance with the OAuth provider is secured in transit. Many prominent API management platforms, such as Apigee, Mashery, and Layer 7, take out the complexity of implementing OAuth and integrating with external identity and access management systems. This should be leveraged instead of natively implementing it.

Protect Against Attacks

Since APIs expose a lot of valuable business data, they are prone to different kinds of attacks. API management platforms come with in-built features to detect and eliminate such attacks. These platforms provide configurable policies or assertions, which when activated or attached in the request pipeline, can detect attacks using malicious contents or malformed XML or JSON. Some API platforms can also detect virus signatures. Schema validation policies or threat detection policies attached in the request-processing pipeline can mitigate the risk of SQL injections, malicious code injection, and business logic or parameter attacks. CORS header validation protects against XSS attacks. IP whitelisting is another approach to reduce risk from untrusted sources.

Preventing APIs against denial-of-service attacks is another important security consideration. Most API management platforms provide protection against DoS attacks using Spike Arrest and Quota policies. The Spike Arrest policy identifies unexpected surge in the API traffics and reject all requests exceeding the configured limit. This maintains

a uniform distribution of request flowing to the back-end systems as per their capacity. The Quota policy, on the other hand, restricts the number of API calls that a client app is allowed over a time interval. Alerts should be sent if APIs are getting overloaded or any suspicious pattern of API calls is detected. Using rate limits and quota policies alongs with a licensing model that establishes a contractual obligation between the API provider and the consumer app and enforces payments for violation of contracts, can minimize the risk of DoS attacks.

Encrypt Message Exchanges

Often, message payloads sent in API calls contain sensitive information that can be the target for man-in-the-middle attacks. An API management platform sits in between the client app and the API service provider as an API gateway. All communication between the client app and the API service provider through the intermediate API gateway should be secured using SSL/TLS encryption by default (see Figure 5-8). A two-way SSL between the client app and the API gateway also helps with client authentication. SSL should also be enforced for all communications between the API gateway and the back-end service. A pervasive security approach for encrypting data using SSL prevents against man-in-the-middle attacks.

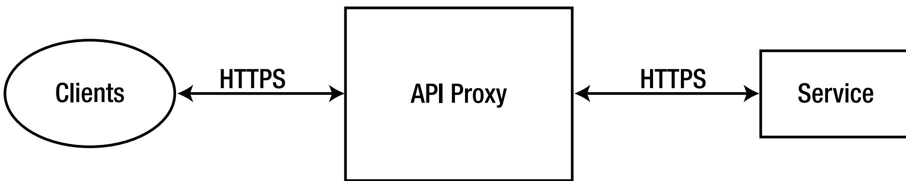

Figure 5-8. *API transport security using HTTPS*

Monitor, Audit, and Log API Traffic

The API management solution should monitor, log and analyze API traffic. It understands API usage patterns. An API provider is interested in knowing which is their most popular API operation, who is the most popular user of the API, what is the rate of growth of API consumers, what is the traffic pattern over a period of time. An insight into all of this information helps with planning the API extensions to strengthen API security.

Logging metainformation from an API traffic flow is also useful in the root cause analysis of any problem. In the event of any security breach, it provides information about the time of the incident, the message payload, and the mechanism of attack. If appropriate information is logged, it can also identify the source of the attack. Hence, monitoring APIs and capturing the right information from the API traffic logs is an essential step in securing APIs.

Logging and Auditing is also one of the major regulatory compliance requirement. Various national and industry-specific laws require minimum logging. Regulatory compliance requirements in financial industry mandate that you log certain API traffic information and make it available as part of the audit and compliance process. For example, you might be required to provide proof that you mask sensitive data, or can detect unauthorized users in the logs captured from API request and response processing.

Build API Security into the SDLC Process

API security is not possible without a comprehensive set of security policies and processes ingrained within the development life cycle of API development. API architects should plan to address security for APIs at the start of the API program. They should provide guidelines for authentication and authorization to make APIs secure. Policies to protect APIs against various forms for attacks and vulnerabilities should be defined as part of the security architecture and design. These security policies should be implemented and thoroughly tested during the development and testing phases. Penetration testing of APIs should be a mandatory step in the testing phase. Post deployment, APIs should be continuously monitored for any potential threats and performance issues that could potentially indicate any security incident.

Use a PCI-Compliant Infrastructure

PCI compliance specifications define a set of guidelines for handling credit card and other sensitive information during a transaction or at rest. The consortium of industries began in 2006 and includes payment card processing companies like Visa, Mastercard, JCB, and Discover. The PCI-DSS compliance requirements apply to all organizations that store, process, or transmit credit card or payment information. The intent of this specification is to protect card holder data and give confidence to the consumers that their sensitive information would not be misused. The following are the some of the important guidelines for PCI compliance.

- Build and maintain a secure firewall. Do not use any default passwords.
- Protect stored data and encrypt sensitive data in transit.
- All application and systems should be secured and protected against unauthorized access using strong access control measures.
- Anti-virus and vulnerability management programs should be kept updated.
- Monitor network access and test systems regularly.
- Maintain an information security policy.

There is no product that will make you PCI compliant. Product and processes can help you implement and enable PCI compliance requirements. If an API is handling any sensitive payment information, it needs to adhere to the PCI guidelines. Also, the API gateway infrastructure on which such APIs are deployed should be PCI complaint.

API Deployment Patterns

APIs need to be deployed on a platform that is scalable and flexible. The platform should simplify API development and deployment. It should also enable the business to manage the entire API ecosystem. The platform should drive the customer reach of the APIs and support business growth. To meet all of these demands, most API platforms provide two deployment models: cloud and on-premise. The decision to choose the right deployment model would depend on the business needs. Let's look at the characteristics of each deployment model.

Cloud Deployment

The cloud deployment of API gateways is hosted and managed by API platform providers on a public cloud, such as AWS or Azure. For example, the Apigee cloud instance is hosted on AWS. Cloud deployment provides customers with seamless product upgrades and improves the pace of innovation. The cloud deployment option leverages the economics of scale. However, it also puts the data and services outside the traditional enterprise firewall, which is can be a security and regulatory concern.

The main advantages of public cloud platform are as follows:

- **Higher reliability and availability**: Cloud platforms provide clustered environments that are distributed across multiple data centers and regions. This mitigates the risk of data center and network outages, and increases the reliability and availability of the platform. The API platform vendor handles traffic fluctuations and makes capacity adjustments to meet the guaranteed SLA.
- **Faster time to market**: The cloud instances of the API platforms can be spun off almost immediately by the API vendors. This saves time and hassle of hardware procurement, setup, and configuration. The cloud instances are up and running very quickly thus reducing the overall time to market for the API program.
- **Reduced capital and operational expenditure**: Cloud deployments are generally available in a subscription model. You pay by usage like number of API calls. This avoids upfront capital expenditures and reduces ongoing in-house operational costs.

- **Reduced management overhead**: Letting the API vendor focus on the data center infrastructure helps enterprises focus on building their API services. The API platform provider takes care of management over heads of running and managing the data center. They address all availability and performance management of the underlying infrastructure. Software updates and fixes are rolled out seamlessly by the vendors. The API provider can focus on creating the API and its back end.
- **Increased scalability and agility**: The licensing for the cloud platforms are generally by API traffic volume. If the traffic increases, API providers only pay additional licensing fee for the increased traffic. They do not need to bother about capacity planning, procurement of hardware, installation, configuration, and training needs for the operations personnel. The platform vendor makes the required changes to provision the additional capacity requested. This makes cloud environments ideal for horizontal scaling to meet the increased demand.
- **Regulatory compliance**: Often regulatory compliance requirements come in the way of adoption for cloud-hosted solutions. But most of the leading API management vendors have achieved industry compliances for their cloud-hosted platforms and their products. PCI DSS for the payment industry and HIPAA for the health industry are the most common industry compliance requirements. Since the platform is already compliant to the industry standards, it helps the client to easily meet the PCI requirements for security and log management on the cloud and other industry compliances.

These are the main disadvantages of cloud deployment.

- **Network latency**: The distributed nature of the cloud infrastructure and additional network hops on the cloud introduces additional network latency. Using an API Delivery Network (API-DN) can route the traffic intelligently and help decrease the latency disadvantages. API-DNs route the request to the closest data center, thus reducing some of the network latencies.
- **Control over data:** On a cloud-hosted platform, all API traffic data is available in the cloud. This reduced the amount of control and security the client can have for their data passing through a cloud-hosted API solution.

On-Premise Deployment

In an on-premise deployment model, the API provider purchases the software and takes the responsibility of setting up and running the entire platform in its data centers. The API provider takes up all the management overhead of installing, running, and maintaining the API platform. They are responsible for the hardware procurement, data center setup, and network configuration. The responsibility to monitor the API platform performance, deal with outages, update and manage software versions and capacity scaling lies with the API provider. Managing the entire API platform also needs additional training about the platform. Though there are initial challenges for setting up the on-premise infrstructure, following reasons can be the main drivers for on-premise deployment.

- **Enhanced security**: With an on-premise deployment model, the API service provider has full control on the data security. They can manage where the under lying data stores would be present, how infrastructure and the data in it is secured, and who can have access to it. This also meets the increased security audit needs of the enterprise.
- **Reduced network latency**: Since the API gateway is installed within the enterprise's network, it cuts down on multiple network hops. API providers may also plan to install the API gateways within the same network as the back-end services. This reduces the network latency and increases the overall performance of the APIs.
- **Better management and control**: On-premise versioning provides better management and control over performance and scaling. You can decide on the number of instances of the product components to be installed to support increased load. You have control over changes to the environment configuration, such as software and hardware upgrades.

API Adoption Patterns

APIs are used by businesses to move ahead with their digital transformation initiatives and increase revenue and customer reach. RESTful APIs are used to expose data and services and deliver an engaging experience to the customers. APIs have also been used by business for internal application integration and partner integration. It makes data more readily available for consumption. As APIs have evolved and used by a greater variety of consumers, there has been a pattern that has emerged in their adoption. The following are the four most common API adoption patterns:

- APIs for internal application integration
- APIs for business partner integration
- APIs for external digital consumers
- APIs for mobile and IoT

Let's look at the business drivers for each of these different adoption patterns and the different considerations for architecting and sharing the APIs for consumption.

APIs for Internal Application Integration

Enterprises use SOA for building services to achieve loose coupling and reusability. These services are used for internal application integration. SOAP and other protocols are used for integration. SOA provided the right level of security and governance, but faced with the challenge of making the services easily discoverable and consumable. The complexities associated with UDDI and service registries to publish and discover service were one of the main. APIs built on top of SOA address the consumption side of it. It makes services easily to publish and discover through the API portal. The developer-friendly and intuitive interface of a REST API makes it easy for developers to consume and build apps using them. APIs have been used for integration within and across lines of business.

With huge investments already placed in SOA services, and with many business processes built around them, companies are less likely to throw it all out to embrace REST APIs. Hence, building an API on a clean slate is a rare opportunity. APIs have to be built on top of the SOA services that expose the back-end services to make them more consumer-friendly. APIs address the consumption side of the equation—making it easy for developers to discover and consume services. API management platforms provide the functionality to create developer-friendly REST APIs from SOAP services that are easy to consume. An internal API portal publishes an API catalog, making the APIs searchable and visible to internal consumers. It brings in an open and collaborative practice for developer, while controlling the visibility of APIs and combining it with the right level of security and governance required for internal consumption.

APIs for Business Partner Integration

Enterprise have been consuming third-party APIs to simplify and expand business partnership. When APIs are used for B2B partner integration, they grow the business rapidly. APIs provide faster integration and an improved partner/customer experience. The technicalities of creating APIs for partner integration are not much different. However, they are more rigorous and have a commercial aspect tied to it. Instead of being open to all, they are available to a select list of business partners. The API consumers and providers are bound by the legal business contracts for the use of the APIs. These business contracts govern the service levels and other aspects of API delivery and consumption. Both API consumers and API providers are responsible for the success of an API program.

APIs for External Digital Consumers

APIs have been adopted by enterprises to accelerate digital transformation, increase customer reach and loyalty, and discover new streams of revenue. Companies can now expose their business assets and service to a larger community of developers with an easy to use and intuitive API interface. External developers and partners adopt these APIs to

build innovative apps. These apps can bring in a completely new business model for the enterprise. Hence, it is important for companies to create external-facing APIs to expose their data and services.

APIs exposed to external digital consumers need a platform that is interactive and provides proactive support to developer community. An API portal provides such a platform. It publishes information about the APIs that developers can use for building apps. Interactive API documentation, blogs, and forums help the developer determine the suitability of an API. It also fosters collaboration with a bigger community of developers. An API portal quickly onboards external developers through a smooth developer onboarding process. Developers register theirs apps to get app keys and secrets on the portal, which are required for secured access to the APIs.

Externalization of APIs and collaboration with other developers build an ecosystem of innovation. It helps developers to share ideas and read about the experiences of others. It generates new and innovative ideas that otherwise would not have been possible. Many companies have seen a northbound trend in their API traffic due to the new experiences brought in by apps created by external developers using their APIs.

APIs for Mobile

Mobile apps have changed the way that humans interact with enterprises. Even though computing power is shifting from server rooms to mobile devices, mobile apps are still limited in resources and restricted by bandwidth. Hence, building a mobile app mandated a simpler interface that can be consumed easily. Also the interface should be such that it can be easily shared with developers to consume them in the apps. RESTful APIs have all of these characteristics, which make them popular for mobile consumption. The API provider should take into considerations the design, security and operational aspects of the API to make them suitable for mobile consumption. Additionally, caching should be looked as an alternative for improving performance and reducing chattiness. Instead of sending bulk payloads, paginations, filtering and other mechanisms should be supported to reduce processing overhead on the mobile app. Standard web API security protocols such as OAuth and OpenID Connect should be supported to secure APIs and make them suitable for mobile consumption.

APIs for IoT

The *Internet of Things* (IoT) refers to the network of devices, sensors, and actuators that communicate with each other over the Internet using API technologies to build a new customer experience. This refers to wearable devices such as iWatch, connected cars, connected sensors—such as Nest thermostats, intelligent bulbs—such as Philips Hue, and many others. It is estimated that by 2020 there will be 50 billion connected devices. APIs form the communication foundation for these connected devices. But the challenge is with the diverse and the newer communication protocols, such as MQTT, AMQT, XMPP, and many others that need to be supported by the API platform. A new generation of infrastructure powered by autoscalling capabilities, may also be needed in furture to support the scale of IoT communication traffic.

CHAPTER 6

API Version Management

Change is inevitable and this is no different with APIs. If APIs are successful, they evolve over time. New requirements may drive you to make changes to your APIs. Advancements in technology are also a contributor to changes to APIs. Handling these changes in a way to minimize the impact on the clients is the art of versioning.

Once you publish a REST API, developers start using it in their client app as per what is defined in the contract. Developers write software that relies on the API contract. So whenever there is a change to the API, there is the potential to break the client software that relies on the contract. Hence, API changes need to be done in a controlled and predictable way. This brings in the need for API version management. In this chapter, we would look at how API versioning is different from normal software versions, the need to version APIs, and the different approaches to API versioning.

API Versioning vs. Software Versioning

Every software release is versioned. The common format for versioning software is as follows:

```
<MajorVersion>.<MinorVersion>.<PatchVersion><OptionalPackageIdentifier>
```

For example, v2.4.16-RC4.

The checksum of the software package is normally used to identify a particular version. If the checksum changes, the version is considered different. However, this approach does not apply to REST API versioning; because if a new version is introduced for every minor change to the API, it would cause a maintenance nightmare. Apps dependent on the API might stop working, leading to a frustrated developer community. Also, maintaining too many versions of the API for the client would be a nightmare.

The REST API version should correspond to the service version and not the software or the package version implementing the service. Every new version of the service implementation need not warrant a change in the service version; hence, the REST API exposing the service. A new API version should be created only when there is a change in the service interface or the contract that is being used by the client.

© Brajesh De 2017
B. De, *API Management*, DOI 10.1007/978-1-4842-1305-6_6

The Need to Version APIs

An API defines a contract for communication between the client and a server hosting a resource to operate on them. The client may want to create, read, update, or delete a resource as defined in the contract. A change in the resource may or may not require a change in the contract. Some changes, such as minor bug fixes, may not require any alteration to the contract. Others, such as a change in the structure of the resource or the way of communication, may require a change in the contract. Changes may or may not be backward compatible. If the change is backward compatible, it may be possible to handle it within the same API version. Changes that are not backward compatible require a new version of the API to be introduced. This lets the consumer know that they may need to make changes to their app code. Versioning APIs helps maintain compatibility, enabling debugging and dependency control.

API Versioning Principles

Some of the main principles of API versioning are as follows:

- An API version should not to break any existing clients
- Keep frequency of major API versions to a minimum
- Make backward-compatible changes and avoid making new API versions
- API versioning should not be directly tied to software versioning

The API Version Should Not Break any Existing Clients

As APIs are adopted by app developers, introducing new versions of the API brings with it the risk of breaking the client's apps. The API versioning strategy should be such it that does not break any existing client apps; otherwise, it will easily frustrate the developer community and slow down the API adoption.

Keep the Frequency of Major API Versions to a Minimum

Every time a new API version is released, it kick-starts a fresh cycle for app developers. They need to understand the new API, analyze the impact on their apps, debug issues, and so on. This is a huge burden on both sides in terms of time and money. Even the API provider has to maintain multiple versions of the API for a sufficient time in order to enable a smooth transition. Supporting multiple versions requires significant investment by both the API provider and the consumers.

Make Backward-Compatible Changes and Avoid Making New API Versions

The simplest way to avoid making new API versions is to make the changes backward compatible. Changes such as the addition of new API resource methods or support for a new data format do not impact the client. Some changes to the input data elements may

be backward compatible; for example, adding an optional element in the input request in the body or introducing support for an additional query parameter does not mandate any change to the client code; these are backward compatible. These kinds of changes do not necessarily require changes to the client app. Hence, it is not necessary to introduce a new version of API backward-compatible changes. However, adding a new mandatory parameter in the request or changing the name of a field in the response requires a new version of the API to be created, because these require changes to the client application code.

API Versioning Should Not Be Directly Tied to Software Versioning

As discussed at the start of the chapter, software evolves very rapidly. Every major release, enhancements, and bug fixes result in a new version of the software. If we start tying the API version to its software implementation, it would result in an unprecedented number of API versions. This would not only frustrate the consumers dealing with the API, but also results in maintenance nightmares for API providers. Hence, the API version should never be tied to the software version of the back-end data/service. A new API version should be created only if there is a change in the contract of the API that impacts the consumer.

Approaches to API Version Management

There are multiple approaches to introduce versioning in your API. The next few sections discuss the most common ways to version an API.

Versions Using URLs

An API is normally identified by its URL. So in API versioning, it makes sense to introduce the version information in the URL as follows:

`http://www.foo.com/v1/customers`

In this URL, `v1` defines the major version identifier. When this identifier changes, it is assumed that all resources under it changes—in this case, the `customers` resource. If a new version of the customer is introduced in the next version, it should use a new version identifier, like `/v2/customers`. The new version can be accessed as follows:

`http://www.foo.com/v2/customers`

This approach to API versioning is followed by the likes of Google, Yahoo!, and many others. Using a dot notation for API versioning to indicate the minor revision eg, V1.1, is also a common practice. However, that does not add much value compared to just using the major version and incrementing the version number to the next integer. Some popular API providers, like Twilio, use the date as a version identifier for their APIs. For example, if the API was released in the year 2015, the provider may use the following URL format:

`http://www.foo.com/2015/customers`

The version that came up in the next year would use this URL format:

http://www.foo.com/2016/customers

This could be extended to include the month, as follows:

http://www.foo.com/2016/02/customers

or

http://www.foo.com/2016-02/customers

This approach might be needed only if there are multiple versions of API releases in a year. However, monthly or very frequent release is against the API versioning best practices.

The advantage of this approach is that it is easy for users to understand which version of the API they are using.

Versions Using an HTTP Header

Another approach to API versioning is to use an HTTP header. With this approach, an HTTP header is used by the client to specify the API version that it wants to invoke. The advantage of this approach is that it helps to keep the API version out of the URI that is used to refer to a resource. The other benefit of this approach is that you can easily ignore or silently upgrade if the user does not specify any version or specifies a deprecated version of the API.

The use of an HTTP `'Accept'` header is one of the preferred choices. The GitHub API follows this approach and expects it to be passed in the request as follows:

```
Accept: application/vnd.github[.version].param[+json]
```

Using a custom HTTP header like `'X-API-Version'` in the request is yet another approach to API versioning. However, this has its own disadvantages. What if the client does not add this header in the request? What should the default behavior be? Should the server respond with an error message or handle it according to the latest API version? If handling using the latest version of the API is the default approach, how do you then handle breaking changes when introducing a new API version?

Versions Using Query Parameters

This is yet another common approach to versioning API. In this approach the client specifies the version number as a query parameter in the request, as follows:

http://www.foo.com/customers?version=v2

The server may choose to honor the query parameter or even ignore it. One advantage of this option is that the `version` parameter can be optional or required, depending on how you want the API to be used. In this case, the `version` is optional, a default behavior may be assumed to be the latest version when it is left off. Being in the URL, this is very easy to see and understand.

This approach works well when the resource representation is versioned. In such cases, it is necessary to also put in transformation logic to transform the resource representation based on the version specified.

Versions Using a Host Name

Another approach to API versioning using a URL is to use a different host name. For example, Facebook's first version of an API is available at `api.facebook.com`, whereas their new graph API is available at `graph.facebook.com`. This approach is used only when there is an extensive revamp of the API.

The downside of this approach is that it not only requires a change in the URL used by the client to access a resource, it may also require changes in the security settings on the client side due to the change in the hostname. Also, this may require setting up a completely new infrastructure to support the new version, adding to the cost. The only advantage of this is that you can completely revamp the URI structure and route client requests to a different server, without the need to change the older version. Client requests for older API versions are processed by the old instance.

Handling Requests for Deprecated Versions

As new API versions are introduced, the API provider should notify the expected behavior when a client makes calls to older and deprecated API versions. During the transitions phase, the provider may choose to handle requests to older versions by responding with a redirection URL pointing to the new API version. Alternatively, the API provider may respond with an HTTP 404 error code that indicates that the requested resource version was not found. If a client makes calls to a deprecated API version, it should fail with the 404 HTTP status code.

API Version Lifecycle Management

Introducing non-breaking changes to APIs is fairly simple and can be pushed out without much fanfare. As a best practice, however, such changes should be published through a blog post, updated in API documentation, or at least logged in the API release notes change log. However, making changes requires a lot more planning, extensive testing, customer handholding, and voluminous communication. Since there may be a lot of consumers for the API, it is important to notify them of any changes to the APIs. There are multiple ways to do that. The following are some of the most common approaches:

- Announce new upcoming version and versioning schedules, if any, in the API developer portal
- Send emails to registered developers about upcoming new API versions
- Introduce "warning" headers in alerts on older versions being deprecated
- Define a migration period and cut-off date for support to older API versions

Releasing a new API version as a beta release to a restricted group of developers is a good approach to introducing new API versions. Provide enough time for the developers to test and provide feedback on the new API version. Only after a successful launch of a beta version should it be taken to production and opened for general availability. Even after the successful release of the new API version, never immediately deprecate or remove support for older versions. Once in production, old and new versions of an API should run simultaneously to give enough time for developers to migrate their code to the new version. Set a date for deprecation of the older version of the API so that developers have a clear target to migrate their apps over to the new API version. There should be clear communication and coordination between the API provider and the consumer community when new versions are introduced. This helps to reduce the risk of breaking trust with any version upgrades.

CHAPTER 7

API Security

APIs provide a very good opportunity to build engaging and innovative customer experiences. They help businesses build new channels of integration and partnership. As companies look to expose their assets and data as REST APIs in an effort to provide new customer experiences and expand their business, security becomes a main concern. To a chief security officer at an enterprise, it is of paramount importance to secure the APIs and protect underlying assets from misuse, attacks, or any kind of threat. The security threats to APIs can be of various types. The security model to protect against these threats depends on the type of asset or service exposed and the associated risk. For example, APIs dealing with sensitive financial data over public networks require stronger security measures than APIs dealing with publicly available data over a restricted network. There is no one-size-fits-all approach that can be applied to protect APIs against various threats. This chapter looks at some of the important security threats to consider when building a security solution. It also looks at the various approaches and the security models for protecting APIs against these threats.

The Need for API Security

APIs allow consumers to interact with and access an enterprise's data and services. It is no good to have assets locked down within the enterprise. Exposing assets helps enterprises grow business and revenue. Creating APIs that enable customers to use their assets builds enriching customer experiences and increases customer engagement and loyalty. However, since assets have a business value, they are prone to theft and attacks to gain unauthorized access. APIs that act as the front door to these assets should therefore be secured. An API without security is like keeping the door to your vault open. Since APIs are used by in-house developers, trusted partners, and third-party developers, they should be protected intelligently. Considering that APIs may sit at the edge of the enterprise and can be accessed by a wide variety of customers through different channels, such as mobiles, smartphones, tablets, web apps, connected cars, kiosks, IoT devices, and more, they should be secured thoroughly to prevent any kind of misuse of the underlying assets.

Today, APIs introduce a new form of security threats by hackers. Earlier hackers used to sit behind a console and try out attacks to find vulnerabilities. Due to the programmable nature of APIs, hackers can now use them to automate their attacks and try out different things to find system vulnerabilities. Hence, the security model in the APIs should be able to identify such attacks and reject requests in order to protect the back-end assets.

© Brajesh De 2017
B. De, *API Management*, DOI 10.1007/978-1-4842-1305-6_7

APIs today form a critical part of any digital strategy. Lack of API security can bring your digital transformation journey to a grinding halt. Hence, having a well-defined strategy for API security is of principal importance.

API Security Threats

APIs provide channel of access to enterprise assets. Hence, they introduce many more types of security threats that were previously non-existent or were not considered a genuine threat. The different API security threats can be broadly classified into the following categories:

- Authentication
- Authorization
- Message or content-level attacks
- Man-in-the-middle attacks
- DDoS attacks (distributed denial-of-service)

APIs allow a new range of third parties to access enterprise assets. Without proper security policies in place, anyone can access these assets—even before a formal relationship has been established with third parties. Apps built by third parties can compromise enterprise security. Hence, it is important to have a proper registration and onboarding process for third-party organizations and app developers. Apps built by third parties should be registered with the API provider before they can use the APIs. Only authenticated systems, apps, and developers should be allowed access to the APIs in order to eliminate any risk of security compromise.

Third-party apps often access information on behalf of the end users. This information may be sensitive and private and can be accessed only after proper authorization has been obtained from the end user. Hence, APIs should be secured to check for the right level of authorization to grant in the request. Access to the resource should be granted only after authorization checks succeed.

Attackers can place malicious content, such as malware, in API requests to attack the system. They can also inject scripts in the request that are executed in the back-end systems. The impact can be devastating. It can corrupt systems and provide an outsider with unauthorized access to sensitive and business critical data. This can put a company's reputation at stake. APIs should be protected to detect any such malware or scripts, or malformed payloads in the request.

In a man-in-the-middle attack, hackers get access to credentials and tokens that can be used to get access to APIs. These credentials and tokens may be harvested and used nefariously. All data should therefore be encrypted in transit and protected from unauthorized access while at rest.

Attackers can launch DDoS attacks from one or more IP addresses via APIs to bring down the system. Since APIs provide a programmatic access to underlying resources, launching a DDoS attack is very easy. An API security model should be able to identify a DDoS attack and take the right action to protect back-end systems.

The next few sections look at how to design a security framework to protect APIs and their underlying resources against various forms of attack.

API Authentication and Authorization

Authentication determines the identity of the end user or the party requesting access to a protected resource. It helps validate who you are.

Authorization determines the access level and permissions of the end user to perform a certain operation. It determines the actions that the client is allowed to perform on the protected resource.

The following are some of the most commonly used forms of authentication and authorization used for API security:

- API keys
- Username and password
- X.509 client certificates and mutual authentication
- SAML
- OAuth
- OpenID Connect

API Keys

An API key identifies the application using an API. It provides a simple mechanism to authenticate the apps. API keys allow an API to determine which applications are using it. API keys are generally long series of random characters typically passed as an HTTP query parameter or header. This makes it easy to use an API key in an API request for application authentication. API keys are also known by other names, such as app ID, client ID, app key, or consumer key.

When a developer registers his app with an API provider, a unique API key is provided to the developer. The developer needs to secretly store this API key and use it in the application requests when making an API call from the application. The key identifies the application making the request and helps the provider monitor which application is making the request. The developer also gets insights into how his application is used by end users.

An API key is normally a long alphanumeric string that is opaque and without any signature or encryption. This makes API keys less secure for authentication purposes. The use of API keys is best for auditing and identification. An API key validation policy in the request flow of the API can help to validate the API key and also capture important metadata information, such as developer, organization, and so forth, related to the application. This may be good for APIs that only need to know who is using it. API keys can also be used to enforce API call quotas for an application (see Figure 7-1).

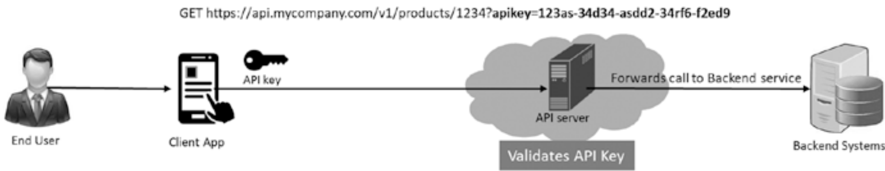

Figure 7-1. *API key usage to secure back-end services*

Thus, API keys can be also used to filter or turn off access to rogue applications that might be flooding the system with API calls. Providers can revoke an API key to block traffic coming from that application. Where enhanced security is required, API keys can be used to generate tokens using OAuth and OpenID flows.

Username and Password

A username and password is the most common form of authentication and is useful when dealing with sensitive data in an API call. In this form of authentication, the client presents the server with a unique name (username) and a secret code (password). The server validates the username and password against its credential store and provides access to the client only on successful validation.

For a REST API call, the client can pass the credentials (username and password) in an HTTP header using the Basic Authentication scheme. As per this scheme, the client sends the server authentication credentials using an `Authorization` header. The `Authorization` header is constructed as follows:

1. The username and password are combined with a single colon (:).
2. The resulting string is then Base64 encoded.
3. The authorization method and a space (`Basic `) are then entered before the encoded string.

The username *John* and password *John@123*, results in a header that looks like this:

```
Authorization: Basic Sm9objpKb2huQDEyMw==
```

HTTP Basic Authentication is the most common form of authentication; it is supported by nearly all clients and servers. It is easy to implement without the need for any special processing. The client needs to ensure that the password is protected and kept secret. If the client needs to store the password, it must be encrypted in some way to protect it against any attackers reading it from the store. SSL should be used for all client-server communications to protect credentials in transit from eavesdroppers.

API key validation with Basic Authentication can be combined for better API security. For example, the app identity may be sent as API key and the end user credentials may be passed in as Basic Authentication header. The API server may first validate the app identity from the API key and then validate the credential of the end-user accessing the client using the Basic Authentication headers.

X.509 Client Certificates and Mutual Authentication

An X.509 certificate contains a public key that validates an end entity, such as a web server or an application. It is a good alternative to a username/password for authentication purposes in application-to-application communication. The X.509 certificate contains the identity of the subject. The subject information is described as a distinguished name (DN), common name (CN), along with other optional attributes, such as country (C), state (ST), location or address (L), organizational unit (OU), and organization name (O). All of this certificate information is digitally signed by a trusted certificate authority (CA). This helps certify the public key of the subject and ensure that the certificate is not tampered with. The certificate's private key is always kept secret with the user and is never divulged to the signing authority or anyone else.

After the subject receives a signed certificate from the certificate authority, it can be used as identification. It allows secure access to protected APIs. For a mutual authentication using two-way SSL, the API resource server needs to import the client certificate in its trust store. The SSL handshake starts with the API resource server sending its X.509 certificate to the client. After the client app has validated the server certificate, it sends its public key to the API resource server. The server validates the client certificate against the list of certificates present in its trust store. A two-way SSL is established after the mutual authentication by the server and the client is successful. Only then can the app can make an API call. Figure 7-2 shows a high-level view of the message exchanges to establish a two-way SSL and make an API call.

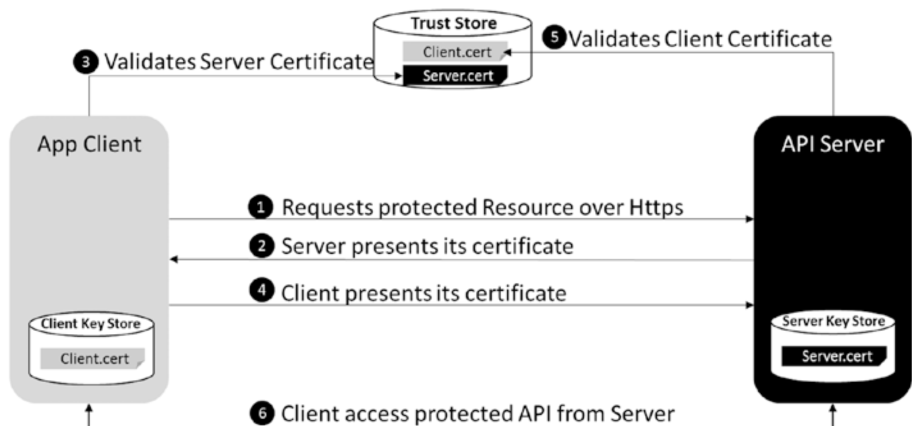

Figure 7-2. *Two-way SSL for mutual authentication*

OAuth

OAuth 2.0 is a protocol that allows clients to grant access to server resources without sharing credentials. As per the IETF specifications, the OAuth 2.0 authorization framework enables a third-party application to obtain limited access to an HTTP service, either on behalf of a resource owner by orchestrating an approval interaction between the resource owner and the HTTP service, or by allowing the third-party application to obtain

access on its own behalf. For example, a shopping app can use access to its customer data in Facebook. When a customer accesses the shopping app, he is redirected to log in via Facebook. The customer is redirected to the shopping app after he has successfully logged in. The shopping app can now access customer data and can even post status updates on Facebook on behalf of the customer (if authorized to do so).

So what problem is OAuth trying to solve? Let's look at the scenario where a user wants to post some reviews about a product from the shopping app (say, Amazon) to Facebook but doesn't want to type their Facebook password on Amazon. This is possible if the Amazon app is able to store the user's Facebook password somewhere and use that to post on their Facebook page. But why should the user trust Amazon with their Facebook password? Also, what happens when the user changes their Facebook password, which is now stored in multiple locations with different apps? The user now has to manually go and update their Facebook password in all the locations that it is stored, which definitely is not a good user experience.

Instead of storing the Facebook password on every application that wants to access the Facebook account, what if we create a token that is authorized to perform limited actions, such as post on Facebook on their behalf. This token is generated after the end user has authorized Amazon to access their Facebook account.

The token has a defined validity and is understood and recognized by Facebook. So when Amazon presents the token to Facebook within the validity period of the token, it is allowed to access and post reviews on the user's Facebook page. In this way, users do not need to share their Facebook password with every other application that needs to access their Facebook account to post any updates. The access is automatically revoked when the validity of the token expires. The token can be revoked even earlier than its expiry time, if required.

Using OAuth tokens for API security makes APIs more resilient to security breaches, since they don't rely on passwords. In the previous example, if the user finds out that their Facebook password has been compromised, they only need to change it in one place, without impacting other applications that need to access their Facebook account. Those applications continue to access the Facebook account using the access token until it has expired or has been revoked.

OAuth Basic Concepts

To understand OAuth better we will now look at some of the basic concepts involved in the next few sections.

Actors in OAuth

OAuth protocol defines a sequence of message exchanges that need to happen between the various parties to grant the client access to a server resource. The various actors involved are resource owner, client, resource server, and authorization server.

- A **resource owner** is the end user who authorizes an application to access various resources in their account. For example, the user of a Facebook account can be the resource owner. The photos and activities like the posts and likes in the Facebook account are the data owned by the resource owner. The list of resources that an application can access or the operation that an application can do, is determined by the "scope" of the authorization granted.
- A **client** is the application that is trying to get access to a resource owner's account.
- A **resource server** hosts the protected resources of the user. In the API world, it is the server where the API resources are hosted. For example, Facebook is the resource server hosting the APIs to view or edit photos and user activities. Access to the API resources is allowed only after the client has been authorized by the resource owner or the user.
- An **authorization server** validates the identity of the user and then issues the access token to the client, which can be used to get access to resources.

Figure 7-3 shows the various actors involved in an OAuth flow.

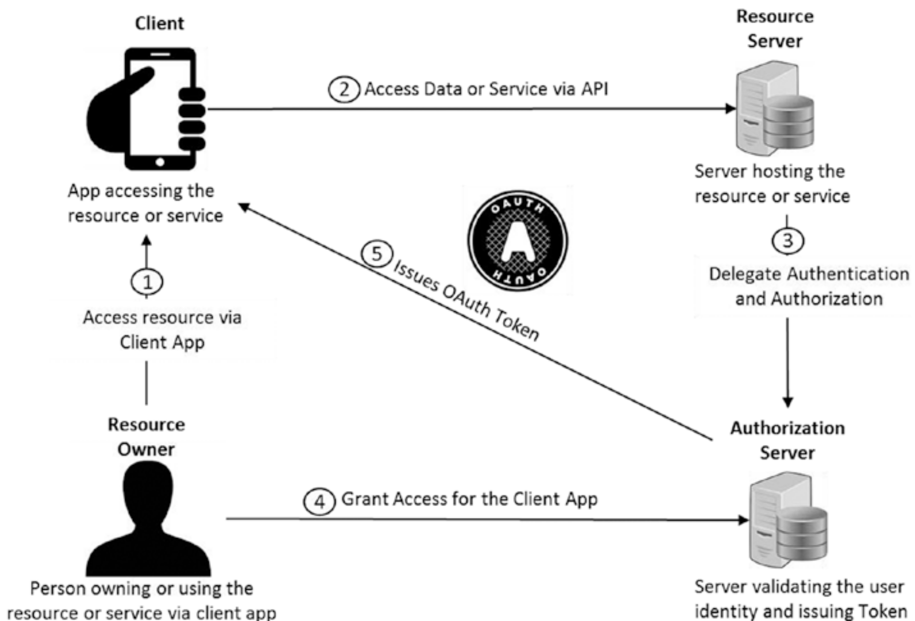

Figure 7-3. *The various actors involved in an OAuth flow*

In most cases, the server on which the API is hosted acts both as the resource server and the authorization server. An API gateway can play this combined role, as shown in Figure 7-4.

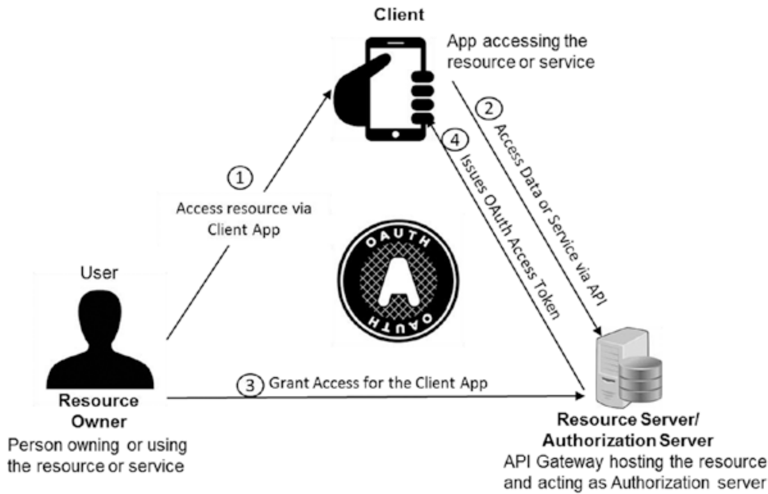

Figure 7-4. *Role of API gateway in OAuth*

Tokens

Tokens are issued to allow access to specific resources for a specified period of time and may be revoked by the user that granted permission or by the server that issued the token. There are two different kinds of tokens used in the OAuth flow: *access tokens* and *refresh tokens*.

- **Access tokens** allow access to a protected resource for a specific application to perform only certain actions for a limited period of time. They are a long string of characters that serve as a credential. They are generally passed as bearer tokens in an authorization header. An access token can also have restrictions or scope associated with it that specify the API resources that can be accessed using the token. An access token generally has an expiry duration and can be refreshed using refresh tokens for certain grant types. In situations where an access token is compromised, it can be revoked to prevent any further use of that token.
- **Refresh tokens** represent a limited right to reauthorize the granted access by obtaining new access tokens.

Scope

Scope identifies what an application can do with the resources that it is requesting access to. Scope names are defined by the authorization server and are associated with information that enables decisions on whether a given API request is allowed or not. When an application requests an access token, the scope names are optional.

Grant Type

An *OAuth grant type* can be thought of as the interactions that an app goes through to get an access token. OAuth 2.0 defines the following four grant types:

- Authorization code
- Client credentials
- Resource owner password credentials
- Implicit

Each of these grant types have their own pros and cons. The grant type used for generating a token depends on the business use case. One of the important considerations for choosing a grant type is the trust in the app accessing the resource.

Let's now look at each of the grant types in detail and learn about the flows involved for generating an access token for them.

Authorization Code

An *authorization code* is one of the most commonly used grant types. It is considered the most secure because it involves authorization from the end user, who actually owns the resource. The experience of using an authorization code grant type is similar to signing in to an app using a Facebook or Google account. This is sometimes referred to as "three-legged OAuth" since it involved three parties:

- End user
- Client app
- Authorization server

The following is the high-level process involved with the authorization code grant type:

1. Generate an authorization code.
   a. The end user logs in and grants consent to the application to access resources.
   b. The authorization server generates an authorization code that contains the scope information for which authorization was given.
2. Exchange authorization code for access token. The client application exchanges the authorization code for an access token from the authorization server. A refresh token is also generated and given to the client.
3. Use the access token. The client app uses the generated access to make API calls.

Figure 7-5 shows a detailed sequence of flow for generating an access token using an authorization code grant type.

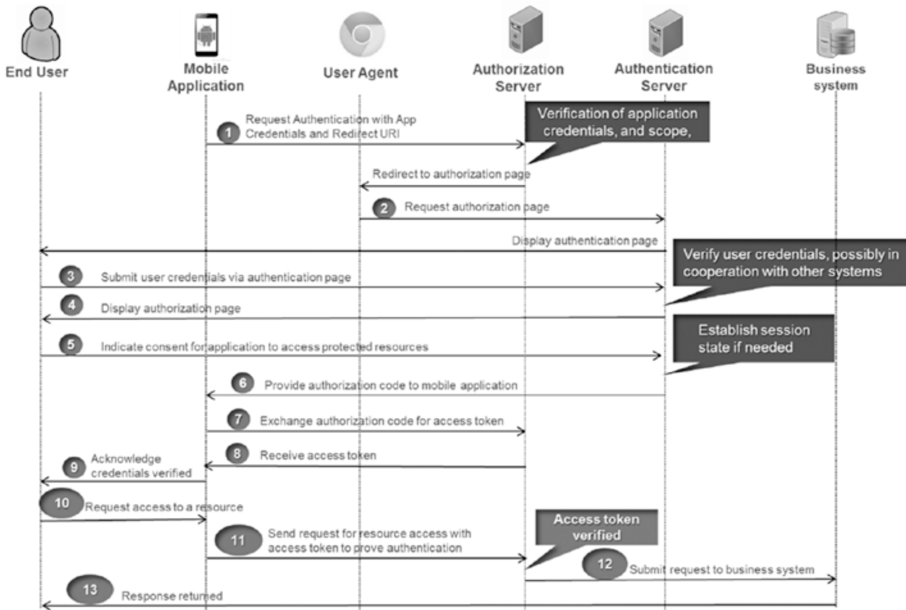

Figure 7-5. Authorization code grant flow for OAuth 2

Client Credentials

The client credentials grant type is suitable for machine-to-machine interaction and does not require any user permissions to access data. The following describes the high-level flow sequence (shown in Figure 7-6).

1. Generate access token.

   a. The client sends a message with its identity and the scope of access required to the authorization server.

   b. The authorization server validates the client's identity and issues an access token.

2. Use the access token. The client app uses the generated access token to make API calls.

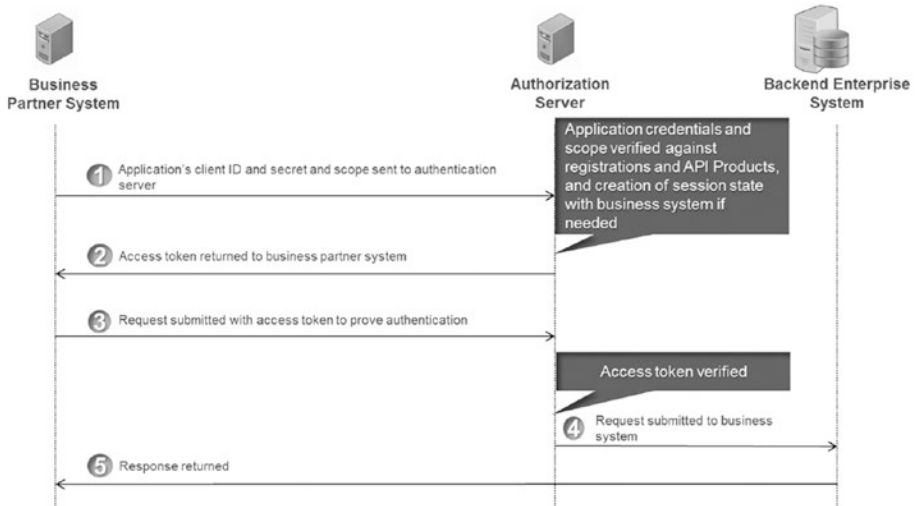

Figure 7-6. *Client credential flow*

Resource Owner Password Credentials

Resource owner password credentials grant type are used when the end user's credentials need to be authenticated before access can be granted. The following describes the high-level flow sequence (see Figure 7-7).

1. Generate access token.
   a. The client sends a request with its identity, scope, and the user's username and password.
   b. The authorization server validates the client's identity and user credentials.
   c. The authorization server issues an access token.
2. Use the access token. The client app uses the generated access token to make API calls.

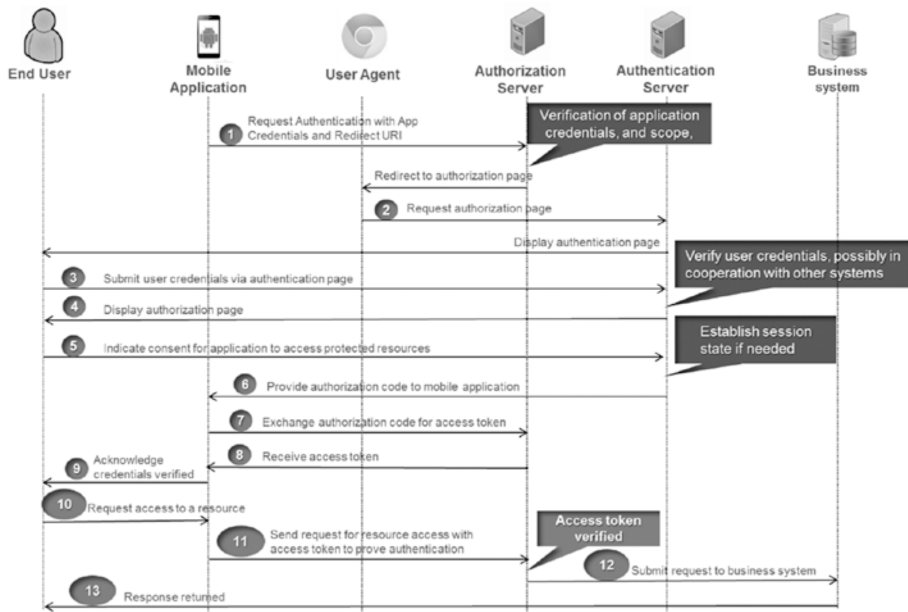

Figure 7-7. *Resource owner password credential flow*

Implicit

Implicit grant type is used by mobile apps and JavaScript applications running in the web browser. In this flow, the access token URL is given to the user-agent to be forwarded to the client app via a redirect URL. Since the access token is encoded into the redirect URI, it may be exposed to the user and other applications running on the same device. The identity of the client is also not validated by the authorization server in this flow. Unlike the authorization code flow, where the client makes separate calls for authorization and for the access token, in the implicit flow, the client gets the access token as a result of the authorization request without any client authentication. The resource server only verifies the redirect URI that was originally registered. This makes the implicit flow easy but less secure. No refresh token is generated with implicit flow. Figure 7-8 shows the sequence of flow for the implicit grant type.

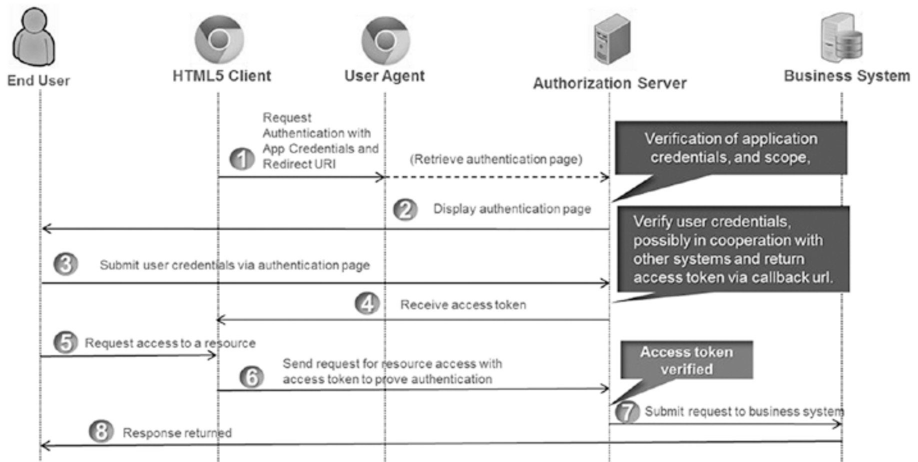

Figure 7-8. *Implicit flow*

OpenID Connect

OpenID Connect 1.0 is an authentication protocol that builds on top of OAuth 2.0 specs to add an identity layer. It extends the authorization framework provided by OAuth 2.0 to implement authentication. OpenID connect introduces an ID token in addition to the access and refresh tokens provided by OAuth 2.0. The ID token contains the identity information of the end user in JWT format. OpenID Connect defines *identity* as a set of claims or attributes related to an entity, which can be a person, a service, or a machine.

Actors in OpenID Connect

The following are the various actors involved in an OpenID Connect authentication flow:

- **OpenID Connect provider (OP)**: An OAuth 2.0 authorization server that provides authentication as a service. It authenticates the end user entity and provides the claims or attributes of the entity to the client.
- **Relying party (RP)**: An OAuth 2.0 client that requires end user authentication or claims from the OpenID Connect provider.
- **End user**: The entity that requests identity or claims information from the OpenID provider. The entity can be a human participant, a machine, or a service and is the owner of the resource that the client is trying to access.

Figure 7-9 is a high-level illustration of how different actors in an OpenID Connect protocol interact with each other.

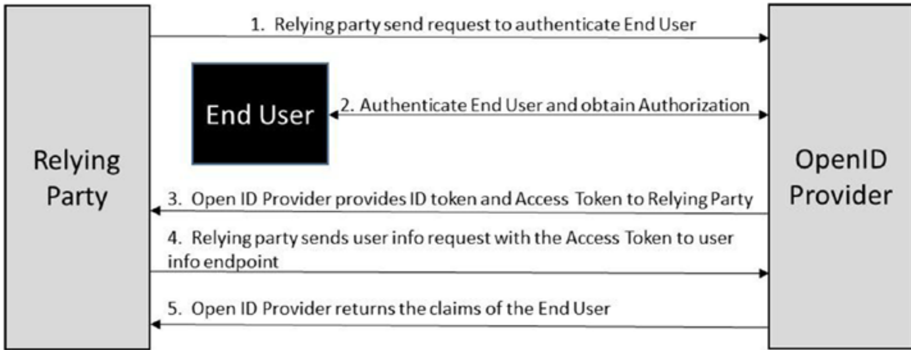

Figure 7-9. *Interaction between parties in OpenID Connect flow*

By using OpenID Connect, clients can request and receive identity and authenticated session–related information about the end user from a central identity provider, and validate it. OpenID Connect can be used by clients of all types—including web-based, JavaScript, and native/mobile clients—to create a distributed and federated model for SSO.

ID Tokens

ID tokens are the main enhancements introduced by OpenID Connect on top of OAuth 2.0. An ID token is like an identity card that contains claims information about the authenticated end user. The ID token is represented as a JSON web token that is signed by the OpenID provider. The ID token contains the following claims information related to the end user in JSON format.

- **Subject identifier** (`sub`) is locally unique and asserts the identity of the end user.
- **Issuer identifier** (`iss`) identifies the issuing authority of the token. It is a case-sensitive URL using the `https` scheme. It contains the scheme, host, and optionally the port and path components.
- **Audience information** (`aud`) is what the ID token is intended for. It identifies the relying party and other audiences that can use this token. The OAuth 2.0 `client_id` of the relying party must be present in the audience information.

- An **alphanumeric string** (`nonce`) associates a client session with the ID token to prevent replay attacks. The nonce value is normally passed unmodified from the client authentication request to the ID token. If this value is present in a client authentication request, it must be included in the ID token response by the authorization server acting as the OpenID provider. If the nonce is present in the response, the relying party must validate that the value received in the response is equal to the value passed in the original request.
- The **time** (`auth_time`) when the end user authentication occurred.
- The **authentication context** class reference (`acr`).
- The **time** that the ID token was issued (`iat`).
- The **expiry date** of the ID token (`exp`).
- Optionally, it may contain other details about the entity, such as name and email address.

The following is a sample JSON format of the set of claims in an ID token.

```
{
  "iss": "https://example.server.com",
  "sub": “2340051",
  "aud": "s6Bhdr4k9qt3",
  "nonce": "n-078_WzB3Mj",
  "exp": 1317881970,
  "iat": 1316780970,
  "auth_time": 1311280969,
  "acr": "c2id.loa.hisec"
 }
```

The ID token is a JWT token created from the JSON format of the claims. JWT generally has three parts: a header, a payload, and a signature.

- The **header** specifies the algorithm used for signing and the token type in JSON format, as follows:

```
header = '{"alg":"HS256","typ":"JWT"}'
```

- The **payload** contains the claims in JSON format.

- The **signature** is calculated by Base64 encoding the header and the payload, concatenating them with a period separator, and then applying the signature algorithm on the concatenated string.

```
key = 'mysecretkey'
unsignedToken  = encodeBase64(header)  + '.' + encodeBase64(payload)
signature  = HMAC-SHA256(key,  unsignedToken)
```

- The ID token is created by concatenating together the Base64-encoded value of the header, payload, and the signature with a period as the separator between them, as follows. This is done so that the token can be easily passed around.

```
token = encodeBase64(header)  + '.' + encodeBase64(payload)  + '.' + encodeBase64(signature)
```

eyJhbGciOiJKUzI1NiIsInR47CI6IkpXVCJ9.eyJsb2dnZWRsdKlbkFzIjoiYWRtaW4iLCJpYXQde5eE0MjI3Nzk2Mzh9.gzSraSYS8EXBxLN_oWnHRDgCzcmJmMjLiuyu5CSpyHI

OpenID Authentication Flows

OpenID performs authentication to log in an end user or to determine if the end user is already logged in. The result of the authentication is securely retuned by the authorization server to the client in an ID token so that the client can rely on it. For this reason, the client is also referred to as the *relying party*. OpenID Connect defines the following three paths or flows for authentication to obtain the ID token:

- Authorization code flow
- Implicit flow
- Hybrid flow

Authorization Code Flow

In this flow's first step, an authorization code is returned directly to the client after authenticating the end user and receiving consent. In the second step, the client exchanges the authorization code to get an ID token and an access token. Since OpenID Connect is built on top of OAuth 2.0, the sequence of message exchange is almost same for both. The main difference being that the end-user is authenticated against an Open ID Provider and an ID token is generated and returned to the client in addition to the access token. The following are the high-level steps for the authorization code flow.

1. The client sends an authentication request to the authorization server containing the client_id, secret, redirect URI, and scope.
2. The authorization server authenticates the end user accessing the client against the identity provider.

3. The authorization server obtains consent and authorization from the end user for the client to access resources owned by the end user.

4. The authorization server sends the end user back to the client with an authorization code via HTTP 302 redirect.

5. The client sends a request using the authorization code to the token endpoint.

6. The client receives a response that contains an ID token and an access token in the response body.

7. The client validates the ID token and passes the access token to retrieve the end user's subject identifier.

An authorization server must implement the following endpoints to support the OpenID connect authorization code flow:

- Authorization endpoint (`/authorize`)
- Token endpoint (`/token`)
- User information endpoint (`/userinfo`)

An *authorization endpoint* is used to authenticate the end user and provide an authorization code to the client. The user agent is sent to the authorization endpoint hosted by the authorization server for authentication and authorization. The authorization request contains the following information:

- `scope`: Mandatory information sent in the request. For OpenID connect flows, this must have the `openid` value. It can also have other values for which the client is requesting access on behalf of the end user.
- `response_type`: This value determines the authorization processing flow to be used. For an authorization code flow, it must have the value of `code`.
- `client_id`: The identifier of the client making the request. The client gets this at the time of registration.
- `redirect_uri`: The redirection URL to which the response is sent. For security reasons, it must match the value of the redirect URI provided by the client at the time of registration to the OpenID provider.
- `state`: An opaque value that is used to maintain the state between the request and the callback. It is typically used to mitigate cross-site resource forgery (CSRF) attacks.

Other request parameters defined by OAuth 2.0 specifications may also be used.

The authorization endpoint must validate all the information sent in the authentication request according to OAuth 2.0 specifications. If the request is valid, the authorization server attempts to authenticate the end user or determines if the end user is authenticated. The method used for authentication is beyond the scope of the OpenID specification. The authorization server may display an authentication user interface to the end user, depending upon the values in the request parameters and the authentication method. The authorization server must authenticate the end user if not already authenticated or if the authenticate request contains the `prompt` parameter with the `login` value. After the end user has been authenticated, the authorization server must obtain consent from the end user before releasing any information to the relying party. The end user consent can be obtained through an interactive dialog with the end user. After the authorization server has successfully authenticated the end user and received the consent, it responds with a successful authentication response containing the *authorization code* and the *state* information. This information is returned as a query parameter added to the `redirect_uri` specified in the authentication request. The following is a sample response from the authorization server.

```
HTTP/1.1 302 Found
Location: https://client.example.com/cb?
code=TplxlOBeZMMYbYS7WxSbIA &state=af2ifjhldbj
```

After successful authentication of the user, the client uses a *token endpoint* to obtain the following:

- ID token
- Access token
- Refresh token

The client or the relying party makes a token request by presenting the authorization code received from the authorization endpoint. The token request can be made using an HTTP POST call over TLS (Transport Layer Security) 1.2, as follows:

```
POST /token HTTP/1.1
Host: myserver.example.com
Content-Type: application/x-www-form-urlencoded
Authorization: Basic bzZCaGRSa4F0MzpnWMFmQmF0M2JZ

grant_type=authorization_code&code=TplxlOBeZMMYbYS7WxSbIA
&redirect_uri=https%3A%2F%2Fclient.example.com%2Fcb
```

The token endpoint must validate the token request, as follows.

1. Authenticate the client and validate its client credentials.
2. Validate that authorization code was issued to the authenticated client.

3. Verify the validity of the authorization code and ensure that it has not already been used.

4. Validate that the `redirect_uri` presented in the token request is same as that included in the authorization request.

After successful validation of the token request received from the client, the authorization server returns a successful token response containing the following:

- ID token
- Access token
- Refresh token

The following is a sample token response containing the three tokens:

```
HTTP/1.1 200 OK
Content-Type: application/json
Cache-Control: no-store
Pragma: no-cache

{
  "access_token": "SlAV32hkKG",
  "token_type": "Bearer",
  "refresh_token": "8xLOxBtZp8",
  "expires_in": 3600,
"id_token":"eyJhbGciOiJSUzI1NiIsImtpZCI6IjFlOWdkazcifQ.ewogImlzcyI6ICJodHRwOi8vc2Vyd
mVyLmV4YW1wbGUuY29tIiwKICJzdWIiOiAiMjQ4Mjg5NzYxMDAxIiwKICJhdWQiOiAiczZCaGRSa3F0MyIsC
iAibm9uY2UiOiAibi0wUzZfV3pBMk1qIiwKICJleHAiOiAxMzExMjgxOTcwLAogImlhdCI6IDEzMTEyODA5N
zAKfQ.ggW8hZ1EuVLuxNuuIJKX_V8a_OMXzR0EHR9R6jgdqrOOF4daGU96Sr_P6qJp6IcmD3HP99Obi1PRsc
wh3LOp146waJ8IhehcwL7F09JdijmBqkvPeB2T9CJNqeGpegccMg4vfKjkM8FcGvnzZUN4_KSP0aAp1tOJ1z
ZwgjxqGByKHiOtX7TpdQyHE5lcMiKPXfEIQILVq0pc_E2DzL7emopWoaoZTF_m0_N0YzFC6g6EJbOEoRoSK5
hoDalrcvRYLSrQAZZlyuVCyixEoV9GfNQC3_osjzw2PAithfubEEBLuVVk4XUVrWOLrLl0nx7RkKU8NXNHqr
vKMzqg"
}
```

The client receiving the token response must validate the received ID token as follows.

- If the ID token is encrypted, the client must first decrypt it using the keys and the algorithms that the client specified during the time of registration with the OpenID provider.
- The client must—at a minimum—validate the following information in the ID token:
    - The issuer identity of the OpenID provider must exactly match the value in the `iss` claim attribute.
    - The audience (`aud`) claim attribute must contain the `client_id` value that was issue to the client by the OpenID provider at the time of registration.
    - The algorithm value (`alg`) must be as negotiated at the time of registration.

- The expiry (`exp`) claim of the ID token must be greater than the current time.
- The issued at (`iat`) claim of the ID token is not too far from the current time. The client can decide on the value of this duration.
- The *nonce* value, if sent in the authentication request, must match with the value received in the ID token.

- Other information, such as `acr` claim and `auth_time` claim, should also be provided by the client.

The *userinfo endpoint* (`/userinfo`) is an OAuth 2.0 protected resource that returns claims about an authenticated end user. The client makes a request to this endpoint using the access token received from the token endpoint to get claims and attribute information about the end user. The end user claims are returned as a name:value pair in a JSON object. All communication to the userinfo endpoint must use TLS. This endpoint must support both HTTP GET and POST methods. The endpoint must be able to accept and process a request containing an accept token in bearer format sent in the authorization header. The following is a sample userinfo request:

```
GET /userinfo
HTTP/1.1 Host: myserver.example.com
Authorization: Bearer SlBV35hkKH
```

On successful processing of the request, the endpoint returns the end user claims in a JSON format, as follows:

```
HTTP/1.1 200 OK
Content-Type: application/json
{
  "sub": "548286761004",
  "name": "Fred Sweet",
  "given_name": "Fred",
  "family_name": "Sweet",
  "preferred_username": "fred.sweet",
  "email": "fredsweet@myserver.com",
  "picture":"http://myserver.com/fredsweet/fred.jpg"
}
```

Implicit Flow

Implicit flow is mostly used for browser (JavaScript)–based apps. In this flow, the client obtains the ID token and optionally the access token from the authorization endpoint. The authorization endpoint does not perform any explicit client authentication, but uses the redirect URI as an alternative way to verify the client's identity. After the client receives the tokens, it may expose the tokens to the end user and applications using the same user agent. Hence, this flow is used only for untrusted clients to obtain identity tokens. Unlike the authorization code flow, no refresh token is generated in this flow.

The implicit flow consists of the following steps.

1. The client prepares and sends an authentication request to the authorization server.
2. The authorization server authenticates the end user.
3. The authorization server obtains the end user's consent.
4. The authorization server sends the end user back to the client with the ID token and optional access token, if requested.
5. The client validates the token and retrieves the end user's subject identifier.

When the relying party wishes to validate the client, it prepares the authentication request and sends it to the authorization endpoint. The client can send this request either using the HTTP GET or the POST methods. For an implicit flow, the value of the `response_type` parameter in the request must consist of `id_token` and `token` as a space delimited list, as shown in the following example:

```
https://myserver.example.com/authorize?
response_type=id_token%20token
&client_id=c6BhdTkqt3
&redirect_uri=https%3A%2F%2Fclient.example.org%2Fcb
&scope=openid%20profile &state=bf1ifjsldkj
&nonce=n-0S7_WzA3Mk
```

After authenticating the end user and obtaining consent, the authorization server responds with the `id_Token` and optionally `access_token`, as follows:

```
HTTP/1.1 302 Found
Location: https://client.myexample.com/cb#
    access_token=TlAV35hkKH
    &token_type=bearer
    &id_token=exJ0 ... NiJ8.eyJ2d ... I6IjJifX0.EfWt4Ru ... Zxso
    &expires_in=3600 &state=af0ifjsldkj
```

Hybrid Flow

The *hybrid flow* is a combination of the authorization code flow and the implicit flow and hence the name. This flow allows the client to make immediate use of the ID token to get access to the client's identity and retrieve an authorization code that can request a refresh token. The refresh token can gain long-term access to back-end resources.

The hybrid flow consists of the following high-level steps.

1. The client prepares and sends an authentication request to the authorization server.
2. The authorization server authenticates the end user.
3. The authorization server obtains end user consent.
4. The authorization server sends the end user back to the client with the authorization code. Depending on the `response_type` parameter, one or more parameters may also be returned.
5. The client requests a response using the authorization code at the token endpoint and received a response containing the ID token and the access token in the response body.
6. The client validates the ID token and retrieves the end user's subject identifier.

In the hybrid flow, the client makes the authentication request to the authorization server. The `response_type` parameter in the request can have the following values:

- `code id_token`
- `code token`
- `code id_token token`

The following is an example request using the hybrid flow that would be sent by the user agent to the authorization server in response to a corresponding HTTP 302 redirect response by the client:

```
GET /authorize?
    response_type=code%20id_token
    &client_id=f6BhdKkqg
    &redirect_uri=https%3A%2F%2Fclient.example.org%2Fcb
    &scope=openid%20profile%20email
    &nonce=n-2S6_XzA2My &state=afligjslfku HTTP/1.1
Host: myserver.example.com
```

On receipt of the authentication request, the authorization server does the following validations before responding with a code and the ID token.

1. Validates the `scope` parameter present in the request.
2. Validates the `client_id` provided in the request and that the `redirect_uri` is the same as provided by the client at the time of registration.
3. Validates that all the mandatory parameters are present in the request as per the specifications.

4. Authenticates the end user or determine if the end user is already authenticated.

5. Obtains end user consent for the client to access the protected resources.

After successfully processing the authentication request, the authorization endpoint returns the authorization code. Depending on the value in the `response_type` parameter, the authorization endpoint returns the `id_token` and optionally the `access_token` in a response format, as shown in the following example:

```
HTTP/1.1 302 Found
Location: https://client.myexample.org/cb#
code=Tplx1OAeZQQYbCS6WxSrIL &id_token=eyK0 ... NiK9.eyW1c
... I0IjIifL5.GeWt1Qu ... ZQsj &state=af9ifjs0dkj
```

Benefits of Integration with an Open Identity Provider

Applications often need to validate the identity of an end user. The following are possible ways to achieve this.

- A local database for user accounts and credentials for each app
- A central identity provider used by all end users to register apps and validate their information

With a local database for each app, end users have to register for each new app that they want to use. Many people find the registration process very tedious and not a good customer experience. For an enterprise providing multiple apps, maintenance of separate user databases brings in additional administrative and operational overhead. Hence, having a central identity provider provides a better option from user experience, as well as maintenance and administrative standpoints. Organizations such as Google and Facebook, which have large registered user bases, provide identity provider services than can be used with OpenID Connect. Organizations can streamline and simplify their customer onboarding and login processes by integrating with identity provider services.

Protecting Against Cyber Threats

In the era of social, cloud, and mobile technologies, where enterprises expose their sensitive data and information via APIs in a zero-trust environment, protecting APIs against malicious attacks is of paramount importance. Adding authentication and authorization to protect APIs is not enough. The API security framework must be able to detect any kind of cyber threat and take necessary actions to protect the back-end resources. To protect its APIs from different types of threats, an organization must build an API proxy in front of the APIs with an API management platform and implement

security policies in these proxies to protect against such threats. Some of the most common types of threats are as follows:

- Injection threats
- Insecure direct object reference
- Sensitive data exposure
- Cross-site scripting (XSS)
- Cross-site resource forgery
- Bot attacks

The next few sections go into detail about each of these threats and exposes options of protecting against them.

Injection Threats

Injection threats are common forms of attacks, in which attackers try to inject malicious code that, if executed on the server, can divulge sensitive information. Malicious code can be in any of the following forms:

- XML and JSON bombs
- Script injection attacks

XML and JSON Bombs

Attacks using XML and JSON bombs try to use structures that overload the parsers thereby crash the service. Parsing corrupt or extremely complex XML/JSON payloads with long list of elements and attributes or long tag names and values or multiple levels of nesting can easily use up system resources—such as memory and CPU—and thus induce an application-level DOS attack. Such attacks can be mitigated by using XML and JSON threat protection policies.

XML threat protection policies can be used to check the message payload for the following, and reject the message if any of the allowed limits are exceeded:

- The length of the names of elements, attributes, and namespace prefixes
- The length of the values of elements, attributes, and namespace prefixes
- The node depth of an element
- The number of attributes in an element
- The number of namespaces defined for an element
- The number of child elements for an element

JSON threat protection policies can be used to check the message payload for the following, and reject the message if any of the allowed limits are exceeded:

- The length of a property's name within a JSON object
- The length of a property's string values within a JSON object
- The container depth of the JSON object
- The number of entries allowed in the JSON object of an element
- The number of array elements entries allowed within a JSON object

Script Injection Attacks

Script injection attacks can be in various forms

- SQL injection
- Script injection

SQL Statement Injection

SQL statement injection is a technique in which a hacker presents a malicious SQL query to an application's input parameter. This can be dangerous if the application takes this input in the request to directly query into the database. For example, an API (`/employees?EmpName=<Employee Name>`) that provides the details of an employee from the employee database. This API is implemented in a way to execute the following SQL statement in the database:

```
"select * from Employees where employeename =" + queryparam.EmpName + ";"
```

In this situation, if an attacker invokes the API with the following parameters, the effect can be catastrophic:

```
/employees?EmpName=Lary;drop table Employees;
```

SQL statements like the following can be used by hackers to bypass authentication:

```
select userid FROM customerdata WHERE username = ' ' OR 1 = 1
-- customer_passwd = 'abcd';
```

Hence, it is important that any API that accepts input that can be inserted into an SQL database must be protected against SQL injection attacks. Regular expressions that match certain SQL keywords can be used to detect malicious SQL content in the API request.

Script Injections

Script injections can be in various forms: JavaScript injection, XPath injection, or Java exception injection.

JavaScript is a powerful technology that modifies and sends data. If such scripts are injected though an API, they can reveal sensitive data. For example, hackers can get an unsuspecting user to execute a script in an API request to get access to their authorization token or cookies. The token or the cookie can then be used to log in to the system and steal sensitive information. This kind of attack is known as a *cross-site scripting* (XSS) attack.

XPath injections are also used by hackers to gain unauthorized access to sensitive stored in an XML format.

Input data in API request parameters should be validated and sanitized to harden the APIs and protect against script injection attacks. Regular expressions can detect the presence of malicious JavaScript and XPath in the payload of an API request. However, no regular expression can stop all content-based attacks. Hence, multiple mechanisms should be combined to enable defense-in-depth.

Insecure Direct Object Reference

In an Insecure Direct Object Reference attack, the hacker modifies an existing API request to get access to information. The hacker may try to modify parameters in the request to get a higher level of access. For example, the following API provides access to user account information identified by the account number specified in the URI:

```
GET http://api.myownbank.com/user/account/1234
```

A hacker can attempt to change the account number to get access to a different account. Alternatively, they may try to get admin access to an account using the following URL:

```
GET http://api.myownbank.com/admin/account/1234
```

This kind of attack can be prevented by using OAuth2/OpenID Connect with the right scopes set for the API.

Sensitive Data Exposure

APIs expose internal services and enterprise data. Some of this data may be customer sensitive and highly confidential. Such sensitive data should always be kept private and hence should always be encrypted and masked. Regulatory compliance standards such as PCI, HIPPA, and so forth, require that all sensitive data— such as credit card information and customers' private data—should always be stored in encrypted mode. When sensitive data is sent in an API response, it should be encrypted and tokenized to prevent inadvertent exposure. Again, there may be scenarios where only certain API users may be authorized to view certain information sent in an API response. If the same API is called by another user, some of the response data may have to be either filtered or masked. Sensitive data logged in debug trace should also be obfuscated.

Encryption of data in transit can be achieved by using SSL. Using SSL to encrypt sensitive data is the least any API should do. Another alternative is to selectively encrypt part of API message that contains the sensitive information. This requires the API provider and the client to take on the additional overhead of managing the private/public key. Hence, deployment of APIs that require selective encryption of sensitive data can be complex.

Cross-Site Scripting (XSS)

Cross-site scripting (XSS) is among the top 10 open web application security threats. It is a type of script injection attack that exploits a vulnerability in a web site that the victim visits. The attacker injects malicious code, generally in the form of JavaScript, into otherwise benign and trusted web sites. When a user visits the web site, the malicious JavaScript is delivered to the victim's browser, which appears to be a legitimate part of the web site. The user information or data is compromised when these scripts are executed on the non-suspecting user's browser.

Figure 7-10 shows an example of how an XSS attack is done.

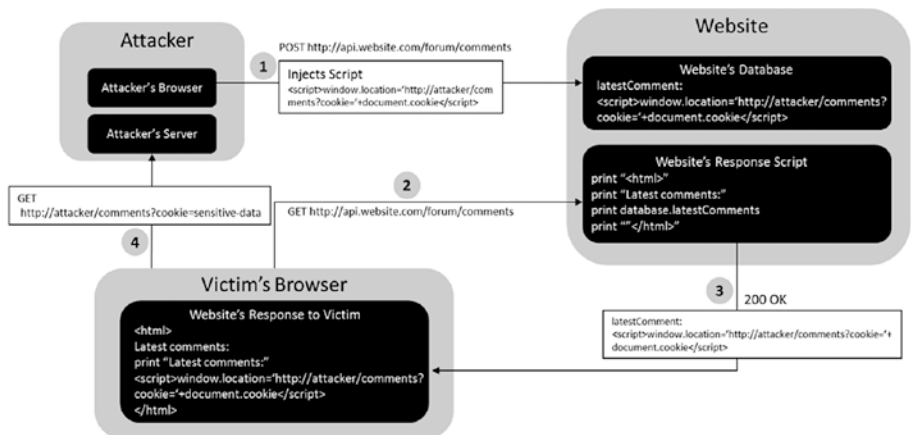

Figure 7-10. *XSS attack approach*

The following describes the process of an XSS attack.

1. The attacker injects malicious JavaScript into a web site's database using a POST request to submit a form.

2. An unsuspecting victim requests a page from the web site using a GET request.

3. The web site responds to the GET request with the malicious script from its database.

4. The victim's browser executes the malicious script in the response, sending sensitive data to the attacker's server.

Hence, to protect against XSS attacks, all user input for an API request must be encoded and validated. Encoding helps to escape the user input so that the browser interprets it only as data and not as code. Validations must include schema and data type validations, and check for the presence of any malicious scripts. Adding a Content-Security-Policy header in the HTTP response constrains the browser viewing a web page to only use resources (script/stylesheet, etc.) loaded from a trusted site. With a properly defined Content-Security-Policy, even if the attacker succeeds in injecting the malicious code, it will not be executed on the browser, since the attacker's site is not among the list of trusted sites.

Cross-Site Resource Forgery (CSRF or XSRF)

Cross-site resource forgery is a type of attack where a user is tricked into executing unwanted actions on a web application in which they are already logged in. This way, the attacker can target the web application, via victim's already authenticated browser. Using social engineering like email or chats, the victim is tricked into clicking a link that sends a forged request to a server where they are already authenticated. Since the user is authenticated, it is difficult for a web application to distinguish between a legitimate request and a forged one. This type of attack is different from XSS. In XSS, the attacker exploits the trust of the user on a web site; in CSRF, the attacker exploits the trust the web site has for the user. Figure 7-11 shows an example of how an attacker can launch and execute a CSRF attack.

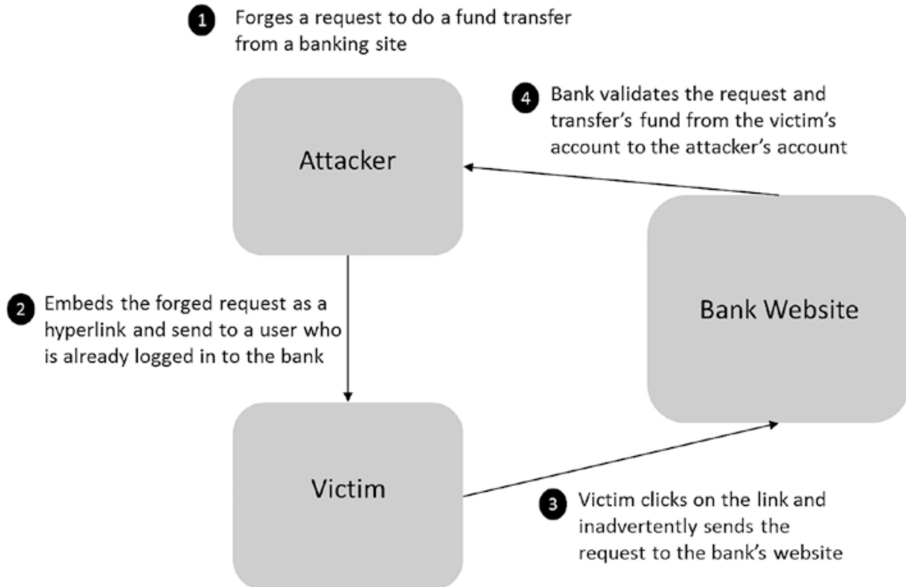

Figure 7-11. *CSRF attack approach*

An attacker uses CSRF to execute unauthorized fund transfers, change passwords or customer data, and many other things that can be detrimental to both the business and the user. The following techniques can be used to protect APIs against CSRF attacks:

- Use OAuth tokens to validate the requests. Tokens are long alphanumeric strings that are difficult for attackers to guess.
- Use a nonce that is unique for every URL and form, in addition to the standard session.
- Check for a “referrer” header in the HTTP request to ensure that the request has come from the original site.

Bot Attacks

In addition to the known threats, a new type of security vulnerability is arising from the use of automated software programs called *bots*. Since APIs provide a programmable interface, it becomes easier for hackers to target APIs using bots. Bot programs constantly scan the application infrastructure for security vulnerabilities. Bot traffic probes for weakness in APIs, abuses guest accounts with brute force, and uses customer API keys to access private APIs. Bot traffic can be identified by analyzing API traffic and access behavior patterns. Using machine learning and statistical models, an adaptive security system constantly learns “good behaviors,” which helps it distinguish “bad behaviors” and enforce dynamic policies that block bots from accessing a protected resource. Bot traffic can be identified in the form of anomalous activities, as follows:

- Logical walk-throughs of the application resource paths by bots.
- Requests originating from a bot network, low-reputation IP address, ISP, or compromised proxies and devices. Malwares installed in rooted devices and PCs may be used to generate bot traffic.
- Unexpected high traffic volumes from certain IP addresses or endpoints.
- High traffic volumes to URIs (resources) that are not generally accessed by end users.
- High rates of form submissions with slight variations in the input parameters. This is one of the common techniques used by bots when applying brute force techniques to get access.
- High error rates on access to resources, especially those that are available to privileged users or applications.

API security strategies must consider how Bot activities can be easily identified through the analysis of API access anomalies. Research has shown that more than 50% of Internet traffic involves bot activities. Retailers and ecommerce service providers that provide dynamic pricing, loyalty programs, financial services, and so forth, are in the radar of bot attacks. Bots are known to target APIs with any valuable or sensitive data.

Bot traffic can have a major load impact on API infrastructure, cause performance concerns, and hurt a company's brand and bottom line due to content theft. Advanced API analytics functionality with machine learning capabilities that can identify malicious bot activities should be considered for building a robust and adaptive API security system.

Considerations for Designing an API Security Framework

There are many aspects to consider in building the right API security framework. Some of the most important considerations include (but are not limited to) the following.

- The nature of the asset or the service being exposed as APIs. What is the impact/loss if data gets into the hands of someone who is not supposed to see it or if a service goes down?
- The regulatory compliance requirements for securing an API. Which regulatory standards should be followed for securing an API?
- The authentication requirements for using the API. Is it OK to authenticate only the client? Or is it necessary to authenticate even the end users before they can use the APIs?
- The authorization needs before a client app can access an API resource. Should the end user authorize access to an API before the client app can access it?
- Threats from API consumers. How can consumers and attackers possibly misuse the API and use loopholes to gain unauthorized access?

API Security Threat Model

To come up with the right security strategy for APIs, the security architect must create a threat model for API exposure and consumption. The following are some of the security threats that need to be considered in API security:

- Unauthorized applications and users may imitate that of another app or user
- Denial of service due to rogue apps or inadvertent errors
- Replay attacks
- Man-in-the-middle attacks
- Data tampering
- Malicious data injection attacks
- Theft of credentials, API keys, and tokens
- Network eavesdropping

API Security Recommendations

An API-centric security architecture that enables defense-in-depth security practices must be adopted to protect data and services from API security threats. This approach builds a security capability that includes role-based access control, fine-grained policies for authentication and authorization, and threat protection against malicious payload content and DoS attacks. The following are some API security recommendations for building a robust API security architecture.

- All API communication involving sensitive data must be secured and encrypted using TLS.
- Build a mechanism to detect malicious content injections and defend against such attacks. This protection is ideally built at the beginning of the API request flow at the edge of the network.
- All incoming and outgoing data must be validated and sanitized. Input data type and format validation must be done at a minimum for all APIs that have input request parameters. This prevents any malicious content from entering the system.
- APIs accepting input parameters via HTTP POST or PUT methods must validate the payload. Such validations help detect large payloads or malformed content that can potentially overload computing resources. Replay attacks and message tampering can also be detected early through these validations. Input parameters passed as query parameters in GET methods should also be validated to check for any malicious contents.
- Use a combination of approaches to identify the source of the request. IP address validation may not be sufficient to identify the originator of a request since IP addresses can be easily spoofed.
- Protection via API key validation can be used only for non-sensitive and read-only data. API key validation identifies the applications and developers making API calls. It also implements API quotas and monitors usage by applications. If the data exposed is non-sensitive and read-only, such as Google Maps APIs, tracking consumer identities through API key validation might be sufficient.
- Use OAuth2 for public or private APIs that are intended for use by native and mobile apps. With OAuth2, the user is not required to share his password with the app and device that he uses. When a user authenticates in an OAuth flow, he enters his credentials in a web browser screen, rather than the application itself. Hence, the application never gets to see the user's password. This becomes a crucial factor when these apps are built by untrusted developers. Since OAuth uses tokens for authorization, API providers can revoke these tokens in any compromise, without the need for users to change their passwords.

- Use OpenID Connect for APIs that need end user identity and authentication. The ID token provided in the OpenID Connect flow can be used by the client or relying party to validate the end user. It can also be used by the API provider to validate the end user trying to get access to a protected API resource.
- Use two-way SSL or TLS with mutual authentication for APIs that are used by a limited number of internal or partner systems authenticating the client. If the API is open to all, maintaining client certificates for a large number of clients to implement two-way SSL may become a real challenge. The Basic Authentication scheme can also be a suitable alternative for authenticating partners.
- All sensitive information must be encrypted in transit using SSL.

Figure 7-12 shows the recommended order in which API security policies must be implemented in an API gateway.

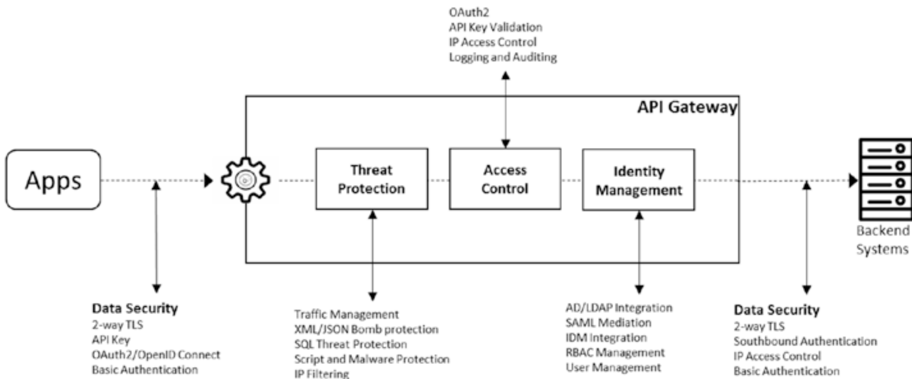

Figure 7-12. *Approach for building end-to-end API security*

CHAPTER 8

API Monetization

APIs securely expose digital assets and services that are of real value to end users and partners. Since they provide value to end users, it makes more sense to monetize the services and the APIs and build a business model for them. Having the right monetization model for APIs helps businesses reap the benefits of their investment in APIs. In this chapter, we look at the various API monetization models and how an API management platform can monetize APIs and open the doors to new revenue opportunities.

Which Digital Assets Can Be Monetized?

APIs share your data and services with front-end applications in an easy and scalable manner. With APIs, you can track usage and billing information in real time. But what kind of data or services can be monetized via APIs? Here are some examples:

- Digital content such as maps, images, and analytical data are assets that developers are willing to pay to get access to
- Digital services such as address verification, credit checks, messaging, and location services
- Payment gateway providers charge a certain percentage fee for every payment transaction processed through their gateway
- APIs facilitating the sale of products on an ecommerce platform

How to Increase Revenue Using APIs?

Now let's look at how APIs can be used to increase revenue.

Increase Customer Channels

APIs expose enterprise services to third-party consumers. Developers build apps that consume these services via the APIs. These apps solve different business problems for different use cases. New avenues of customer interaction are opened up by these apps. Traffic through the APIs increase as more number of third-party applications are built

© Brajesh De 2017
B. De, *API Management*, DOI 10.1007/978-1-4842-1305-6_8

and consumed. APIs can also be used by other applications for faster integration with your services. With this integration, additional traffic gets routed to services. Thus, with a variety of apps and newer integrations using APIs, the inbound traffic to your services continues to grow as they are used by end consumers.

To summarize, APIs can be used to increase the inbound traffic as follows:

- To provide extensions to build apps that can be used by end users
- To build a platform for integration on which numerous apps can be created and marketed

Using APIs to build a platform for integration takes advantage of the cross-marketing potential of the app economy. For example, APIs provided by Uber can be used by travel aggregators within their applications to book cabs and provide a completely integrated travel experience for customers. This helps the aggregator to provide a better customer experience and also brings in increased traffic and revenue for Uber.

The goal of an API provider should be to create a growing ecosystem of third-party developers who can build innovative apps using APIs that provide a richer customer experience. This helps multiply the chances of acquiring new consumers and increasing revenue. It also reduces business risks due to referral traffic brought in by other third-party apps using you APIs.

Increase Customer Retention

Customer retention is the next important factor in generating more revenue and profit. After a certain point, business coming from repeat customers is more than that from new customers. Hence, customer retention becomes a key to the success of your business. The more people use apps that rely on your API, the higher the market share for that type of API. After a critical mass of users using an app is reached, it becomes difficult for the third-party app developer to migrate them to another API provider. When end users become accustomed to your apps and APIs, it is even harder for competitors to beat you. Thus it becomes harder for uses to switch to the competition, which increases your retention rate. For example, Evernote is showcasing the work done by third-party developers in their app center. This promotes the work done by third-party developers and encourages users to build products using Evernote APIs. This increases user engagement and retention for Evernote.

As an API provider, you need to build a platform that grows the customer base and increases retention. Build APIs that are easy to use. A smooth and easy developer onboarding process with well-documented and easy-to-use APIs encourages more third-party developers to use your APIs. A well-designed developer portal aids the process. Happy and engaged developers who build good apps increase customer retention and company revenue.

Upsell Premium and Value-Added Services

Value-added and interesting API features can be made available at a premium to those who have purchased access. For example, a communication app may provide two-party voice services for free, but a multi-party voice conferencing service is available at a cost.

Alternatively, customers may be charged after they have exceeded a particular limit. For example, the Google Maps APIs is available for free and with paid options. By default, Google Places API users get free access to 1,000 requests per day. Enhanced access requires credit card validation. This model attracts users to use the API and they pay for increased usage only if they see a value in it.

To use this model, the business should be well established in the market and have a good consumer base. Alternatively, the service provided by the API should be of high business value to the consumer.

Increase Affiliate Channels

As an API provider, sometimes it makes sense to turn third-party developers building apps using company APIs into an affiliate. With an affiliate program, the third-party app developer is also motivated to build apps for your APIs and drive more traffic. This promotes your APIs and drives additional revenue for the company. If any of the apps from the affiliate partners become successful, it can drive tremendous revenue to the API program.

A reverse model can also be adopted, in which the API provider becomes the affiliate for the app. As an API provider, you may want to showcase how various apps have integrated your APIs. From the showcase page, you may drive traffic to your partners and gain a finder's fee for yourself. In this way, third parties help you with app development and joint marketing, and even pay you to drive customers to them.

Increase Distribution Channels

Often a company's business depends on the number of people that get access to its contents and services. The greater the number of people interacting or using the content, the more revenue generated. In this situation, it makes sense to increase the number of distribution channels for its content. New distribution channels via APIs can be used to share the content. Smaller companies may use APIs provided by large API providers to access data and resources shared by them in a revenue-sharing model. For example, small travel companies may use APIs provided by Expedia and Booking.com to provide hotel information within their web sites. Expedia and Booking.com use APIs as a distribution channel to share the hotel information in a revenue-sharing model with others looking to use it. This model of content distribution helps the API provider increase their revenue though integration of their APIs into a third-party platform that needs to use their services. Smaller companies can quickly start their business by integrating these APIs into their platforms, while large companies providing these API gain from the additional business brought in through the integration of these newer distribution channels.

API Monetization Models

An organization's data and services can be exposed and shared with partners via APIs. APIs extend the reach of an organization's core assets and bring in new channels of revenue. The monetization model can be simple or complex depending on the value and use of the assets. However, they can be broadly classified into the following four categories.

- **Free model**: This model is used when the organization has a set of lower-value assets that it wants to advertise through different channels and devices. This model can be used even when the asset has a high demand, but the organization does not yet have the budget to develop and market all use cases for asset use. APIs that provide information about a store branch, or a store location, or product catalog information are examples.
- **Fee-based model**: This model can be used when the organization had assets that are of high value to the consumer. The consumer of this API is ready to pay for the value derived from it. The value to the consumer can be based on per use or based on the kind of data provided. APIs for payment processing, or credit checks, or that provide valuable analytics data are examples.
- **Revenue-sharing model**: In this model, the organization shares revenue earned from the use of its service or product with the app developer consuming the APIs. This serves as an incentive for the developer to build apps for the API provider that can expand its customer reach. An advertising API is a good example. Developers can embed advertisements within their apps by using the advertising APIs. The revenue earned by the organization through advertisements served on these apps can be shared with the app developer. Revenue sharing can also be through an affiliate program. Affiliates get paid a share of the revenue as long as the customers brought in through their network programs remain customers of the API provider. For example, the Rdio affiliate program paid a cut of the subscription fee to their affiliates as long as the subscriber recruited by an affiliate remained a Rdio subscriber. The revenue sharing for an affiliate program can also be based on the type of service the subscribers sign up for.
- **Indirect model**: In this model, the API provider and the consumer mutually benefit. For example, using Facebook and Twitter APIs provide an easy way to sign up users, which helps to continuously expand their consumer base.

The first three monetization models can be further categorized, as described in the next few sections.

Free Model

Free APIs are available for consumption at no charge to the consumer or the end user. Making APIs available for free drives adoption and popularity. As the adoption increases, the brand value of the provider organization goes up. This can also help the API provider expand into newer channels to increase customer reach. Facebook APIs are an example of free APIs. The company's Like and Share APIs embed the Like and Share buttons into any web site or app. This helps Facebook expand its reach and enrich the company's social reach and position. Facebook is a leader in the social recommendation space. As of

2015, Facebook had about 2.7 billion likes per day and around 2.5 billion web sites using its Like button.

The *freemium* model is a variation of the free model. The different variations are based on duration, quantity, or a combination thereof. In a freemium model, the API consumer gets to freely access the API for a certain duration or usage quantity, or a combination of both. Another approach to freemium model is based on the API's features. With a photo API, the free model may provide photos with watermarks or in a lower resolution. However, a paid model may provide images with higher resolution without any watermark.

- **Duration-based free model**: In this model, the consumer is not charged for API usage for a certain duration. For example, the consumer may sign up and get free access to the APIs for the first month and then be charged from the second month onward.
- **Quantity-based free model**: In this model, the API provider provides free access to the API for a certain number of calls. For example, Pearson provides 5,000 API calls for free for its FT Press API. This means that the developer does not get charged for the first 5,000 calls, which gives them the flexibility to try out the APIs before they decide to buy. The Google Maps API provides geocoding services for free for up to 2,500 requests per day. So if a consumer app is making up to 2,500 requests per day, it continues to use the API for free, but has to pay if the daily traffic exceeds this limit.
- **Hybrid free model**: In this model, the API provider combines duration and quantity with free access. The consumer is charged as soon as either of these thresholds is reached. For example, an API call can be free for the first 5,000 calls or the first 30 days. In this case, the API consumer is charged as soon as 5,000 calls are made or after 30 days have passed, whichever occurs first.

Fee-Based Model (a.k.a. Developer Pays Model)

An organization often exposes many assets of high value to its consumers. An organization assigns a price point to its digital assets that consumers are willing to pay to get access to it. For example, Amazon Web Services (AWS) provides a host of services, including storage, databases, computing power, deployment, and management options via APIs. Consumers are willing to pay to use these services rather than hosting them in their own data centers. A fee-based model is perfect for monetizing these assets via API. With a pay-as-you-go model, Amazon generated $750 million in revenue in 2011. NASA saved $1 million after it moved its IT assets to AWS. A fee-based model can be used to monetize APIs that provide access to such assets. Analytical data or payment processing services are examples. The fee-based model can also have different variations, as follows.

- **One-time fee**: In this model, the provider charges the consumer a one-time fee for subscribing. The consumer then gets unlimited access to the APIs that she paid for.

- **Subscription fee**: In this model, the consumer is charged at a regular interval—weekly, monthly, or any other period that the API provider chooses—for use of APIs. A subscription fee for a group of APIs is a typical example of this model. The volume of API calls allowed in a time period may be fixed or volume-banded, in which the subscriber pays for the excess use of APIs beyond the set limit.
- **Pay-per-API transaction**: In this model, there is no minimum fee and the consumer pays for the number of API transactions made. AWS uses this model to monetize its APIs; developers pay only for what they need to use.
- **Pay by transaction volume**: This monetization model is based on the volume of API calls made or the volume of data accessed or returned in the response. This leads to a tiered approach for monetization, in which the rate applied depends on the usage tier. Google charges its AdWords API consumers a certain fee for every 1,000 API calls.
- **Tiered pricing model**: With a tired pricing approach, the consumer is charged different rates for different bands of API calls. For example, 0 to 1,000 API calls in a month may cost $0.02 per API call; whereas 1,001 to 5,000 API calls may cost $0.01 per transaction; and an even lower rate for more than 5,000 calls within the same time period. Typically, higher-usage band rates are less per call than lower bands. Charging lower rates for high-volume usage promotes developers/partners to use higher volumes. The API provider may also set other custom attributes for payment, such as the number of records accessed or returned in the response or the number of bytes/megabytes stored.

Revenue-Sharing Model

In a revenue-sharing model, the API provider exposes its digital assets with partners who sell them on their web sites and via apps. The provider shares a percentage of the revenue earned through the sale of these assets with the third party. Companies like Walgreens, Expedia, and Sears have successfully used this model to sell their products through third-party apps and web sites hosted by their affiliate partners. This model helps the API provider extend reach by expanding business through various digital channels, increase sales through affiliates, and reduce overhead cost with reduction in physical branches. There are various types of revenue-sharing models, as follows.

- **Cost per action (CPA)** : The API provider pays only when a specific action happens, such as a product is purchased or a video is watched.

- **Revenue sharing**: The API provider shares a part of the revenue earned through API traffic routed from third-party apps. The revenue sharing can be as follows.
    - **Fixed revenue share**: The API provider shares a fixed percentage of the sales revenue earned.
    - **Flexible revenue share**: The API provider shares a variable percentage of the sales revenue earned through API sales. The percentage varies based on the volume of sales made over time.
    - **One-time revenue**: In this model, the affiliate partner gets a one-time referral payment for every subscribing customer routed through its web site or app.
- **Recurring revenue**: In this model, the affiliate partner receives a recurring referral payment for every customer routed to the API provider through a web site or app until the subscriber remains a customer of the API provider. For example, Rdio paid their affiliates each time a new subscriber signed up.

In a revenue-sharing model, the API provider needs to generate periodic billing documents and apply a commonly used tax model to the generated statement.

Monetization Concepts

In order to set up monetization of your APIs, you need to be aware of the various concepts for API monetization. This section explains the basic concepts of API monetization.

API Product

APIs should be sold as a product that developers or consumers are willing to use and pay for. An API product is a collection of APIs. Related API resources can be bundled together into an 'API product' and published to the developer community. Developers sign up to use APIs in an API product of their choice.

An API provider can create different products by combining APIs for different use cases. So instead of providing all APIs as a list of resources, related APIs that solve a specific business need can be combined into separate API products. For example, in the telco industry, APIs for sending SMS and MMS and retrieving their statuses can be clubbed together into a single *Messaging APIs* product, whereas billing-related API resources can be combined into a *Billing APIs* product.

API products can also control access to a specific bundle of API resources. Internal APIs resources can be bundled into one API product, while external APIs can also be bundled into another product. API product attributes can also be used to limit the number of API calls allowed for a consumer within a given time interval. So, multiple API products can be created to club the same resources, with different limits set for each of them. For example a Silver API product might allow a consumer to make 1,000 API

requests per day, while a Gold API product could allow unlimited API requests in a day. Another way to configure API products is to club APIs that provide read-only access to resources into a free API product, while APIs that provide read/write access to resources are in a paid API product.

Developers can register their apps and select one or more API product to associate with their apps. The API key associated with an app gets access to all the API resources available within the associated API product.

API Package

An API package is a collection of API products that an API provider wants to monetize. An API provider may create one or more API packages with different combinations of API products. An API package is presented to the developer, who selects the rate plan that they want to sign up for. One or more rate plans for monetization may be associated with an API package.

Rate Plan

A *rate plan* specifies the monetization approach of your APIs. It specifies how you want to charge developers for API usage or share revenue. The rate plan can be a prepaid or a post-paid plan with a charging model that is a fixed fee, or a variable fee, or a freemium model, or may even be customized for the developer. The rate plan depends on the model followed for monetizing the APIs. At the time of registration, developers select an active rate plan associated with the API package. If an API package does not have an associated rate plan, then developers can use the APIs within that package without any fee.

The rate plan can have an associated scope, which controls the availability of the plan to all developers, a select group of developers, or a developer category. This controls the rate plans, which a developer logged into a developer portal can view and select while registering an app.

A rate plan normally defines the following:

- The name and a brief description of the rate plan
- The developers (or the developer category) who can view the plan
- The currency for payment of the rate plan
- Frequency of payment for the rate plan, such as weekly, monthly, quarterly, or yearly
- Payment due dates
- Any recurring or setup fee information

Figure 8-1 below shows the relationship between the API Product, API Package and Rateplan.

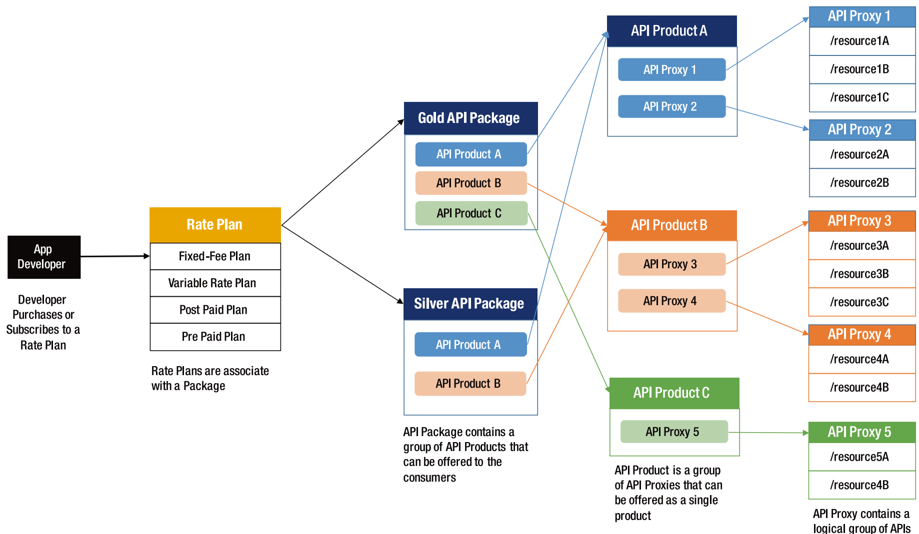

Figure 8-1. *Relationship between components for API Monetization*

Billing Documents

Billing is another important aspect of API monetization. Once APIs have been monetized, it is the necessary to generate consumer bills at regular intervals. Some API management platforms provide an integrated billing solution that automatically generates billing documents such as invoices and revenue share statements at prescheduled intervals. These documents may be viewed in draft state before publishing to their intended recipients. An API provider may want to first make adjustments to the billing documents to increase or decrease the revenue share or fees for a variety of reasons. API monetization also requires that the API provider manage credits, prepaid balances, and refunds. Credits can be provided by reducing the charge in the invoice or by reducing the usage count of the developer's API traffic. API monetization should also allow the API provider to refund a developer's purchase transactions.

Monetization Reports

Monetization without adequate reporting facilities creates havoc. API monetization should also be supported with reports that help reconcile the data, if required. The following are some of the reports that should be generated:

- **Billing report**: This report should provide details of the developer activities that are charged. The report could cover a single billing month or a configurable period.

- **Prepaid balance report**: This report should provide a view of the balance refills that a prepaid developer has done in a month so as to be able to reconcile the payments received from the payment processor
- **Revenue report**: This report should provide a view of the activities that result in revenue through API usage. It helps to analyze the performance and popularity of the API package across developers.
- **Variance report**: This report helps to compare the activities and the revenue generated for two date/time ranges. It shows if there has been any upward or downward trend in API usage.

CHAPTER 9

API Testing Strategy

API testing is different from other GUI-based application testing. It tests the programmatic interface that allows access to the data or business logic. Instead of testing the look and feel of the application, API testing concentrates on testing the business logic of the remote software component and its communication mechanism. Hence, API testing is performed using special-purpose software that sends requests to the API and reads the response received. This chapter looks at the importance of API testing, the challenges therein, various testing considerations, and approaches for testing an API.

The Importance of API Testing

An API exposes business data and services through a defined standard-based interface. Developers use the interface to build applications that are dependent on the API. These applications use the API per the defined contract. The application invokes the API with a wide range and combination of data input. The API response depends on the input data and parameter combination. The API traffic pattern also varies by app usage. The API should be able to gracefully handle the traffic at different loads. With all of these permutations and combinations, the importance of API testing greatly increases. The API should be tested against all expected data inputs to validate that it is behaving as defined in the contract. There should also be testing for different load scenarios. API security testing is yet another important aspect.

Challenges in API Testing

API testing is like white-box testing. Testing an API involves special software that sends messages to an API endpoint as defined in the interface, get the output, and log and analyze the response. The test software should programmatically generate messages with different combinations of input parameters to test the API interface and the underlying business logic. The real challenge with testing APIs is automating the test cases. APIs developed for a business may require the execution of multiple APIs in a particular sequence. It could be a fixed sequence or a dynamic sequence based on the result of the previous API. The test execution may also require passing in dynamic values for different input parameters for the API. Some of these parameters' values may need to be derived

© Brajesh De 2017
B. De, *API Management*, DOI 10.1007/978-1-4842-1305-6_9

from an earlier API execution in the test scenario. All of these requirements necessitate the need for an automated API testing approach. It requires tools and framework that help with automating the test scenarios and reduces the execution time for running API tests with different permutations and input parameter combinations.

Often, APIs are exposed via an API management platform configured to implement different policies on top of the API business logic. These policies may implement security, traffic throttling, data transformation, routing, or orchestration. API testing strategy must consider testing these policies in the API gateway.

Testing API traffic management is tricky. APIs might be protected by policies that throttle traffic based on the nature of the consuming app, the time of day, or other parameters, such as location, originating IP, and so forth. The traffic management policies allow a certain number of requests within a given time interval. The interval can be calendar time or a rotating time. Accurately testing API traffic management rules may require the precise coordination of test executions, which can be a challenge. Traffic management policies may also control the number of simultaneous connections to back-end services. Ensuring that excess connections are not made to the back-end services and that calls are rejected with proper error messages when the threshold limit is reached are also challenges in API testing.

Test data management is another challenge in API testing. Testing a set of APIs may involve managing a wide range of data sets to cover different permutations and combinations of API input parameters. Managing these data sets in multiple test environments could easily become a challenge. It needs a proper test data management approach to effectively manage the test data. If not managed properly, the test results are erroneous and misleading. For example, if an API key or OAuth access token generated in a SIT environment is used to test APIs in a UAT environment, would result in errors.

The other big challenge with API testing is cultural. Since automation is necessary for API testing, testers need to code to test the APIs. Some of the traditional UI testing is manual in nature. And getting testers with experience in coding may not be always easy. Additionally, testers may need to have knowledge of latest API testing frameworks, such as Chai and Mocha, which adds to the challenges.

API Testing Considerations

APIs provide an interface for communicating with back-end business data and assets. The actual business logic is normally out of the scope of the API implementation. Only some exceptional cases—such as peripheral business logic like data validation—may be implemented in the API layer. Hence, API testing should focus on testing the following aspects of the API:

- API interface specifications
- API documentation
- API security

API Interface Specification Testing

An API interface defines the way to communicate with the API. It defines the input parameters required by the API and the expected response from the API. The input parameters may be passed as query parameters, in the body as form parameters, or as payload in JSON or XML formats or even Http headers. API testing should validate the API response when the parameters are passed as documented in the specification. If the API specification describes the parameters to be passed in a query parameter, testing should verify the success and error scenarios for the right and wrong combination of parameters and parameter values. For example, consider testing an API that fetches the product details using an API interface (for example, such as at *https://api.foo.com/v1/products*). This API may accept multiple optional query parameters as inputs—including category, name, and SKU, which may be passed as follows:

https://api.foo.com/v1/products?<queryParamName>=<queryParamValue>

Or, for example, *https://api.foo.com/v1/products?category=Electronics*
The approach to test this API should include the following:

- What is the default API behavior when no query parameters are passed?
- What is the API behavior when the right query parameter with the right value is passed?
- What is the API behavior when the parameter name passed is incorrect?
- What is the API behavior when the parameter does not have any value?
- What is the API behavior when the parameter value is incorrect?
- What is the API behavior when multiple query parameters are passed in the right combination?
- What is the API behavior when multiple query parameters are passed in incorrect combinations?
- What is the default data format for the API response when no information about the requested data format is passed?
- What is the data format for the API response for both success and error conditions?
- What is the HTTP response status code for different success and error conditions?
- What is the API response for unexpected HTTP methods, headers, and URLs?

API Documentation Testing

The API test team needs to validate that the API interface documentation is correct and up-to-date. When new versions of APIs are released, the API documentation should be updated to reflect the changes; otherwise, this can cause frustration among the developer community consuming the APIs due to hindering effective adoption. Hence, with every release of a new version of an API, its corresponding documentation should be tested to ensure that it reflect the latest updates. The API rest team should also ensure that the documentation provides enough information to interact with the API. If the API documentation is interactive, it should also be tested to validate proper responses for every API operation.

API Security Testing

APIs provide an access point for business services and data to consumers—internal and external. Depending on the criticality of the data, the API is a point of attack. Hackers may want unauthorized access to system resources for undue benefits. Hackers are always in search of security holes through exposed APIs. There could also be DoS attacks that put an API in an unavailable or unstable state. Hacked APIs can damage the brand value of an enterprise. Hence, testing API security is very important. This section looks at the various aspects of API security testing.

Authentication and Authorization

Testing access mechanism and access control policies of an API is of paramount importace. If an API is exposing a protected resource, the security testing must ensure that only authenticated clients are able to access the APIs. Testing the security policies can include testing API access protected via API Key or Mutual Authentication using PKI or OAuth/OpenID token. Testing OAuth scope validation to ensure that APIs can be accessed using tokens having the right OAuth scope, should form part of the API testing strategy. It is important to validate that right http error codes are returned in case of authentication or authorization failure while accessing an API.

API Fuzzing

API fuzzing is an attack in which the attacker tries to get information about the API and the system resources by sending random input parameters. The attacker sends all possible permutations and combinations of input parameters and analyzes the response in an effort to gain insight into the system resources. The attacker may try to analyze the error messages for various data combinations to understand system behavior. Hence, APIs should be tested with all permutations and combinations of input parameters, and the responses should be analyzed to ensure that the information provided in the responses is appropriate. Error responses provided under different combinations of invalid input data should only provide optimum information as required for the API. It should not reveal information about the internal data structure or database query, file system information,

or any other information that can potentially be used to get unauthorized access to the system. For example, for an invalid input to fetch data for an entity, the response should not have any information about the SQL queries that failed to execute in the back end. The error response should indicate only the parameters that are invalid.

Malformed Payload Injection

APIs need to be protected against malformed or unexpected message injection attacks. Very large JSON or XML payloads, JSON payloads with long attribute names or values, and payloads with highly nested structures are used as means for attacking the underlying systems. Processing complex payload structures can take up lot of system resources and CPU cycles. A high volume of such requests may potentially bring down the underlying systems, thus impacting the overall system availability. Hence, API testing should test the API's ability to withstand such vulnerabilities. An API testing strategy should include test cases that test API behavior when a request payload is an unexpectedly large size or has an unexpectedly complex and heavily nested structure.

Malicious Content Injection

Injecting SQL scripts, JavaScript, shell script through input parameters or payloads is also a common form of attack. These scripts, if executed on the server or by a third party, may provide vital information to unauthorized users. The scripts might also be damaging enough to modify or delete data—impacting the business severely. Hence, API testing should test the API behavior when such scripts are injected in the API requests. The API should reject such messages with appropriate error messages. Testing the presence of malicious script should include testing the API behaviour when different types of script like SQL, javascript, shell script, regular expression, XPath, XQuery, python or groovy script are injected throuogh the API payload.

Testing API Gateway Configuration

In many scenarios, a business service is exposed through an API gateway. The API gateway enforces policies in the request and response flow of the API, which may perform one or more of these depending on the requirements: security, throttling, data validation, transformation, routing, error handling, caching, and mashup. Most of these are implemented either using policy blocks or filters within the flow. Unit testing of the API proxy must test the execution of these policy blocks and the conditions applied, if any. Verification methods look at the input and output of each of these policy blocks in the debug trace. If a policy block is to set a local variable or an HTTP header, you can validate whether it is being done properly by looking at the trace output. Similarly, if a policy is supposed to transform the message payload, the output of the policy execution should be the successfully transformed payload. It is also important to look at the average execution time of each of these policy blocks, either in debug logs or in the debug trace of the message flow execution. This can help identify potential performance bottlenecks at

an early stage of the testing and reduce efforts to troubleshoot API latency issues at a later stage of performance testing.

API Performance Testing

APIs are no longer seen only as mechanisms for integration but have become mainstream for the delivery of data and services to end users through various digital channels. This increases the demand on APIs to perform well under loads. The overall performance of a client app is dependent on the performance of the underlying APIs powering the app. Hence, the importance of performance and load testing for APIs increases greatly. This section looks at the strategy for load testing an API.

Preparing for the Load Test

It is important to plan well and have a well-defined load testing strategy. Planning for API load testing starts with identifying the list of APIs to be tested. Load testing is most effective when the workload for the API is as close to the real expected traffic. It is not useful to know that an API can handle say 500 transactions per second without knowing whether the real traffic is higher than or lower than that. The first step in preparation for a load test is to gather information on the performance requirements that the APIs are expected to handle. This includes the following information:

- The average throughput in terms of the number of requests per second for each API deployed on the platform
- The peak throughput that projects the maximum number of requests that each API is expected to handle at any given point in time (normally during peak loads)
- Throughput distribution across all the APIs deployed on the platform.
- The traffic distribution patterns of client apps using the API helps predict accurate API usage
- The number of concurrent users expected for each client app using the API, which predicts the total number of concurrent connections that the API platform is expected to handle under load.

Having decided on the performance requirements for API testing, there may be different approaches to actual test execution. Actual test execution can start with the generation of repetitive loads for each of the API endpoints. This establishes the upper bounds of the performance that may be achieved in the test platform. If it is low, you should look at options to tune and optimize the API and platform parameters for a better throughput. Adding hardware or instances in the cluster configuration can be the second option to look at if the results from the repetitive load test are not satisfactory.

Once the platform has stabilized and the upper bounds of load testing have been determined from repetitive load tests, it is time to simulate a realistic traffic pattern. Using

a real traffic scenario might be ideal, but not practical for various reasons. The simulated traffic should consider the following:

- Traffic distribution across various deployed APIs. For example, 45% of calls are to product catalog APIs, 35% of calls are to customer information APIs, and 20% of calls are o payment APIs.
- Traffic growth pattern during the day. For example, gradual increase or sudden spikes or a constant load through out.
- API traffic for both success and failure responses.
- Geographically distributed API traffic to test for any network traffic congestion at high loads.

Data from production traffic logs of already deployed services can provide information to simulate realistic traffic scenarios for load testing.

API performance testing should consist of the following, with an aim to find the different performance parameters of the APIs and the API platform.

- **Baseline testing**: The objective of this testing is to find out how the system performs under normal expected load. The results from this test should be used to analyze the average and peak API response time and error rates. The CPU and memory utilization of the platform should also be looked into to eliminate any resource bottlenecks.
- **Load testing**: During load testing the load is increased to study the API performance under growing API traffic volumes. Performance metrics, such as response time, throughput of the APIs should be looked at to review the performance under load. The aim of this testing is not to find the breaking point, but to understand the expected system behavior and capability to handle expected peak loads. Server performance metrics, such as CPU utilization, heap memory utilization, network port utilization should be analyzed to understand the state of the platform and its ability to handle high load.
- **Stress testing**: The goal of stress testing is to find the breaking point of the platform. It is used to determine the maximum throughput that the system can handle. In this form of testing the API traffic load is gradually increased till a breaking point is reached when the performance starts to degrade or errors from API calls start to increase.

- **Soak testing**: Soak testing determines whether there are any system instabilities in long duration testing. The baseline test may be executed over several days or weeks to learn about any unwanted behaviors that may occur when the system is used for a long time. The aim is to discover any issues with releasing system resources and make them available for the next cycle of execution. If system resources are not getting released periodically, there is a high probability of the system crashing under sustained high loads. Normal baseline testing or load testing may not be able to unearth such problems, and then the importance of soak testing increases.

The load test strategy should consider the environments for doing the load test. A pre-production setup that is a replica of the production setup would be an ideal for load test. However, that may not be available all the time due to practical reasons. Hence, a dedicated load test environment that is a scaled down version of the production environment may be used for load and performance testing. Considerations should be made to scale down expected throughput by the same factor while performing the load test.

Setting up for the Load Test

Having identified the environment for the load test and approach, it is time to identify the right set of tools to execute the load test. There are many tools available to perform API load testing. Let's talk about the most commonly used tools for doing the execution testing.

- **JMeter** is an open source Java-based tools with a powerful GUI used to easily simulate non-trivial HTTP requests to test REST APIs. It allows you to model complex workflows using conditions. A test plan in JMeter allows you to define the thread group that is used to simulate end user behavior in terms of the number of concurrent users, the ramp-up time, and the REST API request sent by them. The HTTP request is parameterized. Parameterizing the test requests reuses it with different parameter values and dynamically passes the execution results from one test to another. Assertions are added to validate the test results automatically. Listeners provide widgets that are used to view the test results. JMeter is one of the best open source tools for functional testing used to model complex user flows using conditions. The availability of a large number of community plugins extends the built-in behavior. Its non-GUI-based option runs JMeter for test execution in an environment that does not support rich GUIs, such as Linux-based environments.
- **LoadUI** is a commercial API load-testing tool from SmartBear. LoadUI has an advanced feature that allows you to do distributed load testing by distributing the load tests to any desired number of LoadUI agents. It also allows running multiple test cases simultaneously and long running tests that may run days or weeks.

- **Wrk** provides a command-line interface to test REST APIs. Being multithreaded, it is able to take advantage of the underlying multicore processor; hence, it is used to simulate really high loads. The default reporting format for Wrk is limited to text only, which sometimes makes it difficult to interpret test results easily. However, its ease of use to simulate high loads makes it one of the best tools, when the goal is to find the load than an API can handle.
- **Vegeta** is an open source HTTP load testing tool for performance testing of REST APIs. It is useful when the aim of the testing is to learn how long the service can sustain a constant load of *x* requests per second. This is important when you have data about the peak load that is expected for an API and you want to find out how long the service can sustain that peak load before you start seeing a drop in performance.
- **BlazeMeter** and **Loader.io** are two tools that run the load test for APIs in a cloud platform. They provide load-testing infrastructures as a service in the cloud. The cloud-based approach reduces efforts to set up the environment for load testing. BlazeMeter provides the option to upload a JMeter test plan and run it from its cloud infrastructure.

API Performance Test Metrics

Performance testing of API should look at the following metrics to measure the performance of the individual API and the platform.

- **API response time**: Measures the overall end-to-end response time of the API. Determines the time in which an end user is expected to get a response from the API. Minimum, average, and peak API response times should be measured as part of API performance testing.
- **API target response time**: Determines the time it takes for the API back-end systems to respond. If an API is exposed through an API gateway, it measures the response time of the target back end for the gateway API proxy.
- **API latency**: Measures the latency introduced by any intermediary, such as an API gateway used in the API architecture.
- **Throughput**: Measures the number of requests processed in a second. Normally, this is measured in transactions per second (TPS).
- **Success and error rates**: These metrics are important for API performance testing. They measure the number of requests successfully processed under load.

- **CPU utilization**: Measures the capacity of the system under load. A low CPU utilization means that the system can handle a higher load. Higher CPU utilization is indicative of a system under stress.
- **Heap memory utilization**: Indicates how system memory is being utilized to process requests under load. The system RAM may need to be increased if heap memory utilization stays at its peak throughout the performance test. Low available memory may impact the overall performance of the APIs.

Selecting The Right API Testing Tool

Having looked at the various aspects of API testing, it becomes important to look at the feature that should be in an API testing tool to make it a success. The following lists can help with selecting the right API testing tools. They cover the features that an API testing tool should have and other nice-to-have features.

Must-Have Features

The following are must-have API testing tool features.

- API test tools should support automated API testing to cover a wide range of scenarios.
  - It should test success conditions with different data combinations
  - It should test error conditions and corner cases
- The automation of functional test cases should support the following and must be repeatable for multiple deployments and environments. The tool must have the following capabilities for API functional testing:
  - It should support the creation of HTTP requests with different combinations of verb, headers, query parameters, and payloads
  - It should support payload generation in multiple data formats (JSON, XML, SOAP) and even binary format
  - It should support automated creation of API request templates by importing WADL, RAML, Swagger, API Blueprint, and so forth
  - It should support automatic data validation for request/response messages based on a defined schema
  - It should support parameterized test creation

    - It should provide the ability to define test flow logic
    - It should support test visualization to understand the failure points of API executions
- Test asset management capabilities
    - It should group and tag test cases
    - It should search test cases and make changes to a group of test cases through find and replace
    - It should easily create new tests or update existing test assets based on changing API demands
    - It should manage test data for different environments
- Security testing capabilities
    - API authentication and authorization using protocols such as OAuth, OpenId, SAML, Basic Authentication, and SAML
    - Message encryption and decryption
    - Penetration attacks, such as SQL/ script injection, malformed payload, virus attacks, parameter fuzzing, and so forth
- Performance testing capabilities
    - It should calculate API response times, throughput, and error rates
    - It should simulate regular performance loads
    - It should simulate unpredictable and volatile performance load with valid payload
    - It should simulate spikes and sudden bursts of traffic in consuming apps

Nice-to-Have Features

The following are some of the nice-to-have features in an API testing tool.

- Record and replay API traffic
- Integration with requirements management and issue tracking systems, such as JIRA and QC
- Integration with CI tools such as Jenkins, Cloud Bees, Cruise Control, and so forth
- Federated and cloud testing ability to execute test cases in a distributed scenario

- The ability to run in non-GUI mode with a command-line interface
- The ability to schedule test cases

Common API Testing Tools

The following are some common API testing tool products.

Unit Testing Tools	Integration Testing Tools	Performance Testing Tools
JUnit	JMeter	JMeter
Curl	SOAPUI	LoadUI
Postman	APICLI	Grinder
Advanced REST Client	Cucumber	Curl-Loader
Mocha	Jasmine	Wrk
Chai	Mocha	Vegeta
TestNq	jBehave	BlazeMeter
QUnit	NSpec	
PyUnit	SpecFlow	
Hurl.it		

CHAPTER 10

API Analytics

As the old saying goes: "You can't manage what you cannot measure." This holds true even for the enterprise API programs. API analytics provide data and trends about APIs. API traffic flowing through an API management platform can provide lot of useful insights to businesses, which can help to effectively govern an enterprise API program. It is important for an enterprise to measure the success of its API program. This can help provide information that can be used by actors in various roles in a wide variety of ways to make the right decision. In this chapter, you look at the various API metrics and learn how to effectively use API analytics to drive the success of an API program.

The Importance of API Analytics

An API management platform collects a wide variety of operational and business data as traffic flows through it. The data collected is then analyzed to provide metering and monitoring capabilities. This data should be regularly analyzed by the business to make improvements for the API program and channelize its investments. Only then can the business reap the benefits of the investments made in its API program for digital transformations. Depending on the data collected, API analytics data can provide answers to the following questions:

- How has API traffic trended over time?
- Who are the top users of the API—apps as well as end users?
- Which developer app is generating the maximum traffic for the API?
- How many developers have signed up for the API program?
- What is the most recent trend for developer adoption of APIs?
- How has been the performance of the APIs? How has been the performance of the backend services?
- What is the API usage pattern across geographical regions?

API Analytics dashboard can provide information about API traffic trend. It shows how APIs are used over time - the peaks and the troughs of API traffic. Aggregated API traffic data can show traffic distribution over a day, week, month, quarter, or even

© Brajesh De 2017
B. De, *API Management*, DOI 10.1007/978-1-4842-1305-6_10

a year. Average API response time aggregated over time can help you understand API performance during peak and low traffic. API traffic distribution identifies the most popular APIs. Data on an API's users identifies the most popular app. All of this information can help improve the quality and performance of APIs and provide valuable insights into API governance.

API Analytics Stakeholders

The data collected from API traffic for analytics can be used by different stakeholders in a variety of ways. The following are the main stakeholders for API analytics:

- API product owner
- API team
- App developer
- Operations team

APIs are products that you sell to your customers. Hence, as a product owner of the API, the business user would be interested in knowing how their product is doing. Without proper insight, it is difficult to make the right investments into the API program and make it successful. A business owner of an API program would be interested to get answers to the following questions:

- **How has the API been adopted?** An insight into API traffic data can provide an answer to this. A continuously increasing traffic trend over a period of time can be a fair indication of the successful adoption of the API. A constant or a falling trend in API traffic means that there has been low adoption of the APIs.
- **How many new applications are using the API?** A report on the new apps registered to use an API can help you understand the interests of the developers. The number of new apps registered over a period of time is a good indicator. But just looking at the new app registration data can be misleading because developers may register apps but not use them to invoke the APIs. Hence, it is also important to look at the traffic generated by these apps to measure the real adoption of APIs.
- **How many active developers are there?** A report showing the top developers' app traffic can provide information about the developers who are actively using the APIs.

- **What is the geographic distribution of API usage?** As an API product owner, I would be interested in knowing how the API has been adopted across different geographic regions. Depending on the services provided by the API, its adoption could be concentrated in only a few geographic locations or it could be widespread. For example, Google Maps APIs have a wide geographic distribution, indicating that it is widely adopted by users in different geographic locations. If an API is designed to be used across the world, traffic distribution by geographic region would be of interest to the API product owner to see their adoption in different countries. If the traffic is coming only from one geographic region, it means that its adoption is limited.
- **How are investments in the API being used? Is the API program bringing in new business?** An API traffic report can help answer these questions for the business owner. An increasing API traffic trend means end users like the API. Depending on the monetization model setup, this would mean an increasing trend in direct or in-direct revenue from the API. Custom analytics reports can help drill down to specific business transactions to get more insight into the API's business impact.

An API team is the technical team involved in the development of APIs. API analytics reports can provide the following information to help analyze and optimize the performance of an API:

- Traffic
- Response time
- Message payload size
- Errors
- Cache performance
- Back-end service performance
- Developer adoption

With this information, the API team knows how the API program is doing overall, how individual APIs are performing, and how to improve the API performance. A higher than expected response time may impact the adoption of the APIs due to a poor overall user experience. Hence, the API team needs to look at the root cause to reduce the response time and improve overall performance. Response caching may help improve response time and may be an option to consider for performance improvement. Message payload size is another consideration in improving API performance. Large payloads not only impact network performance due to bandwidth constraints, but can consume more CPU cycles for message processing. Hence, optimizing the message payload size can improve API performance and help drive its adoption.

App developers are the consumers of APIs. These developers are innovating with your APIs and building creative apps that help drive revenue to your enterprise. Their innovative apps help provide better user experiences. By sharing analytics information with app developers, you get better apps. Analytics help app developers know how their apps are doing and how much they are contributing to the bottom line of your enterprise. App developers want to know how they can improve their apps. Ultimately, everyone wants happy end users.

The operations team uses API analytics reports to understand traffic patterns and anticipate when to add back-end resources or make other critical adjustments. An increasing API traffic trend associated with a degradation of API performance may indicate that the underlying infrastructure is reaching its capacity and may need to be supplemented.

API Metrics and Reports

A lot of operational and business data can be collected from API traffic. The metrics can be divided into traffic metrics and developer metrics.

Some of the key API metrics that should be analyzed are as follows:

- API traffic
  - Total API traffic across all APIs
  - Traffic distribution and trends by API proxy
  - API traffic by business or technical assets
  - Top APIs and methods
- Response time
  - Average response time of the API
  - Target service response time
  - Request and response processing latency on the API gateway
- Error rates
  - Error distribution over a period of time
  - Error distribution by APIs
  - Target service error rate
  - Error distribution by HTTP error code: 5xx, 4xx

- Message payload size
    - Average request payload size
    - Average response payload size

The following are some of the key API developer metrics:

- Developer engagement
    - The total number of developers registered with the API program
    - The number of developers with apps
    - The number of active developers
    - Traffic volume trend by developer
    - Traffic generated from developer apps
- Traffic composition
    - Top 10 apps' traffic
    - Top 10 developers' traffic
    - Top 10 API products' traffic
- End user engagement
    - Geographic distribution of API traffic
    - API traffic distribution by device: device type, OS families, agents, browser type
    - App error rates

Custom Analytics Reports

Many times, default analytics data captured by the API management platform from API traffic may not be sufficient enough to provide all business insights. You may need to capture certain custom data from the message payload and derive useful analytics information from it. Many API management platforms provide the facility to extract custom data from messages and log it into their analytics database. This data may be extracted from message headers, query parameters, or payloads, and used to create meaningful custom analytics reports. For example, in a hotel reservation API, a business might be interested in knowing the distribution of reservations by city or hotel. Such information can help businesses take actions that result in better customer satisfaction and grow business across cities.

Ensuring good API performance and helping highly engaged developers build apps with your APIs is key to the success of an API program. API analytics provide insights into metrics that should be monitored regularly to ensure the success of an API program. A dip in API traffic can mean user interest is shifting away from the API, the reasons for which could be many. It could be due to the API's poor performance or customers moving to services provided by other competitors in the market. Business owners should critically look at API analytics reports on a regular basis and reflect on how they should further fuel and tweak their APIs to make them more competitive and popular in the market. Proper implementation of API analytics holds the key to the success of an enterprise's API program. API management is incomplete without the proper insights provided by API analytics.

CHAPTER 11

API Developer Portal

Success of an API Program for an enterprise depends on the proper planning to build the right API at the right time to meet the current and growing needs of the consumers. APIs that power the digital business should not only be built correctly with clean and well documented interface, but should also be published and socialized with a developer community that can help in the adopting the APIs at pace. A good API developer portal helps to easily onboard developers onto the API program. In this chapter we would look at the role of an API Developer Portal in API Lifecycle and what should be the features of a good Developer Portal so that it can attract developers and facilicate their onboarding onto the enterprise API program.

The API Lifecycle

The life of an API starts with designing the right interface using an API-First approach. Once the interface is designed and documented, it is built and deployed on an API platform that provides the runtime infrastructure. The API platform should also help publicize and socialize the APIs with the developer community to accelerate their adoption. To support the evolving requirements, older versions of APIs should be slowly deprecated and retired, giving way to publish newer versions of the API. An API developer portal plays a vital role in managing the lifecycle of the API by providing a mechanism to publish and socialize APIs.

Publishing and Sharing APIs

A well-built API will not fetch the desired business benefits unless it is publicized. People—especially developers—looking to build apps using APIs need to know about it. So an enterprise needs to have a mechanism to publish the details of the APIs and provide a platform for the developers to easily find and use the APIs. Developers need to know the details about the API.

An API should be well documented to provide information about the endpoint, the input/output parameters, the SLAs, monetization model/rate plans, and other information. The API provider needs a social enterprise API platform to publish information about the APIs, whether the API is for internal or external use.

© Brajesh De 2017
B. De, *API Management*, DOI 10.1007/978-1-4842-1305-6_11

It needs to be marketed well. An API provider needs a platform to market his API with a powerful search-driven catalog that offers social features such as ratings, reviews, likes, and more. The API needs to be published in a catalog with appropriate descriptions and tags to easily search for potential consumers.

After developers have found an API in a catalog, they would like to know its fitness for their app development. This is where developers browse through API documentation, blogs, and forums to read user feedback on the API and evaluate it. They would like to know how interested the community is in the API to better understand the level of adoption and support for the API. An active community allows developers to ask questions and get honest feedback from fellow developers who are using the API. Good feedback in forums helps drive faster adoption.

The Importance of the API Developer Portal

A developer portal provides the platform for an API provider to communicate with the developer community. It helps communicate static information about the API, such as documentation and terms and conditions for use. It can also include dynamic social content contributed by the developer community, such as forums and blogs.

Creating a good API is only a small step in building a successful API program. API providers need to expose and publicize information about the API, provide documentation to educate developer communities about the API, and provide a platform to easily register developers and their apps. Developers and users of the APIs should be able to provide feedback, get support, and make new feature requests that can help the APIs to evolve. App developers should also be able to submit and share their own content for others to use.

An API portal is a single point of information for an app developer looking to use APIs in building an app. In addition to providing documentation for the API, the portal should provide a platform that allows users to easily play around with and test the APIs; this helps developers better understand its usage in building apps. Embedded API test consoles and smart docs generated from API specifications can be used for testing the API interface within the portal.

A developer portal should provide developers with analytics information for API usage. App developers should be able to monitor the API usage pattern for their apps. API analytics information can include traffic trends, API performance metrics, and error rates for the API and apps.

Supporting App Developers

App developers are the real users of the API. Innovative apps built by app developers increases API adoption and usage. Hence, as an API provider, it becomes even more important to effectively support the app developer community to accelerate the adoption of your API. An API provider should provide support to app developers to drive the API's social adoption. The support provided can be in various forms.

- Good documentation to easily understand the API interfaces
- A test bench to play with the API and understand its behavior
- SDKs and code samples that developers can readily use in their apps to invoke the API
- A Q&A forum for developers to help each other by answering questions asked by others
- An indexed forum to search for errors, issues, or questions and get immediate answers to already solved problems

An API provider should put lot of effort and time in building a thriving community. The right investments in building the app developer community with the rights folks can help pay enormous dividends later and make the API program successful. An API Developer Protal should provide the following social collaboration features to support the developer comnunity.

Invitations

Invitations are a popular way to socialize your APIs. They are an easy and effective way to build a community for API users. A developer portal should facilitate sending invitations that create a community of interests around the API. Any user—an API or app developer, or a business administrator—should be able to invite others to start using or following an API. You can encourage people to invite their contacts too. This can help build a huge social community connected to the API.

Social Forums

A social forum helps app developers share their experiences with using APIs. It can connect developers who are building apps with the APIs. They can discuss best practices for using the API, as well as any limitations and how to overcome them. They can post their comments and ideas, ask questions, and even raise support ticket with the API provider. The view available to an app developer can depend on the assigned role. An administrator might be able to see all issues logged and all unanswered questions; whereas an app developer may only see the answered questions and the check status on the issues that they logged. As the community around the API matures, the forum might act as a platform where API users and app developers answer questions or make comments on questions asked by fellow users.

An enterprise API platform needs to be social. App developers and API users should be able to follow APIs, apps, business organizations, developer groups, or other users. A personalized dashboard for each app developer should provide them an aggregated view of all items of interest. It should provide a centralized dashboard where they can keep track of what's going on with everything that they are interested in.

Federated Developer Communities

The success of an API initiative depends on its adoption by the developer community. A developer portal allows developers to sign up for the API program and get access to the API. The portal helps API providers build their own developer community. But a federated developer community might be a better idea. With a federated approach, developers of other API providers, who are partners or are like-minded, may want to share the same API keys with developers. So if a developer signs up with a company's API program and obtains an API key, the same API key can be used to access APIs provided by other partners of the company.

An enterprise API program should support the concept of an API provider federation. This brings together communities by providing developers with access (through proper authorization) to any API from any provider by using a single API key. This helps API providers easily extend the reach of their APIs to a wider community of developers. However, all of this first needs a deeply federated trust and permissions model to be established between the API providers. The model should allow API providers to opt in or out of the federation model and to choose the partners with whom they want to federate.

Types of Portal Users

There are three types of portal users: app developers, the API team, and the API product owner. The **app developers** use the APIs to build apps. They refer to the API portal to learn more about the APIs that they can use for developing apps. They look for API documentations and a sandbox environment to try out the APIs. They register for an account in the portal, register their apps that will use the APIs, review the terms and conditions for API usage, interact with other developers in the community through forums, and view statistical information about their app usage on a dashboard.

The **API team** is the provider of the APIs. They create the developer portal to publish information about their APIs for the developer community. The API team sets up the portal and the workflow for developers to register and obtain an API key. The workflow may be simple automatic approval or it might involve manual verifications and approvals. The API team sets up the API portal to do the following:

- Automatic or manual approval for API key generation
- Publish and maintain API documentation
- Provide and maintain a forum for app developers to connect with other developers in the community
- Provide a test bed for app developers to test the API interface through an embedded test console
- Provide contact and support for app developers

- Enforce a role-based access control mechanism for developers to access various features in the portal
- Customize email notifications sent to administrators and developers for user creation, app registration, and approval

The **API product owner** is the person or organization responsible for the productizing the APIs. They are responsible for identifying the APIs to be built based on market research and user stories. They work with sales, marketing, and other stakeholders to create an API product that will sell. They are responsible for understanding what the app developers want. They help to translate the business requirements into terms that the API team can use to actually build APIs that will sell. The API product owner would be responsible for the following:

- Defining how APIs should be packaged into a product
- Defining the process and rules for app approval
- Define the pricing and billing plans for the API products

API Developer Portal Features

As an API provider, it is important to understand the features that the API portal should have. The portal should attract app developers and provide all the necessary information that they might be looking for to get started with using the API. The following are some of the features to consider while building or customizing a developer portal.

- **User registration and login**: The app developer should be able to easily sign up for the API program and start using the APIs. The registration process should be simple and easy. Requiring a lot of information to register or a complicated registration process may annoy developers and hold them back from signing up for the API program. Hence, the developer registration form should be simple and easy. A minimalistic approach for user registration is recommended. When a developer registers, the approval process can be automatic or manual. In either case, an email should be sent to the developers that confirms registration. The administrator should also be notified of developer registration and be provided with a link to approve, if so required. In a manual approval process, an email should be sent to the developer once their registration request is approved. The login process after the registration should be easy but secure.

- **User management**: A developer portal administrator should be able to create and edit users. Administrators may directly create developer accounts through the portal. Upon successful registration, the portal should send an email to the developer informing her that the account is created. The administrator should be able to modify the status of the developer from active to blocked if so required, or update a developer's profile information. Role assignment is yet another aspect of user management. Admins should be able to assign roles to registered users to control the privileges and access rights of the user based on custom roles, and signed-in and anonymous users.
- **API documentation**: The portal should be the source of all information about the APIs. It should provide all documentation for the API, such as interface specifications, FAQs, tutorials, examples, and sample code. Getting started and how-to guides on using the APIs help accelerate API adoption. Including request and response messages using real-world examples helps developers easily understand the API interface. The API documentation can also include a reference guide that explains common vocabulary, data formats, best practices, common HTTP response codes, and error messages.
- **API test console:** A console for developers to test an API helps them explore and play around with it without writing any code. Developers can use the console to submit a request to the API and view the response. A smart doc for an API also helps developers easily learn how to use the API.
- **Forums and blogs**: Community-contributed content, such as threaded discussion forums and blogs that describe the developers' experiences, help build an engaged developer community.
- **App registration and key management**: When an app developer wants to create an app using the API, he needs to get an API key. For this, developers need to register their apps with the API provider in the portal. The portal should allow developers to register their apps. The approval for the app registration can be automatic or manual. In an automatic approval, the API key is generated immediately upon registration. The approval process can be manual if any background verification needs to be performed before approval. In a manual approval, the API key is generated only after the app registration has been reviewed and approved by the administrator. An administrator may also revoke keys or regenerate new ones.

- **Email configuration**: The API portal should send email notifications when developers sign up for the API program or register their apps. The API portal administration should provide the facility to configure the email templates with the content and format of the emails to be sent. The admin should also be able to configure when emails should be sent to developers.
- **Dashboard**: App developers like to view statistical information about their apps and the APIs used by their apps. They like to know the number of users using their apps, the number of calls made by their apps, and the various APIs and methods used by the apps. The developer portal should provide a dashboard for app developers to view all of this information and much more.
- **Support information**: The API support information in the portal should provide the developers' contact information so as to reach in case of any queries or issues with using the API. The contact information can be a phone number or an email address. The support page in the portal can include quick API status information. The status could be active, under maintenance, deprecated, or retired. The support page can also include FAQs, notices, or coming soon information of interest to the developer community. Notices could cover latest updates or activities related to the API. Coming-soon information provides a list of upcoming API features.
- **Search**: A search facility within the developer portal is very useful feature. It helps developers quickly search for information. They can search for APIs of interest, or for specific information within the API catalog, or specific content within the forums or blogs.

The Relationship Between a Developer Portal and an API Gateway

The API developer portal is the door to an enterprise's API program. It lets developers sign up and register their apps to use the APIs. An API gateway provides the API runtimes. An API key is generated on successful registration and is stored in a database that is referenced by the gateway for API key validation. Not only that, all app attributes, developer information, and details about the organization are provided as part of the onboarding process is stored in the database that the gateway references for validation purposes.

The portal acts as a client for the API gateway to store and fetch API-related information. Normally, the portal makes REST API calls over HTTP or HTTPS to communicate with the gateway. For example, when a developer registers a new app, it makes a request to the gateway to send information about the app to the gateway data store. Every instance of an API developer portal must be associated with an API gateway that hosts the APIs and provides the runtime support. Both the portal and the gateway can be deployed on cloud or on-premises. A hybrid deployment model in which the portal is on the public cloud while the gateway is set up on-premise is also possible.

CHAPTER 12

API Governance

API governance is distinct from SOA governance. API governance provides a policy-driven approach that helps to enforce standards and checkpoints throughout the API lifecycle. It encompasses not only the API runtime, but also design through development processes. It includes the guidelines, standards, and processes to be followed for API identification, interface documentation, development, testing, deployment, run, and operation. Standards and principles defined by API governance provide API quality assurance, such as security, availability, scalability, and reliability. It underpins the API enablement aspect that is critical for the successful adoption of APIs.

The Scope of API Governance

API governance encompasses activities, starting with the API proposal all the way to its adoption, through requirements gathering, build and deploy, and operations during general availability. Figure 12-1 shows the high-level phases where API governance plays a critical role.

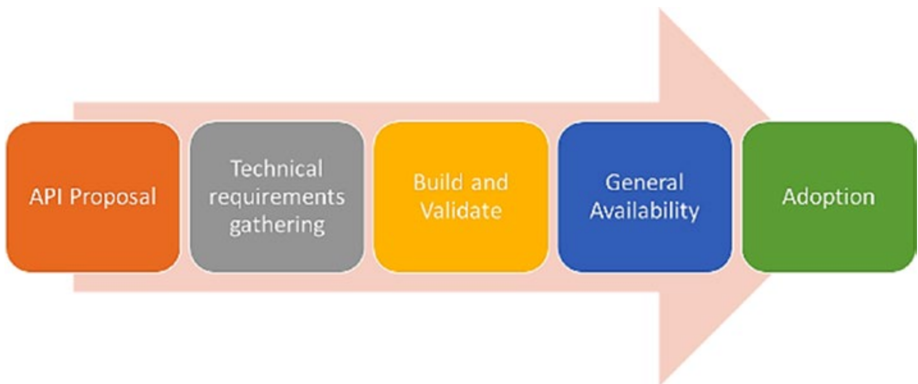

Figure 12-1. *API Governance Phases*

© Brajesh De 2017
B. De, *API Management*, DOI 10.1007/978-1-4842-1305-6_12

The following describes the phases.

- **API proposal**: This is the first stage, where new API or change requests are proposed by the organization. This is done due to new business agreements, changes in existing business agreements, or a new change request submitted to the API governance body. Community managers create an ecosystem: talk to partners, competitors, regulators, and independent developers to identify and propose APIs that are aligned to the business strategy.
- **Technical requirements gathering:** After the API proposal, the next step is to gather the requirements and create the specifications for the API. API architects and business analysts work together to create API definitions. The API governance defines the process and standards for the API interface definition. The following are some important questions that API governance must consider and enforce:
    - Which API specifications standards should be used? Swagger or RAML, or any other standards for API interface documentation?
    - Who is responsible for the review and approval of the API specifications?
    - What is the API versioning approach? When is a new version created?
    - Is there JSON schema versioning?
    - Which back-end services are connected to these APIs? Are there field mappings?
- **Build and validate**: After the API specification and requirements are finalized, the development process starts. The scrum master, API team, and all the members in the API program work together in this phase. Test scripts are created and APIs are validated for compliance to API specification. API governance during the build and validate phase must define guidelines for the following questions:
    - What tools are to be used for the entire API development lifecycle?
    - Which source code repository must be used for configuration management?
    - Which best practices must be followed for API development?
    - What should be the testing approach and the tools to be used for API interface, functional and load testing?

    - What is the review process? What are the checkpoints to ensure the quality of APIs?
    - Which policies must be implemented for APIs?
    - Is there isolation between the non-production and production environments?
    - How should the API interface lifecycle be managed?
    - What is the promotion process, from the lowest development environment to production, and eventually to the retirement of the APIs?

- **General availability**: After the implementation is done and the APIs are deployed, they need to be published to the developer portal for API subscribers. Before publishing an API, consider commercial questions, such as how to monetize the API. Since APIs may expose data to the consumers, the terms and conditions for the use of the APIs and the associated data should be finalized with the legal team. The marketing team should review and ensure that brand use and quality are satisfactory. After approval from the commercial, legal, and marketing teams, the API can be deployed to the production server and released for beta or general availability. After an API has been made available for general use, there must be proper tracking and metrics to provide information that answers the following questions:

    - Which API is deployed to what environment?
    - What is the performance of the deployed API?
    - Which apps are using which API?
    - What are the usage patterns for the API by app, geography, and time?

■ **Note** The API governance process must define all the API metrics to be tracked and how the tracking is done. It must define the necessary steps to be taken in case of any service-level agreement violations.

- **Adoption and sunsetting:** During this phase, developers start exploring and using APIs to build apps around them. API governance should facilitate the easy but secure signup and onboarding of developers and their apps. The governance process should monitor how the APIs are performing and being used by developers. Some important metrics to look for must answer the following:

- What are the top 10 APIs being used?
- Who are the top 10 users of an API?

■ **Note** This step should cover mechanisms that address any issues reported by developers on API usage. When new versions of an API are introduced, the API governance process must address how to sunset and retire older versions with minimal to no impact on the apps still using them.

The Aim of API Governance

API governance must address the following.

- Governance at the time of the API proposal (new/updates) must ensure that the identified APIs align to the business strategy and meet the business requirements. Funding for API development must be approved by the business and other stakeholders.
- API design and development time governance must ensure that the API software quality is maintained. It must address API versioning strategy and focus on development standards and best practices to be followed. Appropriate reviews and checkpoints must be enforced to ensure quality of he API.
- API governance must help define the right API testing strategy to ensure that APIs are delivering the necessary level of security, reliability, and governance.
- API runtime governance must look at aspects such as API monitoring, deployment, and dynamic provisioning to guarantee API runtime quality.
- API governance must ensure that the service-level agreement is followed by the API provider and the consumer.

API Governance Model

The high-level API governance activities, checkpoints, preconditions, and the roles required for each of the phases of the API governance model are described in Tables 12-1 to 12-5.

Table 12-1. *API Proposal*

Title	Description
Input	• New API requests made by business analysts and solution architects. An outline solution document for the new API to be submitted for review. • API change requests can be made by solution architects and an outline solution document for the API is submitted. • Design lead, tech arch and solution architect review the submitted API proposal. Proposal should be approved by architecture review board. The approval process can be similar to an existing SDLC. A lighter version of the existing approval process can be followed for API proposals.
Process	For new APIs • New API request/business agreement to be submitted for review. • API details to be completed in a template that captures the requirements of the API. It should highlight the use cases related to the API. This evaluates the alignment of the API to business needs. The template must also highlight the information model and the related entities that used by the API. This explains the business assets that are API-enabled. The service interface definition of the API should also be documented at this stage. • Governance review by architecture review board (ARB) of the new API proposal decides if the API should be built or not. For API change requests • Business analyst/solution architects submit a change request to an existing API. • API details to be completed in the same template as that of a new API. • ARB governance review of the API change request decides if the API should be built or not.
Output	• *API profile template* document with API specification for new APIs. • Updated API profile template for API change requests. • A new project (for a CR there is a new version of the project) for the creation of API.
Checkpoint	• Fortnightly or monthly governance reviews organized to review new API requests or change requests. The frequency may change depending on the business needs for the APIs.

(*continued*)

Table 12-1. (*continued*)

Title	Description
Exit Pre-Conditions	• Resourcing availability for API development (an API team to be formed). • An API spec should be reviewed and approved by the ARB or API governance body. • Funding for API development should be approved by business and other stakeholders.
Actors and Roles	• *API business owner*: Responsible for establishing and validating the business needs of the API and the requirements for approval of funding. • *API product owner*: Responsible for interfacing with various API delivery teams to ensure the quality and delivery of the APIs. • *API spec lead:* Responsible for the creation of API specification. • *API architect*: Responsible for the technical architecture of the API solution. • *API leadership team*: Responsible for validating the business requirements and providing funding to build the API.

Table 12-2. *Technical Requirements Gathering*

Title	Description
Input	• API profile template
Process	• Create API specifications document from the business requirements. • Define data mappings between API interfaces and back-end services. • Requirements should be stored and/or updated in a central requirements management tool.
Output	• API specification and data mappings document
Checkpoint	• Review with the business analyst and the API architect, and sign off the API specification.
Exit Pre Conditions	• Governance guidelines and rules followed • API profile requirements are updated in JIRA
Cross-functional Implication	• API specification review for any impact on existing functionality

(*continued*)

Table 12-2. (*continued*)

Title	Description
Actors and Roles	• *API business analyst*: Gathers the business requirements for API enablement and identifying the services to be exposed as APIs. • *API solution architect*: Works with the business analyst to define the API specification document and data mapping to back-end services. • *API spec lead*: Defines the API specifications and working with the business analyst and solution architect. • *API project team*: Informed about the new API requirements at this stage. Reviews the API specifications. • *API governance committee*: Ensures that the process is followed, criteria are met, and quality is maintained. • *Scrum master*: Conducts a spec jam.

Table 12-3. *Build and Validate*

Title	Description
Input	• Approved API specification and data mapping documents • Business requirements document
Process	In the API build and validate phase, the API team consists of scrum master, API architect, API designer, API developers, API testers, and DevOps team, who work together to build the API per specifications and business requirements. API development is done in short sprints of three to four weeks using the agile development methodology. The high-level activities are as follows: • The *scrum master* grooms the requirements and fills in action log in a requirements management tool like JIRA. • The *API development lead* reviews the API specifications, captures comments, and updates action logs.

(*continued*)

Table 12-3. (*continued*)

Title	Description

	• The *API teams* are responsible for the following:
	  • Reviewing the specification, update business agreement (after revival) and update action logs
	  • API implementation
	  • Committing to SCM
	  • Creating test kits
	  • Code review
	  • Demoing to the client and validation by the client
	  • Publishing deployable artifacts to repository
	  • Updating developer portal links
	  • Publishing information on the developer portal for API subscribers

Output	• Completed action list captured during previous discussions • Follow-up action plan created in action log • Reference implementation running in development • Artifacts uploaded to a repository, like GitHub • Developer portal updated
Checkpoint	• Implementation should be compliant to API specification document (mappings are validated)
Exit Pre-Conditions	• Final review (config review, demo to the client) • Governance guidelines and rules followed • API conformant to design guidelines • API versioning policies followed • Any deviations are documented
Cross-functional Implication	• Review for any impact on existing functionality

(*continued*)

Table 12-3. (*continued*)

Title	Description
Actors and Roles	• *API program manager*: Responsible for the overall program delivery of the APIs. • *API architect*: Architects the API solution and defines the API REST interface. • *API designer*: Designs the APIs for proxy configurations. • *API developers*: Configures API proxies in the API gateway • *API testers*: Creates automated test cases and testing API interfaces. • *API DevOps*: Builds a DevOps framework to support CI and CD for API enablement.

Table 12-4. *General Availability*

Title	Description
Input	• API interface definition in repository • API test console availability • Developer portal updated • Pre-prod environment running reference implementation • API config uploaded to SCM or repository
Process	During the general availability phase the following activities are performed to publish the APIs: • API deployment from the repository to production and sandbox • API documentation published on the developer portal for API subscribers • Developers access APIs and create apps • API health monitoring is set up
Output	• APIs deployed to production and sandbox environments • API documentation updated in the developer portal • Apps built against APIs
Checkpoint	• Check API's running status • Check API documentation and test console in the developer portal • Check API analytics for API traffic and performance • API health monitors are configured
Actors and Roles	• *Project team*: Responsible for overall API delivery. • *API support team*: Supports reported issues. • *Operations/run team*: Deploys APIs and monitors their health.

Table 12-5. *Adoption*

Title	Description
Input	• The number of developers signed up • API traffic reports • The number of hits on developer portal • The number of mentions in social networks • The number of blogs and forum posts

Process	• During the adoption phase of an API, it is important to have a plan that facilitates easy onboarding of developers and apps, and tracks the usage of the API. For this, the following activities need to be performed:
	  • Develop an adoption plan and identify targets.
	  • Target and inform specific development communities about the availability of the new API.
	  • Target/organize hackathons to support adoption.
	  • Track/follow up with members who are in the member adoption forum.
	  • Update the adoption list, developer portal, and API website.
	  • Ensure the publicity of API through various developer channels
	  • Identify and inform other ecosystems of API availability.
	  • Conduct webinars driven by members to share experience in adoption.

Checkpoint	• The number of developers and apps onboarded • The number of active developers and apps • API traffic patterns • The number of API issues reported from different channels
Actors and Roles	• *API operations team*: Facilitates the developer onboarding and monitors API traffic. • *API support team*: Resolves issues reported about the APIs.

Index

A

Abao, 74
Accept-Charset header, 47
Accept header, 47
Access token, 19
AccuWeather APIs, 9
Activity logging, 24
Amazon APIs, 1, 10
Amazon S3, 4
Amazon Web Services, 3
Analytics services, 17
API adoption patterns
  business partner integration, 103
  external digital consumers, 103–104
  internal application integration, 103
  IoT, 104
  mobile, 104
API analytics, 95
  activity logging, 24
  advanced analytics, 25
  business value reports, 24
  importance, 165–166
  metrics, 168–169
  reports, 169–170
  service-level monitoring, 25
  stakeholders, 166–168
  user auditing, 24
API Blueprint, 75
  document structure, 76–77
  vs. Swagger and RAML, 77–79
  tools, 77
API catalog, 25
API contract, 1, 5
  definition, 2
API deployment patterns
  cloud deployment
    advantages, 100–101
    disadvantages, 101
  on-premise deployment model, 102
API Designer, 73
API developer portal
  vs. API gateway, 177
  in API Lifecycle, 171
  API product owner, 175
  API team, 174–175
  app developers, 174
  features, 175–177
  importance, 172
  publishing and sharing, 171–172
  support, 172–173
    federated developer community, 174
    invitations, 173
    social forums, 173
API documentation, 7, 26, 156, 176
API facade pattern
  callback, 90–91
  composition, 87–88
  HATEOS principles, 88
  two-phase transaction, 89–90
API fuzzing, 156–157
API gateway, 16, 18, 88
  vs. API developer portal, 177
  caching, 22
  interface translation
    format translation, 21
    protocol translation, 22
    service and data mapping, 22
  security
    authentication, 19
    authorization, 19

© Brajesh De 2017
B. De, *API Management*, DOI 10.1007/978-1-4842-1305-6

API gateway (*cont.*)
data privacy, 20
DoS protection, 20
identity mediation, 19
key and certificate management, 20
threat detection, 20
service orchestration, 23
service routing
connection pooling, 23
load balancing, 23
service dispatching, 23
URL mapping, 22
testing, 157–158
traffic management
consumption quota, 21
spike arrest, 21
traffic prioritization, 21
usage throttling, 21
API governance
adoption phase, 181–182, 188
aim, 182
API proposal, 180, 183–184
build and validate phase, 180–181, 185–187
general availability phase, 181, 187
policy-driven approach, 179
technical requirements gathering, 180, 184–185
API interface, 155
API key, 19, 26
API lifecycle management
change notification, 28
creation, 27
issue management, 28
publication, 27
version management, 27
API management patterns
API composition pattern, 87–88
API facade pattern, 86–87
caching, 93–94
logging and monitoring, 94
routing, 91–92
session management, 88–89
synchronous to asynchronous mediation, 90–91
throttling, 92–93
two-phase conversion pattern, 90
two-phase transaction management, 89–90
API management platform
API gateway,18 (*see also* (API gateway))
businesses values, 23–25
capabilities, 16
developer portals (*see* (Developer portals))
lifecycle management (*see* (API lifecycle management))
API message logging pattern, 94
API monetization
API package, 150
API product, 149–150
billing documents, 151
digital assets, 143
fee-based model, 146–148
free model, 146
to increase revenue
customer channels, 143–144
customer retention, 144
distribution channels, 145
upsell premium and value-added services, 144–145
indirect model, 146
rate plan, 150
reports, 151–152
revenue-sharing model, 146, 148–149
API Notebook, 70, 74
API patterns
adoption patterns, 102–104 (*see also* (API adoption patterns))
deployment patterns, 100–102 (*see also* (API deployment patterns))
management patterns (*see* (API management patterns))
pragmatic RESTful API interface, 81–86
security (*see* (API security))
API performance testing, 158–162. *See also* Load testing
baseline testing, 159
metrics, 161–162
soak testing, 160
stress testing, 159
API product owner, 175
API Provider, 5, 6, 13
API registry, 25
API security, 156
authentication, 97
authentication and authorization

API keys, 113–114
mutual authentication, 115
OAuth, 115–118 (*see also* (OAuth))username and password, 114
X.509 certificate, 115
authorization, 97
considerations, 140
cross-site scripting (XSS), 96
cyber threats
bot attacks, 139–140
cross-site resource forgery, 138–139
cross-site scripting (XSS), 137–138
injection threats, 134–136
insecure direct object reference, 136
sensitive data exposure, 136–137
DDoS attacks, 112
demands, 111–112
denial-of-service (DOS) attacks, 95
eavesdropping, 96
logging and auditing, 99
man-in-the-middle attacks, 112
monitoring APIs, 98
OpenID Connect (*see* (OpenID Connect))
PCI compliance requirements, 99–100
Quota policy, 98
recommendations, 141–142
schema validation policies, 97
scripting attacks, 95
SDLC process, 99
session attack, 96
Spike Arrest policy, 97–98
SSL/TLS encryption, 98
testing
API fuzzing, 156–157
authentication and authorization, 156
malformed payload injection, 157
malicious content injection, 157
threat model, 140
API team, 174–175
API testing
API documentation, 156
API gateway, 157–158
API interface specifications, 155
API security,156 (*see also* (API security, testing))
challenges, 153–154
importance of, 153
performance testing, 158–162 (*see also* (Load testing))
tools, 164
must-have features, 162–163
nice-to-have features, 163–164
API value chain, 13–14
API Workbench, 72–73
app developers, 7, 13, 174
app ID, 19
app key, 19
Application programming interface (API). *See also* Web APIs
business models, 14
hotel room booking, 1
AT&T APIs, 10
Audiences, API documentation, 60
Auditing, 99
Authentication, 19, 85, 97
Authorization, 19, 97
Authorization header, 48

B

Baseline testing, 159
B2B partner integration, 103
Billing, 151
BlazeMeter, 161
Blogs and forums, 26, 176
Bot attacks, 139–140

C

Cache-Control general header, 49
Caching, 22, 30–31, 85, 93–94
client ID, 19
Client-server constraint, 30
Cloud computing, 4
Cloud deployment
advantages
capital and operational expenditure reduction, 100
management over heads, 101
regulatory compliance, 101
reliability and availability, 100
scalability and agility, 101
time to market, 100
disadvantages
control over data, 101
network latency, 101

Code-on-demand constraint, 31
Communication, 17
Content-based routing, 91–92
Content-Type header, 49
Cross-Site Resource Forgery (CSRF or XSRF), 138–139
Cross-site scripting (XSS), 96, 136, 137–138
Custom Search APIs, 9
Cyber threats
  bot attacks, 139–140
  cross-site resource forgery, 138–139
  injection threats
    script injection attacks, 135–136
    XML and JSON bombs, 134–135
  insecure direct object reference, 136
  sensitive data exposure, 136–137
  XSS, 137–138

■ D

Data privacy, 20
DDoS attacks, 112
DELETE verb, 40–41
Denial-of-service (DoS) attacks, 20, 95
Developer portals, 17, 25
  access credentials, 26
  API catalog and documentation, 25–26
  API documentation, 26
  community management, 26
  monetization, 25
Documentation, 59
  API Blueprint, 75–77
  app developers or API consumers, 60
  audiences, 60
  bottom-up approach, 66
  endpoint, 62
  error codes, 64
  frameworks, 80
  header parameters, 63
  HTTP response codes, 64
  importance, 59–60
  message payload, 62
  method, 62
  RAML, 69 (*see also* (RESTful API Markup Language (RAML)))
  sample HTTP calls, 65
  SLAs, 66
  Swagger, 61 (*see also* (Swagger))
  title, 61
  top-down approach, 66
  tutorials and walk-throughs, 65
  URL parameters, 62

■ E

Eavesdropping, 96
eBay API, 3
Elastic Compute Cloud platform, 4
ETag (entity tag) response header, 49

■ F

Facebook APIs, 1, 3, 8
Federated developer community, 174
Filtering, 51–52
Filtering criteria, 83
Flickr APIs, 3, 9
Foursquare APIs, 4
Freemium model, 147
Free model, 146–147

■ G

GET verb, 38
Google APIs, 9
Google Maps APIs, 1, 3

■ H

Handle requests, 109
HEAD method, 42
Host request header, 48
HTTP error response codes, 85–86
HTTP headers, 84, 108
  Accept-Charset header, 47
  Accept header, 47
  Authorization header, 48
  Cache-Control general header, 49
  Content-Type header, 49
  ETag (entity tag) response header, 49
  Host request header, 48
  Location response header, 48
  naming conventions, 49–50
  types, 46
HTTP status code, 84
HTTP verbs, 62, 81
  RESTful web services
    DELETE verb, 40–41
    GET verb, 38
    HEAD method, 42

idempotent and safe methods, 42
OPTIONS verb, 41
PATCH method, 41
POST verb, 39
PUT method, 39–40
PUT *vs.* POST, 40
Richardson Maturity Model, 55–56
Hypermedia as the Engine of Application State (HATEOAS), 33–34

I

Idempotent HTTP method, 42
Injection threats
script injection attacks
script injections, 136
SQL statement injection, 135
XML and JSON bombs, 134–135
Insecure direct object reference, 136
Instagram APIs, 4, 9
Internal APIs, 6
Internal application integration, 103
Internet of Things (IoT), 104
Invitations, developer portal, 173

J, K

JMeter, 160
JSON format representation, 46

L

Layered system, 31
Load balancing, 23
Loader.io, 161
Load testing, 159
preparation, 158–160
tools, 160–161
LoadUI, 160
Location response header, 48
Logging, 99

M

Malformed/unexpected message injection attacks, 157
Man-in-the-middle attacks, 112
Message payload, 62
Mobile apps, 104
Monetization, 24, 25
Monitoring APIs
analytics, 25
management patterns, 94
security, 98

N

Naming conventions, 49–50

O

OAuth, 97
API gateway, 117–118
authorization server, 117
client, 116
grant types
authorization code, 119–120
client credentials, 120–121
implicit grant type, 122–123
resource owner password credentials, 121–122
protocol, 116
resource owner, 116
resource server, 117
scope names, 118
tokens, 116, 118
On-premise deployment model, 102
OpenAPI specification, 66–67
OpenID Connect
authentication flows
authorization code flow, 126–130 (*see also* (OpenID connect authorization code flow))hybrid flow, 131–133
implicit flow, 130–131
end user, 123
identity provider integration, 133
ID tokens, 124–126
interaction between parties, 123–124
relying party (RP), 123, 126
OpenID connect authorization code flow
authorization endpoint, 127–128
token endpoint, 128–130
userinfo endpoint, 130
OPTIONS verb, 41

P

Pagination, 83–84
Partner APIs, 6
PATCH method, 41
PCI compliance specifications, 99–100

POST verb, 39
Private APIs, 6, 8
  security and access control, 8
Public APIs, 6
  app developers, 7
  security risks, 7
  success, 7
PUT method, 39–40
  vs. POST, 40

Q

Query parameters, 108–109
Quota policy, 98

R

RAML API specification
  data type, 70
  methods, 71
  resources and subresources, 71
  resource types and traits, 72
  response, 71
  security scheme information, 70
  security schemes, 72
Rate plan, 150
Refresh token, 19
Regulatory compliance requirement, 99
Relying party (RP), 123, 126
Representational State Transfer (REST), 9, 17
  caching, 30–31
  client-server constraint, 30
  code-on-demand constraint, 31
  HTTP headers
    Accept-Charset header, 47
    Accept header, 47
    Authorization header, 48
    Cache-Control general header, 49
    Content-Type header, 49
    ETag (entity tag) response header, 49
    Host request header, 48
    Location response header, 48
    naming conventions, 49–50
    types, 46
  HTTP status code
    categories, 43
    error codes, 44–45
    success codes, 43
  HTTP verbs
    DELETE verb, 40–41
    GET verb, 38
    HEAD method, 42
    idempotent and safe methods, 42
    OPTIONS verb, 41
    PATCH method, 41
    POST verb, 39
    PUT method, 39–40
    PUT *vs.* POST, 40
  layered system principle, 31
  query-string parameters
    filtering, 51–52
    offset and limit, 51
    pagination, 51
  resource identifier design, URIs
    best practices, 35
    modelling resources and subresources, 34
    naming conventions, 37
    resource naming conventions, 34
    URI design, 35–36
    URI format, 36–37
  resource representation design, 45–46
  Richardson Maturity Model
    HTTP verbs, 55–56
    hypermedia controls, 56–57
    resources, 54–55
    Swamp of POX, 53–54
  statelessness, 30
  uniform interface, 30
    HATEOAS, 33–34
    resource identification, 31–32
    resource manipulation, 33
    self-descriptive messages, 33
  versioning (*see* (Versioning))
Resource type, 72
RESTful API Markup Language (RAML)
  Abao, 74
  API Designer, 73
  API Notebook, 70, 74
  API Workbench, 72–73
  code generation tools, 69–70
  JAX-RS, 74
  for .NET, 74
  RAML 0.8 and RAML 1.0, 75
  Restlet Studio, 73
  specification, 70
  structure, 70–72
  structure (*see* (RAML API specification))

vs. Swagger and API Blueprint, 77–79
tools, 74
RESTful web services,11. *See also* Representational State Transfer (REST)
Restlet Studio, 73
Richardson Maturity Model
HTTP verbs, 55–56
hypermedia controls, 56–57
resources, 54–55
Swamp of POX, 53–54
Roy Thomas Fielding's dissertation, 3, 34

S

Safe HTTP method, 42
SalesForce, 3
Scripting attacks, 95
Script injection attacks
script injections, 136
SQL statement injection, 135
Search APIs, 9
Sensitive data exposure, 136–137
Service-level agreement (SLA), 66
Service orchestration, 23
Service-oriented architecture (SOA), 11, 103
Session attack, 96
SMAC (social, mobile, analytics, and cloud) technologies, 4
Soak testing, 160
SOAP (Simple Object Access Protocol) messages, 10
Social forums, 173
Spike Arrest policy, 97–98
SSL/TLS encryption, 82, 98
Streaming APIs, 9
Stress testing, 159
Swagger, 66
bottom-up approach, 66, 68
file structure, 68
frameworks, 66
goals, 66
vs. RAML and API Blueprint, 77–79
tools, 69
top-down approach, 66, 68
Swagger Codegen, 67
Swagger Editor, 67
Swagger-UI, 67

T

Traits, 72
Twitter APIs, 3, 9

U

Uniform Resource Identifier (URI), 32
components, 36
naming conventions, 37
Uniform Resource Locators (URLs), 36, 81
versioning, 107–108
Uniform Resource Name (URN), 36
URL mapping, 22
URL parameters, 62
User auditing, 24

V

Vegeta, 161
Versioning, 50, 82
demands, 106
handle requests, 109
host name, 109
HTTP header, 108
lifecycle management, 109–110
principles, 106–107
query parameters, 108–109
vs. software versioning, 105
URLs, 107–108

W, X

Web APIs
definition, 5
evolution, 3–4
vs. SOA, 11–12
vs. web services, 10–11
vs. web sites, 5
Web sites, 5
Wrk, 161

Y, Z

Yelp APIs, 9
YouTube API, 10