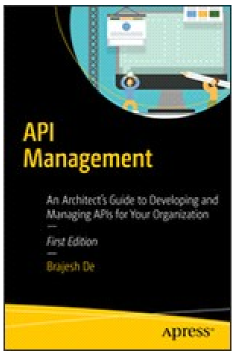

API Management: An Architect's Guide to Developing and Managing APIs for Your Organization

by Brajesh De

Chapter 5: API Patterns

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

Overview

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

Best Practices for Building a Pragmatic RESTful API

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

- **Design APIs with RESTful URLs**. Design an API based on the logical grouping of identified resources. The API URL should point to either a collection of resources/subresources or an individual entity within the collection. For example, `/customers` should refer to a collection of customers, while `/customers/{customerId}` should refer to an individual customer entity with in the collection. The URL should be intuitive enough to identify the resources and navigate through them easily.

- **Use HTTP verbs for CRUD action on resources**. Use the HTTP verbs to perform CRUD action on the resources. Use POST to create a new resource, GET to read, PUT to update, and DELETE to delete a resource. Additionally, you may consider providing support for the PATCH verb in the API resource for partial updates. OPTIONS verb can be used to determine the metainformation about the resource, such as the methods supported, HTTP headers allowed, and so forth.

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

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

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

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

- **Design the API interface to support filtering on the result set**. The response to a GET request for an API resource may sometimes be quite large. Displaying this large response in the consumer app may be quite challenging considering the limited form factor and processing power on the device. Also transmitting a large payload over the network would also impact the bandwidth and the overall performance. Hence, the client app using the API would obtain a lean and filtered response for a GET request. This can be achieved only if the API supports filtering on the result set. Filtering criteria may be specified as unique query parameters for each field that supports filtering. For example, when querying for a customer's orders, you may want to limit it by the order date, such as orders placed in the previous month, six months, or year. This can be specified using a GET request with a `orderDate` such as `GET/customers/customer123/orders?orderDate>'YYYY-MM-DD'` query parameter.

- **Design the API to support pagination**. Pagination is yet another feature that is useful in handling large responses from an API. Even the filtered response from the back-end service for an API may contain hundreds of records. In such a scenario, it makes sense to display only ten of them on a page in the consumer app and provide a link to the next page with the next set of records. Supporting pagination for an API response can address this need to the app developer. This includes pagination parameters in the request as query parameters. `'limit'` and `'offset'` are the most commonly used query parameters to specify the pagination requirements. For example, `orderDate` is like `GET/customers/customer123/orders?orderDate>'YYYY-MM-DD&limit=5&offset=0. 'limit'` indicates the number of records to be included in a page and `'offset'` denotes the page number. Also the API response should include the pagination metadata in the response. This can be included in human-readable format as envelope within the response or in a machine-readable format using the `Link` header. Following is an example of pagination metadata included as an envelope within the response:

```
"_metadata":
  {
      "offset": 2,
      "limit": 5,
      "page_count": 25,
      "total_count": 127,
      "Links": [
        {"self": "/orders?offset=2&limit=5"},
        {"first": /orders?offset=0&limit=5"},
        {"previous": "/orders?offset=1&limit=5"},
        {"next": "/orders?offset=3&limit=5"},
        {"last": "/orders?offset=25&limit=5"},
      ]
  },
  "orders": [
    {
      "id": 1,
      "item-name": "Widget #1"

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

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

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

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

- **Return resource representation in response to creating and updating**. The response for POST, PUT, and PATCH operations results in creating and/or updating a resource. The API response payload for these methods should include metainformation such as `'created_at'` or `'updated_at'` along with the created or updated representation of the resource. Successful execution of a POST request should return a HTTP *201 Created* status code along with the `'Location'` header containing the URL of the newly created resource. Successful update requests should return with HTTP status code 200 OK.

- **Use HTTP headers to specify the media type for the message payload**. When sending a request or a response, include the `'Content-Type'` header to specify the content type for the message payload. This helps the message recipient to easily identify the parser to be used for processing the message. For example, the `'Content-Type'` header with the value *`application/json`* indicates that the payload is in JSON format and the recipient should use a JSON parser to process the message. Similarly, use the `'Accept'` header in the request to indicate the format of the response expected by the consumer from the provider.

- **Use HTTP headers to support caching**. HTTP provides built-in features to support caching. HTTP provides a number of useful headers to efficiently communicate caching information. The `'ETag'` header contains a hashed value of the resource information. Instead of including the entire resource representation in the message payload in XML or JSON format, the `'ETag'` header with the hash value can be used to communicate information about the resource. Similarly, headers can be used to communicate resource modification date/time, expiry time of the cache, validation rules, and the cacheability of a resource. These headers should be used effectively for handling cache.

- **Secure APIs using authentication information in HTTP header**. APIs exposing data and assets should always be secured. Authentication credentials should be included in the HTTP `Authorization` header. Since REST services are stateless, cookies should not be used. Session information if any should be passed in custom HTTP headers. Basic Authentication should be used when the API needs to identify the end user. OAuth-based authentication can be used when a third-party application needs to access the API on the behalf of another user. In case of authentication failure, the API should respond with *401 Unauthorized*. All communications containing sensitive information or any security credentials should be encrypted using TLS.

- **Handle Errors using HTTP Status code and appropriate error messages**. In case of errors, APIs should respond with useful error messages in a consumable format. Appropriate HTTP error response codes in 4XX or 5XX series should be used. 4XX response codes should be used if the error occurred due to fault of the client. 5XX response codes should be used in case of server errors in processing the request. The error response payload should at a minimum communicate the following:

  - **Error message code:** A unique alphanumeric code to uniquely identify the error.

  - **Error message:** A brief summary of the error.

- **Error description or reason:** A description of or reason for the error.

This is an example of an error response payload:

```
{
  "code" : Err_P0L0001,
  "message" : "Address missing",
  "reason" : "No Address specified in the 'Address' field"
}
```

API Management Patterns

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

API Facade Pattern

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

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

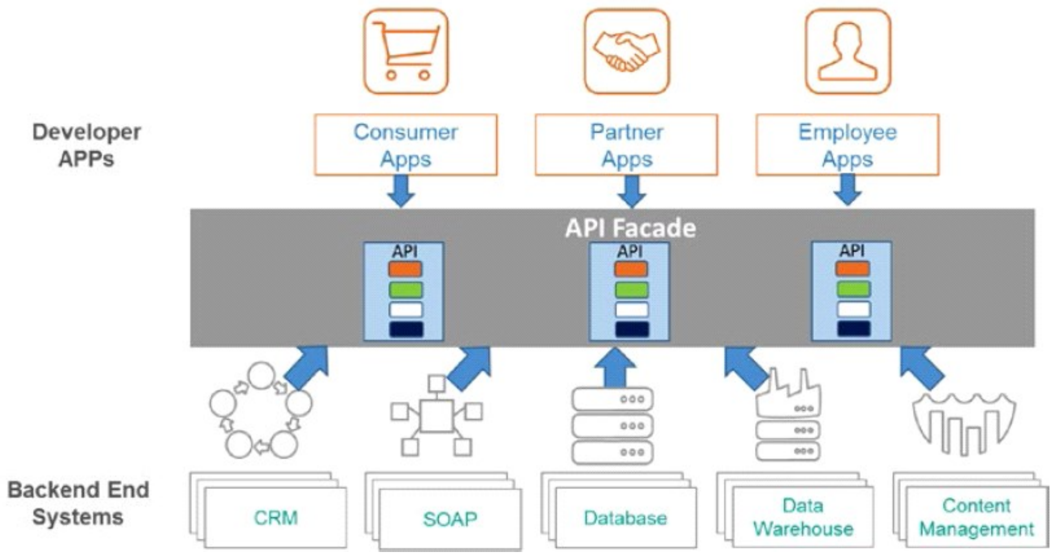

Figure 5-1: API facade pattern

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

API Composition

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

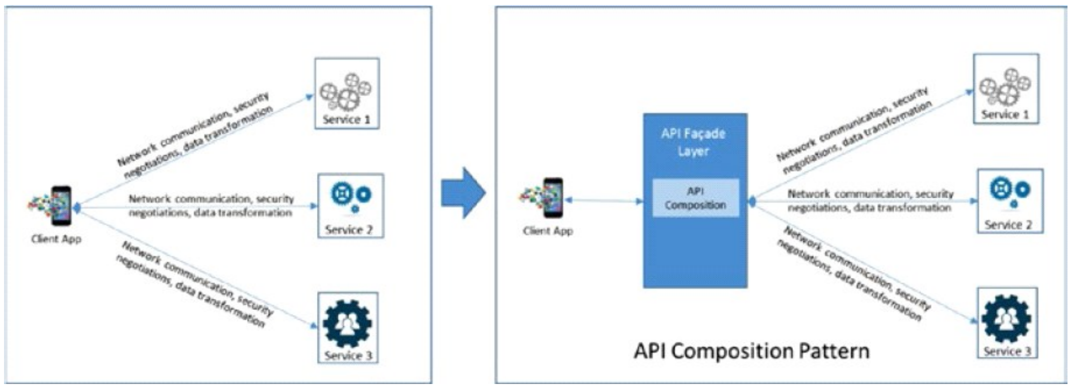

Figure 5-2: API composition pattern

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

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

Session Management

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

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

The response for this request can be as follows:

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

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

Two-Phase Transaction Management

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

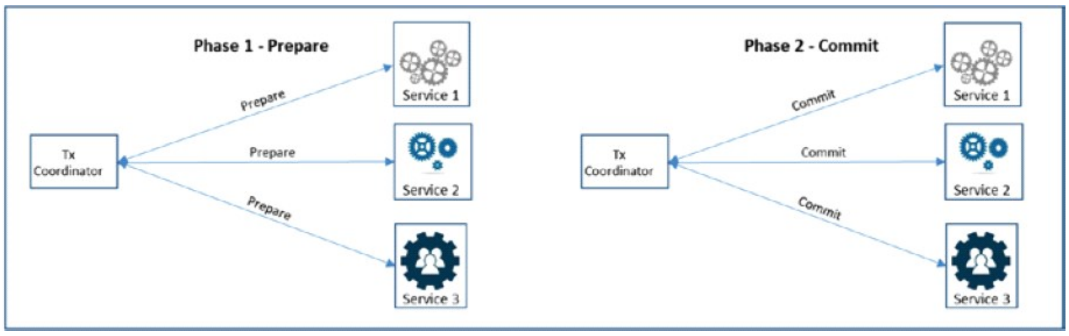

Figure 5-3: Two-phase transaction management of APIs

Exposing each transaction phase as an API and expecting the client app to coordinate the transaction, and roll over in case of failure, is an over kill. Managing all transaction from the client app is going to result in a chatty conversation. The complex processing logic in the app for transaction coordination and management definitely yields a poor app performance. The solution is to handle the conversation from an API facade. The logic to prepare, commit, and roll back two-phase transaction management is implemented in the facade. The facade exposes only one API that is invoked by the client. For example, a hotel booking service can expose only one endpoint to access it (`/hotelbooking`). This endpoint may in turn invoke two separate endpoints: one to reserve the hotel (`/reserve`) in the prepare phase and the second to make the payment (`/payment`) to confirm the reservation in the commit phase. This way the client need not directly access both `/reserve` and `/payment` services and nor does it have to manage the two-phase transaction. Figure 5-4 shows how the two-phase transaction is handled by APIs.

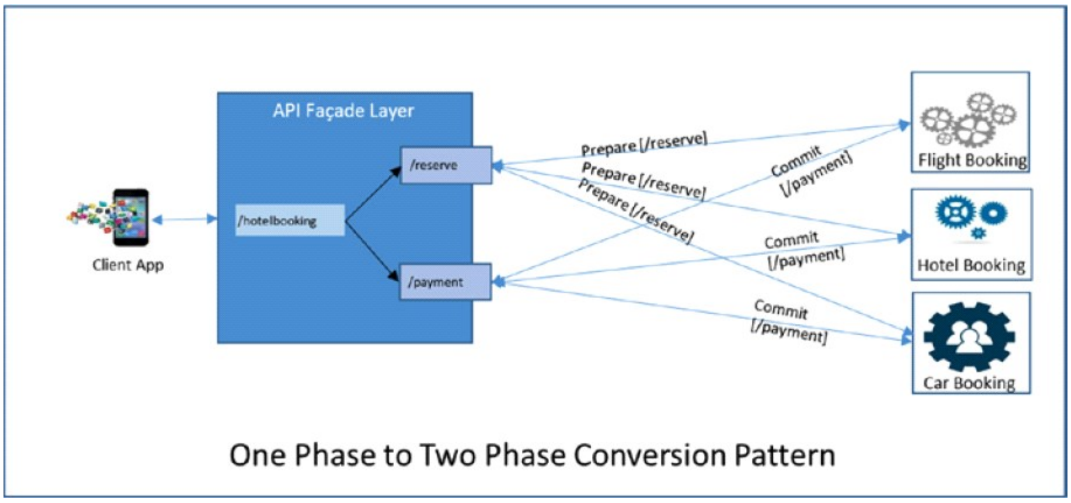

Figure 5-4: Two-phase conversion pattern

Synchronous to Asynchronous Mediation

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

1. The client app makes a call to the API facade.
2. The API facade makes a call to the back end with a callback URL pointing back to the facade layer.
3. The API facade sends response back to the client with URL to check on the response status.
4. After sometime the target system sends the updates (Eg. delivery status) to the API Facade at the callback URL. API facade layer forwards the notification to the notification URL of the mobile app.

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

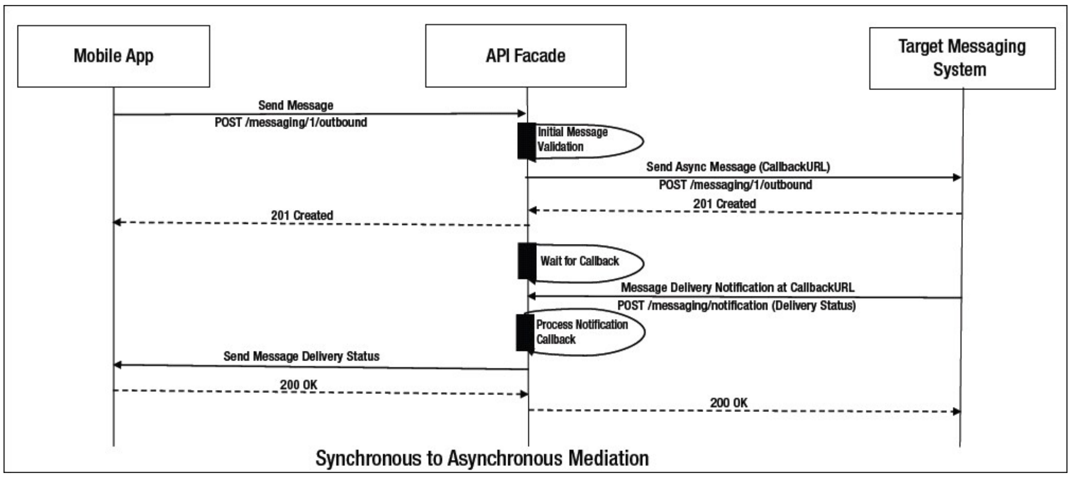
Mobile App
API Facade
Target Messaging System
Send Message
POST /messaging/1/outbound
Initial Message Validation
Send Async Message (CallbackURL)
POST /messaging/1/outbound
201 Created
201 Created
Wait for Callback
Message Delivery Notification at CallbackURL
POST /messaging/notification (Delivery Status)
Process Notification Callback
Send Message Delivery Status
200 OK
200 OK
Synchronous to Asynchronous Mediation

Figure 5-5: Synchronous to asynchronous mediation pattern

Routing

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

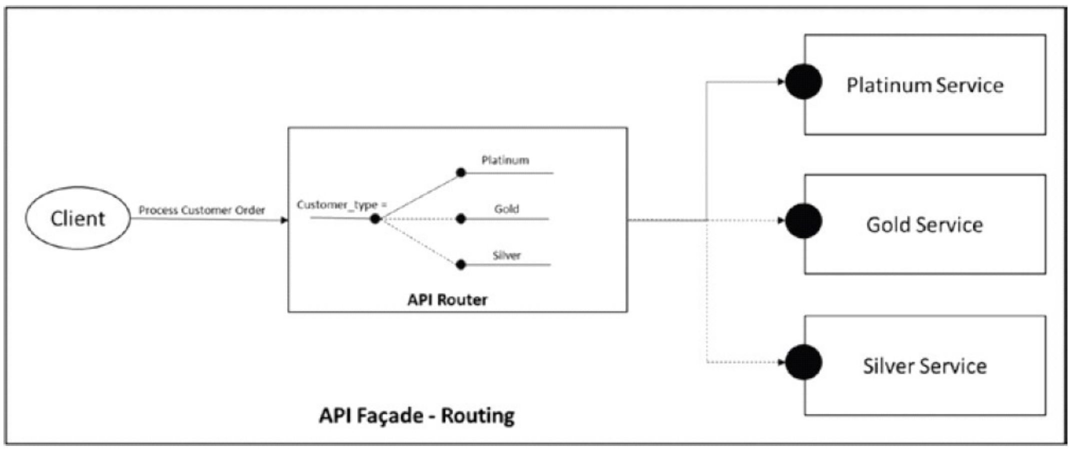

Figure 5-6: API routing pattern

API Throttling

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

- **Spike Arrest:** With Spike Arrest, you can detect sudden unexpected changes to the traffic pattern. Applying a Spike Arrest policy smooths out the traffic by uniformly distributing the traffic across each smaller interval. For example, if the set spike arrest limit is 60 per minute, then only one request is allowed every second. If in any second there is more than one request, they would all be throttled. Similarly, if the spike arrest is 200 per second, then only one request is allowed per 5 milliseconds. If there is more than one request in any 5-millisecond interval, subsequent requests is throttles. The value of the spike arrest should be calculated based on the capacity of the back-end services. The limits should be configured for shorter intervals such as sec or minutes. This feature protects the back-end services against sudden traffic burst coming from some malicious users or apps.
- **Rate Limit or Quota:** With a Rate Limiting approach (also sometime referred to as Quota), the requests are throttled based on the originating app or user, region of origination, time of the day and various other factors over a period of time. The request within the specified limit is routed successfully to the target system. Those beyond the limit are rejected. For example, if the quota is defined as 1,000 requests per day, all requests after the 1,000th request are rejected. It doesn't matter when these 1,000 requests are made. They could have been made in the first minute, or in the final minutes, or evenly paced. Additional requests are allowed only after the quota is

reset at the end of the time interval. The rate-limit values depend on the API product sold to the user. It controls the number of calls allowed for APIs in that product. For trail product, the limits might be less. For high-value products, the limits allowed could be more. Unlike spike arrest, rate limit allows calls to go through till the limit is reached. Hence, the rate-limit values should be carefully derived by looking at the overall capacity of the back-end systems and the expected load. The rate limit values are normally specified for a longer duration such as minute, hour, day, or month.

- **Concurrent back-end connections:** Sometimes legacy backend systems might have the strict restrictions on the number of connections that can be made. By implementing throttling using concurrent connections, you can limit the number of simultaneous connections that can be made from the API to the back-end services at any given point of time. Based on the value specified, the API gateway container controls the number of connections made to the back-end and rejects requests once the connection limit is reached. The limits to be set should be determined based on the capacity of the back end services.

Caching

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

Logging and Monitoring

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

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

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

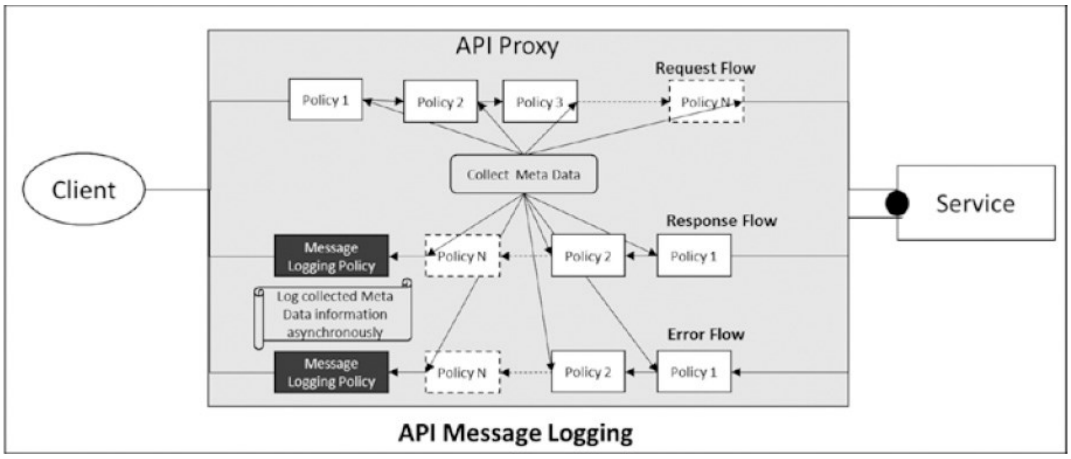

Figure 5-7: API message logging pattern

API Analytics

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

API Security Patterns

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

Common Forms of Attack

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

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

API Risk Mitigation Best Practices

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

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

Authentication and Authorization

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

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

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

Protect Against Attacks

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

Preventing APIs against denial-of-service attacks is another important security consideration. Most API management platforms provide protection against DoS attacks using Spike Arrest and Quota policies. The Spike Arrest policy identifies unexpected surge in the API traffics and reject all requests exceeding the configured limit. This maintains a uniform distribution of request flowing to the back-end systems as per their capacity. The Quota policy, on the other hand, restricts the number of API calls that a client app is allowed over a time interval. Alerts should be sent if APIs are getting overloaded or any suspicious pattern of API calls is detected. Using rate limits and quota policies alongs with a licensing model that establishes a contractual obligation between the API provider and the consumer app and enforces payments for violation of contracts, can minimize the risk of DoS attacks.

Encrypt Message Exchanges

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

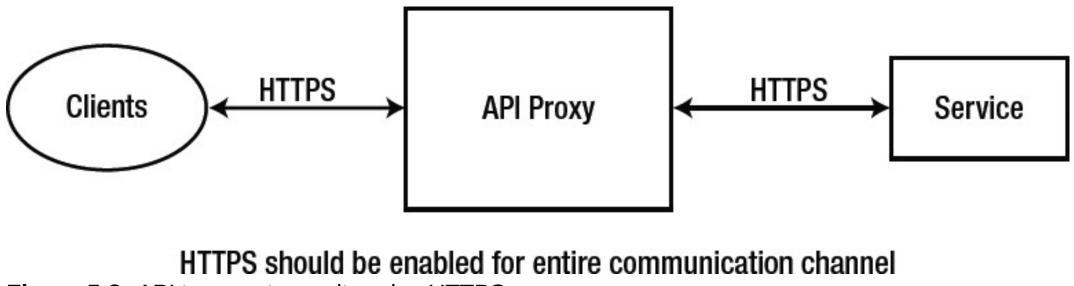

Figure 5-8: API transport security using HTTPS

Monitor, Audit, and Log API Traffic

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

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

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

Build API Security into the SDLC Process

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

Use a PCI-Compliant Infrastructure

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

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

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

API Deployment Patterns

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

Cloud Deployment

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

The main advantages of public cloud platform are as follows:

- **Higher reliability and availability:** Cloud platforms provide clustered environments that are distributed across multiple data centers and regions. This mitigates the risk of data center and network outages, and increases the reliability and availability of the platform. The API platform vendor handles traffic fluctuations and makes capacity adjustments to meet the guaranteed SLA.
- **Faster time to market:** The cloud instances of the API platforms can be spun off almost immediately by the API vendors. This saves time and hassle of hardware procurement, setup, and configuration. The cloud instances are up and running very quickly thus reducing the overall time to market for the API program.
- **Reduced capital and operational expenditure:** Cloud deployments are generally available in a subscription model. You pay by usage like number of API calls. This avoids upfront capital expenditures and reduces ongoing in-house operational costs.
- **Reduced management overhead:** Letting the API vendor focus on the data center infrastructure helps enterprises focus on building their API services. The API platform provider takes care of management over heads of running and managing the data center. They address all availability and performance management of the underlying infrastructure. Software updates and fixes are rolled out seamlessly by the vendors. The API provider can focus on creating the API and its back end.
- **Increased scalability and agility:** The licensing for the cloud platforms are generally by API traffic volume. If the traffic increases, API providers only pay additional licensing fee for the increased traffic. They do not need to bother about capacity planning, procurement of hardware, installation, configuration, and training needs for the operations personnel. The platform vendor makes the required changes to provision the additional capacity requested. This makes cloud environments ideal for horizontal scaling to meet the increased demand.
- **Regulatory compliance:** Often regulatory compliance requirements come in the way of adoption for cloud-hosted solutions. But most of the leading API management vendors have achieved industry compliances for their cloud-hosted platforms and their products. PCI DSS for the payment industry and HIPAA for the health industry are the most common industry compliance requirements. Since the platform is already compliant to the industry standards, it helps the client to easily meet the PCI requirements for security and log management on the cloud and other industry compliances.

These are the main disadvantages of cloud deployment.

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

On-Premise Deployment

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

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

API Adoption Patterns

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

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

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

APIs for Internal Application Integration

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

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

APIs for Business Partner Integration

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

APIs for External Digital Consumers

APIs have been adopted by enterprises to accelerate digital transformation, increase customer reach and loyalty, and discover new streams of revenue. Companies can now expose their business assets and service to a larger community of developers with an easy to use and intuitive API interface. External developers and partners adopt these APIs to build innovative apps. These apps can bring in a completely new business model for the enterprise. Hence, it is important for companies to create external-facing APIs to expose their data and services.

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

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

APIs for Mobile

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

APIs for IoT

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