**Chapter 7**

**API Management and API Client**

Sanjay Patnia\*

a UCSC Excention, Santa Clara, California, USA

**Abstract**

In this chapter we will start with Façade and review API Management requirements/solutions available.

In this chapter we will start with Façade and review API Management requirements/solutions available. Then we will continue with the framework and build demo client calls of RESTful APIs for the podcast application, followed by how clients need to be supported by Cross Origin Resource Sharing (CORS).

**Façade**

In this section we will first review the Façade design pattern and then in the second part we will get into details about how Façade is applied to the APIs.

**Façade Pattern**

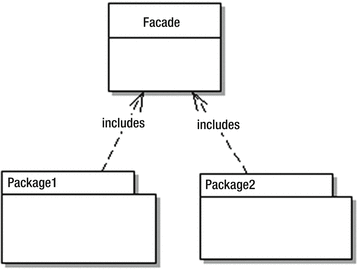
Before we discuss the Façade Pattern, let's consider what a façade is in the real world. The most obvious example is that of buildings, which all have an exterior to protect and decorate, hiding the internal workings of the interior. This exterior is the façade.

Now we can get closer to APIs by considering operating systems. Just like in buildings, an operating system provides an exterior shell to the interior functionality of a computer. This simplified interface makes an OS easier to use and protects the core from clumsy users.

This is where the definition of the Façade Pattern in On Design Patterns (Gamme, et al.) comes in handy:

*Provide a unified interface to a set of interfaces in a subsystem.* Façade defines a higher-level interface that makes the subsystem easier to use.

Consider Figure [7-1](#bookmark); you can see how the Façade Pattern puts an intermediate layer between the packages of the application and any client that wants to interact with them.



***Figure 7-1.*** *Façade Pattern*

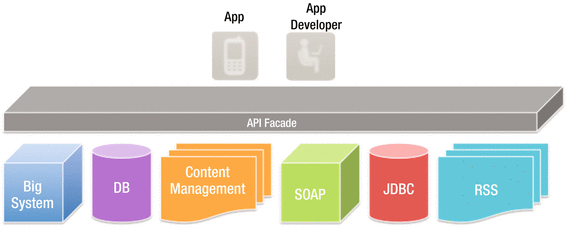
**API Facade**

Like all implementations of the Façade Pattern, an API Façade is a simple interface to a complex problem. Figure [7-2](#bookmark1) shows internal subsystems in an enterprise. As shown, each internal subsystem is complex in itself: for example, JDBC hides the inner workings of database connectivity.



***Figure 7-2.*** *Internal subsystems*

Figure [7-3](#bookmark2) shows an API Façade layer on the top of internal subsystems of the enterprise, providing a unified interface to apps.

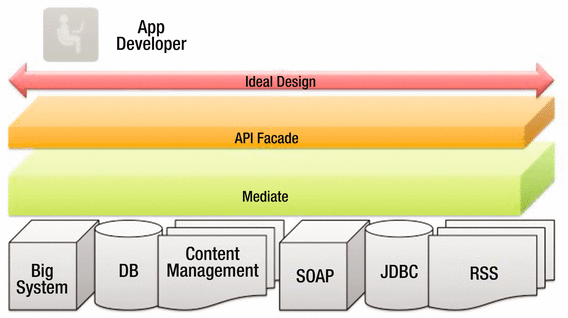


***Figure 7-3.*** *API Facade*

Implementing an API façade pattern involves three basic steps.

1. Design the API: identify the URLs, request parameters and responses, payloads, headers, query parameters, and so on.
2. Implement the design with mock data. App developers can then test the API before the API is connected to internal subsystems, with all the complications that entails.
3. Connect the façade with the internal systems to create the live API.

Figure [7-4](#bookmark3) shows these layers.



***Figure 7-4.*** *API Façade*

**API Management**

An API management tool provide the means to expose your API to external developers in an easy and affordable manner.

Here are the features of an API management service:

* Documentation
* Analytics and statistics
* Deployment
* Developer engagement
* Sandbox environment
* Traffic management and caching abilities
* Security
* Availability
* Monetization
* API lifecycle management
* API management vendors implement their solution in three different ways:
* Proxy. All traffic goes through the API management tool, which is placed as a layer between the application and users.
* Agents. These are plug-ins for servers. They do not intercept API calls like proxies.
* Hybrid. This approach picks features of proxies and agents, and integrates them. You can then pick which features you need.

**API Life Cycle**

The default API life cycle has the following stages:

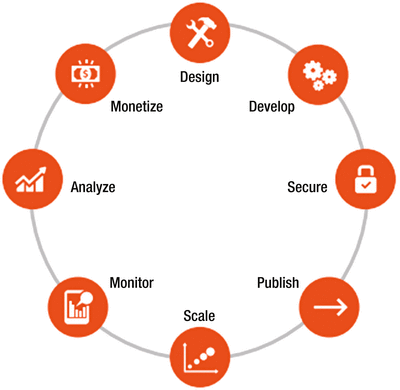
* ANALYSIS: The API is analyzed and mock responses are created for a limited set of consumers to try out the API and provide feedback. It’s also analyzed for monetization, as discussed in the following section.
* BEING CREATED/DEVELOPMENT: The API is being created: designed, developed, and secured. The API metadata is saved, but it is not visible yet, nor deployed.
* PUBLISHED/OPERATIONS: The API is visible and eventually published and is now in the maintenance stage, where it is scaled and monitored.

In addition, there are two more stages:

* DEPRECATED: The API is still deployed (available at runtime to existing users), but is not visible to new users. An API is automatically deprecated when a new version is published.
* RETIRED: The API is unpublished and deleted.

These are discussed in the next section.

Figure [7-5](#bookmark4) shows an API life cycle.



***Figure 7-5.*** *API life cycle*

**API Retirement**

As old age comes we get to retire, and the same is true with APIs. With time and due to the following reasons, API is retired or deprecated.

* Lack of partner or third-party developer innovation
* Losing market share due to expose of data by API
* Changes in technology stack e.g., REST replacing SOAP
* Security concern: making public API private due to security requirement of the information or data exposed by API
* Versioning: most common reason due to functionality changes

Some of the examples of API retirement are Netflix, Google Earth, and Twitter V1.0, etc.

**API Monetization**

Digital assets or services provide real value to customers, partners, and end users and hence they should be a source of revenue for your company, as well as an important part of your business model.

There are three business models for monetizing APIs:

* The revenue share model, where the API consumer gets paid for the incremental business they trigger for the API provider.
* The fee-based model, where the API consumer pays the provider for API usage.
* The third and final business model is freemium. Freemium models can be based on a variety of factors such as volume, time, or some combination; they can be implemented as standalone or hybrid models (in conjunction with the revenue share or fee-based).

**Cross-Origin Resource Sharing (CORS)**

“Cross-Origin Resource Sharing” (CORS) is a mechanism that allows Typescript or JavaScript on a web page to make XMLHttpRequests to another domain, not the domain the JavaScript originated from. Such “cross-domain” requests would otherwise be forbidden by web browsers, per the same origin security policy. CORS defines a way in which the browser and the server can interact to determine whether or not to allow the cross-origin request. It is more useful than only allowing same-origin requests, but it is more secure than simply allowing all such cross-origin requests.The Cross-Origin Resource Sharing standard works by adding new HTTP headers that allow servers to describe the set of origins that are permitted to read that information using a web browser.

How to implement CORS? Example:

return Response.ok() //200

.entity(quote)

.header("Access-Control-Allow-Origin", "\*")

.header("Access-Control-Allow-Methods", "GET, POST, DELETE, PUT").

allow("OPTIONS").build();