

Practical API Architecture and Development with Azure and AWS

Design and Implementation of
APIs for the Cloud

Thurupathan Vijayakumar

Apress®

Practical API Architecture and Development with Azure and AWS

**Design and Implementation of
APIs for the Cloud**

Thurupathan Vijayakumar

Apress®

Practical API Architecture and Development with Azure and AWS

Thurupathan Vijayakumar
Colombo, Sri Lanka

ISBN-13 (pbk): 978-1-4842-3554-6

ISBN-13 (electronic): 978-1-4842-3555-3

<https://doi.org/10.1007/978-1-4842-3555-3>

Library of Congress Control Number: 2018946567

Copyright © 2018 by Thurupathan Vijayakumar

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, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Nikhil Karkal
Development Editor: James Markham
Coordinating Editor: Divya Modi

Cover designed by eStudioCalamar

Cover image designed by Freepik (www.freepik.com)

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/978-1-4842-3554-6. For more detailed information, please visit <http://www.apress.com/source-code>.

Printed on acid-free paper

This book is dedicated to my loving parents and friends.

Table of Contents

- About the Authorix**
- About the Technical Reviewerxi**
- Acknowledgmentsxiii**
- Introduction xv**

- Chapter 1: Practical Introduction to APIs 1**
 - APIs: A Practical Introduction 2
 - Programmable Language Constructs 2
 - Systems of Data & Operations Flow 5
 - API Economy 6
 - APIs in the Public Sector 9
 - G2C: Government to Citizens 9
 - G2B: Government to Business 10
 - G2G: Government to Government 11
 - Summary..... 11

- Chapter 2: API Strategy and Architecture 13**
 - API Strategy 14
 - API Strategy Use Case 17
 - API Value Chain 18
 - API Architecture 19
 - API Management..... 22
 - Summary..... 24

TABLE OF CONTENTS

- Chapter 3: API Development25**
 - API Development Considerations 25
 - Explicit Parameters 26
 - Avoid Consumer-Commanded Endpoints 26
 - Documentation 27
 - Security 27
 - Versioning..... 27
 - API Development Standards..... 28
 - HTTP Verbs 28
 - HTTP Status Codes 29
 - Error Handling 31
 - URI Syntax 33
 - Versioning..... 34
 - Kick-Start API Development 34
 - Implementation: ASP.NET Core 35
 - Setting Up Swagger..... 45
 - Run the API and Swagger 48
 - Team Orientation in API Development..... 49
 - Summary..... 50
- Chapter 4: API Gateways51**
 - API Gateways in a Public Cloud..... 51
 - Endpoint Mappings..... 53
 - Azure API Management..... 56
 - Creating an Azure API Management Service 57
 - Connecting to the Backend Service..... 61
 - Configuring API Endpoints 64
 - Configuration Policies..... 68
 - Products in Azure API Management 72

Azure API Management Developer Experience.....	76
Structure of the Azure API Management Components	81
AWS API Gateway	82
Creating an AWS API Gateway Service	82
Configure Methods	84
Deploy AWS API Gateway.....	89
Creating API Usage Plans	92
Structure of AWS API Gateway Components	94
Summary.....	95
Chapter 5: API Security	97
Request-Based Security	97
Azure API Management	98
AWS API Gateway	102
Authentication & Authorization	104
API Security Design	105
Authorizers in AWS API Gateway	127
Summary.....	132
Chapter 6: Serverless APIs	133
Serverless Computing.....	133
Serverless APIs in Azure	135
Azure Functions.....	135
Azure Function Proxies.....	144
Azure Logic Apps	149
Serverless APIs in AWS	150
AWS Lambda	150
Creating an AWS Lambda Function	151
Setting Up AWS Lambda with AWS API Gateway.....	156
Summary.....	158

TABLE OF CONTENTS

Chapter 7: Practical Design and Development159

- Contract-First Design 159
 - Preparation 161
 - Key Challenges 162
 - When Not to Try It 163
- APIs in Microservices 163
 - Client-Coordinated Design 164
 - API Gateway Pattern 165
- APIs for Enterprise Integration 166
- Summary 167

Index 169

About the Author



Thurupathan Vijayakumar (Thuru) is a technology architect at Tiqri Corporation. He has experience in a range of fields in the software industry such as software architecture, development, cloud computing, data security, business intelligence and etc. In his role as a technology architect, he works closely with business stakeholders and developers and help them mapping business requirements to the latest technologies and driving innovation.

Thuru is a hard-core developer and experienced in many different languages, including GW Basic, FoxPro, Pascal, C, C++, Visual Basic, Action Script, Java, C#, JavaScript Go, Rust and the list grows.

He authored the book *Practical Azure Application Development* in 2017, which is a handy guide for getting started in Azure development.

He's a Microsoft MVP for Microsoft Azure and a speaker in international conferences and user group meetings. Check his blog for his most recent work.

Blog: www.thuru.net

Twitter: @thurutweets

About the Technical Reviewer



Dheeraj Swami is a Technical consultant at Microsoft India. He has more than 10 years of experience in the IT industry. He has worked on Microsoft full stack, Hybrid Mobile Development (Ionic), and Salesforce. Dheeraj is a Microsoft certified Architect in Azure technology stack and a certified Salesforce developer, advanced developer, and administrator.

Acknowledgments

First, I declare my acknowledgement to Apress for another successful engagement with this book.

The content of this book is mostly influenced by the knowledge and the experience gathered during my tenure at Tiqri Corporation, and I'm thankful for the organization and my great colleagues, especially Ms. Thushara Wijewardena, who has been relentlessly pushing me to write this book, and Mr. Buddhika Jaywardhena, who tirelessly argued with me about minimum API standards. The content in chapter 3 is greatly influenced by many discussions we had before I had even thought of writing a book about API development.

I declare heartfelt gratitude to my parents and friends, who've been always there for me and supported me in completing this book.

Finally, I extend my acknowledgement to the Apress team, especially to Rita Fernando, Nikhil Karkal, and Divya Modi, who helped me in many ways in reaching the completion of this book.

Introduction

This book, *Practical API Architecture and Development with Azure & AWS*, is meant to fill the gap between the demand for cloud-based API architecture and design and the skills of developers. In API implementation, developers often miss the real depth of API design, including some critical technical aspects. Mostly this occurs because APIs are thought of as JSON emitting endpoints to a client application. The growing field of automated frameworks and tools fuels this misunderstanding and hides the business complexity and technical criticality of API implementation and standards.

This book is structured with a specific focus on addressing these two issues. The first section (the first three chapters) explains the business aspect of APIs to developers and provides a quick-start guide to API standards implementation, with code samples. The second section (chapters 4 through 6) provides a more technical, practical approach to API gateways, API security, and serverless APIs using tools from Azure and AWS. The last chapter offers a holistic approach to common design and development use cases of APIs, including an exploration of contract-first design.

The technical implementations are illustrated with figures and screenshots, which allow the reader to easily follow the processes described in the book. Additional code samples can be found at this URL: <https://github.com/thuru/paadaa>

The book is targeted to developers who work in .NET stack and use Azure or AWS.

CHAPTER 1

Practical Introduction to APIs

Data is God: the ever-growing demand for data is one of the fundamental factors influencing the disruption of many communication and integration technologies. The preference for data over operations is a challenge for the modern software development, orchestrating and managing the data integrations and flow have become the key success factor in the modern software solutions.

The demand for the data-centric software applications and the evolving development landscape which favors adopting existing tools and services to achieve speed and flexibility over developing everything from scratch, make traditional integration technologies obsolete. APIs are the current state of evolution in integration technologies and we can see the evolution is moving towards integration as a language space. In addition to that, modern software architectures like microservices and serverless favor APIs over traditional integration technologies and APIs help those architectures to foster; the benefit is mutual and complimentary in both ways.

In theory, any programmable interface can be referred as an API. Though the term Application Programming Interface (API) is not new and it takes many flavors of implementations, the current usage of the term commonly refers to a HTTP-based RESTful service.

Businesses expose data and operations via APIs for various reasons, such as achieving business agility, monetizing data and business operations, integration, enabling innovation, enabling business ecosystems, adhering to regulatory requirements and etc.

This chapter provides an introduction to APIs with a practical explanation, and builds the discussion toward generic topics like API economy and how APIs are used in the public sector.

APIs: A Practical Introduction

The software industry is already overwhelmed with buzzwords, and most of them are created with the intention of satisfying both technical and business stakeholders. Because of this dual audience obligation, most of these terms do not address either business or technical stakeholders properly.

The practical approach of this book explores the meaning and builds a common understanding of APIs for the readers' context. This is not a quest for a new definition, but an attempt to help readers understand the context.

Note APIs have dual personalities: one is based on language constructs or in the form of libraries/frameworks, and the second is, as systems exposing data and operations.

Programmable Language Constructs

First, let us look at the first “personality.” As developers, we write code using language constructs and expose behaviors. In a typical Object Oriented Programming (OOP) language, this is achieved by classes implementing interfaces.

Generally, these interfaces are implemented, so that parameters are included in method signatures only when the caller is required to supply them. If the required parameters can be acquired from the execution context then we try to have those cross cutting-data available to different layers by shared classes, rather than explicitly declaring them in the method signature. The below example shows it.

The exposed interface:

```
internal interface IRegistrationService
{
    Task<int> RegisterBookAsync(Book book);
}

// One of the implementations

public class RegistrationService : IRegistrationService
{
    public async Task<int> RegisterBookAsync(Book book)
    {
        book.UserId = SessionProvider.GetUserId();
        // rest of the logic
    }
}
```

Here, the `UserId` is not parameterized in the current execution context, but the implementation works because it knows from where to get the `UserId`. This is fine, but this interface implementation is limited because the service implementation is tightly coupled with the current execution context. And others who implement this interface expected to acquire the `UserId` internally. If, down the line, different clients or assemblies want to use `RegistrationService` and wants to provide `UserIds` from different sources, they will fail or need to handle it in the interface implementation, resulting different implementation for each `UserId` source or any other cumbersome logic.

For an interface to be consumed by external callers, parameterization is important in the method signature itself, as shown here. This makes the implementation and the

```
public interface IRegistrationService
{
    Task<int> RegisterBookAsync(Book book, string UserId);
}
```

Creating interfaces with the consideration of external callers outside the execution context, helps achieving loose coupling in different layers of the code. This change is the first level realization of an exposed interface in the development context. An interface becomes published and exposed to external callers during this transition, so being consumable by an external party is a fundamental characteristic of a consumable interface.

These interfaces and implementations can be published as packages. Packages are referenced in other execution contexts, ensuring usability for external parties, and bundled with other development tools, components, base classes, documentation, and sample implementations. These bundles are commonly known as Software Development Kits (SDKs). SDKs are APIs with additional features and tools that target more experienced developers.

packages and SDKs has helped with code reusability and code sharing. These changes influence the way systems are programmed; developers are not required to write all the code for a system they develop. Frameworks and libraries came into the picture, and now we are all familiar with package managers like NuGet, NPM, and many more. Now, we cannot imagine a development scenario without packages.

Systems of Data & Operations Flow

The second type of API enables data and operations flow between different systems. The need for distributed, service-based architectures is trivial due to disparate systems and growing demand for integrations.

In this model, service contracts (data agreements) are important for communication between different services. Initially, language-specific interface implementations took the stage. Remote Procedure Call (RPC) is the first level of implementation to transform the first API type to the second.

RPC implementations are highly dependent on language and runtime. Due to this limitation, systems developed using different languages never had significant success in service-based architecture using RPC. The second API type needed a different approach.

The need for language- and implementation-agnostic message exchange between different services was acknowledged. This paved the way for the highly used—and highly confused—industry term Service-Oriented Architecture (SOA).

Most, SOA implementations have some form of a queue or service bus technology and service contracts are implemented in XML/JSON. In the SOA world, XML was leading the pack with varying types of implementations, like SOAP.

Later with the time, the advent of web and HTTP paved the way for web services, web services came into the picture due to the growing need for data in the Internet world. Compared to traditional SOA implementations, web services are Internet friendly, but due to the overwhelming standards and implementation practices borrowed from the SOA they are not nimble enough to cater to a growing Internet.

Eventually, more Internet-friendly—or in other words, more Internet-native—data exchange technology was proposed. RESTful services are based on HTTP and HTTP verbs, while JSON emerged as a lighter data exchange format and gained popularity. The second type of API has evolved as HTTP-based RESTful services.

So now, when we use the term API, it most often refers to a HTTP-based RESTful service with JSON-based service contracts.

But as we have discussed, APIs come in two different types: in the form of programmable language constructs and programmable RESTful services.

The commonality between these two is the ability to be consumed by external parties. In the context of this book, the term API refers to the second type, which is the Internet-friendly, HTTP-based programmable interface.

Now we understand what an API is and the two different forms it can take in the engineering world, as well as the meaning of API in the current industry context. In the next section of this chapter, we will focus on API economy and how APIs are used in the public sector.

API Economy

Rapid technology changes and growing trends, like the Internet of Things (IoT), the cloud, service collaboration, AR, VR, MR, and many other buzzwords, highly influence almost every business to seek new models that cater to industry pressure and ensure survival. This trend helps the proliferation of APIs and API economy.

Any use of APIs for economic benefit can be generalized as “API economy.” Businesses have their own unique data capabilities and operations, and APIs expose data and operations to create new opportunities.

Businesses with valuable data and operations that are expensive to created sell them for consumption through APIs. This is direct selling. A good example is the cognitive services that sell AI as a service via APIs. Microsoft & Google have a wide array of such services.

Businesses have increased their focus on creating new customer experiences and finding new opportunities to serve customers better and more efficiently. APIs help to do this, not only by exposing data and operations, but also by enabling innovation and design thinking from outside the organization. For example: a fashion retailer exposes his product catalog and search operations via an API, triggering an external developer to create an app with capabilities such as location-based or picture-based searching. Though the fashion retailer provides the API free of charge to the developer, it gains innovation and design thinking, which may not be fully tapped by the internal organization.

This external innovation and design thinking is a common pattern we can observe in many mashup applications. Because the external developer consumes the fashion retailer's API and mashup with other location- and picture-based searches available from other providers to create a new customer experience, it is a win-win situation for all parties involved.

Another way to create rich customer experiences is by focusing on frictionless experiences by enabling ecosystems consisting of different systems. For example, most wearable gadget manufacturers have integrations with healthcare services. The user's step count, distance, heartbeat, and other details are commonly integrated with providers through APIs, enabling a seamless customer experience ecosystem.

Consider the below example, where a smart refrigerator automatically orders more milk when it runs out. Figure 1-1 shows the basic services, the different stakeholders, and the high-level integrations among them. The refrigerator manufacturer has exposed the API (IoT gateway), which receives sensor data from the refrigerators. Assume some magic milk sensor is able to gauge the amount of milk in the refrigerator, and the sensor data is collected and processed in the cloud.

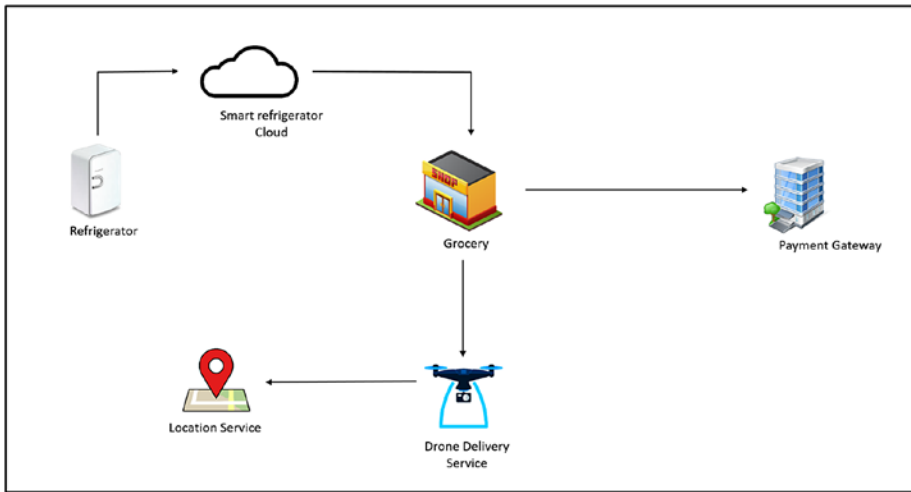


Figure 1-1. *Smart Refrigerator Scenario*

If the milk level is below a configured threshold, the appliance connects to a grocery store API. This API takes the order and confirms the payment by talking to the payment gateway, which is another API. Then it sends a message to a drone-based delivery service to pick up the milk from the grocery store and deliver it to the customer.

The drone delivery service uses location and weather APIs to complete the delivery. In a real-life scenario, there would be more APIs involved from various parties, but this is a decent example for understanding business models that use APIs from different parties.

For the refrigerator manufacturer, having a smart cloud and processing IoT sensor data enables a value-added customer experience. For the grocery store, accepting orders via API is an opening for e-commerce. The refrigerator manufacturer may enable an ecosystem with multiple grocery stores, placing orders that boost business, and may charge the stores for this benefit.

The payment gateway employs a commission-based business model around the API delivery service, exposing its API to manage delivery orders—this is service-based e-commerce. It talks to other utility service

APIs to complete its task, like location and weather APIs, which are either free or paid under commercial use. So the location and weather data providers sell their data directly via APIs.

In the above case, we see different purposes of APIs for different parties under different business models. Some do direct selling, some are able to provide richer customer experience and eco-system enablement, some expose their operations based on commission or e-commerce. All these models are different versions of API economy, and represent different ways that APIs can yield economic benefits for organizations.

APIs in the Public Sector

Enabling and empowering governments through integration and data strategy is a key purpose of APIs in the public sector.

Many governments have been working actively to provide better experiences to citizens, businesses, and other governments via digitization and API strategies.

There are three government service models.

- *G2C*: Government to Citizens
- *G2B*: Government to Business
- *G2G*: Government to Government

G2C: Government to Citizens

This model serves citizens by providing data and operational efficiency of a government engine through APIs. Exposing APIs allows developers and state offices to come up with new integration models and applications, allowing the automation and digitization of many services.

The government of India has introduced a new digital identification number for residents through the Unique Identification Authority of India. The primary goal of this project, known as Aadhaar, is to enable unique digital identification, integrating with myriad other government APIs to provide a seamless citizen experience. One of the mission statements of Aadhaar clearly identifies this goal: “Encourage innovation and provide a platform for public and private agencies to develop Aadhaar linked applications.”

Another good example is Japan’s open government data project. The Information Technology Promotion Agency (IPA), a 100% government-funded entity, executes a set of API and open data programs. This includes APIs that expose data from disaster management sources, especially for earthquakes. This open data strategy allows organizations and individual application developers to come up with useful applications to serve citizens.

G2B: Government to Business

Government API services can also benefit businesses; it is important for a good e-government strategy to support businesses and stimulate the economy. Integrations and business-related government services can be digitized to serve businesses, improve efficiency, and bring new revenue streams to the government.

Singapore government is a leading player in data strategy, its Smart Nation initiative is a masterpiece of data and API. On the Smart Nation site (<https://www.smartnation.sg/resources/open-data>), you will find an array of government APIs and data endpoints that serve citizens and businesses. Notable government-to-business initiatives include datastore APIs for developers, which reveal millions of public sector data points such as LTA Data Mall (a variety of transport-related datasets), and MAS APIs (APIs from the monetary authority of Singapore that help other financial institutions and application providers).

Through these APIs, businesses and startups have access to government resources for establishing a business, taxation, business information, and regulatory audits.

G2G: Government to Government

Government-to-government service interactions occur between public sector departments of the same government or between two different governments.

The United States uses a huge number of APIs to facilitate data transactions between its federal agencies and security departments, mostly information related to national security, narcotics, and public safety. These APIs act as either integration points or data delivery endpoints.

The New Zealand government employs data strategies that integrate with internal government departments in other countries to fight human trafficking. Also, central banks and similar authorities execute many system integrations between nations using APIs.

Summary

APIs are the current state of evolution in integration and data exchange between different systems, but the nature of their implementation and the business agility they provide make them more than just integration mechanisms.

In a business context, APIs are crucial elements of economic value creation. This has made APIs a discussion point not only among developers, but also in boardrooms and governments.

Modern service-based architectures favor APIs and embrace API-based service communication and integrations. The next chapter details API strategy, and how it is developed and executed to support a business's vision. You will also read about how the value delivery of API happens at different levels, and the fundamental pieces of API architecture.

CHAPTER 2

API Strategy and Architecture

The rise of APIs and the realization of the value they can bring to the business have made APIs a relevant topic beyond the world of technology and brought them all the way into the boardrooms. Business and technology decisions fuel API strategy and API architecture, respectively.

In terms of implementation, APIs can either be private or public. Private APIs serve a specific set of stakeholders, and are usually not exposed outside these specific parties. Public APIs are open for consumption by anyone (this does not mean they are free). There are several factors that dictate whether an API is public or private, including security, monetization strategy, data trends, and regulatory standards.

It is important and inevitable for business stakeholders to involve themselves in API creation and strategizing, not only to make informed decisions related to the data and operations APIs expose, but also to set expectations, goals, and constraints for the operational environment; support the business vision through API strategy; and evaluate different business models.

The chosen business strategy should be implemented to ensure the business goals are achieved. Integrating technical aspects into the identified API strategy is a major role of the API architecture. In most cases, the areas of API strategy and API architecture goes together by either complementing or conflicting with each other in some cases.

API Strategy

Any business decision involving the planning, organization, or governance of an API is considered to be API strategy. Business stakeholders propose a vision for the API, and technical stakeholders work on its design and development in order to achieve the set business goals. An API implementation often triggers a number of integrations, while data cleansing flows from legacy systems and exposes contradictory domain specifications between different parties.

For example, a fashion retailer looking to attract innovation and an app ecosystem from external developers would require an API that exposes the product catalog and sales, at bare minimum. Based on existing IT systems and implementation, this would trigger an integration of two or more systems to deliver the functionality of the API.

Initial API implementation of big systems or business operations often triggers or exposes the complexity of the business and lack of common domain understanding within the organization. As a result, API implementations sometimes induce various collateral developments, like data cleansing, modeling a common language, refactoring the code base, and updating tools and frameworks. These developments are not considered to be the components of the API implementation, but are part of the project's mission.

The expected value of APIs is critical to many business decisions related to innovation, business operations, integration, and monetization. These considerations can be categorized under the following strategic aspects of a business:

1. *Business orientation:* APIs are strategized based on the business orientation of the organization. This exposes the purpose of business operations, to be achieved by the API implementation. For example:
 - a. An API developed for the integration of two systems may technically reveal the point of integration, but in business terms this is done due to the consolidation of two business operations which is part of a high level business strategy.
 - b. Some organizations expose APIs in order to create and develop business ecosystems. An airline may offer hotel bookings in addition to their core flight booking operations. This is possible through integrations and business collaboration between the hotels and the airline company.
 - c. A health care service provider might expose certain trends in health data to government bodies. In some countries, this is a regulatory requirement.

2. *Attracting innovation and disruption:* Organizations expose data and operations (organizational IT assets) through APIs to attract innovation and disruption from outside the organization, injecting new thinking and skills into the business. Organizations get the benefit of insights on their business data by exposing them to data scientists, while data scientists often search for large data dumps to do research. Here, the benefit is mutual.
3. *Monetization:* Organizations with valuable data and business operations sell them directly via APIs.
 - a. Organizations with valuable data or operations expose them as APIs, resulting in direct cash flow. Consumers often pay a fee to use these APIs. Examples include maps APIs and cognitive APIs.
 - b. Exposing operations as APIs offers greater flexibility for integration and the opportunity to be part of an ecosystem, thus enabling more business opportunities.

The above aspects of API strategy determine whether an API is private or public, the data and operations to be exposed, the security and authentication, monetization strategies, usage policies, and restrictions.

Exposing organizational assets through a public API has the benefit of developer adoption, innovation and monetization. At the same time, it brings the risk of exposing the organization's business IT assets to a wide range of external audience and increases attack surfaces, it also may have the disadvantage of exposing certain data to the competitors, if not planned well. In order to avoid those negative impacts and leverage key benefits, API strategy and architecture should work in hand in hand to determine, govern, and implement correct measures and correct exposure level of the correct data.

API Strategy Use Case

Imagine an organization with more than 30 years of heritage that provides a web-based tool for real estate valuation. The web application is licensed to qualified valuation professionals as a monthly subscription. These valuation professionals visit a particular location, record their findings in the tool, and generate reports, which are mailed to relevant parties.

In 2015, the organization came to the realization the real estate valuation market was under technology disruption. Emerging technologies challenged standard practices; predictive maintenance by machine learning and IoT-based sensor technologies began to dominate the market, thus creating a challenging business environment for the organization.

The organization's business and technical stakeholders developed an API strategy consisting of two major items:

- Expose data to selected machine-learning institutions to bring innovation to the organization, via a private API exposed to selected partners.
- Expose APIs to integrate with banks and insurance companies, which are the main users of the valuation reports. Previously, the reports were sent manually, and with this new strategy, a private, partner API ecosystem is employed to achieve smoother data flow.

In this case, API strategy is clearly laid out with two key focus areas. One is to bring innovation to the organization and stay relevant in a rapidly changing business environment; the second is to strengthen integrations and enable ecosystems to create more coupled business relationships and improve system experience.

API Value Chain

An API implementation touches different levels and layers of an organization. Modern API implementations often include external stakeholders and other value providers like partners, suppliers, customers, and developers.

APIs integrate and facilitate the digitization of business flows by connecting different stakeholders with organizational IT assets. The term “API value chain” refers to the entire ecosystem and the affairs between assets, API providers, and API consumers.

For example, a fashion retail store exposes its product catalog and sales operations as a public API. The decision was made to take advantage of app-based consumer purchases. In order to achieve this, the API should be able to access internal IT systems—at minimum, the inventory and sales systems.

Assuming these two systems are already in place, the decision to expose data and operations as an API will trigger data flow between these systems and the API layer. Then, the API will be consumed by app developers. These developers will publish apps that consume this API, and the API should facilitate a developer community and deliver a proper developer experience in order to maintain steady engagement with the developers.

This developer experience comes in two flavors. One is based on the technical experience, including documentation, the onboarding process, and SDKs. The other one is the financial gain for the external developers, such as commission strategy and advertising policies.

Finally, published apps will create an app ecosystem. Apps will be utilized by end users—users are mostly unaware of this entire value chain and the complexity.

Figure 2-1 shows how different layers, different stakeholders at each layer, tools and processes at different levels are connected and forms the API value chain.

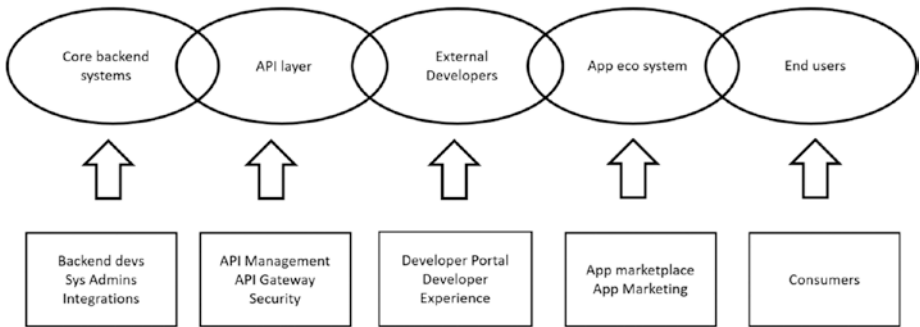


Figure 2-1. *API Value Chain*

The interconnection between these parties is a good example of a basic API value chain.

API Architecture

Business stakeholders—and other stakeholders who live in the intersection of business and technology, like enterprise architects, data stewards, and other organizational evangelists—define the purpose and strategy of an API.

This decision will be evaluated in terms of the effort, budget, organizational context, and current model of IT assets needed to execute API implementation. API implementation next falls to the technical stakeholders, who work closely with business stakeholders to understand the purpose, execution, and limitations of the strategy.

For example, the CEO of a retail store chain has the goal of increasing revenue. He collaborates with other stakeholders, like salespeople and the CTO, finds that mobile-based purchases from their e-commerce site are increasing, and decides to launch a mobile app to establish a more convenient consumer experience, along with the other benefits of an app ecosystem. The organization decides on an API and strategy, and the technical stakeholders are responsible for implementing it.

In order to plan the implementation, technical stakeholders must decide on the API architecture and identify the key constraints and forces at play.

A typical API architecture includes six key aspects, shown in Figure 2-2. Each of these attributes has its own constraints and influence on the overall API design in a given business context.

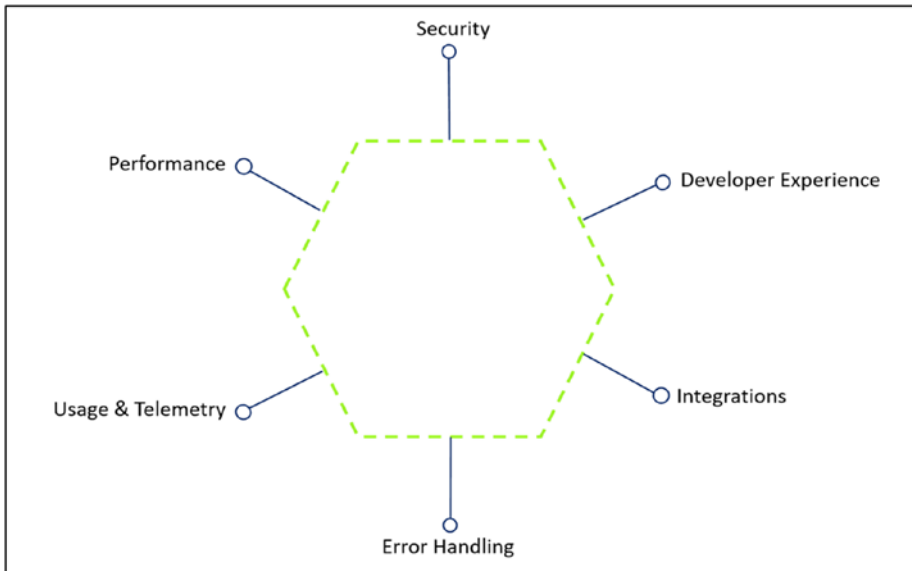


Figure 2-2. *Aspects of API Architecture*

- *Security:* Security is critical. This aspect should address technical concerns like authentication, authorization, injection attacks, and DDoS attacks. There are security concerns in the business context as well, like what data and operations need to be exposed, how to expose them, and to whom. Exposing data and operations that reveal internal information about the business could create an advantage for a competitor.

- *Developer experience:* The success of an API implementation depends on adoption. Developers are the primary audience of an API, and should have a good experience with all aspects of its design. This includes the developer portal, forums, developer tools, and trail API endpoints.
- *Performance:* Performance of an API is expected at every level, not only in terms of responsiveness, but also availability. Caching is a common technique used to increase the responsiveness of APIs. Many-chained integrations and slow data translations from legacy systems are common culprits of performance issues.
- *Integration:* Most enterprise-grade API implementations have integrations with many systems, and the majority of these systems are legacy. Either the APIs talk to some sort of legacy integrations or they talk to a wrapper API, which does the internal dirty work. Combining results from different systems and performing data operations before exposing them is common practice for many API implementations. Some API gateway tools have out-of-the-box basic data translation and transformation capabilities.
- *Usage & Telemetry:* Measuring usage and logging and analyzing telemetry are other important aspects of API architecture, and there are a number of tools available to help with this. Monitoring helps in understanding the usage and adoption of the API, and endpoint-based analysis reveals patterns in how API endpoints are consumed and which endpoints are consumed together. These details will help to continuously optimize the API design.

- *Error handling*: Error handling determines how an API should behave during application-level errors and system failures. System failures are addressed under the system architecture, while application-level failures are addressed under the API architecture. Application-level error handling should address error contracts, error documentation, error contract information level, security, and certain access limits.

As mentioned above, each of these attributes has its own force. The API architecture should identify the correct level of influence for each attribute depending on the business requirement and the context.

API Management

API Management is a solution encompassing the collections of tools used to design and manage APIs, referring to both the standards and the tools used to implement API architecture.

There are several API Management products on the market, and API Management tools are some of the most highly regarded enterprise tools. Microsoft; AWS; IB; and vendors like Apigee, MuleSoft (now Salesforce), and WSO2 all offer API Management tools.

Though different vendors load their API Management tools using different products, they all offer solutions for API design, API gateway, API analytics, and API catalog.

- *API design*: API design includes the ability/features like importing an API from specifications, create API endpoints, define service contracts and generate documentation.

- *API gateway*: An API gateway allows configuring an API gateway engine, manipulating requests and responses, URL rewriting, caching, security enforcement and pre-authentication, and applying request-based security rules.
- *API analytics*: API analytics includes the analytics of API usage, which is often configured as part of the API gateway and tracked and monitored. Analytics provide usage and telemetry insights and reporting dashboards.
- *API catalog*: An API catalog can take different forms in vendor-specific implementations, but in general, it lists available APIs and other access configurations. For example, one API Management solution can have many APIs; some may be public, while the rest could be private access associated with a certain authentication.

The above mentioned API Management tools include granular features like API documentation, API publishing, protocol translation, data translations, data transformation, security capabilities, developer portals, caching, versioning, client SDK generation, usage and telemetry monitoring, URL rewriting, and much more.

API Management and the tools ecosystem is a big business in the IT industry. Many integration service providers offer API Management tools, and enterprise usage of API triggers the demand for API Management solutions. In the following chapters, we will look at different features of the API Management tools offered by Azure and AWS. These API Management tools are offered as cloud services.

Summary

API strategy defines what an organization wants to achieve using APIs. This is the critical decision-making point of API design and initiation.

API implementations touch different layers of an enterprise, as well as external stakeholders. The flow of value between these layers keeps a successful API implementation in place.

API architecture creates the architectural vision derived from the API strategy and puts the technical foundation of API implementation using the API architecture components. API Management tools are packaged with tools to facilitate the API implementation.

In the next chapter, we will consider the fundamentals of API design and a foundational guide for writing a proper API.

CHAPTER 3

API Development

A modern-day developer has access to myriad of tools and frameworks to create a RESTful web service. Spinning up a quick service with CRUD endpoints for database models is easy, but this kind of development does not yield a fully managed API. The key difference is that not all web services are APIs. An API implementation should be strategized beyond just CRUD endpoints; it should facilitate business process and data flow, follow semantics of request URIs and HTTP verbs, have proper developer experience and documentation, implement required security measures and have proper definitions of service contracts and versioning.

When an API development begins, mostly development teams either jump in with their favorite API framework and start development without considering the standards or the teams are stuck with the details of the API implementation standards such as URI, HTTP verbs, exceptions, developer experience, HTTP status code and naming wars. This chapter provides a compact yet thorough set of guidelines to kick-start your API development with the perfect balance between development and API standards.

API Development Considerations

APIs are different from web applications and web services. Technically, modern-day APIs are more similar to a RESTful service implementation, but the purpose of an API is different and beyond the expectation compared to a normal RESTful service. All APIs with RESTful semantics can be considered to be RESTful services, but not all RESTful services are not APIs.

APIs are implemented to expose data and operations to callers. Usability and adaption of an API by various external callers is a critical success factor for any implementation. To achieve this, API implementations have additional considerations to contend with.

As stated, modern tools and frameworks offer rich features for delivering required API implementations. Understanding fundamental API development considerations is necessary to implement a good evolvable API.

Explicit Parameters

API implementations should be stateless and consumer-state agnostic, meaning that APIs should receive parameters from callers and should *not* rely on any client-side, state-persistent models.

A common example is: there are RESTful services developed in order to cater a specific web application. In such cases, it is common to notice these services are developed to accept data from cookies, which is acceptable from a web application to RESTful service call. But if the same RESTful service is intended to be serve as an API for different consumers then relying on cookies will break things. So API implementations should have explicit parameters and accept data from URL parameters or HTTP headers or via HTTP request body.

Avoid Consumer-Commanded Endpoints

APIs and consumers send and receive messages using defined service contracts. Service contract definitions are defined either by the service or by consumers. Both are accepted methods, but consumer-defined contracts are application-centric, whereas service-defined contracts are based on entities and business operations.

However, it is best to avoid endpoints that serve consumer-commanded data—consumer commanded data is about the service contract definitions which contain specific application view model, like various formattings of data and APIs exposing endpoints for simple

data aggregations. In some cases, API responses contain visual stylings like color codes. These kinds of implementations should be avoided whenever possible, since they tightly couple the API implementation to a specific client and a specific application view.

Documentation

APIs should have documentation about the used standards, version, URI syntax, and error codes. Developer experience is not an optional element in API development; tools like Swagger and TRex are helpful in creating documentation and developer experience. Full-fledged API Management tools provide rich documentation and developer experience.

Security

API security is mandatory. Security is not only about authentication and authorization, but also about what data is exposed in which service contract and how endpoints are consumed by the consumers.

Displaying a data property in an error response might help the consumer to correct the request and retry with the correct request parameters, but at the same time, this might open a security loophole that can be exploited by a hacker to obtain certain data. It's important to strike a balance between APIs having helpful responses to clients whilst not exposing sensitive information.

Also, public APIs should have security measures such as IP-based security, tracking the usage of the API key, or limiting the call rate. Modern API Management tools offer the aforementioned out of the box request based security aspects.

Versioning

APIs are software, and software evolves. API development should consider the versioning; versioning of APIs cover two aspects, one is the versioning of the URI and second is the versioning of the service contract. There are many API

versioning techniques available, and the proper technique should be chosen in the early stages of development. Prompt notification to developers about new versions and especially the depreciation of old versions are essential.

API Development Standards

There are a few best practices to follow in API development. As stated in the introduction of this chapter, sometimes developers get overwhelmed with the available information on these standards.

The purpose of this section is to give the best minimum set of standards to get started with the API development with less friction, at the same time these standards allow the developers to extend to more comprehensive implementations if required. The below five standards are extracted from many modern API implementations and compiled as a handy developer guide.

HTTP Verbs

HTTP verbs are key action elements in HTTP communication. Table 3-1 shows the HTTP verbs that are most commonly used in API development.

Table 3-1. HTTP Verbs Quick Guide

HTTP Verb	Common Usage
GET	Get a single entity or list of entities
POST	Create an entity
DELETE	Delete an entity (this can be a soft delete)
PUT	Replace an entity
PATCH	Update properties of an entity

PUT vs PATCH: Developers often confuse these two verbs. PATCH is the latest addition to the HTTP verbs, and well defined in RFC 5789. The difference between PUT and PATCH requests is reflected in the way the server processes the enclosed entity to modify the resource identified by the Request-URI. In a PUT request, the enclosed entity is a modified version of the resource stored on the origin server, and the client is requesting that the stored version be replaced by the new version in the request body.

With PATCH, however, the enclosed entity contains a set of instructions dictating how a resource currently residing on the origin server should be modified to produce a new version. The PATCH method affects the resource identified by the Request-URI, and it may also affect other resources; i.e., new resources may be created, or existing ones modified, by the application of a PATCH. It is commonly observed PUT is being used in many edit operations, some APIs use PATCH.

HTTP Status Codes

HTTP status codes indicate the state of a response from the server, and are defined in ranges. Some API implementations have their own HTTP codes as well. Table 3-2 shows the most common HTTP status codes to quick-start API development.

Table 3-2. *HTTP Status Codes Quick Guide*

Status Code	Common Usage
200	OK - Any successfully processed request. May or may not contain a payload.
201	CREATED - Typical response code for a POST request. Body contains the URI of the newly created entity.
202	ACCEPTED - Request is accepted. Instruct clients to proceed. May contain a URI to check the status of this request, which client can use to do the follow-up.

(continued)

Table 3-2. (continued)

Status Code	Common Usage
204	NO CONTENT - Typical response code for a PUT/PATCH/DELETE request. Typically, body does not contain a payload.
400	BAD REQUEST - Generic status to indicate the issues in requests, i.e., validation
401	UNAUTHORIZED - Used to indicate client authentication / authorization has failed.
403	FORBIDDEN - Used to indicate authorization has failed or a precondition has failed.
404	NOT FOUND - Requested resource is not found.
408	REQUEST TIMEOUT - Server could not process the request in the determined time. Sometimes payload contains retry information.
500	INTERNAL SERVER ERROR - Any unhandled exceptions and server errors fall under this category. Instructs clients that the issue is with the server.

There are additional codes available for more granular responses, and there are variations on these responses; for example, a validation error in the entity can return “400 - Bad Request” with detailed information, or “412 - Precondition Failed.”

Another concern developers have is about using 404 as a status code for a resource retrieval request, such as getting an entity by ID. Some argue that returning a 404 is not valid, since the endpoint to retrieve the resource *is* valid, but the entity itself is not available. So they advocate for returning a 400 as BadRequest or a 200 with empty body instead of 404, because the endpoint is found.

There are many arguments like this, but it is good design and development practice to be consistent in using HTTP response codes and to have detailed messages in the body, particularly during 400- and 500-based scenarios.

Error Handling

Error handling is important, and should be implemented in a way that is helpful to consumers in rectifying issues in the request and guiding them to reach the API back. A common best practice is to return an error response with at least three parameters:

- Correct HTTP status code
- API-specific error code
- Human-friendly error message

API-specific error codes help implementing the client logic easily, rather than processing the human-friendly string message. This also helps in implementing good flow control logic in clients. The error response can contain details such as retry links, retry time interval, and additional helping parameters to modify the response object.

A very latest, RFC 7807 - 'Problem Details for HTTP APIs' has good information on constructing problem detail service contracts. Based on the RFC, the problem detail response includes the following properties. Table 3-3 shows the details of each problem detail entity. The object can be extended with custom properties.

Table 3-3. *Problem Detail Service Contract Properties*

Property	Type	Description
type	string	A URI for the type of the error
title	string	Short description of the error
detail	string	Detailed description of the error
instance	string	Instance of the error

Listing 3-1. Sample Problem Detail Message as Specified in RFC 7807

```
{
  "type": "https://example.com/probs/out-of-credit",
  "title": "You do not have enough credit.",
  "detail": "Your current balance is 30, but that costs 50.",
  "instance": "/account/12345/msgs/abc",
  "balance": 30
}
```

Note Sending information in an error response is an important decision and should be considered with care. In the above example, sending the current balance in the error response has its own pros and cons. Overall security consideration in the development should determine such decisions. For example, if the development emphasizes high security with assumed breach in mind, then it is better not to expose the balance in the error response.

URI Syntax

In a HTTP-based RESTful service, parameters are passed in the request URI or HTTP headers, or in the request body. It is up to the API developers to determine what parameters are sent on which path. Commonly, GET requests pass the parameters in URI, unless there is a specific requirement to do so such as complex search parameters are sent in request body.

URI syntax can be query string-based or based on URI fragments, and there are endless discussions on which method is better. Modern API frameworks support both syntaxes if a specific naming convention is followed, but URI fragment-based syntax is generally preferred due to its semantic approach:

`api/orders/1` over `api/orders?id=1`

It is possible to slot the parameters in the middle of the URI; this makes it more readable:

`api/order/1/products` over `api/order/products?orderId=1`

At the same time, URI fragments do not cope well with all scenarios, like search/filter operations, especially when we require a search endpoint with arbitrary parameters. If we have only one search key value parameter at any given time, then easiest approach is introducing a filter operation and take two parameters, one is filter property type (the key parameter) and second property is the value parameter. In the below example, both “category” and “fashion” are parameters:

`api/products/filter/{category}/{fashion}`

But more complex search scenarios are better served with a request payload. Sending a GET request with the payload that defines properties, values, and filter criteria in the request body would be more suitable than chaining long URI fragments, also this might hit the limitation of the URI length.

RESTful does not define much on the URI syntax, but developers often argue over this topic. Using the above URI segmented approach allows the URIs to be more human friendly, and keeps the request URIs clean from characters like “?”, “=”, and “#.”

Versioning

API versioning standards are generally handled in three different ways. Whichever method is used, APIs expect the version from the client as a parameter, and if the client does not specify the version, the API either falls back to a default version or throws an error.

An API version can be stated in the request URI as a fragment (more common) or as a query string parameter:

```
api/v1/products or api/products?api-version=1
```

Or else, state the API version in HTTP headers. Generally, the header key “api-version” is used to pass the version to the server, but developers can use their own custom header keys.

Or state the API version in the standard Accept header key. This header is used for content negotiation; in a JSON-based API, the Accept header contains the application/JSON as the default value, but some APIs expect the version number to be in the Accept header:

```
application/massrover.v2+json
```

Regardless of the method used, implementation should be consistent across all endpoints of the API. Also, one API can have more than one method enabled in its implementation. The easiest and most straightforward method of implementation is having the version in the URL.

Kick-Start API Development

This section describes a concise approach to getting started with API development using ASP.NET Core and related Visual Studio tooling. We will also explore how to use an API specification to describe a RESTful API. OpenAPI Specification (OAS) is used for this purpose, along with other available Swagger tools.

The OAS uses a standard, language-agnostic interface to describe RESTful APIs, which allows both humans and computers to discover and understand the capabilities of the service without access to its source code. There are rich tools available for implementing the OAS specification and generating API documentation from the API description.

Note OpenAPI Specification (OAS) is formally known as Swagger. Earlier versions of Swagger included both API specification and tools. The owner of Swagger, SmartBear, donated its specifications, making it independent of the tooling and making the specification vendor neutral. The tooling remains under the branded name Swagger. So, in the current context, OpenAPI refers to vendor-neutral specifications, and Swagger refers to the tools used to implement those specifications. There are various other tools available for implementing OAS.

Implementation: ASP.NET Core

The sample API (MassRover API) has CRUD endpoints for a single entity. To begin development, create a simple ASP.NET Core Web API application in Visual Studio 2017. Figure 3-1 shows the selected project template in Visual Studio.

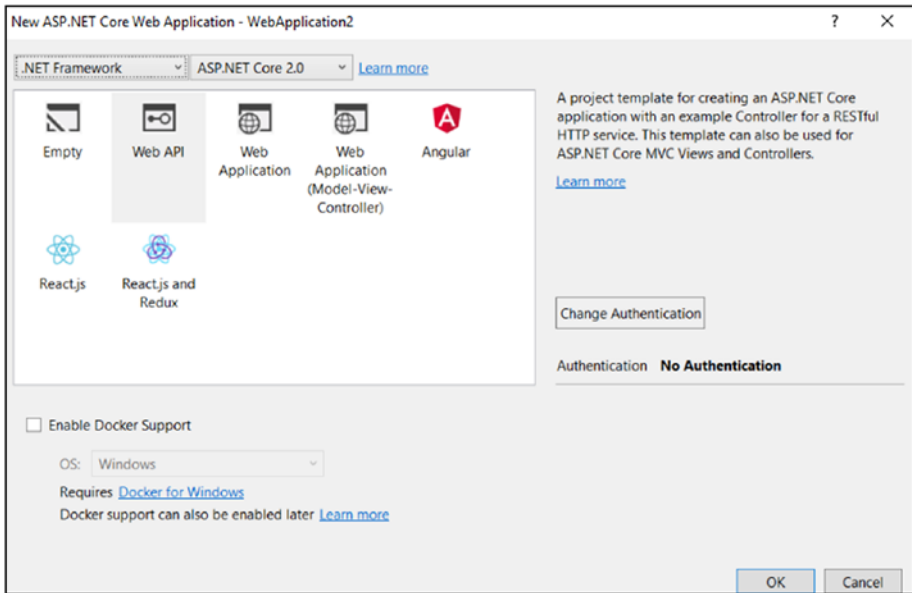


Figure 3-1. ASP.NET Core API Project

Add a folder in the project and name it “Models.” Create the Product model in this folder.

Product.cs

```
public class Product
{
    public int Id { get; set; }
    public string Name { get; set; }
    public DateTime? ModifiedDate { get; set; }
}
```

Add another folder and name it “Errors.” This folder will contain all the used classes and enums to provide error handling implementation of the API as defined in the RFC 7807.

ErrorCode.cs

```
public enum ErrorCode
{
    RequestContentMismatch = 18000,
    EntityNotFound = 18500
}
```

The sample `ErrorCode` enum does not contain any standards; the implementation is specific to the application. When you develop an API, make sure the error codes are consistent across the application and documentation, as API consumers should make decisions and write code based on defined standards.

Add an abstract class in the Errors folder, named “`ErrorMessage`,” with the base properties of the error contract, and implement two specific `ErrorMessage` concrete classes named “`RequestContentErrorMessage`” and “`EntityNotFoundErrorMessage`.”

ErrorMessage.cs

```
public abstract class ErrorMessage
{
    public ErrorCode Code { get; set; }
    public string Type { get; set; }
    public string Title { get; set; }
    public string Detail { get; set; }
    public string Instance { get; set; }
    public string Info { get; set; }
}

public class RequestContentErrorMessage : ErrorMessage
{
    public RequestContentErrorMessage()
    {
        Code = ErrorCode.RequestContentMismatch;
        Type = $"https://massrover.com/doc/errors/#
```



```

        {ErrorCode.RequestContentMismatch.ToString()}";
    }
}

public class EntityNotFoundErrorMessage : ErrorMessage
{
    public EntityNotFoundErrorMessage()
    {
        Code = ErrorCode.RequestContentMismatch;
        Type = $"https://massrover.com/doc/
errors/#{ErrorCode.EntityNotFound.ToString()}";
    }
}

```

Next, add another class to return the correct `ErrorMessage` instance in the right context. This class simulates a service that produces the correct `ErrorMessage` instance. In a real-world implementation, this would be part of the business logic.

ErrorService.cs

```

public static class ErrorService
{
    public static ErrorMessage
    GetRequestContentMismatchErrorMessage()
    {
        return new RequestContentErrorMessage
        {
            Title = $"Request content mismatch",
            Detail = $"Error in the request context."
        };
    }
}

```

```

public static ErrorMessage GetEntityNotFoundErrorMessage
(Type entity, int id)
{
    return new EntityNotFoundErrorMessage
    {
        Title = $"{entity.Name} not found",
        Detail = $"No {entity.Name.ToLower()} found for
the supplied id - {id}"
    };
}
}

```

Now, let's create an ASP.NET Core controller with the actions for the CRUD operations of Product entity. In order to do this, add an empty Web API controller named "ProductsController" (endpoints are listed in Table 3-4). If you'd like, you can delete the ValuesController generated with the Visual Studio template.

Table 3-4. *ProductsController Endpoints*

Action Name	HTTP Method	Response	Error Response
GetProducts	GET	200 List of Products	-
GetProductById	GET	200 Product	404 - Entity Not Found
CreateProduct	POST	201 New Product	-
UpdateProduct	PUT	204 No Content	400 - Bad Request 404 - Entity Not Found
DeleteProduct	DELETE	204 No Content	404 - Entity Not Found

ASP.NET Core provides resourceful attributes in decorating the API. These attributes are helpful during development, and Swagger-like tools leverage those attribute descriptions in generating the API descriptions.

Note MassRover API is a sample reference implementation and does not provide any standards in coding, application structure, code-level architecture, or separation of concerns. It is a simple reference application used to explain and test the detailed information in API development, rather than a development/architectural reference.

Follow these instructions and develop the code for `ProductController.cs`. First, add a product collection (hardcoded) in the controller as shown below.

```
[Produces("application/json")]
[Route("api/products")]
public class ProductsController : Controller
{
    private static List<Product> _products = new
    List<Product>
    {
        new Product {Id = 1, Name = "Lithim L2",
        ModifiedDate = DateTime.UtcNow.AddDays(-2)},
        new Product {Id = 2, Name = "SNU 61" }
    };
}
```

Nest, let's add two HTTP GET actions: one to retrieve all the products, and another to retrieve a product by its ID, parameter is passed via URI path. Also, note that the action methods are decorated with proper attributes for the HTTP method, route parameters, and response types. Each action has its own XML comment as well.

```
/// <summary>
/// Gets list of all Products
/// </summary>
```

```

/// <returns>List of Products</returns>
/// <response code="200">List of Products</response>
[HttpGet]
[ProducesResponseType(typeof(List<Product>), 200)]
public IActionResult GetProducts()
{
    return Ok(_products);
}

/// <summary>
/// Gets product by id
/// </summary>
/// <param name="id">Product id</param>
/// <returns>Product</returns>
/// <response code="200">Product</response>
/// <response code="404">No Product found for the
specified id</response>
[HttpGet("{id}")]
[ProducesResponseType(typeof(Product), 200)]
[ProducesResponseType(typeof(EntityNotFoundException), 404)]
public IActionResult GetProductById(int id)
{
    var product = _products.SingleOrDefault(p => p.Id
    == id);

    if (product != null)
        return Ok(product);
    else
        return NotFound
            (ErrorService.GetEntityNotFoundExceptionMessage
            (typeof(Product), id));
}

```

Add an action to create new products. HTTP POST creates a new product, taking the parameter in the request body.

```

/// <summary>
/// Creates new product
/// </summary>
/// <param name="product">New Product</param>
/// <returns>Product</returns>
/// <response code="201">Created Product for the
request</response>
[HttpPost]
[ProducesResponseType(typeof(Product), 201)]
public IActionResult CreateProduct([FromBody]Product
product)
{
    product.Id = _products.Count + 1;
    _products.Add(product);

    return CreatedAtRoute(new { id = product.Id }, product);
}

```

Add the PUT method for replacing products with the specified ID. This action requires two parameters: the ID of the product to be replaced (this is passed as a path variable) and the product to be replaced with the new values, which is passed in the request body.

This action returns a NoContent response with the HTTP status code 204 for the successful replacement of the product. Otherwise, it responds with two different error contracts. One is “400 Bad Request,” with the request content mismatched when the ID value in the path does not match the ID value of the product in the request body. The other is “404 Not Found,” when the requested entity with the specified ID is not found.

```

/// <summary>
/// Replaces a product
/// </summary>
/// <param name="id">New version of the Product</param>
/// <param name="product">New version of the
Product</param>
/// <returns></returns>
/// <response code="204">No Content</response>
/// <response code="400">Request mismatch</response>
/// <response code="404">No Product found for the
specified id</response>
[HttpPut("{id}")]
[ProducesResponseType(204)]
[ProducesResponseType(typeof(RequestContentError
Message),400)]
[ProducesResponseType(typeof(EntityNotFoundEr
rorMessage), 404)]
public IActionResult UpdateProduct(int id, [FromBody]
Product product)
{
    if (id != product.Id)
        return BadRequest(ErrorService.GetRequest
ContentMismatchErrorMessage());

    var existingProduct = _products.SingleOrDefault
(p => p.Id == product.Id);

    if (existingProduct != null)
    {
        existingProduct = product;
        existingProduct.ModifiedDate = DateTime.UtcNow;
    }
}

```

```

        else
            return NotFound
                (ErrorService.GetEntityNotFoundErrorMessage
                 (typeof(Product), product.Id));

        return NoContent();
    }

```

Add a delete endpoint using the HTTP DELETE action.

```

/// <summary>
/// Deletes a product
/// </summary>
/// <param name="id">Product id</param>
/// <returns></returns>
/// <response code="204">No Content</response>
/// <response code="404">No Product found for the
specified id</response>
[HttpDelete("{id}")]
[ProducesResponseType(204)]
[ProducesResponseType(typeof(EntityNotFoundErrorMessage), 404)]
public IActionResult DeleteProduct(int id)
{
    var product = _products.SingleOrDefault(p => p.Id
        == id);

    if (product != null)
        _products.Remove(product);
    else
        return NotFound
            (ErrorService.GetEntityNotFoundErrorMessage
             (typeof(Product), id));
}

```

```
        return NoContent();
    }
}
```

This concludes the code in the `ProductController`. Next, we'll set up the Swagger tools in ASP.NET Core and leverage the tools with the decorated attribute elements.

Setting Up Swagger

Install the package `Swashbuckle.AspNetCore` in the project by executing the following command in the Package Manager Console (PMC):

```
Install-Package Swashbuckle.AspNetCore
```

Next, set up the `Startup.cs` to activate Swagger tooling and get the Swagger UI up and running in the project.

In the `Startup.cs`, update the `ConfigureServices` method, as shown below. Update the name of the API (MassRover API) and the version of the API (v1) in the `SwaggerDoc`, and update the path for the XML documentation for Swagger to use via `IncludeXmlComments`.

In the sample, in order to provide the XML path, install the package `Microsoft.Extensions.PlatformAbstractions` using PMC. Execute the following:

```
Install-Package Microsoft.Extensions.PlatformAbstractions

public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc();
    services.AddSwaggerGen(c =>
    {
        c.SwaggerDoc("v1", new Info { Title = "MassRover
API", Version = "v1" });
        c.IncludeXmlComments
```



```

        (Path.Combine(PlatformServices.Default.
        Application.ApplicationBasePath,
        "MassRoverAPI.QuickStartSample.xml"));
    });
}

```

Update the `Configure` method as shown below. This enables the Swagger UI and sets the Swagger definition endpoint.

```

public void Configure(IApplicationBuilder app,
    IHostingEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }

    app.UseSwagger();
    app.UseSwaggerUI(s =>
    {
        s.SwaggerEndpoint("/swagger/v1/swagger.json",
            "MassRover Open API");
    });

    app.UseMvc();
}

```

Finally, we will instruct Visual Studio to generate XML documentation based on the comments.

Navigate to Project Properties, then the Build tab, and enable the XML documentation file. By default, this is the path for the bin. Refer to Figure 3-2.

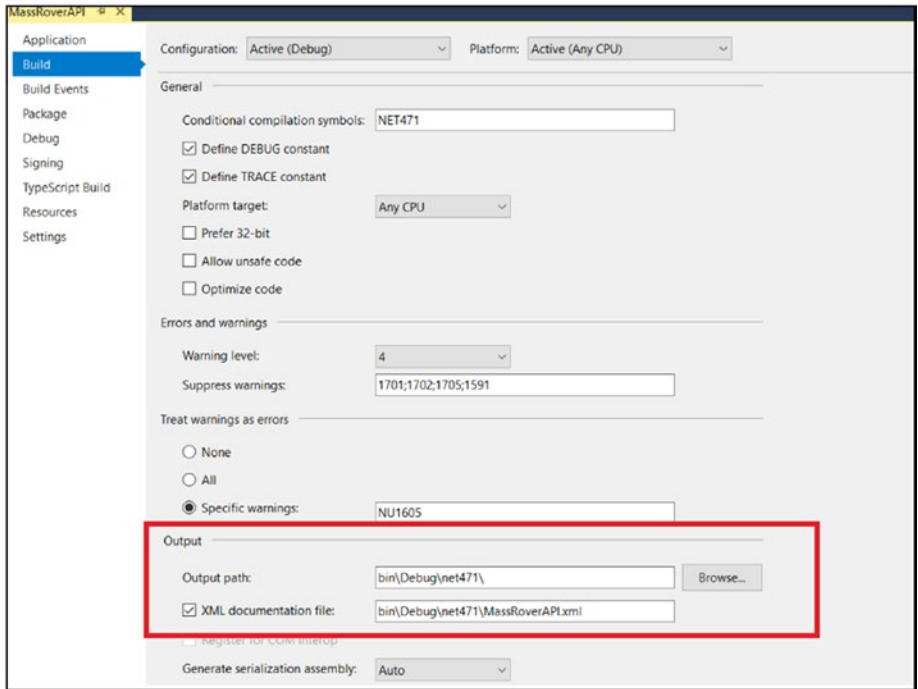


Figure 3-2. XML Documentation Settings in Visual Studio

Note The build configuration in Visual Studio has dedicated setting for each configuration, so you have to perform this step for each build configuration (debug, release and etc) to generate the XML file in the respective build configuration setting.

Note Green Line Problem: Enabling XML documentation causes Visual Studio to look for the XML comments for each implementation. This creates a Visual Studio warning green line everywhere. To suppress this, you can add the rule number (1519) in the Suppress Warnings text, as in Figure 3-2.

Run the API and Swagger

Now you can run the application. Navigate to the URL `http://{host}:{specified port}/swagger`.

Swagger's UI is straightforward, and leverages the XML documentation generated by Visual Studio (Figure 3-3). It provides details of the endpoints, parameters, response types, and response codes. It also provides a comprehensive description of these components as described in the XML comments.

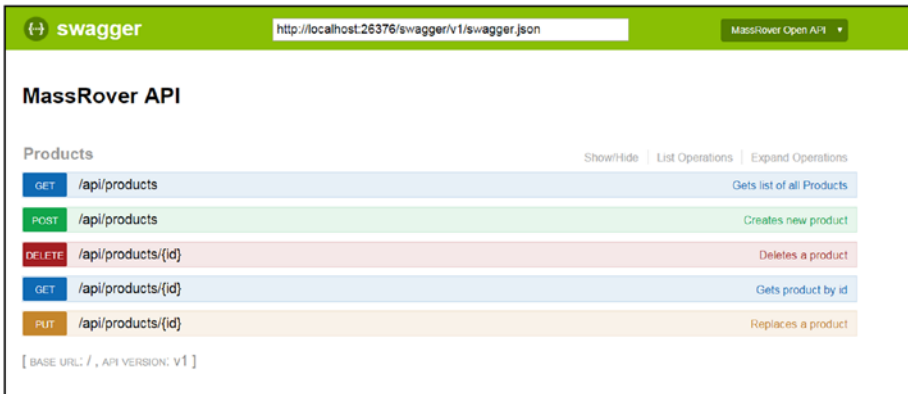


Figure 3-3. Swagger UI

If you expand the PUT method, you will see a similar screen to the one shown in Figure 3-4, including detailed explanations of HTTP response codes and response messages. Note that Swagger leverages the `ProducesResponseType` attribute to generate a model of the error contract.

Response Messages							
HTTP Status Code	Reason	Response Model	Headers				
204	No Content						
400	Request mismatch	<table border="1"> <thead> <tr> <th>Model</th> <th>Example Value</th> </tr> </thead> <tbody> <tr> <td></td> <td> <pre>{ "code": 10000, "type": "string", "title": "string", "detail": "string", "instance": "string", "info": "string" }</pre> </td> </tr> </tbody> </table>	Model	Example Value		<pre>{ "code": 10000, "type": "string", "title": "string", "detail": "string", "instance": "string", "info": "string" }</pre>	
Model	Example Value						
	<pre>{ "code": 10000, "type": "string", "title": "string", "detail": "string", "instance": "string", "info": "string" }</pre>						
404	No Product found for the specified id	<table border="1"> <thead> <tr> <th>Model</th> <th>Example Value</th> </tr> </thead> <tbody> <tr> <td></td> <td> <pre>{ "code": 10000, "type": "string", "title": "string", "detail": "string", "instance": "string", "info": "string" }</pre> </td> </tr> </tbody> </table>	Model	Example Value		<pre>{ "code": 10000, "type": "string", "title": "string", "detail": "string", "instance": "string", "info": "string" }</pre>	
Model	Example Value						
	<pre>{ "code": 10000, "type": "string", "title": "string", "detail": "string", "instance": "string", "info": "string" }</pre>						

Figure 3-4. Generated Swagger UI for PUT

The complete source of the MassRover API Quick Start Sample is found at this repo: <https://github.com/thuru/MassRoverAPI>

Team Orientation in API Development

API implementation requires both business knowledge and technical skills. When multiple teams work together to produce services, a horizontally cross-cutting API team is required to consolidate different endpoints under a single API experience. API teams are commonly utilized in microservices-based architecture implementations.

In a microservices environment, different teams work on different services, thus creating APIs with different standards. A client application for a business user accessing these microservices requires a single standardized API experience. API teams play a key role in standardizing multiple streams of services under one standard channel of API. Figure 3-5 depicts this scenario.

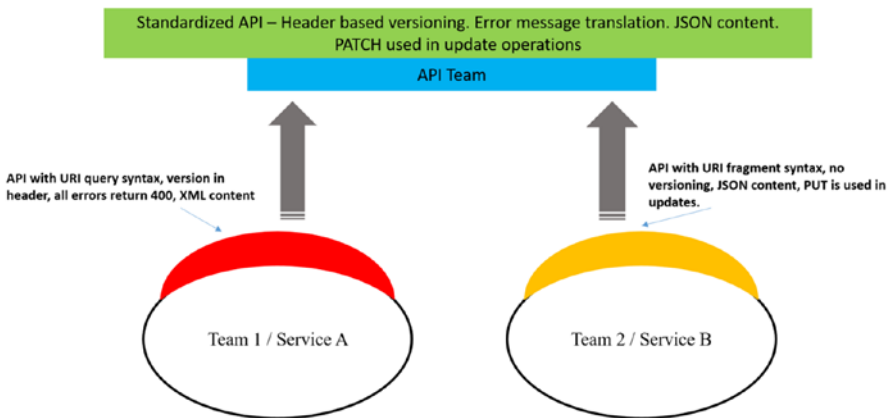


Figure 3-5. API Team Works Across Multiple Teams

Summary

Developing a complete software product requires certain considerations at the outset to avoid confusion and determine development standards. This is quite common in any software project, but API implementations require more significant effort in defining these standards, documentation, and developer agreement, since they are consumed by external parties.

This chapter provides a quick-start guide with the required standards with the balance between quick development and API standards.

Most standards described in this chapter are sufficient to kick-start an API development. Later in the process, developers must consider more complex standards and practices of API development as the business evolves.

CHAPTER 4

API Gateways

API gateways are the front layer for APIs, acting as CDNs. Though they do not necessarily operate on edge servers like CDNs, their main function is to abstract the underlying API or service and provide a uniform access point to consumers. In addition to that, API gateways provide other associated value-added features like caching, security, management, content negotiation, and policy management.

This collection of features, stacked with other architecture components such as developer experience, enterprise integration, telemetry, access and request policies, and access control, form the service known as API management. Most commercial API gateways are offered as a part of the API management tools and services.

This chapter covers two different API management services available from two major public cloud platforms: API Management from Azure and API Gateway from AWS. Note that API Management and API Gateway are the respective names of the API management services from Azure and AWS; these are product names and should not be confused with the generic terms “API gateway” and “API management.”

API Gateways in a Public Cloud

Modern public cloud platforms, especially Azure and AWS, have numerous services and platform offerings. API management is one service most public cloud providers sell.

An API gateway service in a public cloud gets requests from the Internet, most commonly via HTTP/HTTPS. When a request comes to the API gateway, it performs actions on the inbound request, passes the request to the backend service (or sometimes not, depending on the configured rules, such as cached requests or default replies), then performs actions in the outbound response. Figure 4-1 shows this scenario.

In Figure 4-1, consumers make a request through the public Internet. There is a range of possible consumer types, including mobile applications, websites, IoT devices, other API services, and direct human consumers.

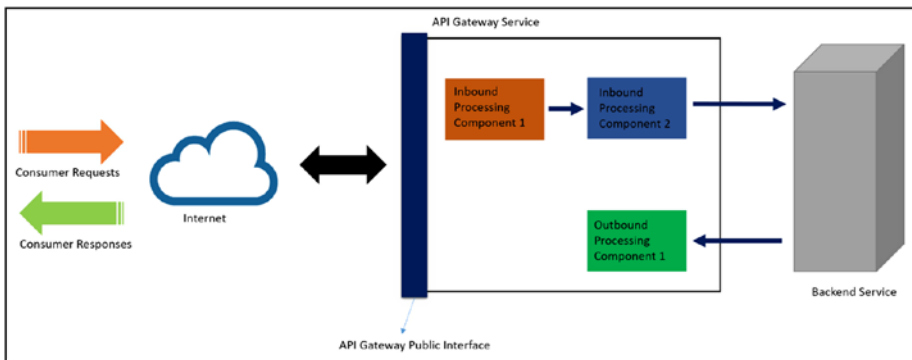


Figure 4-1. *API Gateway Overview*

API gateway services in a public cloud receive requests from the Internet and process those requests based on its configuration. The process acts as a pipeline, similar to any modern web server. We configure static and dynamic rules in the pipeline and, based on this configuration, various components process the request before sending it to the backend service. Not all the processing components modify the request—some work as monitoring and security checks. Some inbound processing components may not send a specific request, depending on validations and set rules.

After the inbound request processing is complete, if validations and rules and configurations allow, the request will hit the backend service. The backend service sees the processed request, not the original request made by the consumer. Also, the API gateway decides which backend endpoint to send the request to—this is not controlled by the consumer. The backend service delivers a response (either success or error) to the API gateway, and the API gateway outbound processing components apply the configured rules and validations and send a response to the consumer. The API gateway has full control over the requests it receives, and has the same control over the response it delivers to the consumer.

Figure 4-1 depicts a fundamental overview of an API gateway in a public cloud platform. Internals of implementation, configuring APIs, connecting backend services, and configuring rules vary based on vendor-specific implementations. In the next section, we will look at implementation details using Azure API Management and AWS API Gateway. As an example, we will use the MassRover product API as the backend.

Endpoint Mappings

As depicted in Figure 4-1, API gateways are the frontiers for client requests, and sit between backend services and requests. In this model, API gateways can have different mappings between backend service endpoints and API gateway interface endpoints. This section describes the various possible mappings and patterns related to these endpoints.

One-to-One Mapping

One-to-one mappings are straightforward: one backend service is mapped to one API gateway interface endpoint. Figure 4-2 depicts this.

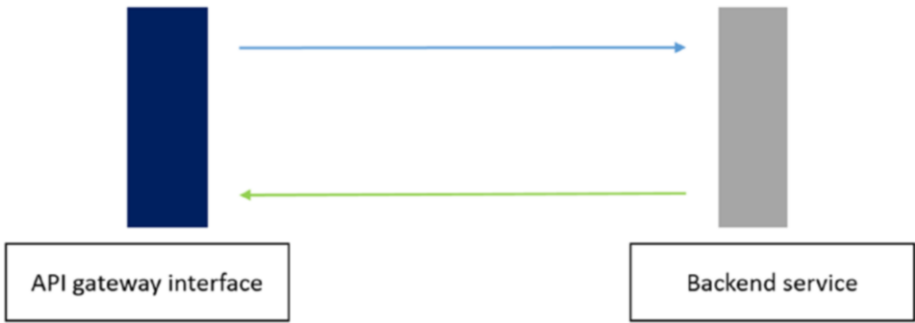


Figure 4-2. One-to-One Mapping

One-to-Many Mapping

In this case, one API gateway interface endpoint is mapped to many different endpoints of the backend service. This mapping is used in the API composition pattern to reduce the roundtrip in fetching data, such as product review information and product details, in a single call. Figure 4-3 depicts this.

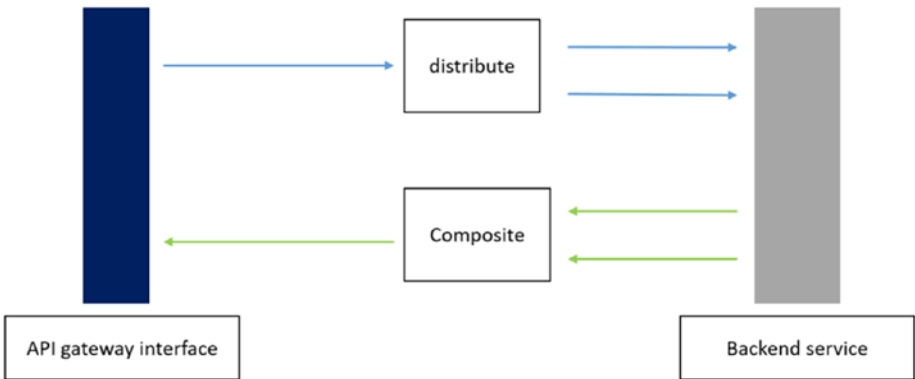


Figure 4-3. One-to-Many Mapping

Many-to-One Mapping

This mapping is used to abstract the direct operation of the backend service and give business meaning. Many API gateway interface endpoints are mapped to one backend service endpoint. For example, an entity has different states, and each state change is technically an update operation. In a business context, each state change can be a different operation in terms of the authorization and semantic meaning, so each API gateway interface has different endpoints exposed with a meaningful URL pattern, but they are mapped to a one-update endpoint of the backend service. Figure 4-4 depicts this.

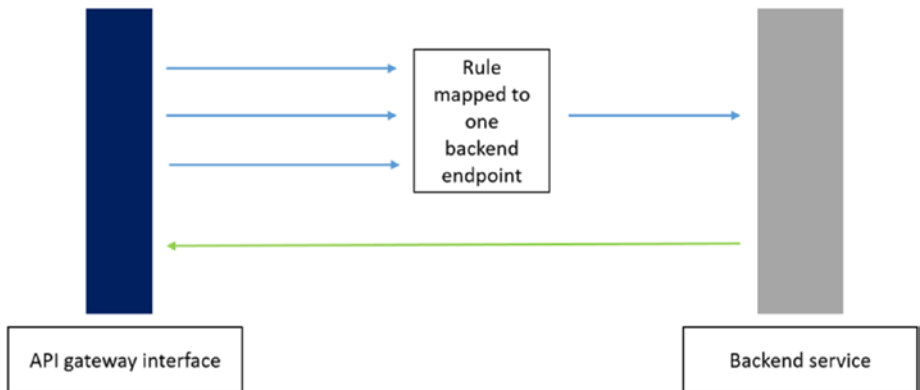


Figure 4-4. *Many-to-One Mapping*

One-to-None Mapping

API gateways are good at handling requests, and are able to execute some fixed logics and return a response without connecting to a backend service. Often, this mapping is used in mocking during development. Also, in some cases, though the endpoint is mapped to a backend service, the API gateway can produce a response without contacting that service. Figure 4-5 depicts this.

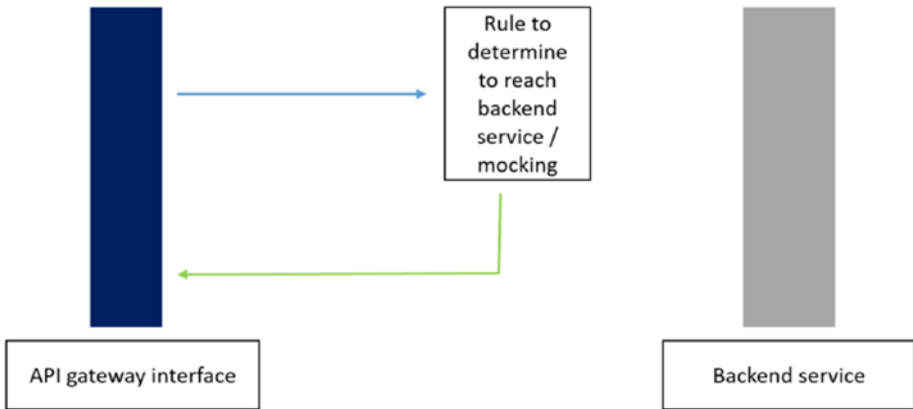


Figure 4-5. One-to-None Mapping

Azure API Management

In this section, we'll look at how to implement an API gateway using the Azure API Management service. API Management is bundled with API Gateway and other API management features, like the developer portal, security, the API catalog, and caching.

Azure API Management has two components: API Gateway and the portal. The portal provides two different experiences, publisher and developer.

Traditionally, these two experiences are delivered via two different web applications, shown in [Figure 4-6](#).

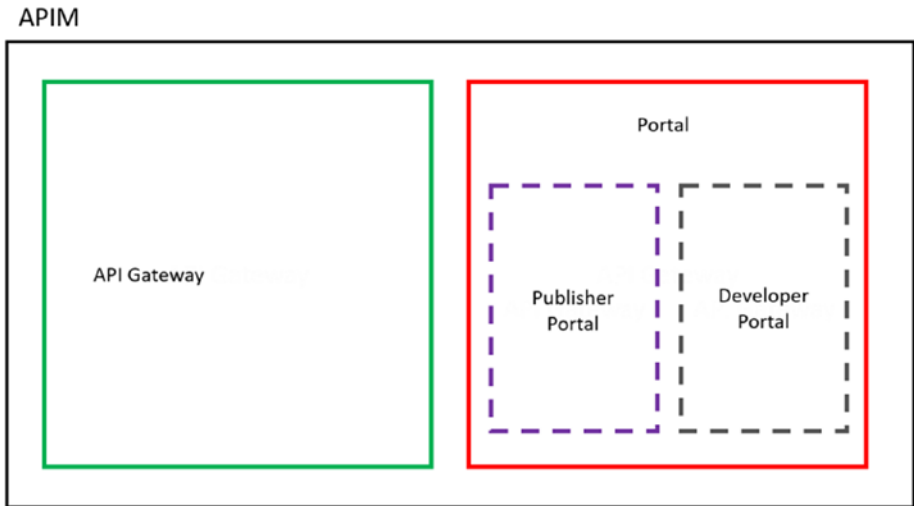


Figure 4-6. *Logical Composition of Azure API Management Service*

When creating an Azure API Management service instance, Azure provisions both the API gateway and the portal. API Gateway is the core engine, receiving requests, processing them, connecting to the backend service, and responding to requests.

The publisher portal provides an administration interface to configure the API gateway and developer portal. The developer portal includes the interface and workflows for developer onboarding, API subscriptions, and other developer experience–related features.

As an API developer, you will spend a lot of time in the publisher portal configuring both the API gateway and the developer portal.

Creating an Azure API Management Service

Navigate to your Azure subscription and search for “API Management,” then select API Management in order to create a service instance. You will see the Azure API Management service creation blade as shown in Figure 4-7.

API Management service [X]

* Name
 ✓

* Subscription
 ✓
.azure-api.net

* Resource group
 Create new Use existing
 ✓

* Location
 ✓

* Organization name ⓘ
 ✓

* Administrator email ⓘ

Pricing tier ([View full pricing details](#))
 ✓

Figure 4-7. Azure API Management Creation Blade

Follow these steps in the Azure API Management creation blade.

1. Provide a name. This name sets the URL of the API gateway and portal. Later, you can configure the DNS for this URL. The API gateway URL appears as `yourname.azure-api.net`, and the portal URL appears as `yourname.portal.azure-api.net`. Appending “/admin” to this will open the publisher portal.
2. Select the subscription and resource group (or create a new resource group), and select the location.

3. Specify the organization name. This name will appear in the developer portal as the organization that publishes the API.
4. Specify the email address of the administrator. The user who creates the service instance will be the default administrator, so it's best to provide the email address of this user until you want someone else to serve as administrator.
5. Select the pricing tier. The Developer tier is the most comprehensive offering, with sufficient request/response limitations in dev/test scenarios. Upon completing the form, you can create the Azure API Management service instance.

Figure 4-8 shows the Azure API Management service instance immediately after creation.

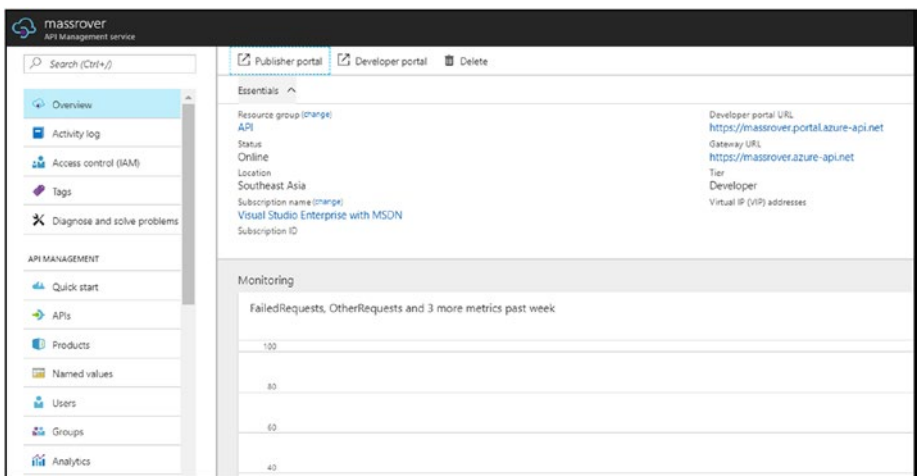


Figure 4-8. Azure API Management Overview Blade

You may notice that the links for the publisher and developer portals appear at the top. Also, the overview blade header section displays the URL of the developer portal and API gateway. The publisher portal URL is the same as the developer portal URL; you should add “/admin” at the end.

Note The Azure portal offers two different experiences. When you click the publisher portal and developer portal links in the overview blade, these portals will be opened (as long as they are valid) in different tabs. But Microsoft Azure is in the process of creating the experience at portal.azure.com. In the future, we can assume that portal.azure.com will be the primary workspace for both the publisher and developer portals, replacing the publisher portal. This book refers to the experience at portal.azure.com as often as possible.

In the overview blade, shown in Figure 4-8, on the left-hand side, you will notice several different menu items under the API Management section. These items define the structure of the Azure API Management service. Click on the APIs option—this will open the APIs blade as shown in Figure 4-9.

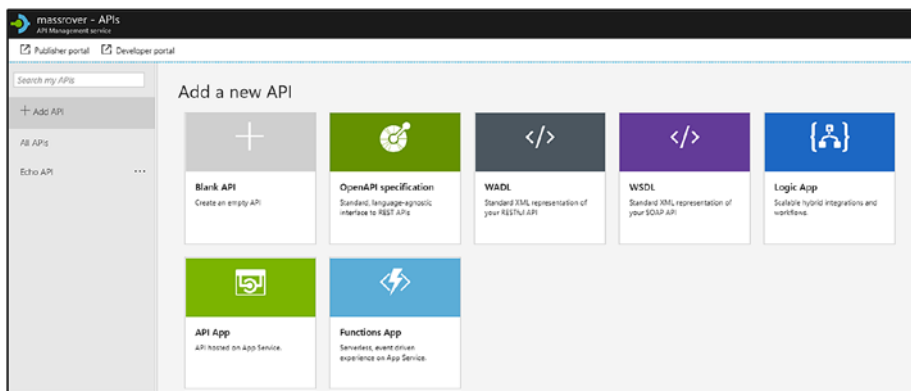


Figure 4-9. APIs Blade

This API blade is the primary starting point for creating APIs in the Azure API Management service, with several options available. In terms of Azure API Management, an API is a collection of endpoints that may or may not connect to a backend service.

An API can also include endpoints from more than one backend service, and one backend service can be included in multiple APIs as well. Backend service-to-API gateway endpoint mappings are discussed in the “Endpoint Mappings” section of this chapter.

Connecting to the Backend Service

First, we should establish a backend service for the Azure API Management service to connect to; we will use the MassRover API. Be sure to host the MassRover API so that it can be accessed from the Azure API Management service.

Assuming the MassRover API is hosted in an Azure API app service, let's import it using the Open API specification. Since we have configured the Open API specification in the MassRover API implementation, we can do this with the portal GUI. As shown in Figure 4-9, click on the “Open API” specification tile to begin the process.

Note Azure API Management offers various options for importing and creating APIs. This chapter focuses on creating an API from the Open API specification; this is commonly used practice in modern development.

This will open a popup, as shown in Figure 4-10.

Create from OpenAPI specification

- OpenAPI specification: or
- Display name:
- Name:
- Description:
- URL scheme: HTTP HTTPS Both
- API URL suffix:
- Base URL:
- Products:
- Version this API?:
- Versioning scheme:
- Version identifier:
- Version header:
- Usage example:

```
GET https://massrover.azure-api.net/[operation]
api-version: v1
```

Figure 4-10. *Open API Specification Pop-up*

You can specify the Swagger definition URL of the hosted MassRover API or upload the JSON file. Fill in the display name and internal name of the API. You can also include a description if you so choose. Select the URL protocol—HTTP, HTTPS, or both. HTTPS is ideal for security and content compression.

An API URL suffix is optional; the base URL will change based on your suffix. Then, select a product. In Azure API Management, products are consumable API packages. We will look at them in more detail later in this chapter, but at this point you will select “Starter Product” (one of the predefined products).

We can version the API by selecting the checkbox. It is good practice to version your API from the very beginning, even though the backend does not support this. Select a versioning scheme (“Header” is selected in this example). Next, provide a version identifier. This can be a whole version number, date, time, or any other string value (the whole version number is used in this example, with a “v” prefix). Finally, provide the header key for the version—the typical setting is “api-version,” which is used in this example.

After completing the form, click “Create” to import the MassRover API to Azure API Management. Now the gateway is fronting the MassRover backend service. After this step, you will see the MassRover API in the APIs section (in addition to the Echo API, which was provisioned by default). Figure 4-11 shows this.

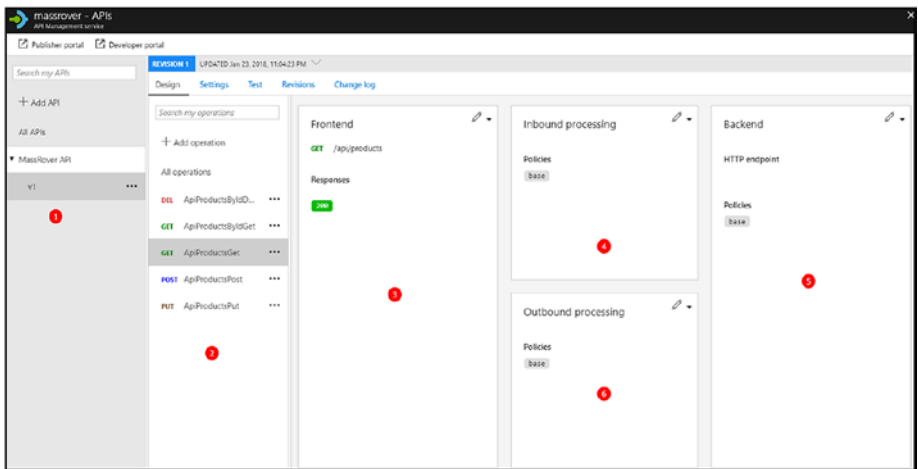


Figure 4-11. API Configuration Workspace (New Portal)

On the left-hand side, you can see the created APIs and their versions. On the right-hand side, you will see the list of endpoints and related workspace. These endpoints are the ones available in the backend service (MassRover API). We can add, edit, or remove endpoints as per the requirement.

1. APIs in the Azure API Management Service. Each has a dedicated sub-section for each version.
2. Endpoints of the selected version of an API.
3. Front-end configuration panel.
4. Inbound processing configuration section.
5. Backend service configuration section.
6. Outbound processing configuration section.

Configuring API Endpoints

After importing the API using the Open API Specification, we need to connect the backend service in order for the API to function. We can provide the backend service URL in the settings of the API. When you select a version of the API, before choosing any endpoint, navigate to the settings from the top menu, as shown in [Figure 4-12](#).

There, you can fill in the backend service URL. This should be the base URL of the hosted MassRover API. Note that this URL should be accessible by the Azure API Management service. Save the settings to persist the changes.

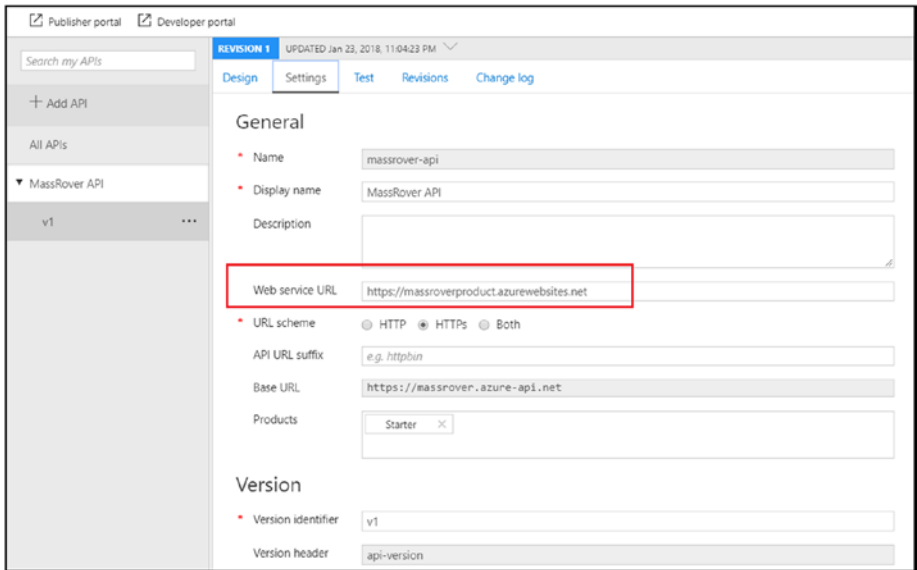


Figure 4-12. *Configuring the Backend Service URL*

Since the MassRover API implementation adheres to standard API practices and Open API Specification, the endpoint mappings are straightforward. Click on the Test tab to test the API gateway.

Figure 4-13 shows the Test tab with the ApiProductsGet endpoint selected. This is the endpoint mapped to “get all products” in the backend service.

As you can see in Figure 4-13, this test blade has several sections:

1. *Query parameters:* Set the query parameters for the request URLs in this section.
2. *Headers:* Set the request headers. You can see some pre-defined headers in this section. One of them is “api-version,” which was configured in the previous step. Other headers, mainly “ocp-apim-trace,” instruct the trace URL for the request. During testing, it's better to set this as “true,” as Azure API

Management will produce a temporary URL of the trace log that can be used outside the portal. The other header is “option ocp-apim-subscription-key.” This header value has the authentication to the Azure API Management service. Every consumer should have a subscription key in order to make calls to Azure API Management. We will discuss these subscription keys later in this chapter.

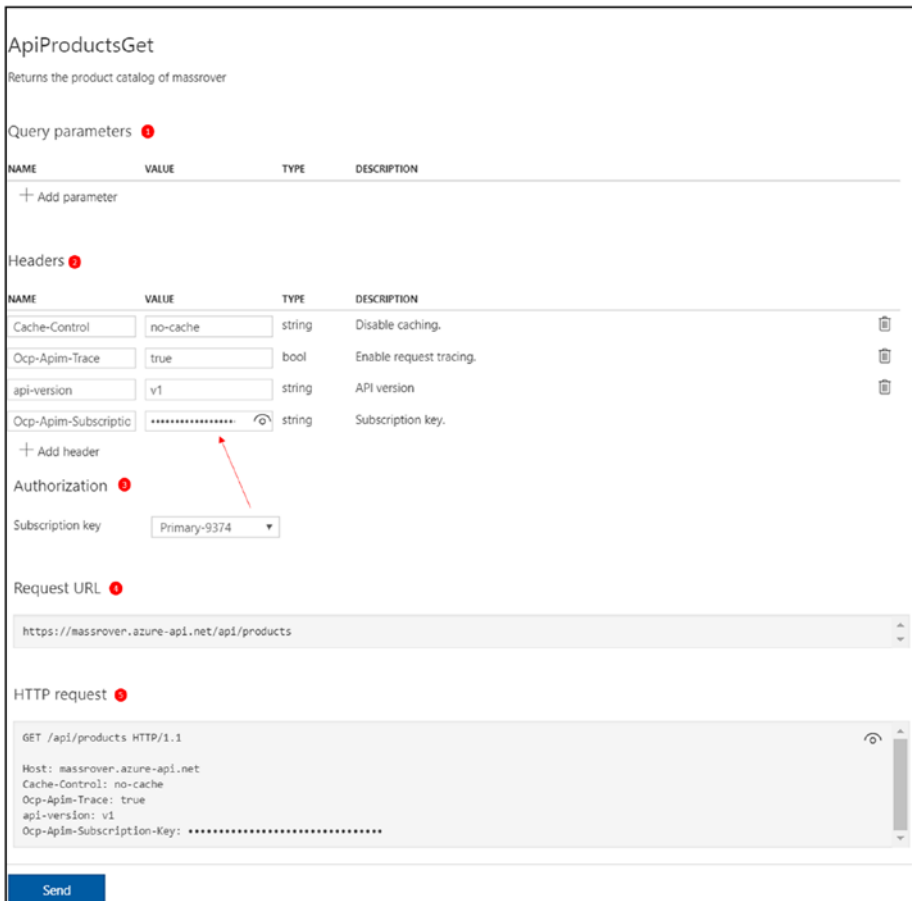


Figure 4-13. Azure API Management Admin Test Console

3. *Authorization*: This is the section where we choose the subscription keys. These are auto-generated by the Azure API Management service, and we can choose either the primary or secondary key. Two keys are available to support the fallback during key cycling.
4. *Request URL*: You can view the request URL here. If you add query parameters, you will notice that this preview section changes dynamically.
5. *HTTP Request*: This section previews the request before it is sent.

Click the Send button at the bottom of the test console blade. This will send the request to the hosted MassRover API.

Since the `ocp-apim-trace` header is set to true, we can see the trace results URL in the response header “`ocp-apim-trace-location`.” The same information is structured in the Trace tab. Figure 4-14 shows the response message in the portal.



```

HTTP response
Message Trace
HTTP/1.1 200 OK
date: Wed, 31 Jan 2018 05:55:28 GMT
content-encoding: gzip
x-powered-by: ASP.NET
vary: Accept-Encoding, Origin
ocp-apim-trace-location: https://apimgmtstc80phda7krgejer.blob.core.windows.net/apinspectorcontainer/veda9ArAuo6Wbef1D5ER0g2-147sv-2015-07-08x
sre68&size8303DorUeEoE004z8R21v5zMRa1zW2Ed.gerUT32RevsERlyx33D&sep2018-02-01T0553A5553A167&spur&traceIdac15aa78422f543fb92f2818467775adc
content-type: application/json; charset=utf-8
transfer-encoding: chunked

[
  {
    "id": 1,
    "name": "Lithium L2",
    "modifiedDate": "2018-01-29T05:55:29.0695181Z"
  }, {
    "id": 2,
    "name": "SMJ 61",
    "modifiedDate": null
  }
]

```

Figure 4-14. Azure API Management Admin Test Response Message

Configuration Policies

Next, let's configure some policies for the API using the Design tab. We can configure individual endpoints with more customized settings using inbound and outbound rules. Click on an endpoint, then click the inbound processing section (#4 in Figure 4-11).

In the right-hand corner, you will see the Edit icon. The Azure portal provides two different edit modes—the form-based editor, which offers less functionality but is still handy enough for most quick edits, and the code-based editor, where we can set policies based on XML policy snippets. Figure 4-15 shows the code editor, and on the left-hand side you can see the list of rules that can be used.

In this section, let's remove the api-version header from the request, as this is not implemented in the backend service yet (sending this in the header will do no harm). We'll also set caching policies for the `ApiProductsGet` endpoint.



Figure 4-15. Inbound Policies Code Editor

Note Though we navigated to the code-based policy editor from the inbound rules section, the policy XML is a single document with all the policy sections included—inbound, outbound, backend, and error.

In the inbound section, click on the Set HTTP Header policy under the Transformation policies section. This will inject the Set HTTP Header XML snippet template. See Figure 4-16.

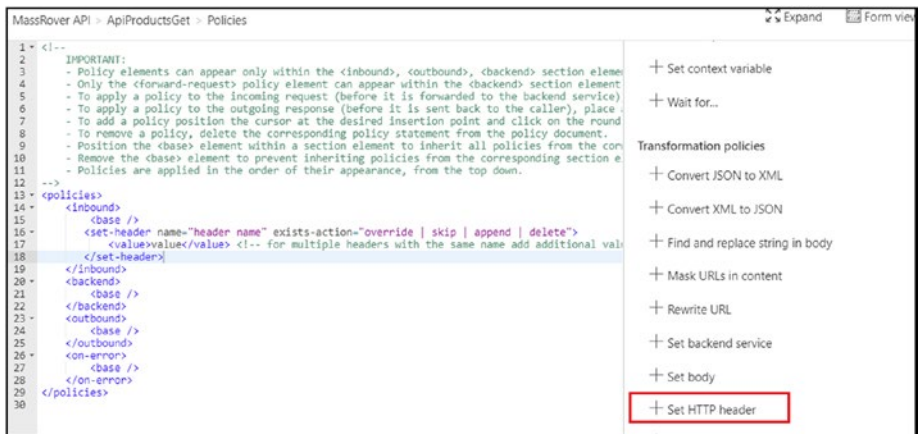


Figure 4-16. Policy XML Snippet (Set HTTP Header)

Below is the XML snippet template.

```

<set-header name="header name" exists-action="override | skip |
append | delete">
  <value>value</value>
</set-header>

```

Edit the template by entering the header name = api-version and exists-action = delete. Since the action is delete, we do not need the value element. After the replacement, the policy will appear as follows:

```

<set-header name="api-version" exists-action="delete">
</set-header>

```


Next, we'll set a caching policy for this endpoint for 60 seconds. The Azure API Management service will cache the response for 60 seconds and respond to requests with low latency.

The caching rule should be set up with two policies. When a request comes in, we should check the cache for the cached value, and for this we need a cache-lookup policy in the inbound section. When the response is received from the backend service, we should store the value of the specific request—for this, we need a cache-store policy in the outbound section.

In the cache-lookup policy, we will set to look the cache, under specified request context. In the below policy configuration, the request context is cached for all the developers in all the development groups (developer experience is described in the next section). The Accept header and Accept-Charset header attributes will vary, meaning individual cache entries are set for varying values for those headers. We can also set any number of header and query string values.

```
<cache-lookup vary-by-developer="false" vary-by-developer-
groups="false">
  <vary-by-header>Accept</vary-by-header>
  <vary-by-header>Accept-Charset</vary-by-header>
</cache-lookup>
```

The cache store policy is set with a timeout period. The default cache duration is 60 seconds, with a maximum of 2,592,000 seconds (one month) for this policy.

```
<cache-store duration="60"/>
```

Next, let's manipulate the response headers. It's assumed safe to hide the implementations and server details of the backend service, so let's add a policy to the outbound rules to remove the "X-powered-By" header value from the backend service.

```
<set-header name="X-Powered-By" exists-action="delete">
</set-header>
```

After we configure all the policies, the full applicable policy set can be verified in the portal using the option “Calculate effective policy.” Using this option, you will see additional policies that come from the product scope policies. We will look at these products in the coming sections. Below is the total set of custom configured policies for the selected endpoint.

```
<policies>
  <inbound>
    <base />
    <set-header name="api-version" exists-action="delete" />
    <cache-lookup vary-by-developer="false" vary-by-developer-groups="false">
      <vary-by-header>Accept</vary-by-header>
      <vary-by-header>Accept-Charset</vary-by-header>
    </cache-lookup>
  </inbound>
  <backend>
    <base />
  </backend>
  <outbound>
    <base />
    <cache-store duration="60" />
    <set-header name="X-Powered-By" exists-action="delete" />
  </outbound>
```

```
<on-error>  
  <base />  
</on-error>  
</policies>
```

You can test some the endpoint and experience the effects of the configured policies.

Note The Azure API Management service has a rich set of policies where conditional and complex decision-making logics can be configured. Azure API Management also supports augmentation of these policies via C#. You can read more about these policies here: <https://docs.microsoft.com/en-us/azure/api-management/api-management-policies>.

Products in Azure API Management

Products are the consumable packages in Azure API Management. Products contain APIs, and one product can have multiple APIs. We can create products using the portal, and each product has different settings. By default, the Azure API Management service is provisioned with two products: Started and Unlimited.

Let's create a product and see what options are available. Navigate to the Products section and click Add to create a new product (see Figure 4-17).

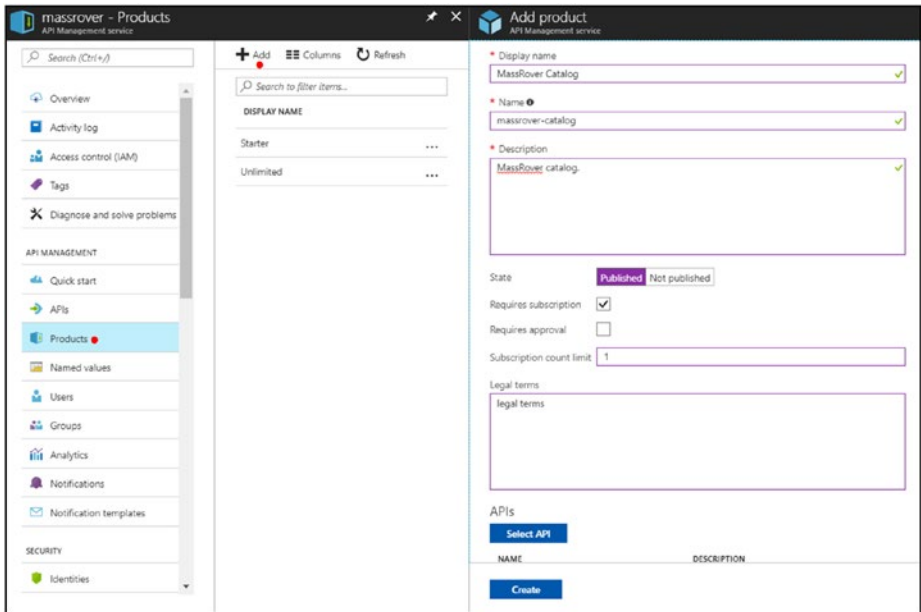


Figure 4-17. Azure API Management Products Blade

A product has the following properties:

1. *Display name*: The publicly visible name, used in the portal, among other places.
2. *Name*: The Azure resource name—this cannot be edited later.
3. *Description*: A short description of the product. This is a mandatory property.
4. *State*: A product can be either published or not published. Only published products are available in the developer portal. The not-published state is analogous to draft mode or disabled mode. By default, the state is not published, which means the product will be available to administrators only.

5. *Requires subscription*: Indicates whether a developer subscription is required to consume the product.
6. *Requires approval*: Indicates whether the subscription requests from developers (which are made via the developer portal) require approval from an administrator in the publisher portal.
7. *Subscription count level*: The number of subscriptions for this product that can be granted to a single developer. One subscription per developer is sufficient in most cases.
8. *Legal terms*: Optional legal terms the developer should agree and adhere to in order to consume the API.

Set the State to Published and check “Requires Subscription.” Finally, select the API to be included in the product (we can add more than one). We can add the MassRover API v1 to this product.

After we create the product, it will be visible in the products blade. Click on the newly created MassRover Catalog product to configure it. Figure 4-18 shows the product blade.

Here, we can configure the settings for the products. You will see a number of tabs:

1. *Overview*: This section shows an overview of the product. At the top we can see the options for changing the state of the product, and we can add or remove APIs. Additionally, this blade shows a summary of the access control.
2. *Settings*: This section is the same as the product creation blade. Here, we can edit the properties of the product, except the Azure resource name.

3. *APIs*: Use this section to add or remove APIs to or from the product.
4. *Policies*: This is the policy configuration for the product. Earlier, we configured policies at the endpoint level, and here we can configure them at the product level, where policies are applied to all the endpoints of all the APIs included in the product.
5. *Access control*: In this section, you can add or remove groups who have access to the product.
6. *Subscriptions*: Here, you can suspend, cancel, or delete subscriptions to the products.

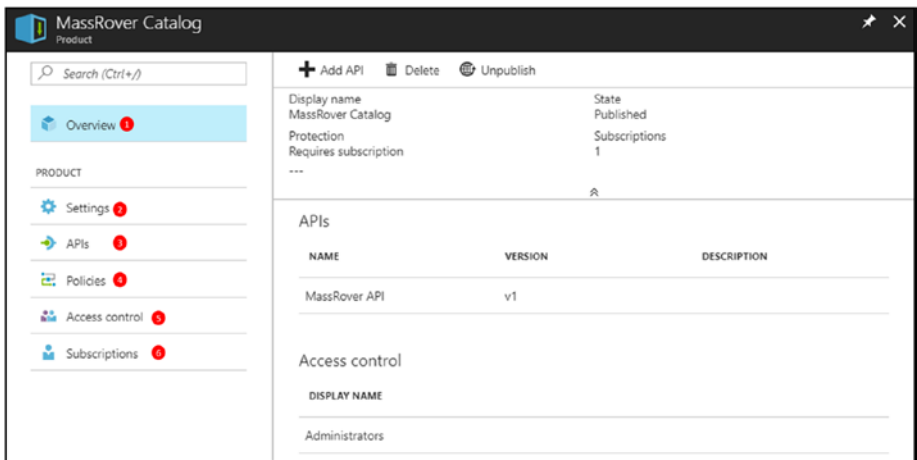


Figure 4-18. Product Configuration Blade

The Azure API Management service has three in-built access control groups with defined permissions.

- *Administrators*: Administrators manage Azure API Management service instances, creating the APIs, operations, and products that are used by developers.
- *Developers*: Authenticated developer portal users fall into this group. Developers are the customers who build applications using your APIs. Developers are granted access to the developer portal and build applications that call the operations of an API.
- *Guests*: Unauthenticated developer portal users, such as prospective customers visiting the developer portal of an Azure API Management instance, fall into this group. They can be granted certain read-only access, such as the ability to view APIs but not call them.

Additionally, we can add custom groups, but the access control level stays in the built-in groups. Especially when adding the all the developers to single developer group will expose all the products to all the developers. In order to avoid this, we can use custom groups to group the developers and assign them access.

Azure API Management Developer Experience

Now we have a product, and we can deliver it to developers who will consume the API. As stated, products are the consumable packages from the Azure API Management service.

First, navigate to the developer portal. You can get the developer portal URL from the overview blade of the Azure API Management service, shown in Figure 4-8. Copy the URL and navigate to it using a private browser session. This will keep you from logging in to the developer portal as an administrator.

The raw developer portal will look similar to the screenshot shown in Figure 4-19. At the top, you will see the tabs Home, APIs, Products, Applications, and Issues.

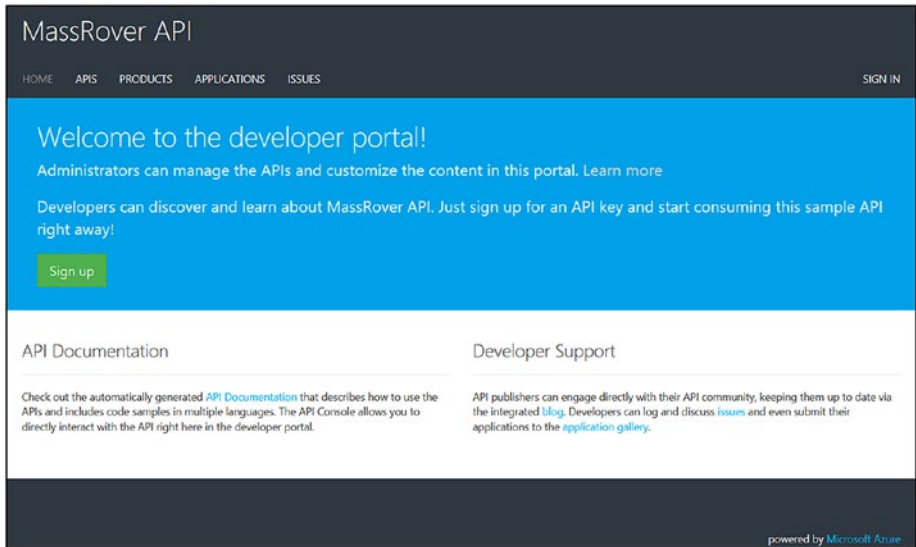


Figure 4-19. *Developer Portal*

The APIs section lists the APIs in the Azure API Management service instance, and you can see the available products in the Products tab. If you click the Products tab, you will see Started and Unlimited products (which are default products created by the Azure API Management service during the provision), but not the MassRover Catalog. This is because we haven't signed in to the developer portal, and as per the product access controls, MassRover Catalog is only accessible to administrators.

The Applications tab shows a list of registered applications using different APIs. The Issues tab is a developer issues reporting section. Overall, the developer portal provides a comprehensive experience, from authentication to subscriptions to application cataloging to issue reporting.

As an anonymous user, you can see the APIs and products (which have guest access enabled). If you have a valid subscription key, you can use it and test the APIs as well. Guest access is enabled for prospective API consumers, and they can view the available endpoints and documentation. They cannot make calls unless they have a valid subscription key.

In a typical workflow, a developer can access the developer portal in one of three ways.

1. A developer can sign up in the developer portal itself using a simple registration or a configured login provider, like Facebook or Google. This can be configured by an administrator in the Azure portal.
2. An administrator can add a developer via the portal using basic authentication. In this case, the administrator sets the username and password for the developer. The developer can change the password later.
3. An administrator can send an invitation to a developer. The administrator completes the basic profile and an invitation will be sent to the developer along with a link. The developer can then complete the registration using the link and set a password.

Register yourself as a developer, or send an invite and log in to the developer portal as a developer.

In order access the MassRover Catalog product, we need to add the developer groups. You can perform this action in the access control section of the product configuration blade (see Figure 4-18).

After adding a developers group to the product, log in to the developer portal using developer credentials. You will now see the MassRover Catalog under the Products tab (see Figure 4-20). As a developer, you can subscribe to this product in order to consume it.

Note The Azure API Management developer workflow is quite comprehensive, and can be configured to your preferences. Emails are sent at different stages of the developer workflow, and email templates are fully customizable in the portal. The developer portal branding can also be customized. This book deliberately does not cover those areas.

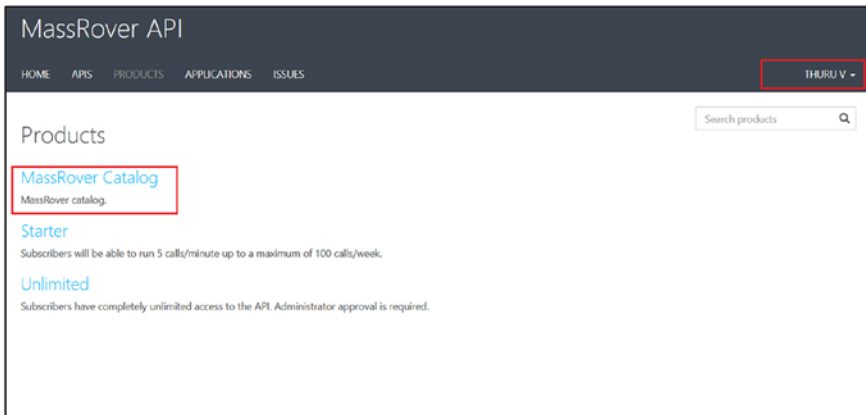


Figure 4-20. Developer Portal Products—Developer Logged In

Click on the product, accept the terms and conditions, and click subscribe. This is shown in Figure 4-21.

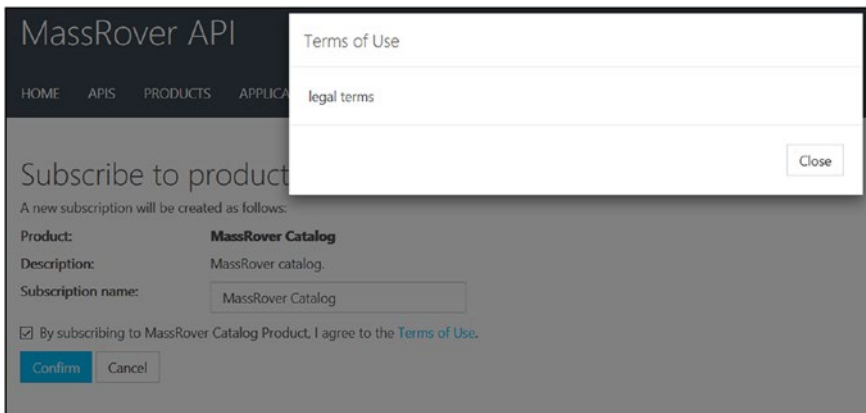


Figure 4-21. Subscribing to a Product—Developer Experience

Since we configured the MassRover Catalog product to accept subscriptions without approval, you will see the product on the developer page immediately after you subscribe, as shown in Figure 4-22.

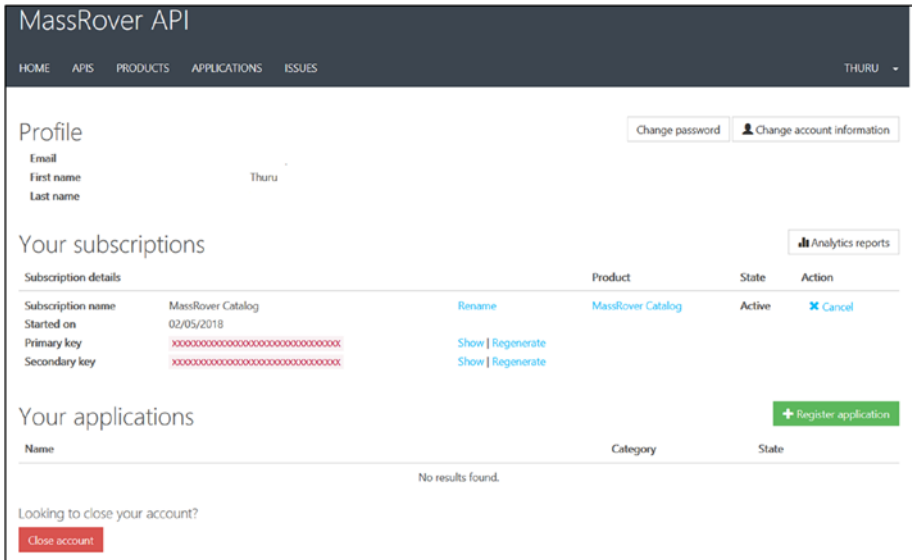


Figure 4-22. *Developer Subscriptions Page—Developer Experience*

A developer can access and generate subscription keys on the developer subscription page. The keys should be submitted under the `ocp-apim-subscription-key` header. The subscription keys act as a first-level security feature, but more importantly, they are used to track developers, usage, and policy configurations.

Developers can use their subscription keys and test APIs in the developer portal itself. The Azure API Management developer portal includes a test console with code samples to consume the endpoints. Developers can also register their applications using the developer portal.

Structure of the Azure API Management Components

In Azure API Management, an API is a collection of endpoints, which may or may not connect to a backend service. Also, these endpoints can connect to different backend services as well. Each version of the API is treated as a separate API. Policies can be configured at both API and endpoint levels.

APIs are linked with products, and products are consumable packages. We can configure policies at the product level as well. Products can include multiple APIs. Developers can consume published products (endpoints of the APIs included in the product). A developer can have more than one subscription to each product. Each subscription is identified by a subscription key, and each subscription key is scoped to a product. A developer cannot use a single subscription key for two different products (see Figure 4-23).

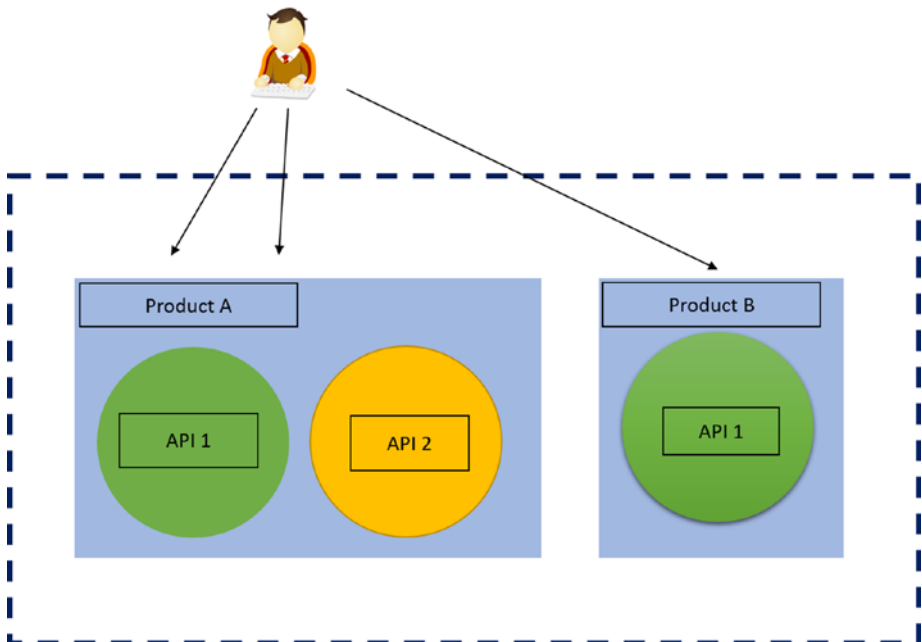


Figure 4-23. Structure of Azure API Management

Subscriptions are granted via the developer portal. As illustrated, the developer has two subscriptions to Product A and one subscription to Product B. In the developer portal, you can configure additional workflows during the subscription process, like external sign-in and payment processing.

AWS API Gateway

API Gateway is the commercial name of the API management service offered in AWS. You can create an AWS API Gateway instance in AWS. AWS API Gateway offers features like caching, request control, authentications, mocking, and API publishing via AWS Marketplace.

API Gateway is tightly coupled with other AWS services like VPC and Lambda. It also allows client SDK generation for popular platforms like iOS and Android. It has pipeline of API gateway public interfaces, request processing, connection to the backend service (or mocking), and response processing.

Creating an AWS API Gateway Service

Assuming you have an AWS account with IAM permission to perform actions, log in to the AWS console, search for API Gateway, and create an AWS Gateway instance. Click “Import from Swagger” and copy the Swagger definition of the MassRover API in the panel (Figure 4-24), or upload the definition file. AWS also has a sample API implementation option.

In the Settings section, select the endpoint type. There are two options for this.

- *Edge optimized*: This the default option, which enables the AWS Cloud Front distribution and improves connection time. This is a good choice in most cases. API requests from clients will be routed to the nearest CloudFront edge servers across AWS regions (similar to the CDN).

- *Regional*: This option will route client requests to the region-specific API gateway, bypassing CloudFront distribution. A request from the same region has the benefit of avoiding an unnecessary round trip to CloudFront, but requests from other regions may have latency. This can be achieved by deploying region-specific API gateways in target regions.

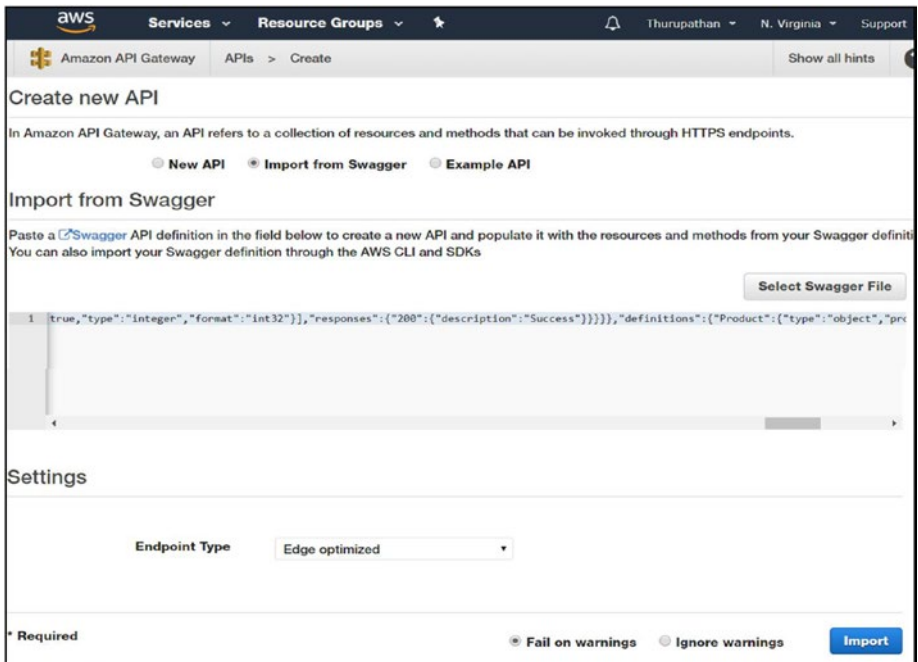


Figure 4-24. AWS API Gateway Provisioning

Click Import to import the MassRover API definition. After the import, you will see the API in the panel. AWS API Gateway structures APIs as resources and methods. The API URI segment is a resource, and HTTP actions are referred to as methods.

In a URI segment like `api/products`, `API` and `products` are two different resources. A resource can have other resources and methods, and path parameters can also be added as resources. You can use curly braces to indicate the path parameters, as shown in Figure 4-25.

In the MassRover API, the `products` and the path parameter (`ID`) are different resources and have methods within them. Methods are associated with the HTTP verb.

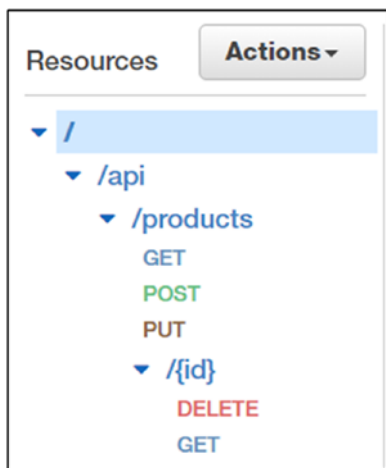


Figure 4-25. API Structure in AWS API Gateway

You can select a resource and add a resource or method to it using the Actions drop-down menu on top. When adding a path parameter as a resource, use curly braces. Click on “method” to configure it.

Configure Methods

Since we have only imported the definition of the API, the configuration should be performed. There are a few different options for configuring the backend service for a method. Click on the GET method under `api/products`, and you will see the panel shown in Figure 4-26.

This is the phase 1 panel, and we must choose the integration point for the selected method. From a basic configuration, select the HTTP option and paste the URL of the hosted MassRover API product's endpoint in the Endpoint URL text box. This will complete the backend integration for the selected method. You can also choose "mock" as an option and later connect to the right service.

Content Handling offers three different options to handle the request body of the specific method.

- *Pass-through*: This is the default option, used when no content conversion is required.
- *Convert to binary*: This is used when the backend service requires the body to be in binary format, when the original request is submitted in base64 format.
- *Convert to text*: Converts the binary input body request to base64 string. This is used when the backend service requires the binary body to be in a string-based format.

HTTP proxy integration is used to streamline the backend service as a single API entry point. The HTTP verb ANY is used in this case, as this will allow the method to accept any HTTP requests from the client.

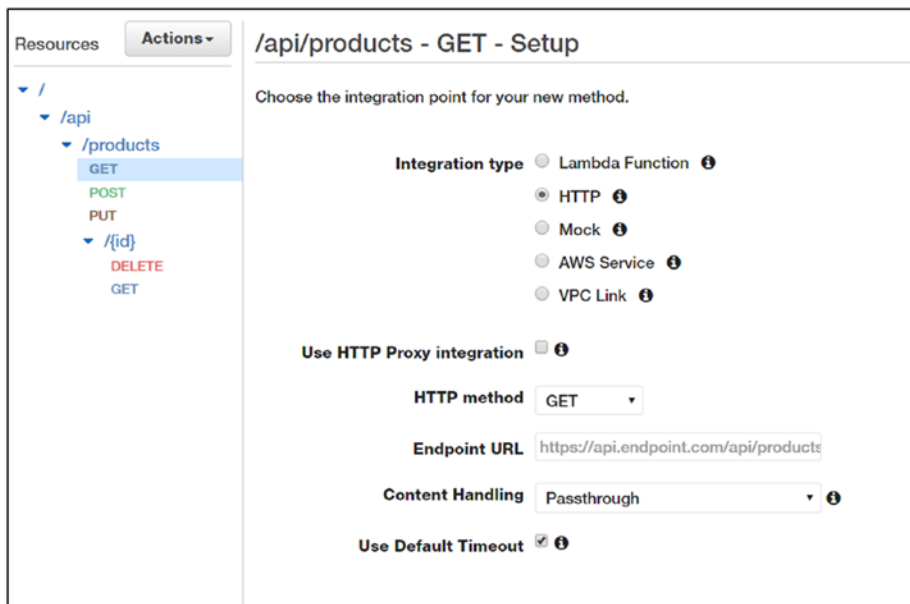


Figure 4-26. API Method Configuration—Panel 1

After configuring the integration point, we can test the method. Click on the specific method, and you will see a panel as depicted in Figure 4-27. This shows the AWS API Gateway request and response flow.

The requests are received by the AWS Gateway public interface, then passed to the method request, then to the integration request process, and finally to the backend service. Similarly, responses are received in the integration response, processed, and forwarded to the client.

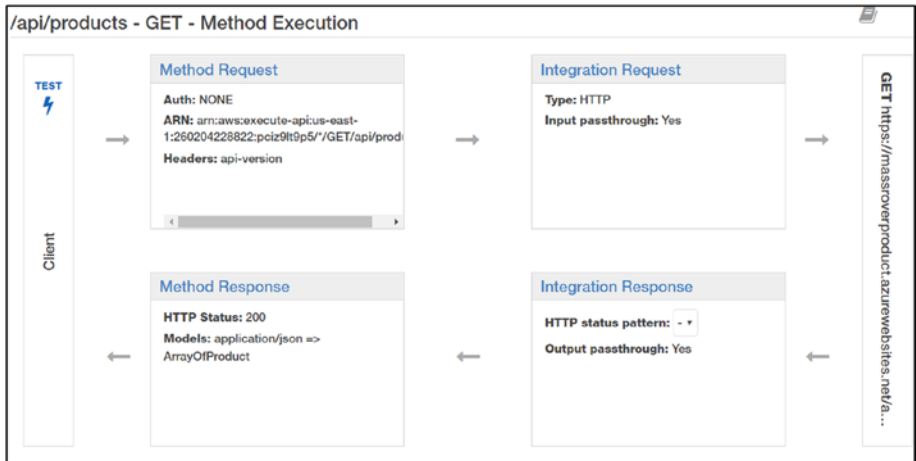


Figure 4-27. API Method Configuration—Panel 2

In this pipeline, the path parameters, query string parameters, header values, and content payload body mappings should all be mapped between the method request and integration request. These mappings are used to add, edit, or drop specific values.

If the client sends a request with the header “version” but the backend service expects the header “api-version,” then this should be mapped between the method request and integration request. In order to do this, we should first define the method request variable. Click on Method Request and expand the HTTP Request Headers section, shown in Figure 4-28. Add a header from the client request.

Name	Required	Caching
version	<input checked="" type="checkbox"/>	<input type="checkbox"/>

Figure 4-28. Method Header Mapping

Next, we will create the mapping of this header value in the integration request. Go back to the pipeline panel (Figure 4-27). Click on Integration Request and expand the HTTP Request Headers section, shown in Figure 4-29.

Here, we can set the mappings using the AWS API Gateway mapping syntax. This syntax is very straightforward:

```
method.request.{path|header|querystring}.{parameter_name}
```

In this case, we should map the incoming header (version) to the api-version. Create a header value and apply the following mapping in the text box (shown in Figure 4-29):

```
method.request.header.version
```



Figure 4-29. Integration Request Header Mapping

After the integration, test the method endpoint. You will provide a header value named “version” in the request. We can check the mapping of the header value in the logs.

```
Method request headers: {version=v1}
```

```
Endpoint request headers: {api-version=v1, x-amzn-
apigateway-api-id=pciz9lt9p5, Accept=application/json, User-
Agent=AmazonAPIGateway_pciz9lt9p5, X-Amzn-Trace-Id=Root=1-
5a8001d3-ad377f51777323dd3897df2}
```

In order to drop a header being passed to the backend service, you can simply remove the mapping in the integration request, as shown in the logs below. There's a header (`abc`) in the request, but it is not in the request forwarded to the backend service. Note that when you add parameters in the method request, they are automatically mapped in the integration request, so you have to explicitly remove the mappings.

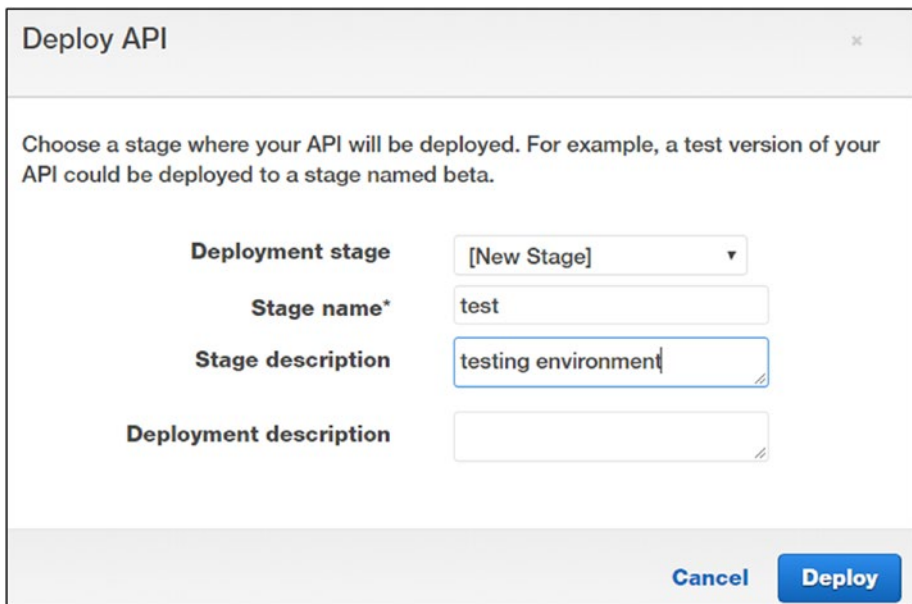
```
Method request headers: {abc=zzz, version=v1}
```

```
Endpoint request headers: {api-version=v1, x-amzn-  
apigateway-api-id=pciz9lt9p5, Accept=application/json, User-  
Agent=AmazonAPIGateway_pciz9lt9p5, X-Amzn-Trace-Id=Root=1-  
5a800263-ed2e227b694e156930d6176e}
```

You can also specify the models for the request/response payload in the Models section of an API and use them in the mapping. Also, Gateway responses are used to configure the HTTP status code, response messages, and header values of the API gateway.

Deploy AWS API Gateway

We must deploy the API in order to allocate usage plans and define API keys. In the panel shown in Figure 4-27, select the Deploy API action from the Actions menu. This will prompt you to create a stage (as shown in Figure 4-30). Stages are the different environments of the deployed APIs. All methods should have integrations; once this condition is fulfilled, you can deploy the API.



The screenshot shows a dialog box titled "Deploy API" with a close button (X) in the top right corner. Below the title bar, there is a text instruction: "Choose a stage where your API will be deployed. For example, a test version of your API could be deployed to a stage named beta." The form contains four fields:

- Deployment stage:** A dropdown menu currently showing "[New Stage]" with a downward arrow.
- Stage name*:** A text input field containing the text "test".
- Stage description:** A text input field containing the text "testing environment".
- Deployment description:** An empty text input field.

At the bottom right of the dialog, there are two buttons: a "Cancel" button and a blue "Deploy" button.

Figure 4-30. *Creating a Stage and Deploying the API*

Once the stage is created, we can configure more things at the stage level. Also, since we can create any number of stages, we can deploy the same API in different environments. Figure 4-31 shows the stage options. In this panel, the Settings section includes options for the request level, caching, and client certificate. You can also tag the stage with the necessary key value pairs.

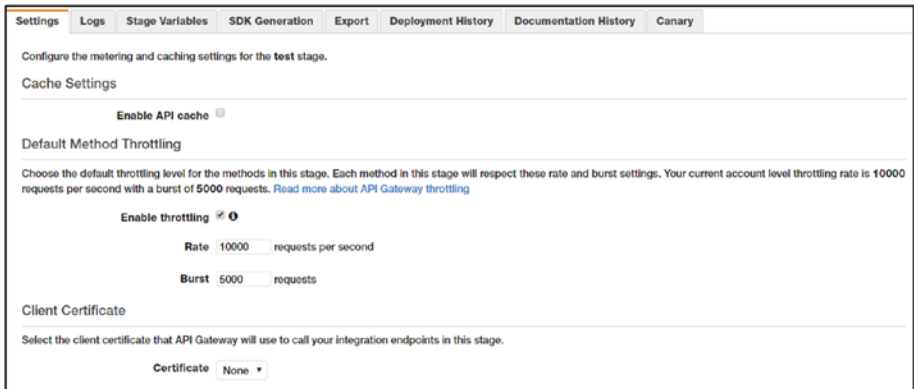


Figure 4-31. API Stage Settings

- *Settings:* In the Settings section, we can configure caching. AWS API Gateway allows us to select the cache size, and it is priced accordingly. In the method level, we can configure whether a specific method should be cached or not. We can also configure the method throttling in this section; throttling is used to secure the rate limit of the API. Finally, we can set the client certificate for authentication. Chapter 5 explains the rate limits and authentication in more detail.
- *Logs:* Here, we can configure the logs for the API. In the AWS world, this is typically the CloudWatch service.
- *Stage variables:* We can set variables as key value pairs, which can be accessed in the mappings.
- *SDK generation:* AWS provides support for generating client SDKs for the API. This includes popular mobile programming languages Android and SWIFT, and languages like Java and Ruby.
- *Export:* Here, we can export the API definition in different formats.

- *Deployment history*: This section contains the deployment history of the stage.
- *Documentation history*: This section displays the history for the attached documentation of the stage. Documents can be created, edited, and published in the Documentation section of the API.
- *Canary*: This section allows us to set a canary for the stage. Canary releases make sure certain portions of traffic go to the mentioned canary, while the rest go to the older version. Consider a scenario where a new version of the backend is available; this can be reached with the header value `api-version = v2`. But we do not want to route all the traffic to the `v2`, so we will set a canary release on the stage and select the percentage of the traffic that should go to the new version. The new version is identified by the variable, and we can override it in the canary settings by submitting a new canary variable.

Creating API Usage Plans

Next, we will create API usage plans to be consumed. Usage plans help to meter API usage and regulate it via throttling limits. We can assign different API stages to a single-usage plan, and it can contain stages from different APIs.

Click on the Usage Plans option in the main menu on the left-hand side. Create a usage plan as shown in [Figure 4-32](#).

The screenshot displays the configuration interface for creating an API Usage Plan. It is divided into several sections:

- Name***: A text input field containing "Plan 1".
- Description**: A large text area for providing details about the plan.
- Throttling**: A section with a dropdown arrow, containing:
 - Enable throttling**: A checked checkbox with an information icon.
 - Rate***: A text input field with "10000" and the unit "requests per second".
 - Burst***: A text input field with "1000" and the unit "requests".
- Quota**: A section with a dropdown arrow, containing:
 - Enable quota**: A checked checkbox with an information icon.
 - Quota value**: A text input field with "250000" and a dropdown arrow.
 - Unit**: A dropdown menu showing "requests per" followed by "Month".
- * Required**: A legend indicating that fields with an asterisk are mandatory.

Figure 4-32. *Creating an API Usage Plan*

Once you name the plan, you can enable throttling and define the request quote for the plan. In the next step, we will select the API and the stage of the API to be included in the plan. In order to track the throttling and quota limits, we should assign API keys to the usage plan. AWS API Gateway uses the subscription to meter the requests.

We can either import an existing key or request the AWS API Gateway to autogenerate the key. Click on Create API Key and add to Usage Plan (refer to Figure 4-33). Assuming you don't have any keys available, here you can choose to autogenerate the key. The new key will be assigned to the usage plan.



Figure 4-33. *Creating API Keys for Usage Plan*

The newly created key can be found in the API Keys section. Finally, you will create the usage plan. Once it has been created, you can click on the plan and edit its properties. When you select a plan, you will see a tab called Marketplace, which is the developer delivery channel for the API. We can associate the usage plan with an AWS Marketplace product using the product code.

Structure of AWS API Gateway Components

AWS API Gateway can have multiple APIs, and each API has set of resources. Each resource can have more resources or methods in it, and each method has an associated integration with a backend service or a mock.

An API is deployed in stages. One API can have many stages, and each stage can have its own settings and configurations. A usage plan can have many APIs. Usage plans have quotas and throttling settings, tracked by subscription keys. Developers access usage plans from AWS Marketplace; this is the API monetization mechanism in the AWS context.

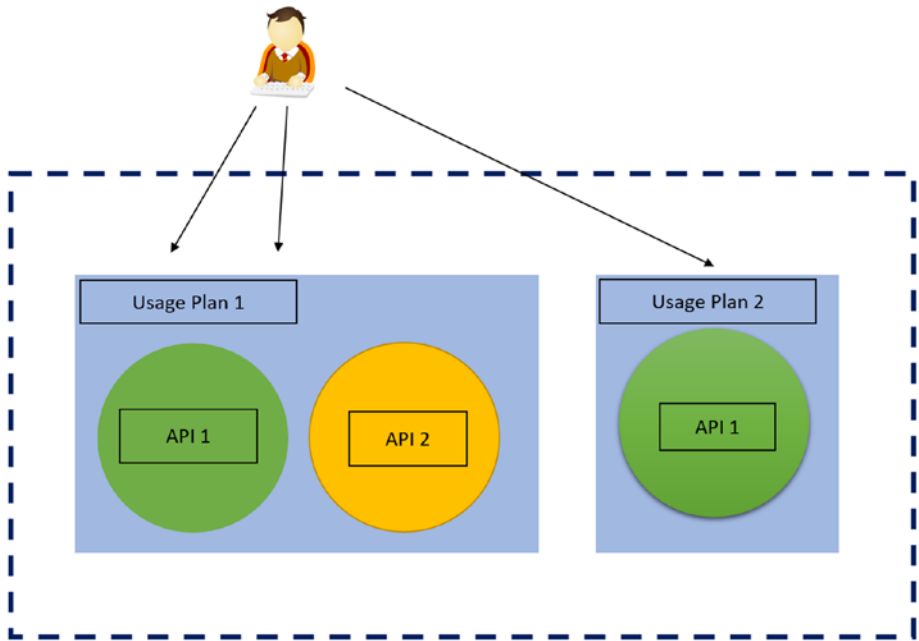


Figure 4-34. AWS API Gateway Structure

Summary

API gateways act as the middleware for backend services. They handle request and response orchestration, flow control, and security. Vendor-specific API gateways are often bundled as API management services, which include other elements of API architecture and value delivery elements such as developer experience, performance features, and monitoring features.

In the modern service-based architecture context, API gateways play a primary role in surviving separation and management. Internet-based API gateway services like Azure API Management and AWS API Gateway act as public-facing proxies for backend services.

CHAPTER 4 API GATEWAYS

This chapter provided information about API gateways, the different endpoint mapping scenarios, and the practical implementation of API gateways from Azure and AWS service offerings. We also introduced the fundamentals of using API gateways in mocking and other architectural patterns.

CHAPTER 5

API Security

APIs expose data and business operations, so they must be secure. As an API developer, you should protect APIs from unauthorized consumers, control the consumption rate, and govern the data. Strategic API design helps to achieve the protection and governance of business-sensitive data.

The scale of the cloud and the flexibility it offers yield many different challenges in securing APIs. Developers combine different security implementations at different levels of a system in an API implementation. This chapter covers common security measures implemented in Azure and AWS API layers. The details are covered under two major topics: request-based security implementations and authentication.

Request-based security implementations dictate policies and constraints on API consumption—mainly, who can consume and how much—while authentication dictates policies and constraints on authentication and authorization, mainly in terms of who the consumer is and what the consumer can access.

Request-Based Security

Request-based security implementations identify a consumer and apply constraints on their consumption, either by limiting the consumption rate or allowing or blocking the consumer. In this implementation, identification of the consumer is simple and does not include a complex authentication mechanism. In general, API keys are used to identify the consumer.

In the API economy especially in the direct selling model, the consumption rate is a fundamental parameter in defining different API SKUs. In rare cases, we can observe limits on API features as a selling model. Cloud API management services such as Azure API Management and AWS API Gateway have these settings built in, and we can configure them to implement request-based security rules.

Azure API Management

Azure API Management has configurable request-based security settings, the rules of which are applied using Azure API Management policies. If you're new to Azure API Management policies, I recommend reading chapter 4 before reading this section.

Subscriptions and Subscription Keys

In Azure API Management, in order to make a successful request to an API offered as an Azure API Management product (for more about products, see chapter 4), a consumer should have a valid subscription key associated with that product. API developers can request or automatically retrieve (depending on the setting) subscription keys for products in the developer portal.

A single subscription to a product provides two subscription keys—primary and secondary. Having two subscription keys helps manage the downtime during key rollover. Subscription keys are sent to the Azure API Management gateway in the request header `ocp-apim-subscription-key`. Usage is tracked at the subscription level regardless of which key is used.

In most cases, subscription keys are used to track usage, but Azure API Management allows developers to configure usage limits based on other parameters, like IP address or response codes.

Request Rate Limits

Navigate to the Azure API Management service and select the Azure API Management product MassRover Catalog we created in chapter 4. Then navigate to the policies section of the product and add the rate-limiting policies.

Product-level policies are applied to all the endpoints in an aggregated manner. If the rate is 10 requests per minute for a product and the product has two API endpoints, the consumer can make a maximum of 10 requests per minute as a combined request rate, but is not allowed to make 10 requests to each endpoint separately. We can apply endpoint-level policies to enforce such rules. In general, request-based policies are applied at the product level.

The following policy statement in the inbound section allows 2000 calls in 60 seconds for the API endpoints included in the specific product.

```
<rate-limit calls="2000" renewal-period="60" />
```

If the caller exceeds this limit, Azure API Management will respond with a 429 HTTP code (too many requests). The body will contain the message and the retry time period, and the caller must wait for that time period for the request to go through. The body containing the remaining time helps to implement smart retry logic rather than hitting the gateway in random intervals.

With the above policy in place, the rate limit is applied based on the subscription key; technically, the subscription key works as the increment counter key for the requests.

We can control the request rate using an arbitrary key value as well. The key can be any variable that can be accessed by Azure API

Management, and the key value can be any accessible value in the request context. The below example shows the request rate limit by the IP address.

```
<rate-limit-by-key calls="1000"
    renewal-period="90"
    increment-condition="@context.Response.
    StatusCode == 200)"
    counter-key="@context.Request.IpAddress"/>
```

The above policy allows 1000 requests, and returns 200 in 90 seconds for each IP address. So a caller with an IP address cannot make more than 1000 calls in 90 seconds, which returns 200.

Another example would be rejecting calls that cause the server to return 500. The below policy statement applies restrictions based on a header value; if a request with a particular header value causes the server to fail 10 times in 60 seconds, then the consumer will be blocked until the renewal period refreshes.

```
<rate-limit-by-key calls="10"
    renewal-period="60"
    increment-condition="@context.Response.
    StatusCode == 500)"
    counter-key="@context.Request.Headers.massrover_
    token"/>
```

Quota Limits

Quota limits are applied primarily to control request quotas. In practical scenarios, this is used for the monetization of APIs, where different SKUs can have different quotas. In order to apply quotas, we can use quota policies.

```
<quota calls="100000" bandwidth="80000" renewal-period="3600" />
```

In the above policy, we can apply a quota—with a renewal period of one hour—for a certain number of calls or a certain bandwidth in kilobytes. Whichever threshold is met first will trigger the condition.

The above policy tracks usage based on the subscription key. As with the rate-limiting policies, we can apply quota limits based on arbitrary keys.

```
<quota-by-key calls="10000" bandwidth="3000" renewal-
period="3600"
    counter-key="@context.Request.Headers.massrover_
token)"
    increment-condition="@context.Response.StatusCode
>= 200 && context.Response.StatusCode < 400)"
    increment-count="@context.Request.Method == "POST" ?
1:0)" />
```

This quota policy counter has the increment for the requests with the header value `massrover_token`. The request increment is triggered for the POST requests that have HTTP response 200 or above and below 400. The policy will block more when the requests number of requests satisfies above condition goes more than 10000 or the request bandwidth exceeds 3000KB in one hour—whichever condition is met first.

IP restrictions

Azure API Management has IP-based restriction policies that let us allow or block certain IPs.

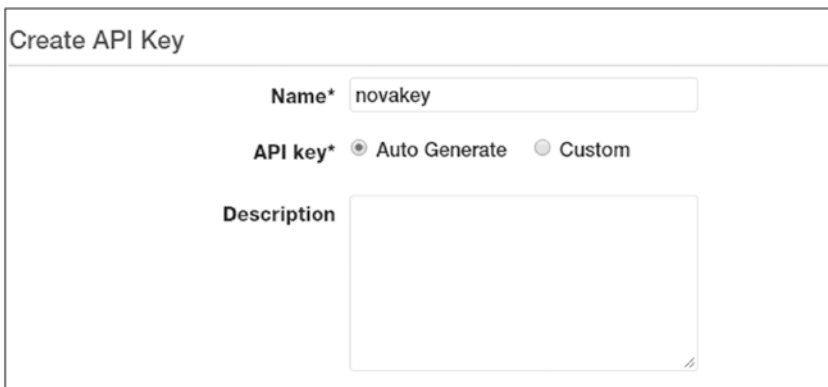
```
<ip-filter action="allow">
    <address-range from="10.1.1.2" to="10.1.1.16" />
</ip-filter>
```


AWS API Gateway

AWS API Gateway has configurable, request-based security settings, applied via API Usage Plans. Chapter 4 describes the fundamentals of AWS API Gateway and Usage Plans, so if you're new to AWS API Gateway, I recommend reading chapter 4 before reading this section.

API Keys

We can generate keys for the APIs under “API Keys.” These keys are either auto-generated by the AWS API Gateway service or custom provided. Figure 5-1 shows the screen on which we can generate API keys. API keys are associated with API usage plans and used for tracking requests.



The screenshot shows the 'Create API Key' form in the AWS API Gateway console. The form has a title 'Create API Key' at the top left. Below the title, there are three fields: 'Name*' with a text input containing 'novakey', 'API key*' with two radio buttons, 'Auto Generate' (which is selected) and 'Custom', and 'Description' with a large text area.

Figure 5-1. AWS API Gateway API Key Generation

We can customize how consumers send API keys to the AWS API Gateway. API keys are sent either in the HTTP header X-APIKEY-HEADER or through the configured custom authorizers. This configuration can be done under the Settings section of the selected API (Figure 5-2).

The screenshot shows the 'Settings' section for API Key Source. It includes a sub-header 'API Key Source' and a description: 'Choose the source of your API Keys from incoming requests. Configure deployments to receive API keys from the X-APIKEY_HEADER or from a Custom Authorizer'. At the bottom, there is a dropdown menu labeled 'API Key Source' with 'HEADER' selected.

Figure 5-2. Settings for the API Key Request Source

Rate Limits

In order to configure rate limits, we must associate the API keys with the usage plans. We set the rate limits in the usage plans, and requests are tracked by the API keys.

One usage plan can include many APIs, and all the APIs in a usage plan are constrained by the rate limit. In order to set the rate limits, navigate to the selected usage plan. You will see two different settings available, Throttling and Quota. Refer to Figure 5-3.

The screenshot shows the 'Throttling' and 'Quota' settings for an AWS API Gateway Usage Plan. Under 'Throttling', there is a checked checkbox for 'Enable throttling'. Below it, 'Rate*' is set to 1000 requests per second, and 'Burst*' is set to 400 requests. Under 'Quota', there is a checked checkbox for 'Enable quota'. Below it, the quota is set to 250000 requests per Month.

Figure 5-3. AWS API Gateway Usage Plan Settings

As you can see, throttling has been enabled, the rate is set to 1000 requests per second, and the burst is set to 400 requests. This means that this particular usage plan can handle 1000 requests per second without any throttling, when those 1000 requests are evenly distributed at one request per millisecond. The burst is the maximum number of requests the usage plan can handle when they arrive simultaneously. The following examples describe two different scenarios for the burst setting.

- If a consumer makes 500 calls in 500 milliseconds, in an evenly distributed manner, then the API will respond to the requests without throttling any of them. In the 501st millisecond, if the caller makes 405 requests, then the requests will be throttled, since the burst rate has been exceeded (even though the rate limit has not been exceeded).
- If in the first millisecond a consumer makes 400 requests and then the caller tries to make 1 request per millisecond for the remaining 999 milliseconds, the requests will be throttled starting at the 602nd millisecond, as this is the moment the call rate will exceed the 1000-requests-per-second limit.

Quota Limits

Usage plans have a quota limit setting as well, available right under the throttling section (see Figure 5-3). We can set the quota by the number of request per day, week, or month.

Authentication & Authorization

In simple terms, authentication identifies a caller and authorization provides information on whether the caller has access to the secured resources or not. An API can implement authentication and authorization

in many different ways; in general, an API expects a value in the request that includes required security information. This value can contain the security information within itself, or in a format and scheme API can understand, or it would be a reference to the real security information the API can retrieve.

There are many standards and protocols available in API security, and all of them boil down to the above mentioned criteria. Issuing a specific value; securing it from unauthorized access and tampering; cycling the value; validating the value; and other management tasks related to the creation, validation, maintenance, and destruction of the value can take many different forms depending on the technology, vendor specification, open standards, and compliancy.

There are myriad such standards and protocols on the market. Some are very common and widely used, and some are highly specific to a particular vendor or technology. Details of such standards and protocols are beyond the scope of this book, but this section covers the most common cloud-based API authentication and authorization scenarios, especially targeting Azure and AWS.

API Security Design

One size does not fit all, so there is no one security design that suits all business cases, but as stated earlier, it's helpful to consider some key implementation patterns.

APIs are stateless, meaning they do not maintain state information between requests, so each request should contain all required elements for fulfillment. The security information should also be part of each request, assuming the API needs this information in order to fulfill the request. The information can be in the request body, query string, or request header.

API Keys

API keys are simple identifiers of a caller. As a developer, you register yourself with the API provider or prove your identity to an API provider to obtain a key to access the API. For example, in Google Maps' API, developers prove their identity via Google login before receiving the key.

This key is a random string issued by the API provider—each request should contain the key. The API provider identifies the caller and the access limits. Generally, this implementation is used where there's no specific requirement for identifying the user, but the usage needs to be monitored. The direct-selling API model utilizes API key implementation.

Public APIs often follow this pattern; the full key management is controlled by the API or handled at the gateway service. Recall the Azure API Management subscription keys and AWS API Gateway API keys. The issuing and management of the keys is handled at the gateway service, and the backend service is unaware of the key.

In the direct-selling API model, most keys do not expire, but may be invalidated by the API due to too many requests, suspected request-based attacks, or missed payments. However, API consumers are not allowed to recycle their keys for obvious security reasons (see Figure 5-4).

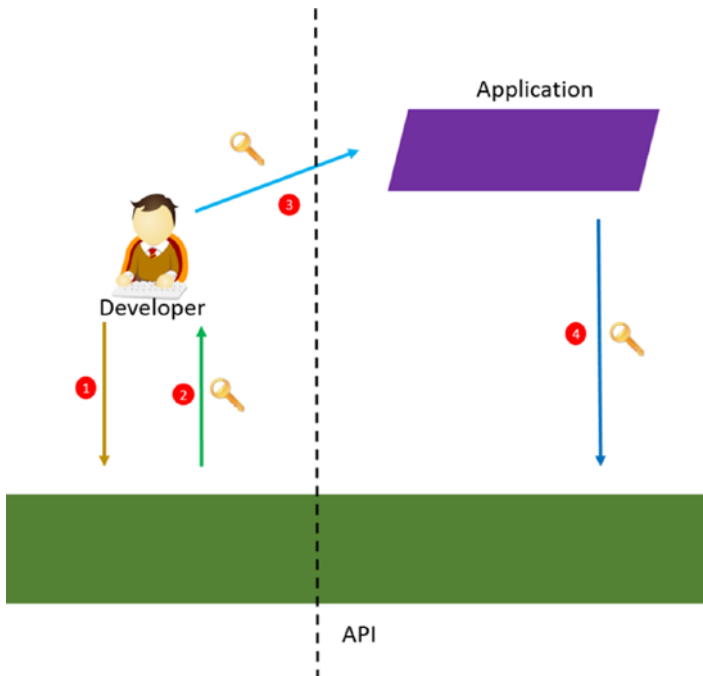


Figure 5-4. *API Key-Based Security*

1. The developer requests an API key from the API/API provider. This request may have preconditions, like certain validations or payments.
2. When the preconditions are satisfied, the developer receives the key from the API/API provider.
3. The developer stores this key in the application.
4. The application sends the key to the API in the requests.

The developer can use the same key across different applications; the API is not aware of it. The only information the API cares about is whether the key is valid in terms of usage conditions and limitations.

As discussed, in Azure and AWS, the gateway service handles key management. This offloads the key management logic from the API implementation, but if the API itself is open and allows direct access, then the callers can successfully make direct requests to the API. Figure 5-5 depicts API key implementation at the API gateway.

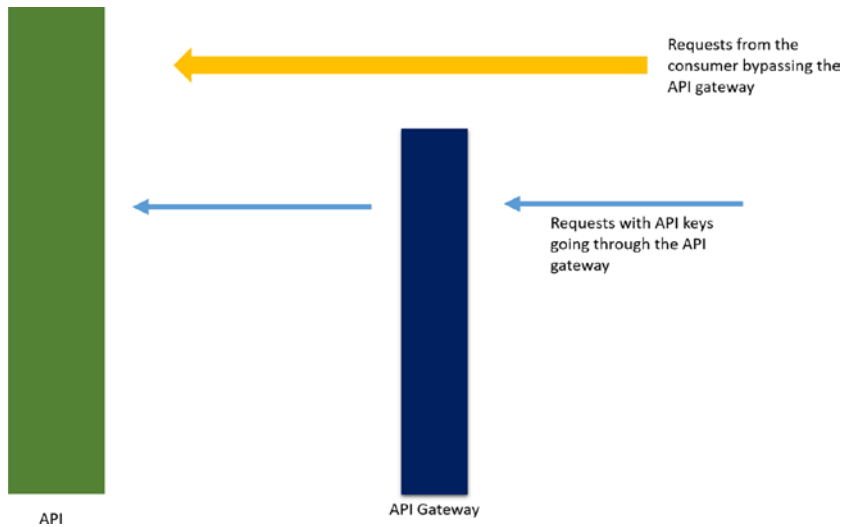


Figure 5-5. API Gateway-Level API Keys

As you can see, the API gateway has the full context of the key, and the backend service is not aware of the key implementation. Key implementation can be brought to the backend service and used in the business logic, but this kind of a design introduces lots of key management logic to be written in the API itself, thus preventing the benefits of API gateway key management.

On the other hand, if the backend API is open—and the consumer is aware of the backend service endpoints, structure, and other details—then requests can be made directly to the backend service by passing the API gateway. This is a common flaw in cloud-based API implementations.

Deploying the API in open Internet services like AWS Beanstalk or Azure API Apps and assuming that API gateway services (Azure API Management or AWS API Gateway) will take care of the security just because the API keys are configured is a major security loophole.

The real implementations should have security implemented in the backend service, and public access to the backend service should be locked down. Preventing public access to the backend service can be achieved with infrastructure configuration and software implementation. The above problem can most easily be rectified by introducing some context between the backend service and the API gateway. Figure 5-6 depicts this.

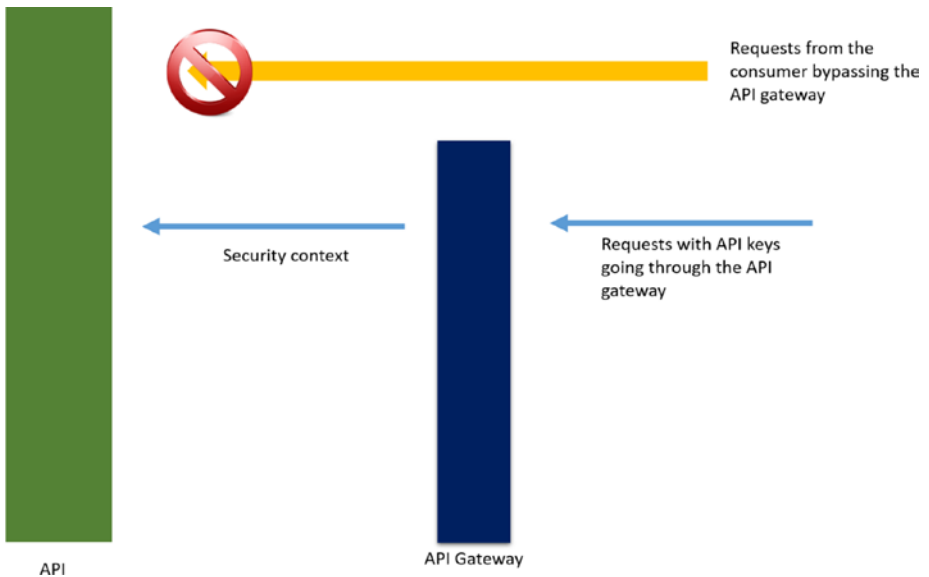


Figure 5-6. *Trust Between API Gateway and Backend Services*

As indicated in the previous chapters, API gateways can manipulate a request by injecting the agreed security context into the requests they make to the backend service.

The backend service looks for the right security context in any request it receives, and rejects requests that do not satisfy these expectations. This security context can take different forms; the following are the most common:

- A shared key (static string) between the API gateway and the backend service. The backend service expects this value as a symmetric key identifier of the API gateway service.
- Certificate authentication between the API gateway and the backend service, the asymmetric implementation as backend service trusts specific certificate authentication.
- The backend service itself, or any other trusted authentication provider, issues tokens for the communication between the API gateway and the backend service.
- A mix of the above, with other network-based security implementations, such as firewalls and IP settings, allowing only the calls from the API gateway.

API keys at the gateway do not offer comprehensive security, so combining them with one of the above methods secures the backend service while offering the flexibility of API key management from the available API gateway service.

OpenID and OAuth

If you're a web developer or work in API development, it is highly unlikely you haven't heard the terms OpenID and OAuth. Though they are very common, there's a lot of confusion among developers and users between these two standards. This section explains the difference between them, as this is a required piece of knowledge to understand the upcoming sections.

Note OpenID is for authentication and OAuth is for authorization.

Yes, it is simple as that, but it is tricky to differentiate them in real-world implementations because authorization includes authentication, meaning that OAuth depends on OpenID. But OAuth as a standard does not specify the particular authentication should happen via OpenID for it to function.

In terms of technical implementation, there are some similarities. For example, both OpenID and OAuth depend on browser redirects.

OpenID identifies the user. For example, imagine you're using Facebook login to access a website (let's call it example.com). When you're navigating to example.com and you see an option to "Log in with Facebook," you click the button to avoid filling out a long registration form.

When you do this, if you're already logged in to Facebook, you do not need to enter your Facebook credentials again. This is OpenID; it facilitates user authentication from the same provider to multiple entities. The fundamental principle of single sign-on is based on OpenID.

Sometimes Facebook will respond with a prompt, saying that example.com is trying to access a user's email, photos, or other information. This is OAuth. In this case, example.com needs to access some information from Facebook, and requests permission by prompting you, as you're allowing example.com to read your data stored in Facebook.

By giving consent, you're allowing example.com to read your data. Technically, Facebook gives some access to example.com to make requests to its resources. Once you give permission, future communication can take place between example.com and Facebook based on the allowed grants. This is also OAuth.

In the coming sections, we'll see how OpenID and OAuth are used in securing APIs.

Securing APIs with Azure Active Directory V2

If you're in the Azure application development space, securing services with Azure Active Directory (AAD) is a common requirement. AAD has a new version of authentication API, known as v2, which has some significant changes as compared to the previous version. In this section, we will see how to secure APIs using AAD v2. You can read more about the difference between v1 and v2 here: <https://docs.microsoft.com/en-us/azure/active-directory/develop/active-directory-v2-compare>.

Note AAD offers comprehensive cloud-based identity management, catering to various complex authentication flows and protocols. These are beyond the scope of this book. For deeper insights into AAD, refer to my book *Practical Azure Application Development*. <https://www.amazon.com/dp/1484228162/>

In order to secure an API using any identity provider, the client should first obtain a valid identity from the identity provider. Next, the obtained identity is sent to the API in the request, and the API validates the identity and serves the request. In order to validate the identity, the API should know the identity provider and the mechanism used to validate the identity.

Now, let's look at the setup for securing an API using AAD. There are two major steps involved:

1. Facilitate the client to obtain the token (the identity)
2. Facilitate the API to validate the token

First, create the setup the client can request token from the AAD v2. This is done via the AAD v2 OAuth authorize endpoint. The client must make a request to this authorize endpoint, submit their credentials, and obtain the OpenID. Below is a sample request from the client.

```
https://login.microsoftonline.com/common/oauth2/v2.0/
authorize?client_id=[client id]&response_type=id_
token&redirect_uri=[redirect uri]&scope=user.read openid
profile&nonce=3c9d2ab9-2d3b-4
```

As you can see, in order to make this request, the client requires, at minimum, a registered client ID and a configured redirect URI, where AAD will redirect the requested response type.

We need to register a client application in AAD and set that information. Visit <https://apps.dev.microsoft.com/> (which is the endpoint to register AAD v2 applications) and navigate to the new Application Registration Portal. Log in with your AAD credentials or Microsoft account and create a new AAD v2 application, as shown in Figure 5-7.

MassRover Dev Portal Registration

[Click here for help integrating your application with Microsoft.](#)

Properties

Name

MassRover Dev Portal

Application Id

76d88779-d888-401f-8565-231aee385b14

Application Secrets

[Generate New Password](#) [Generate New Key Pair](#) [Upload Public Key](#)

Platforms

[Add Platform](#)

Web [Delete](#)

Allow Implicit Flow

Redirect URLs [Add URL](#)

https://localhost:8080

Logout URL [Add URL](#)

e.g. https://myapp.com/end-session

Figure 5-7. AAD V2 Application Registration

Specify the application platform as web and set the redirect URL (in this case, localhost is set, so we can test this without having a real application deployed). The client should specify the client ID and the configured reply URL in the request, as shown below.

```
https://login.microsoftonline.com/common/oauth2/v2.0/authorize?client_id=76d88779-d888-401f-8565-231aee385b14&response_type=id_token&redirect_uri=https://localhost:8080&scope=openid &nonce=3c9d2ab9-2d3b-4
```

The client ID and the redirect URI should match the values configured in the AAD v2 application. Since the requirement is obtaining the identity, OpenID is specified in the scope.

You can test this using the above URL and authenticate with your organizational or Microsoft account and retrieving the id_token (response type). The token will be delivered to the redirect URL (<https://localhost:8080>).

The above application is registered under the author's AAD account and redirects the token to localhost, so you can safely test it by simply pasting it in your browser.

The above request URL accepts any valid AAD authentication because the authorization URL points to the common endpoint. This applies to both Microsoft accounts and organizational accounts. If you want to accept only the organizational authentication, replace “common” with “organizations” as below.

```
https://login.microsoftonline.com/organizations/oauth2/v2.0/authorize?rest-of-the-url
```

If you're designing a single-tenant application, which expects authentication from one AAD tenant, you can specify the tenant ID in the request URL.

```
https://login.microsoftonline.com/[tenant id]/oauth2/v2.0/authorize?rest-of-the-url
```

Consumers can make one of the above requests, authenticate themselves with AAD credentials, and obtain the OpenID information. In the above described context, OpenID information is in the `id_token` issued by the AAD v2 endpoint to the instructed redirect URL.

When you test the below URL:

```
https://login.microsoftonline.com/common/oauth2/v2.0/authorize?client_id=76d88779-d888-401f-8565-231aee385b14&response_type=id_token&redirect_uri=https://localhost:8080&scope=openid &nonce=3c9d2ab9-2d3b-4
```

When you log in for the first time, you will see a consent screen, as shown in Figure 5-8. This is the request from AAD v2 to get consent from the user and issue the OpenID information to the requesting client. Subsequent login attempts will not ask for this consent.

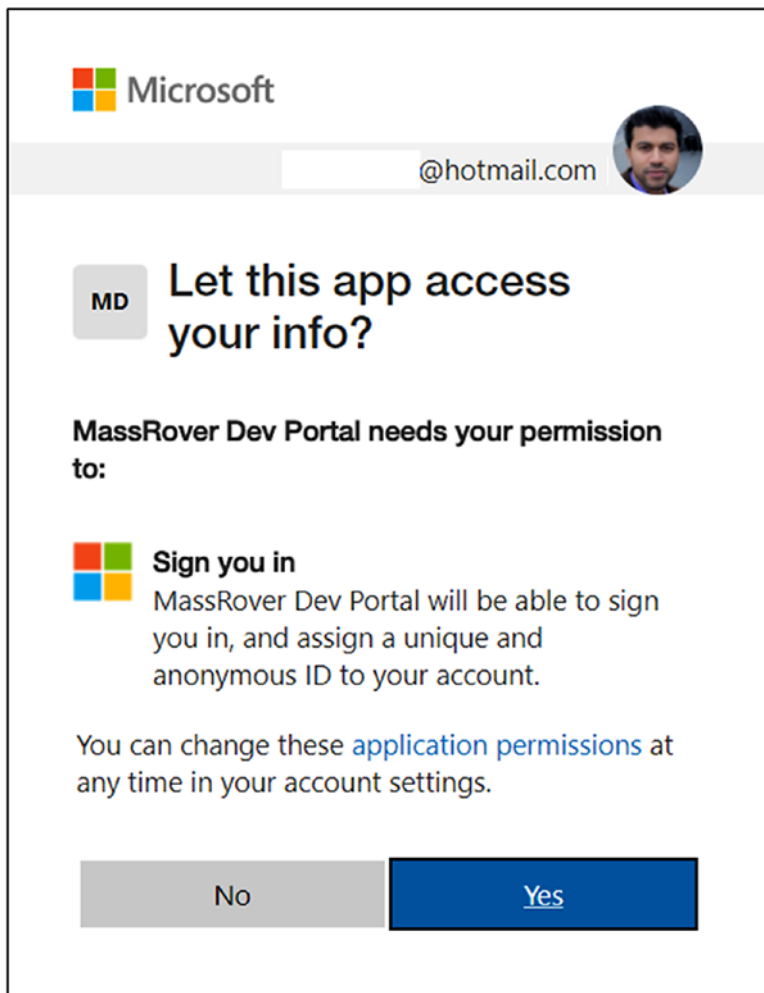


Figure 5-8. AAD V2 Consent Screen

After successfully logging in and obtaining consent, AAD v2 will redirect to the specified URL with the `id_token`, like below.

```
https://localhost:8080/#id_token=eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiIsImtpZCI6IjFjFMVE16YWtpaGlSbGFfOHoyQkVhV01xbyJ9.eyJ2ZXIiOiIyLjAiLCJpc3MiOiJodHRwczovL2xvZ2luLm1pY3Jvc29mdG9ubGluZS5jb20vOTE4ODAOmGQtNmM2Ny00YzViLWIxMTItMzZhMzA0YjY2ZGFkL3YyLjAiLCJz
```



```

aio: "DTH!k!p37Mjhcp0*ZAzi0JovtolxD8Ud99GtbkdZHP*1TUSMpm
JuChu!UmmXs1J*bAdMe7lKScbXn9HY1MsKT5LA0C9jrAv0ZLMWwz2eugJfRlgQ0
cYeigOwyKJbjEetMw$$"
}.

```

After obtaining the token, consumers will send it to the API, which should be able to validate the token and retrieve the information to be used in the business logic.

In an ASP.NET Core application, you can use the following code to perform the token validation and query claims from the token.

```

private async Task<System.IdentityModel.Tokens.Jwt.
JwtSecurityToken>
    ValidateAADIdTokenAsync(string idToken)
{
    var stsDiscoveryEndpoint = "https://login.microsoftonline.
com/common/v2.0/.well-known/openid-configuration";

    var configRetriever = new Microsoft.IdentityModel.
Protocols.OpenIdConnect
        .OpenIdConnectConfigurationRetriever();

    var configManager = new Microsoft.IdentityModel.Protocols
        .ConfigurationManager<OpenIdConnect
        Configuration>
        (stsDiscoveryEndpoint, configRetriever);

    var config = await configManager.GetConfigurationAsync();

    var tokenValidationParameters = new Microsoft.
IdentityModel.Tokens.TokenValidationParameters
    {
        IssuerSigningKeys = config.SigningKeys,
    };
}

```

```

var tokenHandler = new JwtSecurityTokenHandler();

tokenHandler.ValidateToken(idToken,
tokenValidationParameters, out var validatedToken);

return validatedToken as JwtSecurityToken;
}

```

The method receives the token as a string and obtains the token signing keys from the provider (AAD v2) via the secure token service endpoint <https://login.microsoftonline.com/common/v2.0/.well-known/openid-configuration>.

This URL will be used by `OpenIdConfiguration` in order to obtain the signing information. You can change this URL by replacing “common” with “organizations” or “tenant id” as per the request URL pattern discussed earlier.

The obtained signing keys and other validation settings are used to create the `TokenValidationParameters`. Eventually the token is validated, and the output will be converted to a `JwtSecurityToken`, which contains the claims in the token for programmatic access.

The above code snippet “`TokenValidationParameters`” has a minimum validation configuration, meaning it checks whether the token is issued by the correct trusted entity by validating the signing keys. In the case of a real-world implementation, more complex validation rules would be used to perform the validation, along with signing rules such as validating the issuer, audience, and expiration.

Note Tokens and claims-based authentication go hand in hand in implementations. There are many types of tokens, each with a different scope and containing varying information based on the authentication request and authentication flow. But generally speaking, a token is a signed string that contains pieces of information (claims). JWT is a standard token format, widely used by many providers.

In a practical implementation, the above token validation code snippet would be a filter in the ASP.NET Core. Each request header from the client would contain the `id_token` and be validated via the filter. Certain claims will be retrieved from the `id_token` in order to execute the business logic.

Public identity providers like Google and Facebook can be used to authenticate users. Most of the claims in those OpenID information cannot be related to the custom business logic of a typical enterprise line of business application, because the business logic deals mostly with application-specific roles and permissions, which are outside the context of the mentioned identity providers.

To solve this, we can issue the custom token from the application after validating the identity from the external providers. In this mode, we depend on external identity providers for authentication and issue custom tokens for the authorization. The next section explains the process of issuing custom tokens in more detail.

Issuing Custom JWT Tokens

As I explained, there are certain cases where we need to create and issue our own JWT tokens. For example, say you're developing a SaaS application that relies on several identity providers. These identity providers help users authenticate to the application with less friction, and OpenID plays a key role in establishing a single sign-on experience, but once the user is authenticated, the application should be aware of the authorization information of the user, including roles and permissions.

In simple terms, Google or Facebook cannot store a user's details, even if the user is an admin of your custom application. It is the responsibility of the backend service/application to manage the authorization.

In the previous section, I explained how to obtain a JWT token with OpenID claims from AAD and validate it. This would help secure the API using AAD from an authentication point of view, but once the user has logged in, API has to determine the authorization of the user in order to decide what the user can do inside the application.

The below code snippet shows how to issue a custom JWT token using a symmetric signing key.

```
private string IssueJwtToken(JwtSecurityToken aadToken)
{
    var msKey = GetTokenSignKey();

    var msSigningCredentials = new Microsoft.IdentityModel.
        Tokens.SigningCredentials
        (msKey, SecurityAlgorithms.HmacSha256Signature);

    var claimsIdentity = new ClaimsIdentity(new List<Claim>()
    {
        new Claim(ClaimTypes.NameIdentifier,
            "thuru@massrover.com"),
        new Claim(ClaimTypes.Role, "admin"),
    }, "MassRover.Authentication");

    var msSecurityTokenDescriptor = new Microsoft.
        IdentityModel.Tokens
        .SecurityTokenDescriptor()
    {
        Audience = "massrover.client",
        Issuer = "massrover.authservice",
        Subject = claimsIdentity,
        Expires = DateTime.UtcNow.AddHours(8),
        SigningCredentials = msSigningCredentials
    };

    var tokenHandler = new JwtSecurityTokenHandler();

    var plainToken = tokenHandler.CreateToken(msSecurityToken
        Descriptor);
}
```

```

    var signedAndEncodedToken = tokenHandler.
    WriteToken(plainToken);

    return signedAndEncodedToken;
}

```

The above JWT issuing code obtains the signing key from this private method.

```

private Microsoft.IdentityModel.Tokens.SymmetricSecurityKey
GetTokenSignKey()
{
    var plainTextSecurityKey = "massrover secret key";

    var msKey = new Microsoft.IdentityModel.Tokens.
    SymmetricSecurityKey
        (Encoding.UTF32.GetBytes(plainText
        SecurityKey));

    return msKey;
}

```

The token-issuing code uses `ClaimsIdentity` to include application-specific claims in the token subject. For the next step, `Microsoft.IdentityModel.Tokens.SecurityTokenDescriptor` is used to construct a full JWT token along with the custom claims in the subject.

This object is used to create the JWT token using the `CreateToken` method from `JwtSecurityTokenHandler`, and the token is returned as a string.

When we issue the custom token, the API should be able to validate the token as well, when it is returned from the callers in the requests.

The below code snippet shows the token validation code.

```

public bool ValidateMassRoverToken(string token)
{
    var tokenValidationParameters = new Microsoft.
    IdentityModel.Tokens

```

```

        .TokenValidationParameters()
    {
        ValidAudiences = new string[]
        {
            "massrover.client",
        },
        ValidIssuers = new string[]
        {
            "massrover.authservice",
        },
        ValidateLifetime = true,
        IssuerSigningKey = GetSignTokenSignKey()
    };

    var tokenHandler = new JwtSecurityTokenHandler();

    tokenHandler.ValidateToken(token,
        tokenValidationParameters, out var validatedToken);

    return true;
}

```

The code uses the `TokenValidationParameters` object, which includes token validation logic along with signing keys (retrieved from the same private method used to issue the token). The `ValidateToken` method from `JwtSecurityTokenHandler` validates the token using the constructed `TokenValidationParameters` object.

Note that the above code snippets are not production ready, and token flow in the production application requires software implementation along with TLS support. The above pieces of code explain the fundamentals of issuing and validating JWT tokens in ASP.NET Core.

Note There are a number of libraries and frameworks available that provide secure and comprehensive implementations of token management in business applications. These libraries cover many different authentication scenarios. In real-life implementations, you would use one of those libraries or frameworks to handle the authentication token management, rather than writing the entire code by yourself. Identity Server is a popular authentication framework in the .NET world.

Once the JWT token flow is in place, consumers will send the token in each request to the API. In typical scenarios, tokens are sent in the Authorization header. Backend services receive the token from the HTTP request and validate and obtain information from them to coordinate the business logic requirements. In some cases, tokens do not contain any information other than an identifier, but backend services know to get the required information using this identifier. These kinds of tokens are known as reference tokens.

In the next section, we will discuss how to use Azure API Management to pre-authenticate requests that contain a JWT token.

Pre-Authentication in Azure API Management

In many examples above, we saw that Azure API Management can process request and response information. Pre-authentication at the gateway is a good practice and prevents hitting the backend service for all requests. But it is strongly recommended to validate the token in the backend service and obtain the information stored there.

We will use the Validate JWT policy for this pre-authentication step. The below snippet shows a basic JWT validation policy implementation.

```

<validate-jwt
  header-name="Authorization" failed-validation-httpcode="401 "
  failed-validation-error-message="Unauthorized"
  require-expiration-time="true"
  require-scheme="scheme"
  require-signed-tokens="true">
  <audiences>
    <audience>76d88779-d888-401f-8565-231aee385b14</audience>
  </audiences>
  <required-claims>
    <claim name="massrover-role" match="any">
      <value>admin</value>
      <value>user</value>
    </claim>
  </required-claims>
  <openid-config url=" https://login.microsoftonline.com/
common/.well-known/openid-configuration" />
</validate-jwt>

```

In the above snippet, I have provided the openid-config URL and validation comparison of basic claims. The code also checks a custom claim called massrover-role and makes sure its presence in the JWT token and the value it can take either admin or user.

Also, note that require-expiration-time is set to true. In order to pass this validation, the incoming JWT token should contain the exp claim. If the exp claim is not present, validation will fail.

The Azure API Management JWT validation policy supports both HS256 and RS256 signing algorithms. For HS256, they should be provided in the policy itself as a base64 encoded string, like below.

```
<issuer-signing-keys>  
  <key>base64 encoded key</key>  
</issuer-signing-keys>
```

For RS256, the key must be provided via an OpenID configuration endpoint, as shown in the sample policy above.

You can add more validation rules to this policy, including validating custom claims, like the massover-role claim.

Note There are concerns about bringing more business context to API gateways, referred to as overambitious API gateways. Read more about this here: <https://www.thoughtworks.com/radar/platforms/overambitious-api-gateways>.

In the above case, validating custom claims (massover-role) and controlling access may be considered bad practice, since this brings the business logic to the gateway. At the same time, as a tool, API Gateway offers many features like this. You can read more about highly customizable security at API Gateway from this post: <https://thuru.net/2018/02/28/overambitious-api-gateways-security-at-api-gateway-with-azure-api-management/>

Authorizers in AWS API Gateway

Authorizers are set up in AWS API Gateway to authenticate incoming requests. AWS API Gateway handles this with a Lambda function (more on serverless in chapter 6) or AWS Cognito. Lambda is the serverless platform of AWS, and Cognito is the AWS-based access control service.

In order to set up an authorizer, we should first create an authorizer under the selected API. Navigate to the API, select Authorizers, and click Create New Authorizer. You will see the panel as shown in Figure 5-9.

In the panel, provide a name for the authorizer and select the type as Lambda, which we can write the code to do the authentication validation logic. As Azure API Management this is done via policies in AWS API Gateway, this done via an external serverless function with custom code.

Next, select the Lambda function, which has the authorizer logic. In order to do that, you should have an existing Lambda function and implementation, or provide a name (entered in the textbox) of the function, which will be deployed later. Refer to chapter 6 for instructions on how to create and deploy Lambda functions.

Select Token for the Lambda Event Payload. This ensures the token will be present in the specified header. Select Request if the token is present in the event payload request body/header/query string with the specified value.

Create Authorizer

Name *
MassRoverJWTAuthorizer

Type * ⓘ
 Lambda Cognito

Lambda Function * ⓘ
us-east-1 - MassRoverTokenAuthorize

Lambda Invoke Role ⓘ

Lambda Event Payload * ⓘ
 Token Request

Token Source* ⓘ **Token Validation ⓘ**
Authorization

Authorization Caching ⓘ
 Enabled **TTL (seconds)**
300

Create **Cancel**

Figure 5-9. Create AWS API Gateway Authorizer

Specify the corresponding value that has the token in the Token Source. As above, the Lambda context will look for the token in the Authorization header.

The rule to look for the token in headers comes from Lambda Event Payload type, and the value comes from Token Source. The same will apply if you select Request as the Lambda Event Payload; the lookup will happen in a range of places, including headers, query string parameters, stage variables, and context parameters for the specified Token Source key.

If you'd like, you can specify a regex validation in the Token Validation section, which prevents hitting the Lambda for tokens that are not in the right format.

Authorization caching indicates whether to cache the authorization policy document or not. Authorization policy documents are generated by the authorizer for the specified token. AWS API Gateway authorizers return an authorization policy to the AWS API Gateway with access and deny rules specified.

You can read more about these policies here: <https://docs.aws.amazon.com/AmazonS3/latest/dev/access-policy-language-overview.html>. These policies dictate whether AWS API Gateway has access to specific AWS resources.

Producing the policy document is the logic in the authorizer Lambda function. As mentioned above, these policy documents control access to AWS resources, and in terms of the application business logic, the authorizer Lambda should have the token validation logic (something similar to the JWT validation logic shown in the Issuing Custom JWT Tokens section above).

After validation, Lambda will construct the policy document and return it to AWS API Gateway, which will cache the document for the determined period if caching is enabled. Figure 5-10 illustrates this flow.

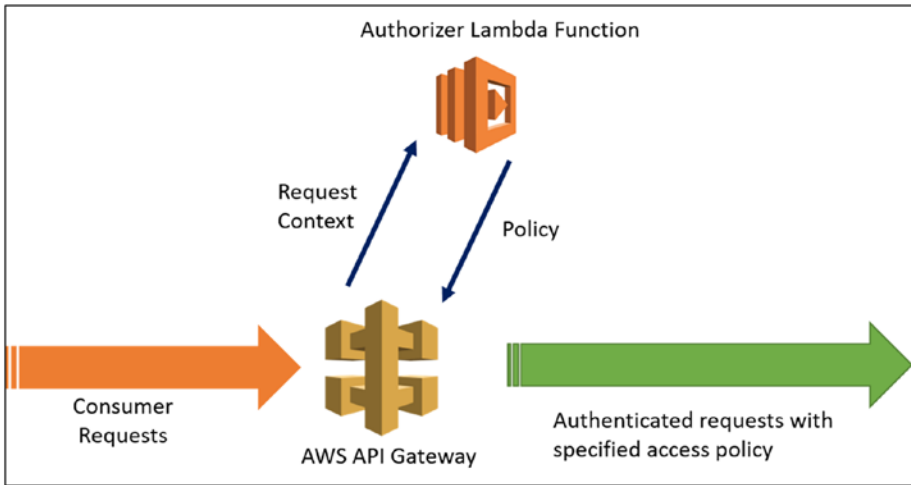


Figure 5-10. *AWS API Gateway Authorizer Authentication Flow*

After creating the Lambda authorizer, we will implement the logic, which validates the token and policy document generation. The following code shows how to do this:

```
public class Function
{
    public Policy FunctionHandler(APIGatewayCustomAuthorizer
    Request authRequest,
                                ILambdaContext context)
    {
        var token = authRequest.AuthorizationToken;
        Policy policy;

        if (ValidateToken(token))
        {
            var statement = new Statement(Statement.
            StatementEffect.Allow);
        }
    }
}
```

```

        var policyStatements = new List<Statement> {
            statement };

        policy = new Policy("TokenValidationPassed",
            policyStatements);
    }
    else
    {
        var statement = new Statement(Statement.
            StatementEffect.Deny);

        var policyStatements = new List<Statement> {
            statement };

        policy = new Policy("TokenValidationFailed",
            policyStatements);
    }

    return policy;
}

private bool ValidateToken(string token)
{
    // JWT token validation here
    return true;
}
}

```

Here, the policy document simply indicates whether access is allowed or not, and does not contain any specific access policy to specific AWS resources. We can create custom policy implementations, but this implementation is sufficient enough to flow the requests to the backend service.

Summary

API security is a large field, and it is one of the security topics that is gaining increasing attention and demand. This chapter covers a few common areas of interest in modern application development. Note that this chapter provides neither a comprehensive guide to API security nor the full flow implementations of it in the selected platforms. It does provide sufficient enough fundamental details and implementation information to get started and discover more, based on the business requirement at hand.

As stated, API security involves different layers of implementation, and this chapter covers them in two main sections: request-based security and authentication- and authorization-based security. Request-based security includes call limits, rate limits, IP restrictions, and API key implementations. Authentication- and authorization-based security includes API key implementations, OAuth implementations from available providers on Azure and AWS, and implementations of custom JWT management without any third-party libraries.

CHAPTER 6

Serverless APIs

Serverless is an ephemeral computational model; serverless endpoints are triggered based on a single event. These events spawn computation and execute the defined logic. Neither the caller nor the developer is aware of the computation infrastructure that emerges to serve the event. It is totally hidden and makes the presence of the computing infrastructure highly unnoticeable. The same goes for development and deployment, since there are no explicit server definitions and infrastructure code packages are given to the serverless platforms and executed. This characteristic, and the short-lived nature of the computation, have yielded the term “serverless” in the industry.

In this chapter, we will focus on serverless technology and economics, available serverless platforms in Azure and AWS, and how to use them in developing APIs.

Serverless Computing

Serverless is a highly trending topic in developer circles. Serverless offerings come in two major types: Functions as a Service (FaaS) and Backend as a Service (BaaS).

The FaaS model allows developers to write custom logic, create code packages, and execute them in serverless context. These functions require an event trigger to be executed. Major serverless platforms support different languages and runtimes in the FaaS model. Azure Functions and AWS Lambda are the FaaS offerings of Azure and AWS, respectively.

The BaaS model allows developers to easily create integrations and workflows that trigger the usage by connecting available services from different providers. Services like authentication, search, data transformation, caching, integration, payments, accounting, and database available as services allow developers to reuse them, thus reducing costs of application design and development. The integrations between different services and coordination of the logic flow allow us to build powerful applications quickly. Logic Apps and Simple Workflow Service (SWF) are the BaaS offerings from Azure and AWS respectively, though SWF is mostly used in internal resource orchestration scenarios.

AWS started offering the serverless model with Lambda, and soon other cloud providers jumped into the market. Now, almost all the leading public cloud vendors provide a serverless platform. Apart from the cloud offerings, there are numerous on-premise serverless platforms available.

The promise of the serverless economy is exemplified by two major benefits. One is the development time. The BaaS model has embraced third-party services and reduced development time dramatically, allowing developers to focus on delivering value. Also, serverless hides the majority of the deployment and management of the application. Second, the cost of running an application in serverless is much cheaper, and though this partly depends on the nature of the application and how it is used, the initial offerings of serverless from major cloud vendors look very promising in terms of cost. But serverless may not always be cheap compared to PaaS and IaaS offerings.

From a technical angle, serverless offers the flexibility to develop fast and deliver fast, as it removes the overhead of resource provisioning and embraces third-party services as much as possible. Serverless is a good candidate to be plugged into any existing system because it works based on events. An event can be initiated by any existing system, with less wire up and serverless logic can pick them and execute the logic. This structure has been embraced by developers, and serverless is being used with various legacy systems to develop new features.

Serverless APIs are API functions backed by serverless platforms. Most serverless platforms can receive HTTP events as triggers, making them readily available to be used as APIs (Azure Functions is a good example of this). In some cases, other API technologies are used to integrate serverless execution with the HTTP pipeline. For example, AWS API Gateway is used with proxy integration with Lambda to serve the API backend using the serverless model.

Serverless APIs in Azure

Azure has many serverless offerings, like Azure Functions, Logic Apps, Azure Storage, Cloud messaging (Event Grid), Azure bot Services two major serverless offerings, Azure Functions and Logic Apps but as per current trend and market demand, we will be exploring Azure Functions and Logic Apps. Azure Functions is a FaaS offering—developers can write custom logic based on available triggers and execute the logic. Logic Apps is a BaaS offering where different actions and workflow conditions are available to execute a logic based on an event trigger. Both Azure Functions and Logic Apps support HTTP events, meaning both can act as standalone APIs.

Azure Functions

Azure Functions is an offering under the Azure App Service. Let's look at how to get started with Azure Functions and create HTTP triggers. We can create Azure Functions in the portal using the Function Apps.

Creating an Azure Function App

Navigate to the Azure portal, search for Function Apps, and create a Function App service. Figure 6-1 shows the Function App creation blade.

The screenshot shows the Azure Function App Creation Blade with the following settings:

- App name:** massoverfunc (with a green checkmark and .azurewebsites.net domain)
- Subscription:** Visual Studio Enterprise with MSDN
- Resource Group:** Create new, Use existing; dropdown menu shows API
- OS:** Windows (selected), Linux (Preview)
- Hosting Plan:** Consumption Plan
- Location:** Central US
- Storage:** Create new, Use existing; dropdown menu shows massoverfunc1
- Application Insights:** On, Off

Figure 6-1. Azure Function App Creation Blade

We'll look at two settings here—the hosting plan and the storage. The hosting plan has two options—the Consumption Plan and the App Service Plan. We also must specify a general-purpose storage account for the Function App.

- **Consumption Plan:** Azure Function host instances are dynamically provisioned and deallocated based on event triggers. This has auto scaling and theoretical limit to unlimited concurrent executions, since the

underlying infrastructure is fully dynamic. This plan is priced by usage. The maximum duration for a function execution is 10 minutes, and the default setting is 5 minutes.

- *App Service Plan:* The Azure Function host runs on the allocated App Service Plan SKU. The Function host has dedicated allocated resources, and scaling can be set up with the auto-scale rules of the App Service Plan. Functions can run for more than 10 minutes. This is a good option to consider for cases when events are almost continuously triggered or function execution time requires more than 10 minutes or the VNET/VPN setup.

Azure Function requires a general-purpose storage account, used to manage event triggers and logs. When using the Consumption Plan, the code and configurations are also stored in the selected account.

Create the Azure Function App and open it. Under the Function App (massroverfunc), you will see three different options: Functions, Proxies, and Slots. One Function App can have many functions; each function can have its own event trigger. This book focuses on HTTP event triggers. Click “Functions,” then click “New Function.” (Figure 6-2).

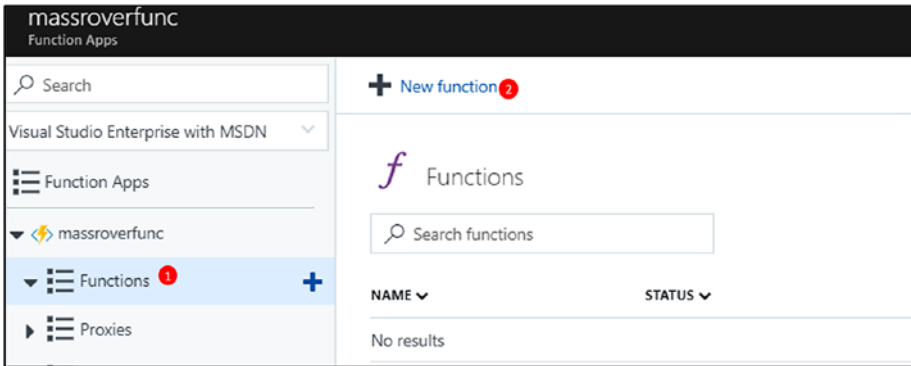


Figure 6-2. Create New Function

HTTP Trigger Function

This will open a list of options. Select “HTTP Trigger,” which will open the HTTP trigger function creation blade (Figure 6-3).

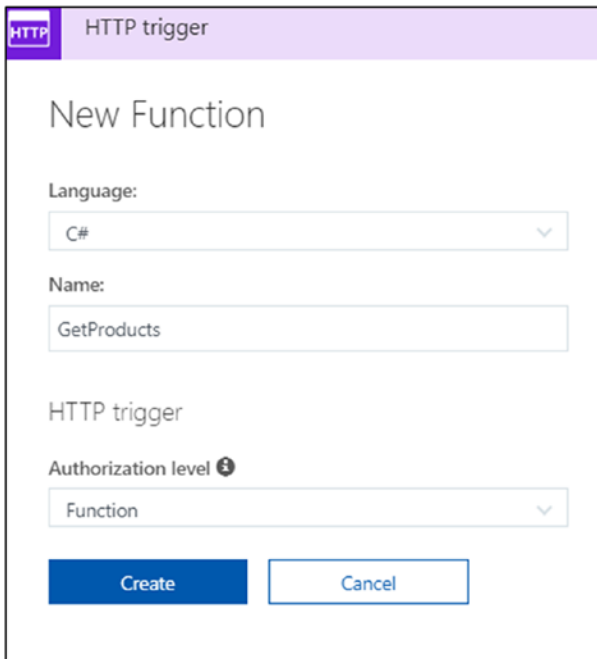


Figure 6-3. HTTP Trigger Function Creation Blade

Select a language and name the function. Additionally, for HTTP triggers, we must select the Authorization level.

Note The development experience available in the portal is not sufficient in most cases, so developers use Azure Functions SDK with an IDE like Visual Studio for real-world development. This book does not cover the development aspect of Azure functions, but includes some important points to consider. One of them is the Authorization level for HTTP triggers. Triggers can be written in C#, F#, and JavaScript. A function always runs as an asynchronous mode.

HTTP Trigger Function Authorization level

The portal offers three options for the HTTP trigger authorization level, but when you create a HTTP trigger using SDK, you will have five options.

1. *Function*: Authorization is based on a function-level key value pair. This key acts as the API key for the function. The key can be passed in the HTTP request, either as a query parameter or in headers. When passing the key in the query, use the query string code, and when passing it via headers, use the header x-functions-key. The function-level authorization is scoped to each individual function, resulting in a different key for each function. This the default option. Sent in query string is as, `https://<app>.azurewebsites.net/api/<funcname>?code=<key>`
2. *Anonymous*: Allows HTTP event triggers without any keys.

3. *Admin*: Requires the host key. Host keys are generated at the Function App scope. So, one host key can be used across multiple functions which has the authorization level set to Admin.
4. *System*: Requires the master key. The master key is a special host key that can be used to manage the Function host. Unless your function does management logic execution, this level of authorization is not needed, and shouldn't be used.
5. *User*: This authorization level is not key based, but requires a valid security header (such as an authorization header) in the request.

The above authorization levels (except User-level authorization) work using two different types of keys.

1. *Host keys*: Host keys are scoped at the Function App level. The master key (named as `_master`) is a special host key that cannot be revoked but can be renewed.
2. *Function keys*: Function keys are scoped at individual functions. One function can have many function keys.

Figure 6-4 shows these keys, under the Manage section of a selected function. Though host keys are managed at the Function App level, they are also shown under the selected function.

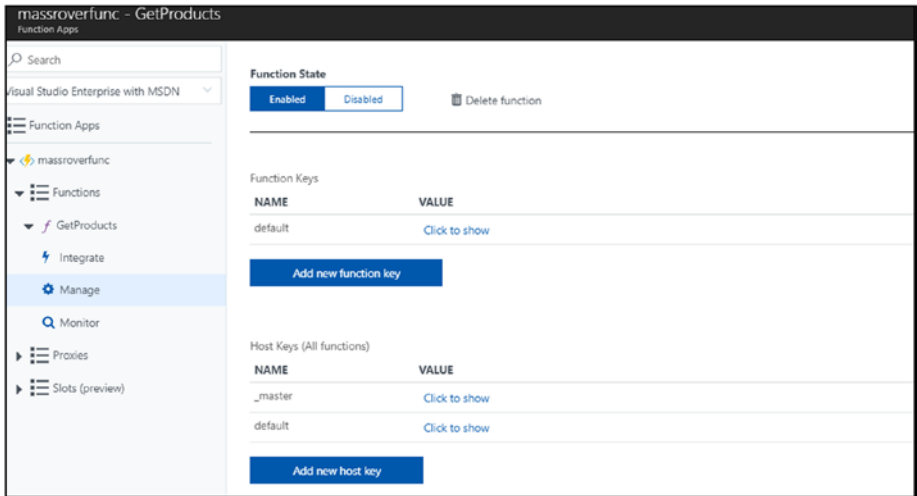


Figure 6-4. Function App Keys

Configuring the Function

Navigate to the Integrate section of the function, shown in Figure 6-5.

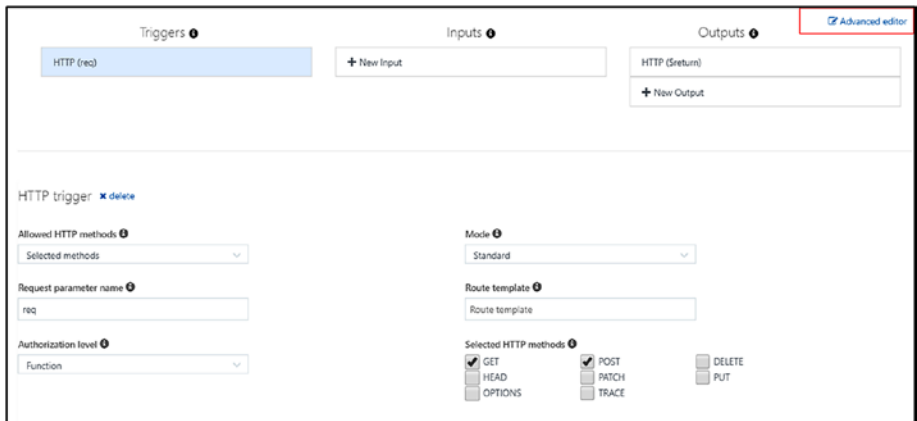


Figure 6-5. Function Integrate Section

In this section, we can set the HTTP request parameter name, the authorization level, the applicable HTTP methods to invoke the function, the mode of the HTTP trigger (whether it is a simple HTTP trigger or a web hook), and the route template. Route template are straightforward; we can specify the parameters using curly braces.

If you click the Advanced editor, you can edit get JSON document for the specified configuration.

```
{
  "bindings": [
    {
      "authLevel": "function",
      "name": "req",
      "type": "httpTrigger",
      "direction": "in",
      "methods": [
        "get",
        "post"
      ]
    },
    {
      "name": "$return",
      "type": "http",
      "direction": "out"
    }
  ],
  "disabled": false
}
```

This JSON document defines the configuration in a more developer-friendly structure. We can edit this document to make configuration changes to the function. Deploy a working version of the function, which

will return some data in the HTTP response. You can choose to write the logic in the portal or using an IDE.

The code snippet below returns some products (like the MassRover API in previous chapters). The code is written in C# using Visual Studio.

Note The development and deployment of Azure Functions is beyond the scope of this book. The easiest way to develop and publish the Azure function is from Visual Studio itself. To do this, the developer must install Azure Function and WebJobs tools. This can be done using Visual Studio tools and extensions.

```
public static class GetProducts
{
    [FunctionName("GetProducts")]
    public static IActionResult Run(
        [HttpTrigger(AuthorizationLevel.Function, "get",
            "post", Route = null)]
        HttpRequest req, TraceWriter log)
    {
        var products = new List<Product>
        {
            new Product { Id = 1, Name = "Lithim L2"},
            new Product { Id = 2, Name = "SNU 61" }
        };
        var data = JsonConvert.SerializeObject(products);

        return (ActionResult)new OkObjectResult(data);
    }
}
```

```
public class Product
{
    public int Id { get; set; }
    public string Name { get; set; }
    public DateTime? ModifiedDate { get; set; }
}
```

Once you've published, you can get the function URL in the portal. Since the authorization level is set to Function, we have to pass the function key either in the query or in the headers. A sample URL would be in the below format:

```
https://<function app name>.azurewebsites.net/api/<function name>?code={key}
```

Also, note that one HTTP trigger function can accept multiple HTTP verbs. This option can be used to create a quick CRUD endpoint of an entity with a switch statement, which can serve the CRUD requests for an entity.

Azure Function Proxies

Azure Functions Proxies provide a core set of API development tools specifically suited for serverless API developers. Azure Functions Proxies allow you to composite multiple APIs across functions and services together into one unified API surface. Azure Function Proxies can link to external backend services or other Azure Functions, or deliver results from mock endpoints.

The portal experience of Azure Function Proxies seems limiting, but Azure provides a JSON document to define the Azure Function Proxies. First, we will create an Azure Function proxy that directly calls an existing backend service. Then, we will edit the JSON document to include more API surface endpoints.

Navigate to the Function App and click on Proxies to create an Azure Function Proxy. You will see the Azure Function Proxy creation blade as shown in Figure 6-6.

The screenshot shows the 'New proxy' creation blade in the Azure portal. It includes a text input for 'Name' with the value 'Products'. Below it is a 'Route template' input with 'api/products'. To the right is a dropdown for 'Allowed HTTP methods' showing 'Selected methods'. A grid of checkboxes allows selecting HTTP methods: GET (checked), POST, DELETE, HEAD, PATCH, PUT, OPTIONS, and TRACE. The 'Backend URL' field contains 'https://massroverproduct.azurewebsites.net/api/products'. At the bottom, there are expandable sections for '+ Request override' and '+ Response override', and a blue 'Create' button.

Figure 6-6. Azure Function Proxy Creation Blade

In the screenshot shown in Figure 6-6, an Azure Function Proxy is created for a backend service. The route template specifies the routing path, and we have the option to override requests and responses. Click Create and an Azure Function Proxy will be created, and you will see the proxy URL.

Now we have created a proxy for a backend service URL. We can have multiple proxies and customized requests and responses, but we'll do this in the proxies.json document rather than in the portal.

Click on Advanced editor—this will open the editor window in the browser, and the proxies.json file will open. The below code sample shows the proxies.json with an additional proxy added as a mock endpoint.

proxy.json

```
{
  "$schema": "http://json.schemastore.org/proxies",
  "proxies": {
    "Products": {
      "matchCondition": {
        "route": "/api/products"
      },
      "backendUri": "https://massroverproduct.
        azurewebsites.net/api/products"
    },
    "Customers": {
      "matchCondition": {
        "route": "/api/customers",
        "methods": [
          "GET",
        ]
      },
      "requestOverrides": {
        "request.header": {
          "api-version": "v1"
        }
      },
      "responseOverrides": {
        "response.statusCode": "200",
        "response.statusReason": "OK",
        "response.body": "[{\\"name\\":\\"customer1\\"},
          {\\"name\\":\\"customer2\\"}]"
      }
    }
  },
}
```

```

    "Partners":{
      "matchCondition":{
        "route":"api/partners",
        "methods":[
          "GET"
        ],
      },
      "backendUri":"https://localhost/api/
HttpTriggerCSharp1"
    }
  }
}

```

As you can see, the above document has three proxies: Products, Customers, and Partners. The Product proxy emits the settings we chose in Figure 6-6, which listens at the route `api/products` and calls an external backend service URL. This backend service URL can be another Azure Function as well.

The Customer proxy exposes a mock endpoint at the route `api/customers`, with additional settings for request and response overrides. The Partners proxy calls Function within the same Function App—thus, using the `localhost` works in Azure as well. If the Function belongs to a different Function App, then it should be called by its full URL.

The above model provides a single API surface consumer, and the Azure Function Proxy abstracts the implementation details. Figure 6-7 illustrates this.

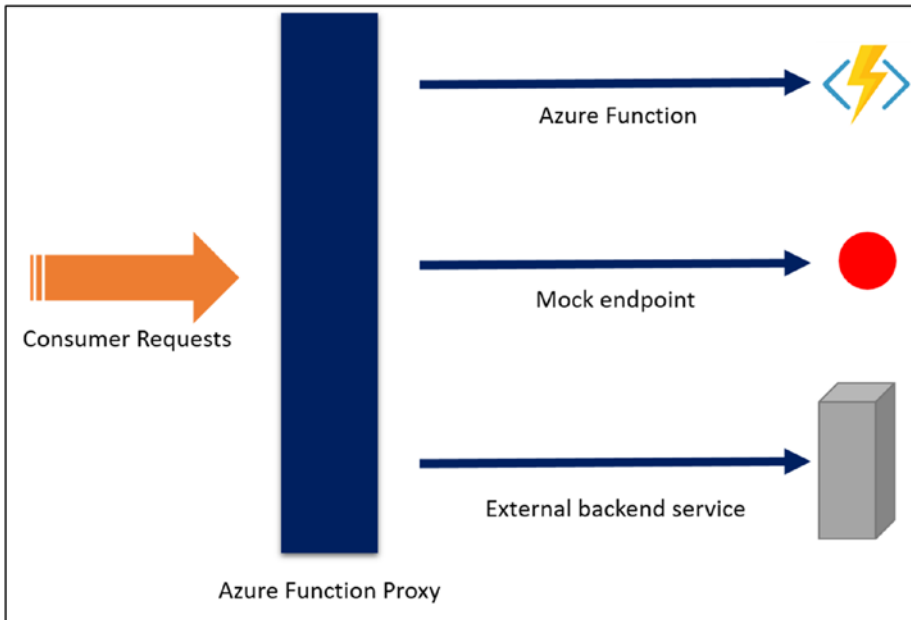


Figure 6-7. *Azure Function Proxy Implementation*

Note Azure Function proxies play the role of an API Gateway, and they can be used to mock APIs as well. In this context, Azure Function proxies provide some of the functionality of Azure API Management, but Azure API Management is a richer, fully functional API management platform service, providing advanced control, features, and additional API management workflows such as a developer portal and subscription management. If you are integrating multiple APIs with simple control over HTTP requests and responses, Azure Function proxies are preferable over Azure API Management (and cheaper).

TR comment: Throttling is missing??

Azure Logic Apps

The Azure Logic App is a serverless BaaS offering, useful for creating integration and workflows. It has many pre-built connectors to applications and services with conditional statement blocks. For granular customization, it offers a logical language. Logic Apps are useful in API development because they have HTTP triggers that can act as API endpoints.

Navigate to the Azure portal, search for Logic Apps, and create one. The creation process is straightforward. You must name the app and choose a subscription, resource group, and location. After the app is created, navigate to the Logic App. There are many templates to choose from, but for this exercise, select the Blank Logic App template to start from scratch. The portal designer provides a number of different triggers. Search for “HTTP” and you will find the HTTP request trigger.

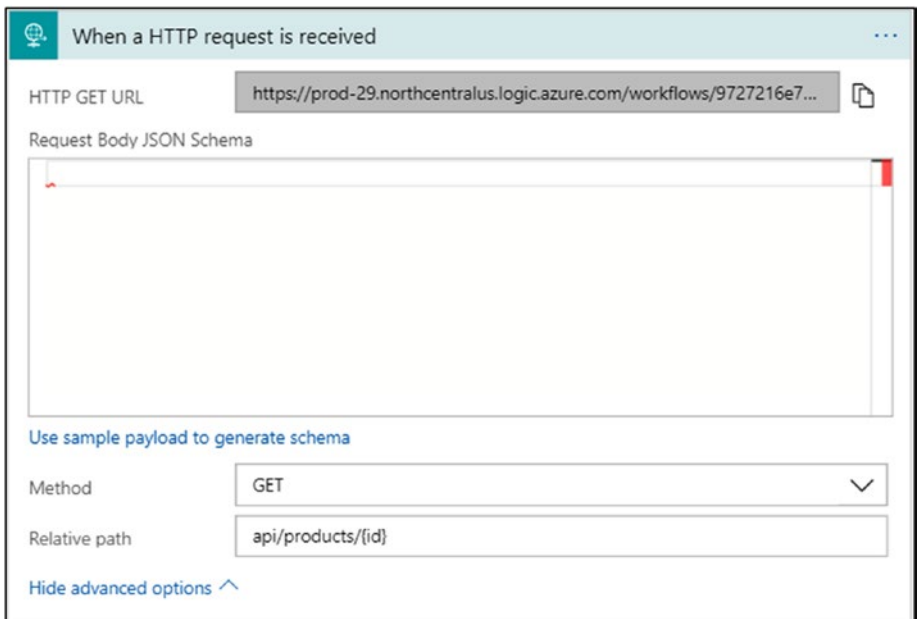


Figure 6-8. Azure Logic App HTTP Request Trigger

We can set the HTTP method and relative path for the request, and we can also set the JSON schema for the request body. We can chain actions to the trigger, and add an action to the HTTP request trigger. In the action, you can get the possible inputs from the previous step, in this case the request body and the ID parameter.

Finally, when you save the Logic App in the designer, it will generate the full URL for the HTTP trigger. The URL will be similar to the one below.

```
https://prod-29.northcentralus.logic.azure.com/workflows/9727216e7a4442ccb4b3f96e863374be/triggers/manual/paths/invoke/api/products/{id}?api-version=2016-10-01&sp=%2Ftriggers%2Fmanual%2Frun&sv=1.0&sig=EuqlcN4fv8qKIflQbteKaTkE3V5v_OkM5w-FgShMnXw
```

You may notice that the URL contains the relative path after the manual/path/invoke segment, which is filled in the request. Additionally, there are some keys and IDs to for the Logic App to identify the request.

Though the URL for the Azure Logic HTTP triggers looks complex, these URLs can be integrated with Azure API Management or Azure Function Proxies, resulting in a unified API surface with consumer-friendly URI fragments.

Serverless APIs in AWS

Serverless is a native term for AWS, and AWS has a FaaS serverless offering called Lambda. In this section, we will build backend services using Lambda and deliver API surface via AWS API Gateway.

AWS Lambda

AWS Lambda is the FaaS offering from AWS. Lambda itself popularized the serverless model and paved the way for serverless traction in commercial software development. It supports multiple languages and platforms.

The AWS Lambda function does not support HTTP triggers by themselves, so we need to equip the serverless API via AWS API Gateway and receive HTTP requests and pass them to AWS Lambdas. In this section, we'll look at how to create and configure AWS Lambdas, and in the next section we will focus on integrating them with AWS API Gateway.

Creating an AWS Lambda Function

Navigate to the AWS portal and search for Lambda to create a function. We can create a function from scratch, from an available template, or from the serverless application repository. In this section, let's create a Lambda function from scratch.

Select Author from scratch and enter the required details. Figure 6-9 illustrates this.

Author from scratch [info](#)

Name*

Runtime*

Role*
Defines the permissions of your function. Note that new roles may not be available for a few minutes after creation. [Learn more](#) about Lambda execution roles.

Lambda will automatically create a role with permissions from the selected policy templates. Note that basic Lambda permissions (logging to CloudWatch) will automatically be added. If your function accesses a VPC, the required permissions will also be added.

Role name*
Enter a name for your new role.

Policy templates
Choose one or more policy templates. A role will be generated for you before your function is created. [Learn more](#) about the permissions that each policy template will add to your role.

Figure 6-9. AWS Creating Lambda from Scratch

Enter the name of the Lambda function, select the runtime, and select a role for the Lambda function. This role defines the permissions for the function. You can use an existing role, create a new one, or create one from a policy template.

In this case, the role named `massrover-lambda` has been created from two policy templates: Simple Microservices permissions and Basic Edge Lambda permissions.

Once the role has been created, AWS will open the function configuration window. The window has many sections, and this book briefly explains them. Figure 6-10 shows the AWS Lambda designer.

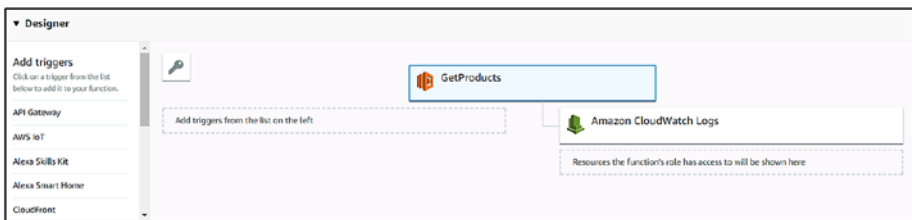


Figure 6-10. AWS Lambda Designer

On the screen shown in Figure 6-10, if you scroll down, you will notice other configuration options:

- *Function code:* This is the section where we upload the Lambda code as a zip file, or we can point to file/folder in AWS S3. In this section, we can set the Lambda runtime as well. Handler specifies a value of the Lambda handler; in the C# language context, this in the form shown below:

```
assembly::namespace.class-name::method-name for
the C# runtime.
```

- *Environment variable:* This sets any environment variables required to execute the Lambda function. In

the Lambda execution context in C#, the environment variables can be read from the system assembly methods, like below:

```
Environment.GetEnvironmentVariable("key");
```

- In the environment variable section, we can set the AWS KMS to encrypt sensitive environment variables. You can also use environment variables to store connection strings and other execution variables.
- *Tags*: These are key value pairs used to group and filter Lambda functions. Lambda functions related one logical work unit can be tagged with the same value and filtered.
- *Execution role*: Execution role the permissions for the Lambda function. This was set at the time of creation (Figure 6-9), and we can change it here.
- *Basic settings*: In this section, we can set the maximum memory for the Lambda function. The available range is 128MB to 3008MB, and the CPU is allocated proportional to the memory. Memory is allocated in 64MB chunks. We can also set the Lambda function timeout. The maximum Lambda execution time is 300 seconds.
- *Network*: AWS Lambda functions run in a system managed default VPC. In this section, we can configure a custom AWS VPC, which makes it secure enough that it can access resources like databases within the specified VPC.

- *Debugging and error handling:* When a Lambda function fails during execution, it will retry twice by default. If the execution is not successful after the retries, we can configure the function to put the messages in an AWS SNS topic or AWS SQS. This configuration option is known as DLQ (Dead Letter Queue). We can specify the ARN of either SNS or SQS to act as the DLQ.
- *Concurrency:* This configuration deals with the execution concurrency of Lambda functions. We can set a specific number or use all the unreserved concurrency of the account. An account is reserved with 1000 concurrency executions.
- *Auditing and compliance:* In this section, we can enable the audit trails of Lambda executions using AWS CloudTrail.

By selecting the Lambda function, we can visit the designer and configure the above settings.

After the creation of the AWS Lambda, we can write code using an IDE and deploy it, or upload the code packages via AWS Console directly or from AWS S3.

The below code snippet shows the returned list of products in AWS Lambda in C#:

```
public class Function
{
    public string FunctionHandler(ILambdaContext context)
    {
        List<Product> products = new List<Product>
        {
```

```

        new Product { Id = 1, Name = "Lithim L2" },
        new Product { Id = 2, Name = "SNU 61" }
    };

    var data = JsonConvert.SerializeObject(products);

    return data;
}
}

public class Product
{
    public int Id { get; set; }
    public string Name { get; set; }
    public DateTime? ModifiedDate { get; set; }
}

```

The injected `ILambdaContext` will give the access the execution parameters.

Note Development and deployment of AWS Lambda are beyond the scope of this book. You can develop and publish AWS Lambda from Visual Studio itself using the AWS development tools for Visual Studio.

After the publish, we can invoke the Lambda from Visual Studio, as we still haven't wired the HTTP event to the Lambda. This is done using integration with AWS API Gateway.

Figure 6-11 shows the AWS Lambda invoke from Visual Studio. You can see the response on the left-hand side. Also notice the log output, which shows the used memory, execution time, and billable time of the Lambda.

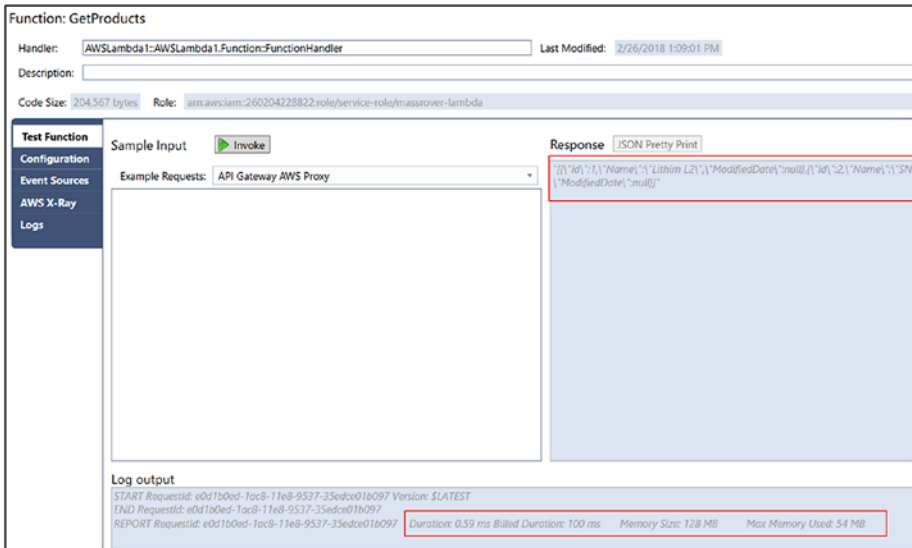


Figure 6-11. Invoke AWS Lambda from Visual Studio

Setting Up AWS Lambda with AWS API Gateway

In this section, we will set up the integration between AWS Lambda and AWS API Gateway, which enables the functional serverless API in AWS.

Navigate to the AWS Console, create a new API in AWS API Gateway, then create two resources under api, named “api” and “products.” (refer to chapter 4 for more details on how to perform these actions).

Select the api/products resource and create a GET method, shown in Figure 6-12. In the method, select the integration type Lambda Function. Select the region and then select the Lambda Function (when you type the name of the Lambda, it will appear in the dropdown).

When you click save, AWS will show a dialog indicating that you are giving permission to API Gateway to invoke the Lambda function, showing the ARN of the Lambda function.

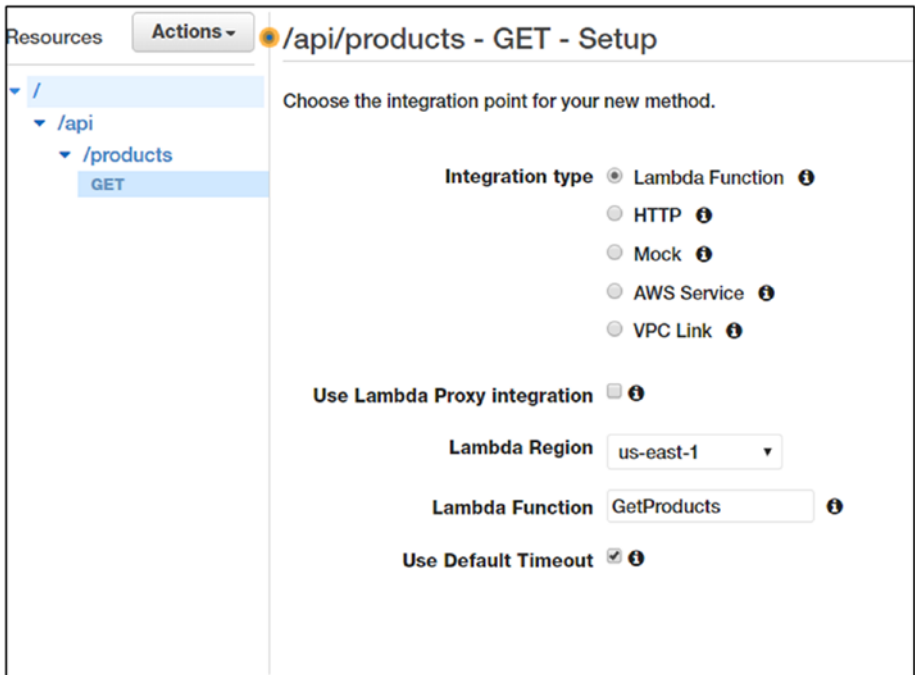


Figure 6-12. AWS API Gateway Lambda Integration

Technically, the integration is complete after this step, and you can test it using the test link in API Gateway.

We can then deploy the API by creating a stage (refer to chapter 4 for more details). The deployed API will have a URL like this:

<https://vtv6xchkkk.execute-api.us-east-1.amazonaws.com/prod/api/products>

Note The integration of the Lambda with API Gateway can be performed from the Lambda designer as well.

Summary

Serverless is an evolution of cloud computing that has been gaining popularity for the development experience it offers and its more granular, utility-based computing. Because of this, serverless is not only a new way of developing applications, but also provides a new economic model in the cloud computing world.

Serverless APIs are in the limelight because of the benefits mentioned earlier, as well as its economic benefits. But serverless platforms can be expensive in some cases.

The event-based and self-hostable nature of serverless platforms are other major reasons developers can easily integrate this model with any legacy application.

CHAPTER 7

Practical Design and Development

The modern application design and development landscape design and development landscape of modern applications is highly influenced by emerging technologies, changing developer mindset, and the urging survival and peer pressure to innovate. Fundamentally, there is an ever-growing demand for data, and developers control the flow of data in applications.

APIs help developers to control and orchestrate the data flow among services and solutions more effectively. In practical design and development, there are some common models used developers to get the maximum benefit from the API development strategy.

This chapter explains a few common scenarios and design practices used in real-world development.

Contract-First Design

As the name implies, contract-first design is focused on the service contracts; contract first design approach is all about beginning the development by defining service contracts and related service endpoints. In addition to the service contract schema and the endpoints, other aspects such as versioning, URI schemes, and naming, are included in the design approach.

Defining the clear service contracts helps to understand the business domain models. There are different techniques for identifying this, but event storming is one widespread practice used to identify the business flows domain models and operations. The identified domain models and operations are the seed for the contract-first design.

In DDD (domain-driven development) and event storming world, the term domain model is used, but in practical contract first design approach and RESTful service design, it makes perfect sense to use the term resources and operations.

When starting the API development with contract first approach, first we have to absorb the business problem and start defining the service contracts (typical JSON objects) at the API level. These JSON objects define the schema of the service contracts; these define either the result set from API or the incoming request body. One common and easy implementation approach for the contract first design is to use API mocking. Mocking APIs is a very powerful in the contract first design, it helps the developers and business stakeholders to validate the domain models while establishing the technical standards. This eliminates lots of frequent rewrites and discussions at the early stages of the development. Both Azure API Management and AWS API Gateway have have mocking facilities, enabling you to define models and operations right from the portal.

The idea here is basically to fast-track implementation and identify business and usage flow gaps as early as possible, and to iterate quickly without compromising the technical standards required for a classy API design.

Figure 7-1 depicts this approach in a high level picture and how different stakeholders are involved in the contract first design approach.

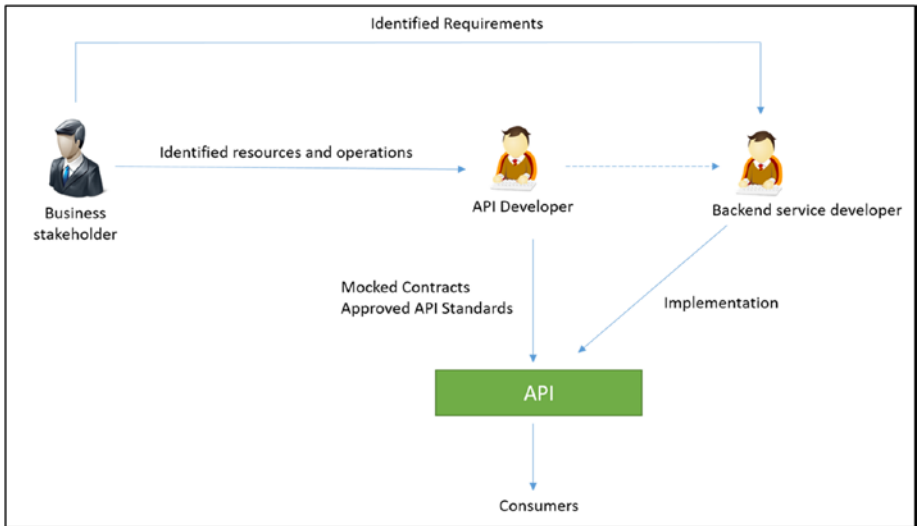


Figure 7-1. *Contract-First Design*

This approach requires considerable skills and strategic practices in order to be successful. Because, the API developer and the backend service developer.

Preparation

The business owner or the product owner or event storming coordinator should be aware of this design model so he or she can facilitate the requirement flow in order to identify the resources and operations and enable the contract first design approach.

The API developer should be skilled in mapping the domain being discovered to the API resources and endpoints. He/she also should experience with the chosen development/API mocking platform to gain speed in and orchestrate the front consumers and backend service developers.

Basic-level API standards differ among developers and teams. During the above exercise the identified operations and the resources can change in high frequency but not the agreed API standards. Chapter 3 provides a good starting guideline of API standards, these guidelines are battle tested and can be extended to the custom requirements.

Key Challenges

The primary challenge of contract-first design is identifying the right resources and operations to be exposed as APIs. There will be changes at high frequency at the initial stages of the domain discovery but this will gradually come to a more over stable state; identified resources will not change (except their properties) since

For example, in a simple clinical system, a medical practitioner is an identified resource. The attributes of this resource and the corresponding operations will change, but the resource practitioner should not change to a different resource in high frequency. The person who facilitate this exercise should make sure the business domain is progressively discovered; if the person does not strategize the domain discovery in a progressive manner that would become a big bottleneck and a challenge for the entire process.

The second key challenge is the technical skill level of the API developer, because the sole purpose of this design approach is to fast-track development and enable API consumers as quick as possible. If there's a requirement for a mobile application, then the mobile developer should be enabled from day one, as soon as the requirements begin to flow, API endpoints with the defined operation and standards should be available from the day in order to achieve this, so an API developer lacking the technical skills in the chosen platform will cause a bottleneck in the implementation.

Finally, the team—the API developers, backend service developers and the consumers—should be aware of and well-disciplined with the approved API standards. Note, as in the [Figure 7-1](#) the approach has many

roles, but in smaller projects this does not map to different individuals playing those roles. There can be situations one single developer/one single team doing all things.

When Not to Try It

This design approach would not be suitable in high-level usage flow scenarios. In an event storming model, discussions reoccur with increasing granularity and detail; in certain cases things are being discussed at very high level, which makes it difficult to extract the required resources and operations to be included in the API. If the domain discussion at very high level then the contract first design approach wouldn't fit.

If the approved API standards or disciplines are not available and not formalized well. in such occasions it is not recommended to use the contract first design approach, as it will create conflicts and unproductive communication among the teams. First, make sure everyone things on the standards on the same way and define the standards to a certain level.

APIs in Microservices

Microservices is an emerging application design architecture.

Microservices are consisted of many different services and these services communicate with each other using defined service contracts (not necessarily in HTTP) and they expose the resources and operations to the consumers (client applications) via APIs. In order to orchestrate the service communication between the clients and the different services in microservices API gateways are heavily used. In this model, there are two different approaches 1) Client Coordinated design - this design let the consumer manage the service orchestration, there is no unified approach in API standards. Each service can have its own API standards and consumers should be aware of them individually to separate services.

2) API Gateway pattern - API gateway is used to do the service orchestration

and unified API standards are implemented at API gateway, allowing individual service development teams to have own standards, this also allows to bring legacy services to the unified API standards.

Client-Coordinated Design

Client-coordinated design is based on individual services that are directly accessed by the client to complete a business flow. In this context, each microservice exposes its own API to consumers.

Figure 7-2 depicts this.

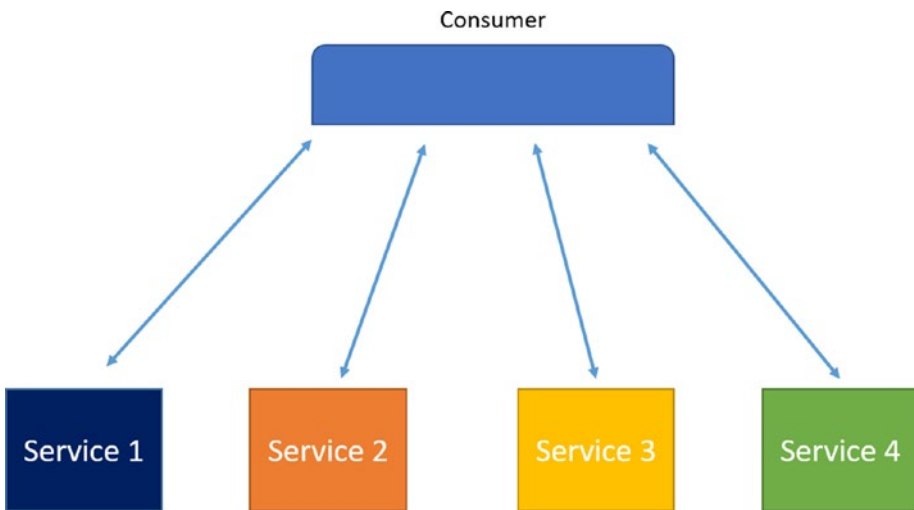


Figure 7-2. *Client-Coordinated Micro Service Design*

The main advantage of this approach is, it allows consumers to determine the business flow rather than a predefined flow from one single party, this give flexibility to the consumers.

Small-scale micro services can use a client-coordinated design, especially when teams are together and have approved standards. API standards should be followed by each team to ensure a uniform development, but this can easily be violated since there's no central governance for the API.

There are common disagreements over this design approach compared to the API gateway design, but this approach wins over the API gateway approach pattern in the argument of with much governance and single point of failure the API gateway introduces.

API Gateway Pattern

The API gateway pattern introduces the API gateway as a centralized, coordination layer between individual services and consumers. Also, this practice helps to achieve additional benefits such as gateway-level pre-authentication logic, caching, flow control and other cross cutting concerns.

Figure 7-3 depicts this design.

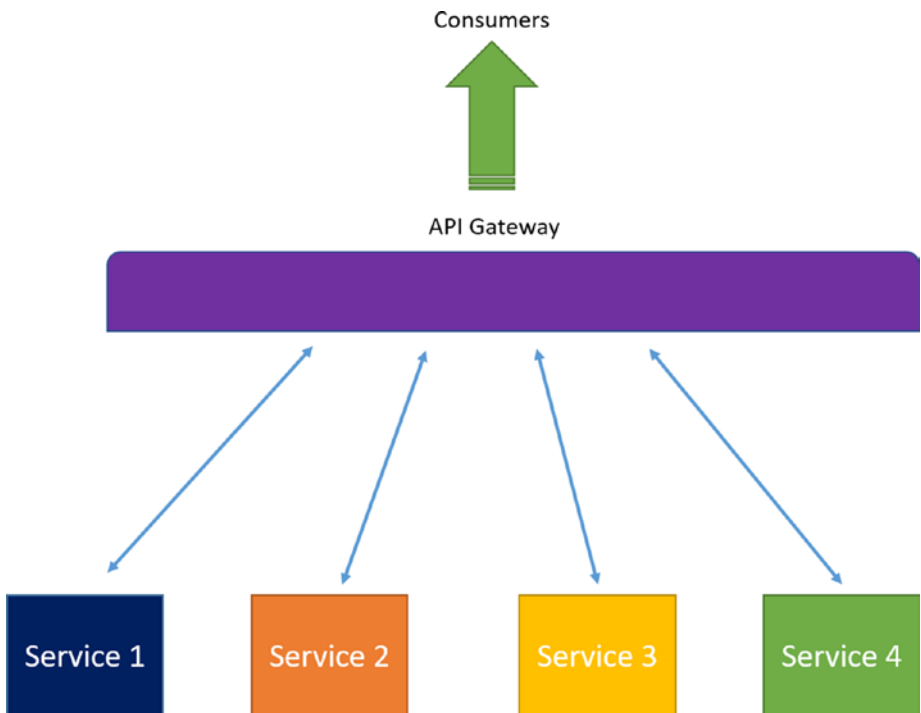


Figure 7-3. API Gateway Implementation

The API gateway pattern issues a central point of communication and standards, which allows each service to use their own API standards. Having different API standards for each service allows flexibility in development and opens the possibility of implementing different protocols, especially in modernizing the legacy systems.

From a deployment point of view, API gateways add layered security for the backend service, allowing the backend service to live outside the public Internet.

There is a general concern, about having more domain specific logic in the API gateways, the point is about, whether is it good or bad to have business logic in the API gateway? With the increasing popularity of the API gateways, the API gateway vendors overload them with features that can be used beyond the gateway model. Since the term gateway is not a functional requirement and serves the purpose of a reverse proxy; it is quite obvious that including business logic in an API gateway is NOT a good design. But again in certain cases utilizing the chosen tool/framework makes the implementation easy and fast. Having business context in the API is known as 'Thick API gateways', you can read more about Thick API gateways from author's blog: <https://thuru.net/2018/04/22/overambitious-api-gateways-security-at-api-gateway-with-azure-api-management/>

APIs for Enterprise Integration

APIs are commonly used in enterprise integration; the usage has been followed from the web services with all the WS* standards. Now, generally RESTful services are used in this context in order to deliver the same functionality.

RESTful services and platforms are lightweight, and emerging modern trends like self-hostable runtimes have made them more flexible so that they can sit between many large systems making them the first choice in enterprise integration.

Enterprise integration APIs often deal with old protocols and messaging standards. Data translations from old to new protocols, and vice versa.

Enterprise API integrations have their modern flavors too, for example: A deep learning company develops an AI engine that can make intelligent predictions using financial data. In order to make this work, the AI engine should have integrations with different accounting systems. Building an integration point for each different accounting platform will be complex and time consuming. So an integration company would help them build a standardized API integration for various accounting platforms.

In the enterprise world, integration is big business. In fact, all enterprise-level application development has some sort of integration. APIs help open the data flow between the systems. Persistence-based messaging platforms like service buses are considered obsolete, and HTTP-based RESTful services are replacing them.

Summary

API design and related tools can help achieve development flexibility and new models of team orientation. There are no hard and fast rules dictating which design principle to practice in a particular situation. Each design has its own pros and cons in any given scenario.

Creating a RESTful service in any modern language is simple and well supported with tools and frameworks. But developing a standardized API practice with well-structured implementation goes beyond the technology platform. This book addresses that in a balanced way, with the proper information to begin the development and also by providing information about the API implementation and architecture. An organized effort around these standards and implementations will yield a good API design.

Index

A

Analytics, 23

App creation blade, new
function, 138

Application programming
interface (API), 1
architecture, 19–20
developer experience, 21
error handling, 22
integration, 21
performance, 21
security, 20
usage and telemetry, 21

attracting innovation and
disruption, 16

business orientation, 15

collateral developments, 14

data and operations flow, 5–6

development, Visual Studio
tool, 34

economy, 6–7

e-commerce, 9

gateway-level API keys, 108

key-based security, 107

management, 51

management tools

analytics, 23

catalog, 23

design, 22

gateway, 23

monetization, 16

programmable language
constructs, 2–4

public sector, 9

G2B, 10–11

G2C, 9–10

government, 11

RESTful services, 6

use case, 17

value chain
definition, 18
layers, 18

versioning techniques, 27

ASP.NET Core implementation

action methods, 40, 42

create product, 36

creation, 35

error codes, 37

error handling, 36

ErrorMessage, 37

ErrorService, 38–39

HTTP DELETE

action, 44–45

HTTP POST, 42

404 Not Found, 42, 44

ProductsController, 39–40

INDEX

- Authentication and authorization
 - security design
 - API keys, [106–110](#)
 - Azure Active Directory V2
 - (*see* Azure Active Directory V2)
 - custom JWT tokens, [120](#)
 - OpenID and OAuth, [110–111](#)
 - pre-authentication, [124–126](#)
- Authorizers, AWS API gateway
 - authentication flow, [130](#)
 - creation, [127–128](#)
 - Lambda Event Payload, [129](#)
 - Lambda function, [127](#)
 - policy document, [129–131](#)
- AWS API gateway
 - components, [94–95](#)
 - configure methods
 - add header, [87](#)
 - content handling, [85](#)
 - GET method, [84, 86](#)
 - integration request header mapping, [88](#)
 - request and response flow, [86–87](#)
 - creation, [82–84](#)
 - deployment API
 - Actions menu, [89–90](#)
 - stage options, [90–92](#)
 - usage plans, [92–94](#)
- AWS Beanstalk, [109](#)
- AWS Lambda
 - creating Lambda function, [151](#)
 - setting up with API gateway, [156](#)
- Azure Active Directory V2
 - application registration, [113](#)
 - consent screen, [116](#)
 - JWT decoding tool, [117](#)
 - JwtSecurityToken, [119](#)
 - mechanism, [112](#)
 - OpenIdConfiguration, [119](#)
 - OpenID information, [115](#)
 - public identity providers, [120](#)
 - request URL, [114](#)
 - setup for securing, [112](#)
 - tokens, [118, 119](#)
 - TokenValidationParameters, [119](#)
- Azure API management
 - API Blade, [60](#)
 - backend service connection,
 - [61–62](#)
 - component structure, [81](#)
 - developer experience
 - developer portal, [76–77](#)
 - developer subscriptions
 - page, [80](#)
 - logged in, [79](#)
 - subscribe to product, [80](#)
 - workflow, [78](#)
 - endpoint configuration
 - authorization, [67](#)
 - backend service URL, [65](#)
 - headers, [65–66](#)
 - HTTP Request, [67](#)
 - query parameters, [65](#)
 - Request URL, [67](#)
 - JWT validation policy, [126](#)
 - overview, [59–60](#)

- permissions, 76
- policy configuration, 68–70, 72
- portal, 56–57
- products, 72–74
- steps, 58
- workspace, 63

Azure functions

- app creation blade, 136
- app service plan, 137
- consumption plan, 136
- HTTP trigger authorization level, 139–140
- HTTP trigger creation, 138–139
- proxies
 - creation blade, 145, 147
 - implementation, 148
 - integrate section, 142, 144

Azure logic app, HTTP request trigger, 150

B

- Backend service
 - connection, 61–62
- Business domain models, 160

C

- Catalog, 23
- Consumer-commanded
 - endpoints, 26–27
- Contract first design, 159, 161
 - challenge, 162
 - preparation, 161

- CRUD endpoints, 25, 35
- Custom JWT tokens, 120

D

- Data and operations flow, 5–6
- Domain Driven Development (DDD), 160

E, F

- Enterprise integration, 166–167
- Error handling, 22, 31
- Explicit parameters, 26

G

- Gateway, 23
 - AWS API gateway (*see* AWS API gateway)
- Government to Business (G2B), 10–11
- Government to Citizens (G2C), 9–10
- Government to Government (G2G), 11

H

- HTTP status codes, 29–30
- HTTP trigger authorization level
 - function keys, 140
 - host keys, 140
- HTTP verbs, 28–29

INDEX

I, J, K

Information Technology
Promotion Agency (IPA), 10
Internet of Things (IoT), 6

L

Lambda function, 127
configuration options, 152,
154-155
designer, 152
invoke from visual studio,
155-156

M

Many-to-one mapping, 55
Microservices
API gateway pattern, 165-166
client coordinated design,
164-165
environment, 50

N

404 Not Found, 42, 44

O

Object oriented programming
(OOP), 2
One-to-many mapping, 54
One-to-none mapping, 55
One-to-one mappings, 53-54

OpenAPI Specification
(OAS), 34-35
OpenID and OAuth, 110-111

P

Package manager console
(PMC), 45
PATCH requests, 29
Programmable language
constructs, 2
Public cloud platforms
gateways, 51-53
many-to-one mappings, 55
one-to-many mappings, 54
one-to-none mappings, 55-56
one-to-one mappings, 53-54
PUT requests, 29

Q

Query string parameter, 34

R

Remote Procedure Call (RPC), 5
Request-based security
AWS API gateway
API keys, 102-103
quota limits, 104
rate limits, 103-104
Azure API management
IP restrictions, 101
quota limits, 100-101

- request rate limits, [99–100](#)
- subscriptions and
 - subscription keys, [98](#)
- RESTful semantics, [25](#)
- RFC 7807, [31](#)

S

- Security, [20](#)
 - IP-based, [27](#)
 - request-based security (*see* Request-based security)
- Serverless APIs
 - Azure functions, [135](#)
 - Azure functions proxies, [144](#)
 - Azure logic app, [149](#)
 - Lambda (*see* AWS Lambda)
- Serverless computing
 - Backend as a Service (BaaS), [133](#)
 - Functions as a Service
 - (FaaS), [133](#)
 - Simple Workflow Service
 - (SWF), [134](#)
- Service contract, [26](#)
- Service-oriented architecture
 - (SOA), [5](#)
- Software development kits
 - (SDKs), [4](#)

- Subscriptions and subscription
 - keys, [98](#)
- Swagger tool, [27](#)
 - ConfigureServices method, [45](#)
 - Microsoft.Extensions.
 - PlatformAbstractions, [45](#)
 - PUT method, [49](#)
 - Swashbuckle.AspNetCore, [45](#)
 - UI, [46](#)
 - XML documentation, [47–48](#)
- Swagger UI, [48](#)

T

- Team orientation, [49](#)
- Telemetry, [21](#)
- TokenValidationParameters, [119](#)
- TRex tool, [27](#)

U

- URI syntax, [33](#)

V, W, X, Y, Z

- Value chain, [18](#)
- Versioning standards, [34](#)