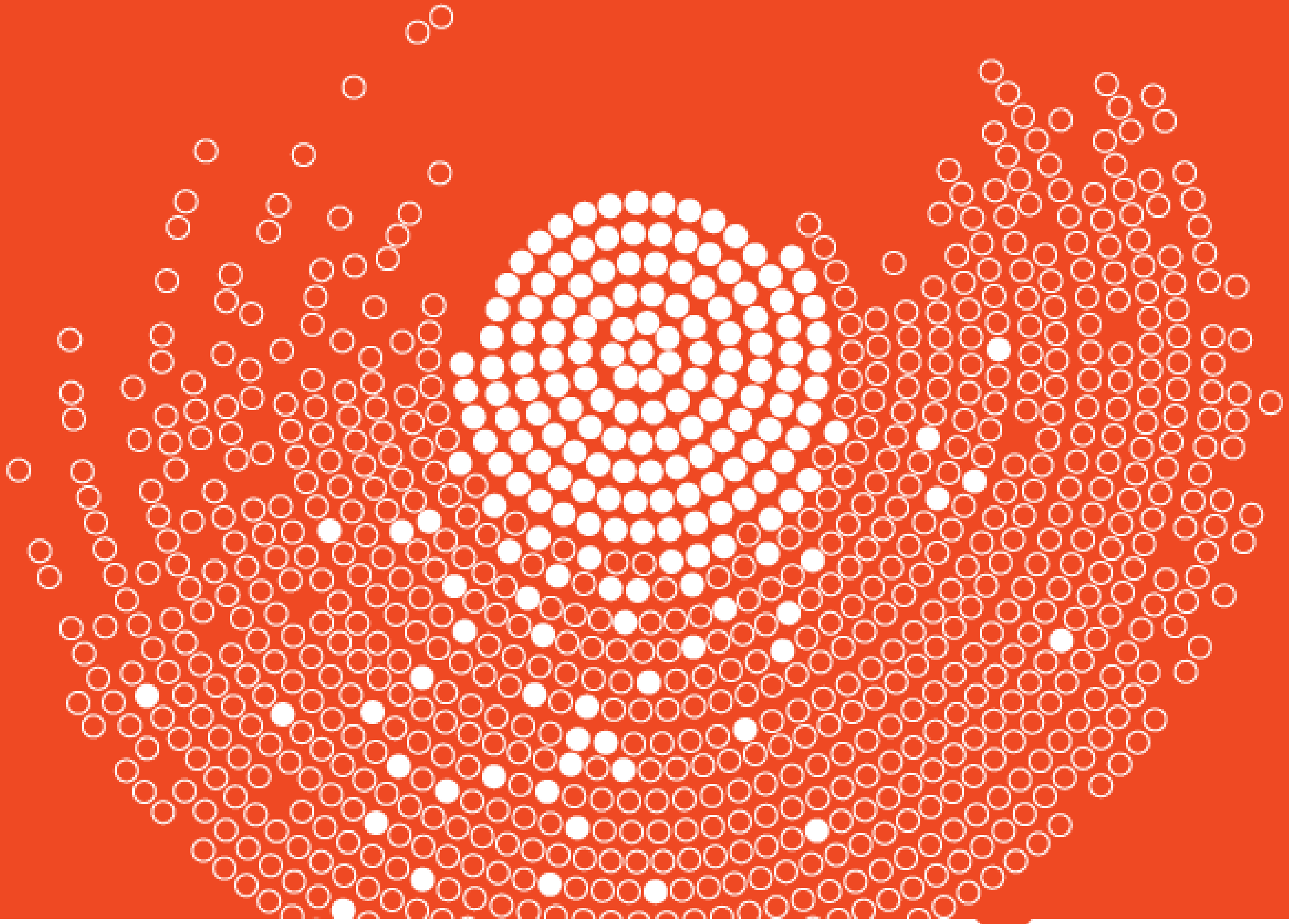
API Facade Pattern

A Simple Interface to a Complex System



apigee

Brian Mulloy

API Facade Pattern - A Simple Interface to a Complex System

Introduction

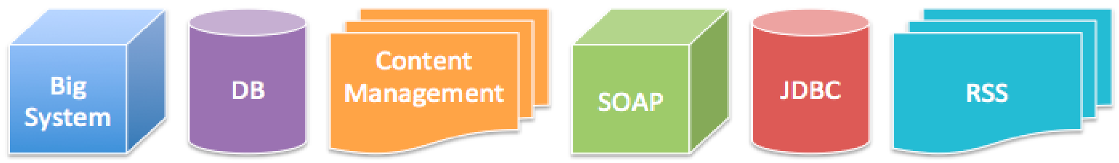
The API Facade Design Pattern is a solution to a design problem that arises for API designers when back-end and internal systems of record are too complex to expose directly to application developers.

Because app developers build apps using the APIs provided by an API Team (API provider), many businesses find that they need to craft solutions to deal with exposing complex systems.

It is beneficial to your API strategy if app developers are as productive as possible and adopt your API. This helps build value within your organization and extend that value proposition out beyond the boundaries of your organization and to the broader market.

The advantages of internal systems of record are that they are stable (have been hardened over time) and dependable (they are running key aspects of your business), but they are often based on legacy technologies and not always easy to expose to Web standards like HTTP. These systems can also have complex interdependencies and they change slowly meaning that they can't move as quickly as the needs of mobile app developers and keep up with changing formats.

In fact, the problem is not creating an API for just one big system but creating an API for an array of complementary systems that all need to be used to make an API valuable to a developer.



The goal of an API Facade Pattern is to articulate internal systems and make them useful for the app developer.

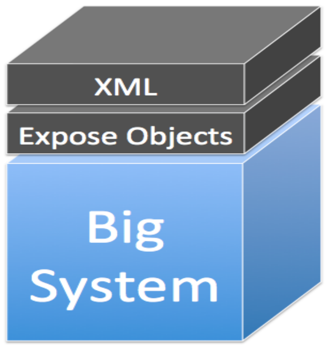
"USE THE FAÇADE PATTERN WHEN YOU WANT TO PROVIDE A SIMPLE INTERFACE TO A COMPLEX SUBSYSTEM. SUBSYSTEMS OFTEN GET MORE COMPLEX AS THEY EVOLVE."

Design Patterns – Elements of Reusable Object-Oriented Software
(Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides)

Patterns and anti-patterns

Let's start by looking at a few anti-patterns that we've seen people use when creating an API for a big system (or for an array of systems) and why we believe they don't work well. The first is what we'll call the "Build Up" approach.

The Build Up approach



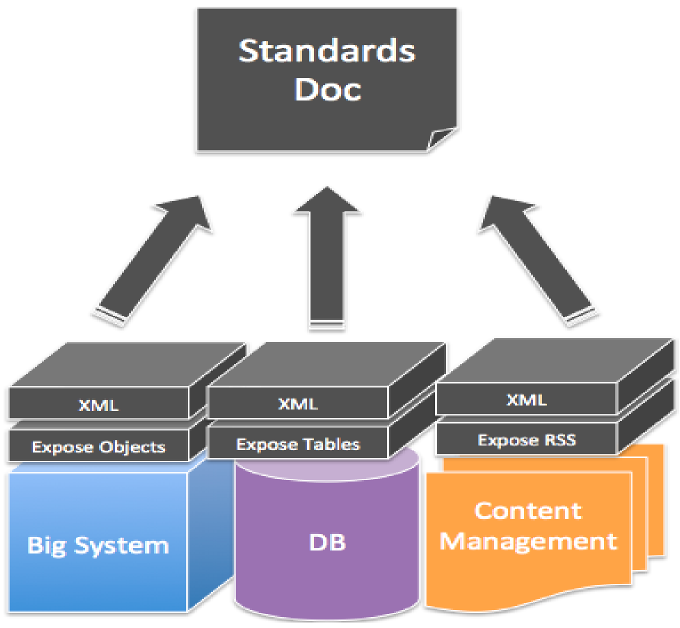
In the build-up approach, a developer exposes the core objects of a big system and puts an XML parsing layer on top.

This approach has merit in that it can get you to market with version 1 quickly. Also, your API team members (your internal developers) already understand the details of the system.

Unfortunately, those details of an internal system at the object level are fine grained and can be confusing to external developers. You're also exposing details of internal architecture, which is rarely a good idea. This approach can be inflexible because you have 1:1 mapping to how a system works and how it is exposed to API. In short, building up from the systems of record to the API can be overly complicated.

The standards committee approach

Often the internal systems are owned and managed by different people and departments with different views about how things should work. Designing an API by a standards committee often involves creating a standards document, which defines the schema and URLs and such. All the stakeholders build toward that common goal.

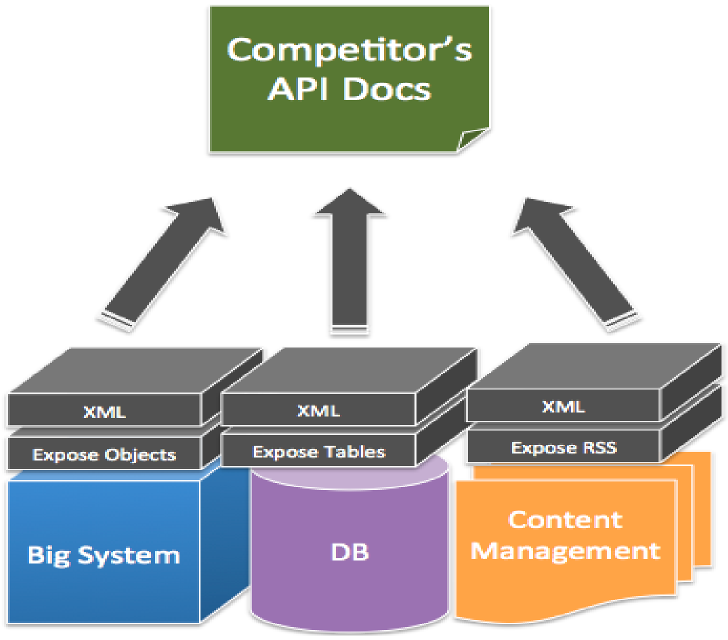


The benefits of this approach include getting to version 1 quickly. You can also create a sense of unification across an organization and a comprehensive strategy, which can be significant accomplishments when you have a large organization and a number of stakeholders and contributors.

A drawback of the standards committee pattern is that it can be slow. Even if you get the document created quickly, getting everybody to implement against it can be slow and can lack adherence. This approach can also lead to a mediocre design as a result of too many compromises.

The copy cat approach

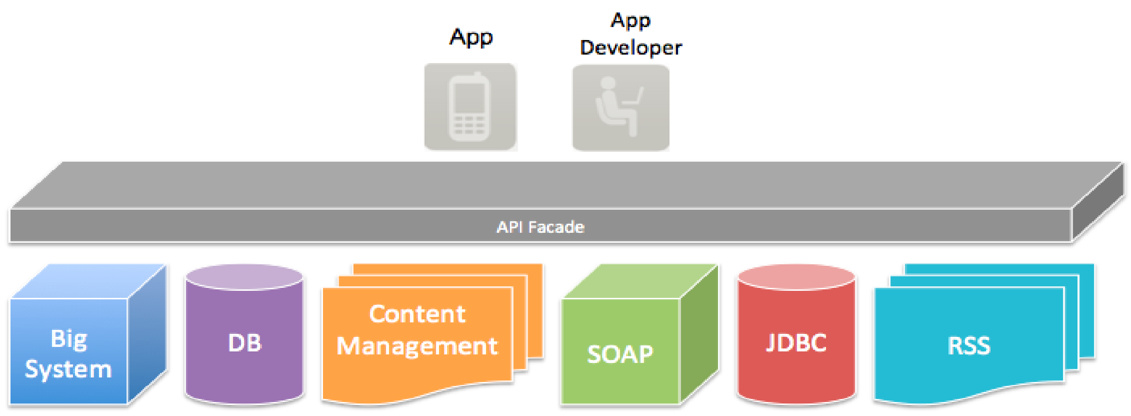
We sometimes see this pattern when an organization is late to market – for example, when their close competitor has already delivered a solution. Again, this approach can get you to version 1 quickly and you may have a built-in adoption curve if the app developers who will use your API are already familiar with your competitor's API. However, you can end up with an undifferentiated product that is considered an inferior offering in the market of APIs. You might have missed exposing your own key value and differentiation by just copying someone else's API design.



A simple interface to a complex system

The best solution to exposing internal systems in a straightforward and usable way for developers to consume starts with thinking about the fundamentals of product management. Your product (your API) needs to be credible, relevant, and differentiated. Once your product manager has decided what the big picture is like, it's up to the architects.

We recommend you implement an API façade pattern. This pattern gives you a buffer or virtual layer between the interface on top and the API implementation on the bottom. You essentially create a façade – a comprehensive view of what the API should be and importantly it is the view from the perspective of the app developer and end user of the apps they create.



Not only will you provide a simple interface to a complex internal system, but you'll also future-proof your environment. As Gamma et. al. observed in *Design Patterns – Elements of Reusable Object-Oriented Software*, systems often get more complex as they evolve and grow.

"USE THE FAÇADE PATTERN WHEN YOU WANT TO PROVIDE A SIMPLE INTERFACE TO A COMPLEX SUBSYSTEM. SUBSYSTEMS OFTEN GET MORE COMPLEX AS THEY EVOLVE."

Design Patterns – Elements of Reusable Object-Oriented Software
(Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides)

Designing the interface

Let's look at the basic steps to implement an API Façade - our recommended solution to the problem of exposing complex internal systems' functionality in a way that's useful for app developers is to implement an API facade pattern.

This pattern gives you a buffer or virtual layer between the interface on top and the API implementation on the bottom. You essentially create a façade – a comprehensive view of what the API looks like from the perspective of the app developer and end-user of the apps they create.

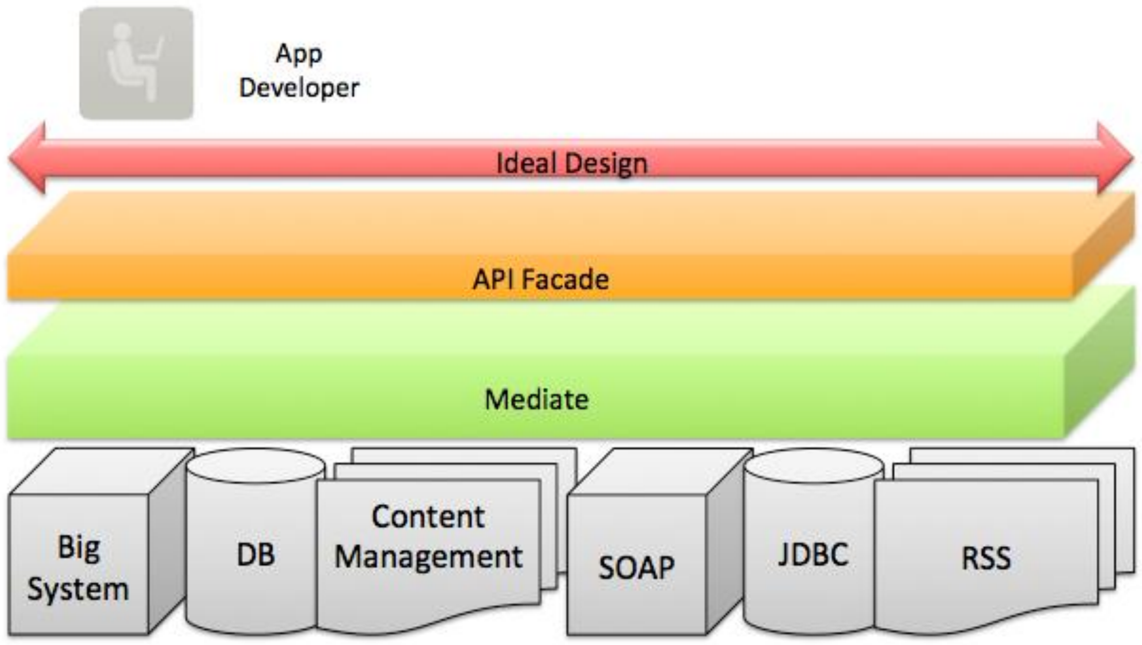
The developer and the app that consume the API are on top. The API façade isolates the developer and the application and the API. Making a clean design in the facade allows you to decompose one really hard problem into a few simpler problems.

Implementing an API façade pattern involves three basic steps.

1 - Design the ideal API – design the URLs, request parameters and responses, payloads, headers, query parameters, and so on. The API design should be self-consistent.

2 - Implement the design with data stubs. This allows application developers to use your API and give you feedback even before your API is connected to internal systems.

3 - Mediate or integrate between the façade and the systems.

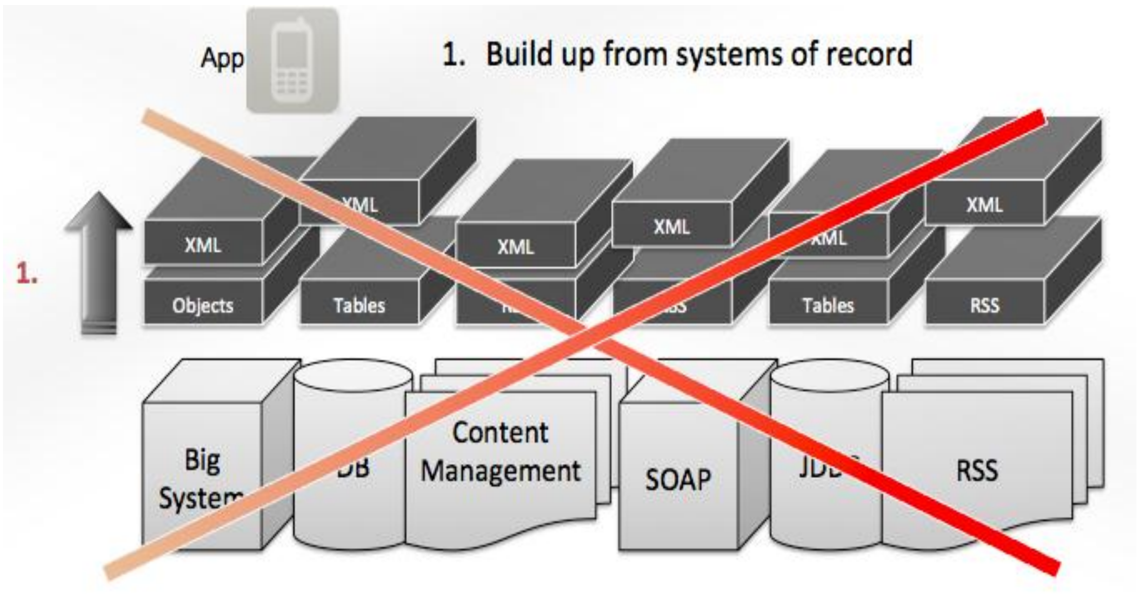


From one big problem to three small problems

Using the three-step approach you've decomposed one big problem to three smaller problems. If you try to solve the one big problem, you'll be starting in code, and trying to build up from your business logic (systems of record) to a clean API interface.

You would be exposing objects or tables or RSS feeds from each silo, mapping each to XML in the right format before exposing to the app. It is a machine–to-machine orientation focused around an app and is difficult to get this right.

API Facade Pattern - A Simple Interface to a Complex System



Taking the façade pattern approach helps shift the thinking from a silo approach in a number of important ways. First, you can get buy in around each of the three separate steps and have people more clearly understand how you're taking a pragmatic approach to the design. Secondly, the orientation shifts from the app to the app developer. The goal becomes to ensure that the app developer can use your API because the design is self-consistent and intuitive.

Because of where it is in the architecture, the façade becomes an interesting gateway. You can now have the façade implement the handling of common patterns (for pagination, queries, ordering, sorting, etc.), authentication, authorization, versioning, and so on, uniformly across the API. (This is a big topic, which we'll delve into later.)

Other benefits for the API team include being more easily able to adapt to different use cases regardless of whether they are internal developer, partner, or open scenarios. The API team will be able to keep pace with the changing needs of developers, including the ever-changing protocols and languages. It is also easier to extend an API by building out more capability from your enterprise or plugging in additional existing systems.

Common patterns

An API façade provides repeatable patterns to help design the common and critical elements of your API. You'll see by examining a handful of common patterns that you can surface simple interfaces to complex systems in a number of contexts, and in predictable and reusable ways. We'll look at the following:

- Errors
- Responses & data stubs
- URLs
- Versions
- Data formats

Errors

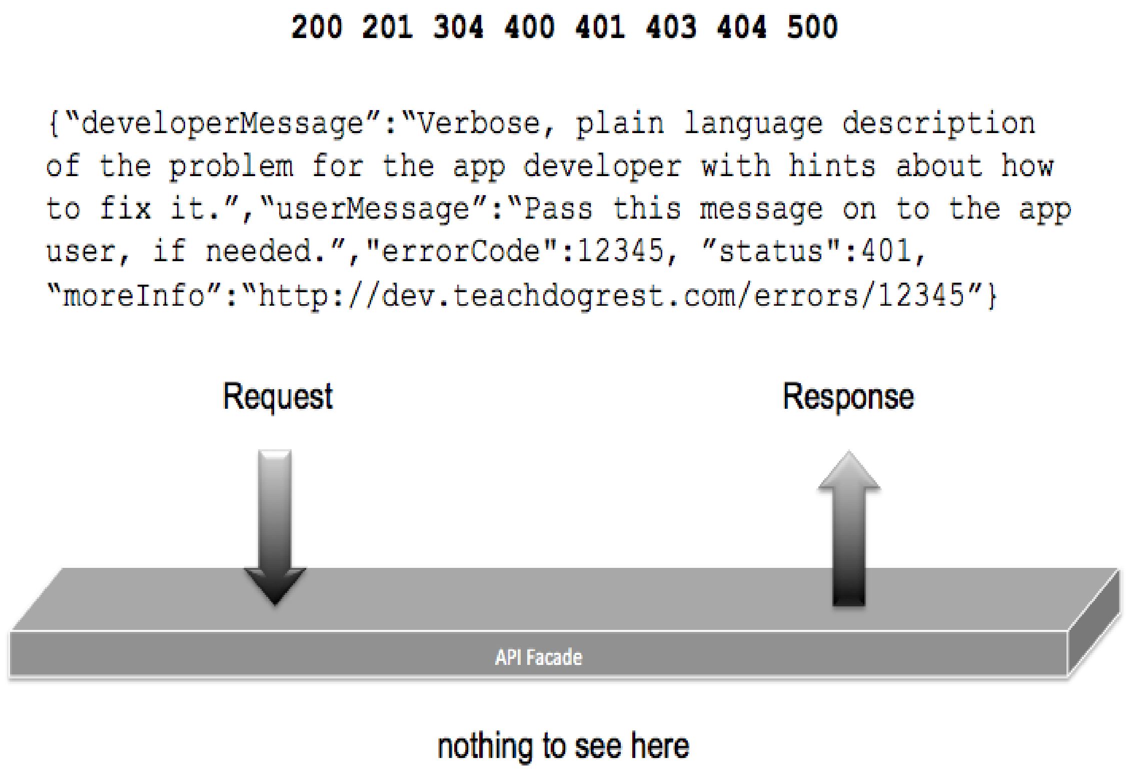
"When I say errors, you say test-driven development"

Test-driven development involves building test cases and when they fail they help guide developers towards creating the right app. Because the black box is more stringently enforced with a Web API, this model and the errors are more important in the world of APIs and apps than in other areas of software development.

Design the HTTP codes and responses you want

Start with a facade, not connected to any internal systems yet and then get the error codes right. In *Web API Design: Crafting Interfaces that Developers Love*, we talked about the 8 error codes we think are most important (shown below too). (The important error codes may vary for your domain.)

Get the error messages in place so that you can start to build your test suite, including the HTTP error codes and the payload response.



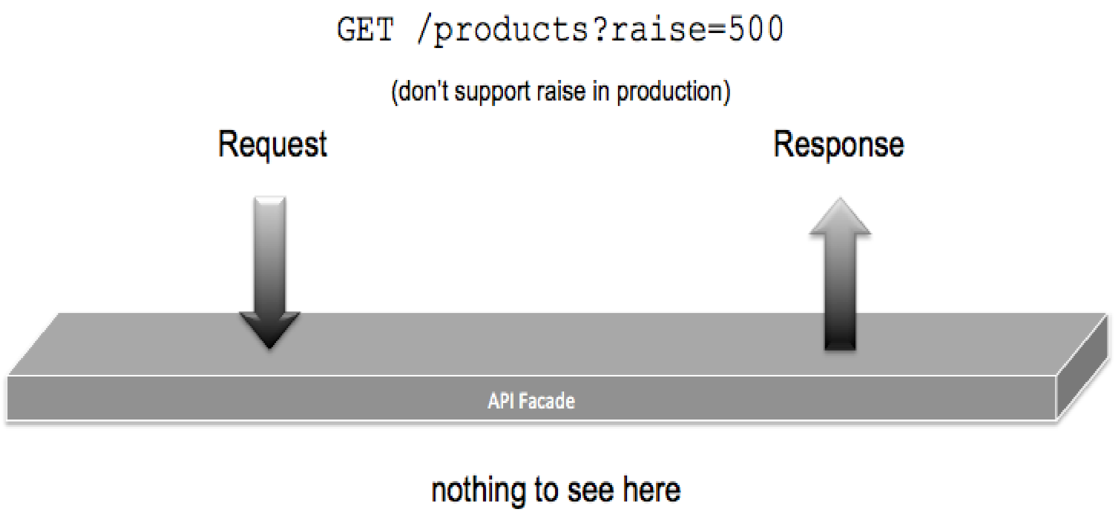
```
200 201 304 400 401 403 404 500

{"developerMessage":"Verbose, plain language description
of the problem for the app developer with hints about how
to fix it.","userMessage":"Pass this message on to the app
user, if needed.","errorCode":12345, "status":401,
"moreInfo":"http://dev.teachdogrest.com/errors/12345"}
```

Control over the facade - raise the error

Sometimes you want to explicitly cause a raise to happen. Here's a useful trick and pattern we created for a large API provider, which worked well. It involves putting a raise query parameter in your HTTP request.

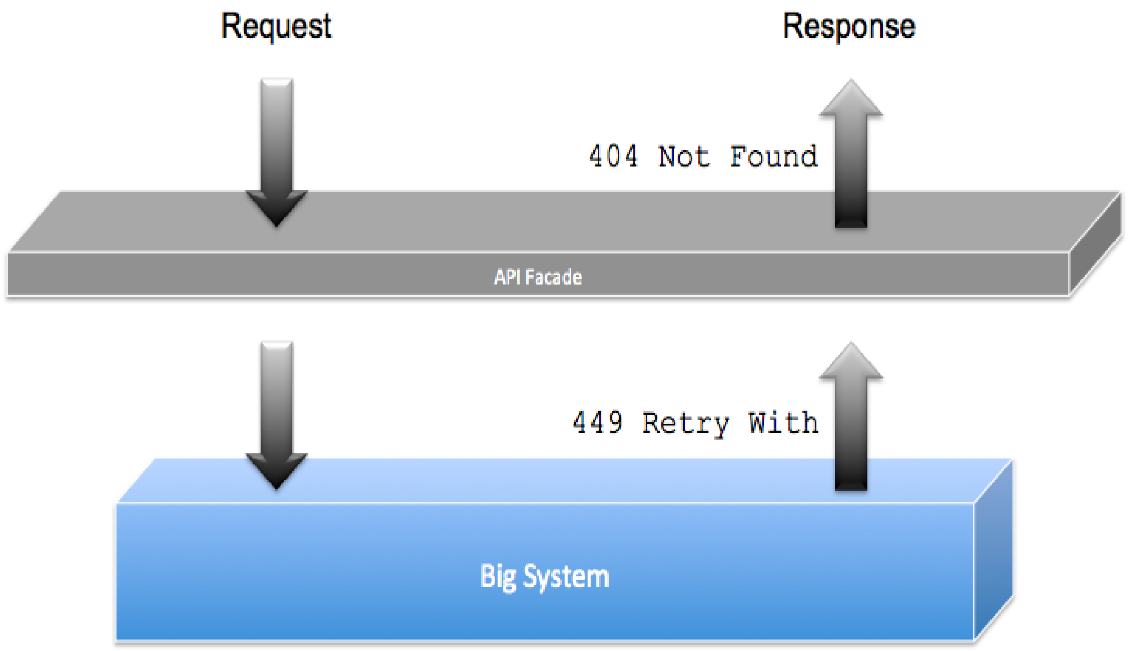
It will raise that HTTP code. When you are building your test suite, this allows you to ensure your app logic handles exceptions properly.



Warning - don't let this make it's way to your production servers.

Test and implement - plug internal system into the facade

You've designed your set of HTTP codes from the outside in. You have a big internal system, which let's say was built on the .NET framework. Microsoft has an extension of the HTTP status codes - `449 Retry With`. You will want to map the 449 to something more aligned with what mobile developers are familiar with today. To do so, you can implement a lookup table and transform the 449 code into a 404 error.

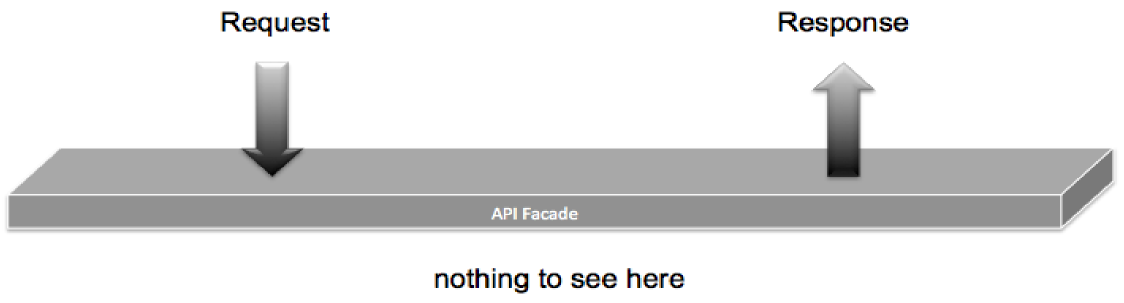


So, you started with the design intent - the HTTP codes and responses. You've exerted some controls over the facade by explicitly forcing the raise. And finally, you've tested it all by plugging internal system into the facade.

Responses & data stubs

In the same way we designed for errors with the facade unconnected to any back-end systems, you can stub out what response data would look like, and have the facade return that to you.

```
{"products”:
  [
    {"product":{"id":"1234”,"name":"Widget”,
"color":"white"}},{"product":{"id":"1235”,
"name":"Gadget”, "color":"brown"}}
  ]
}
```

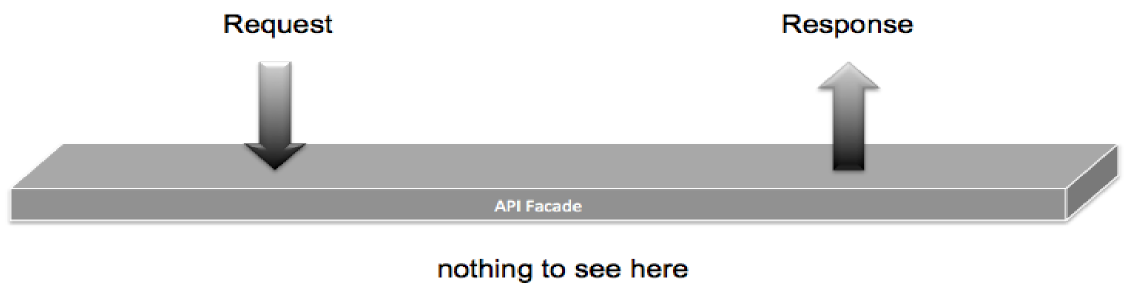


In the same way we designed the forced raise for errors, you can force the mock. Setting mock = true, you have a shunt in the facade that returns the stub. Again, we're looking at predictable behavior to do test driven development.

Warning: It's a good idea to only support the mock attribute on the test server and to raise an error if the mock parameter is included in production.

```
GET /products?mock=true
```

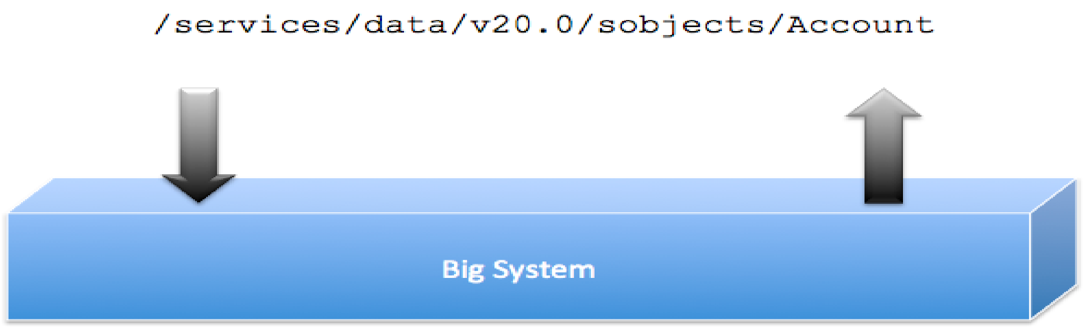
(don't support mock in production)



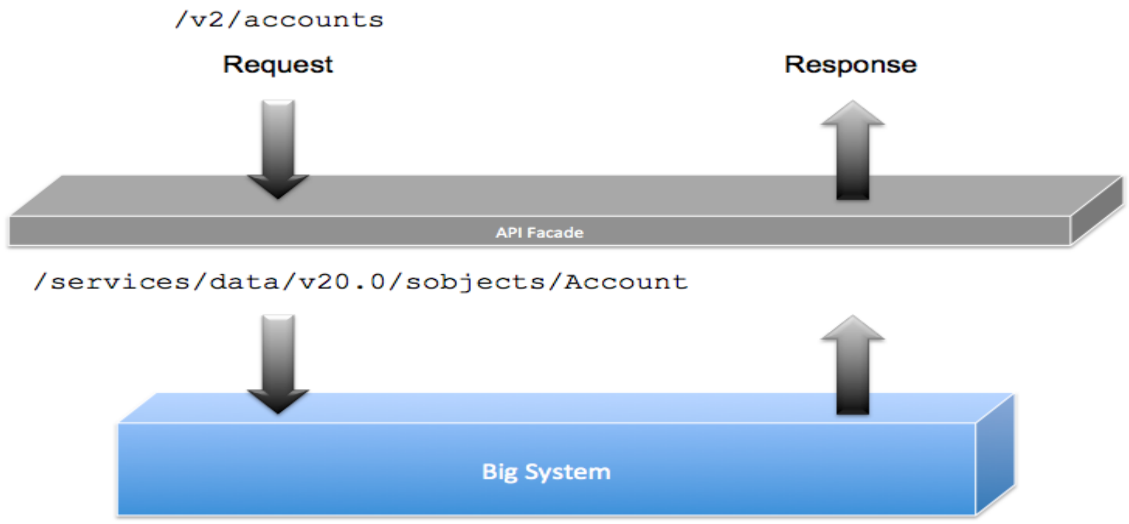
URLs

We think of the URL as the strongest affordance for a well-designed API. This is where the facade pattern begins to shine.

The goal is something like this - an app developer wants to do something as simple as see a collection of accounts by doing an HTTP GET on /v2/accounts. However, the internal system may be far more complex than /v2/accounts, like this Salesforce URL.



Doing URL mediation through the facade, you can show the developer on the outside the simple /v2/accounts interface while keeping the complexity of the internal system behind the facade.

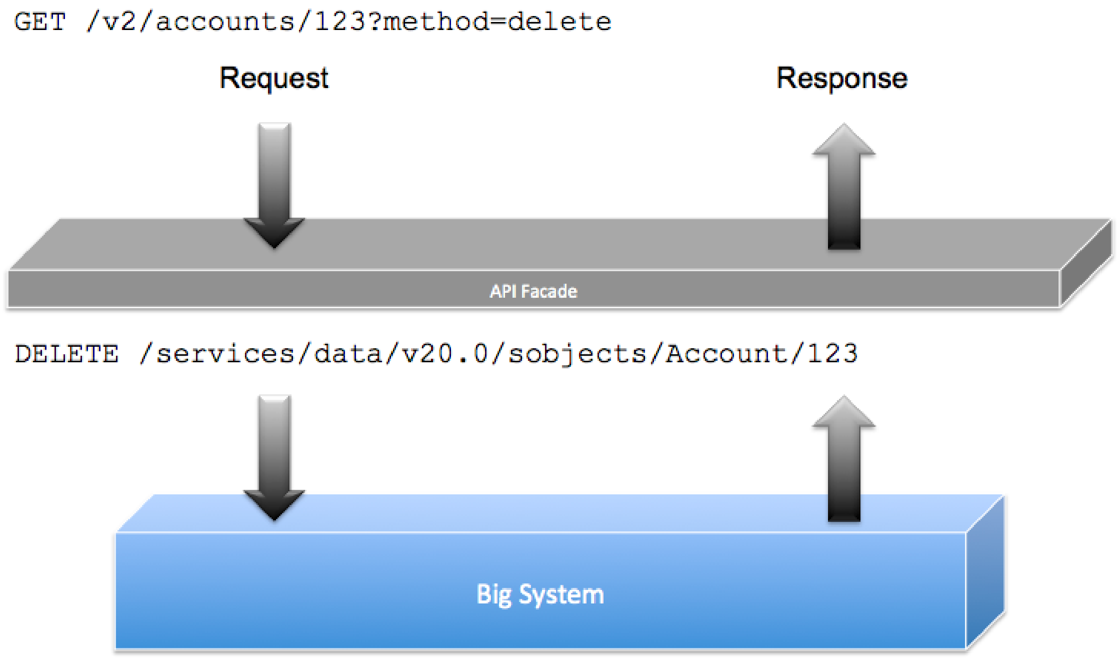


Limited clients

Certain clients have limitations on the HTTP methods that they support. This is a good scenario to consider in the context of the URL pattern.

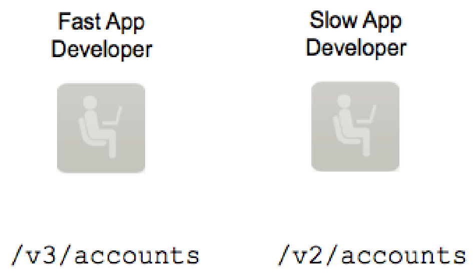
Take for example a client app that doesn't support HTTP DELETE.

You can handle this through the facade by making the method an optional parameter. As the request comes into the facade, the facade changes the HTTP method from GET to DELETE. It also strips the method=delete query parameter and translates to original request of the backend system.

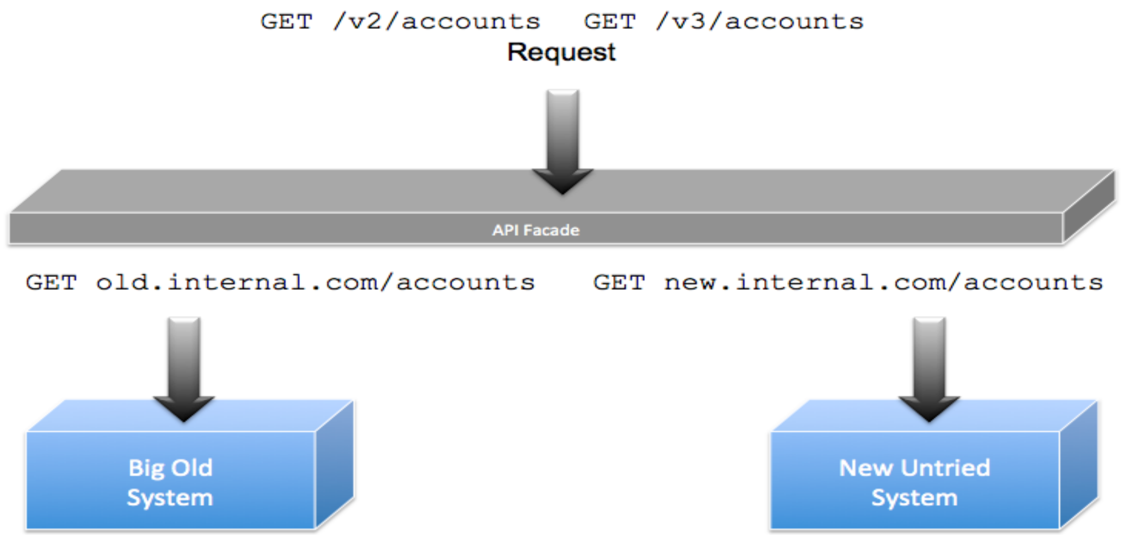

GET /v2/accounts/123?method=delete
Request
Response
API Facade
DELETE /services/data/v20.0/sobjects/Account/123
Big System

Versions

Best practices and principles for versioning your API are described in RESTful API Design: Tips for versioning and in the *Web API Design: Crafting Interfaces that Developers Love* e-book. Here we'll focus on designing for a scenario in which you need to support more than one version. This is common scenario especially in certain phases of your API's life cycle.



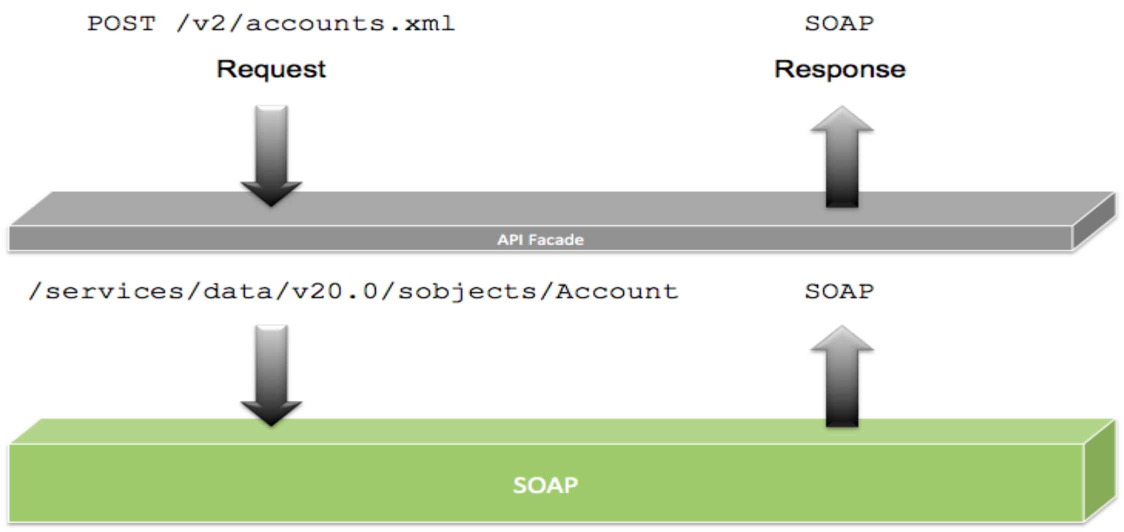
The way to handle this with a facade is to design it such that regardless of which request comes into the facade, you have a shunt in place that points the request to the proper internal system, which serves the response.


GET /v2/accounts GET /v3/accounts
Request
API Facade
GET old.internal.com/accounts
GET new.internal.com/accounts
Big Old System
New Untried System

Data formats

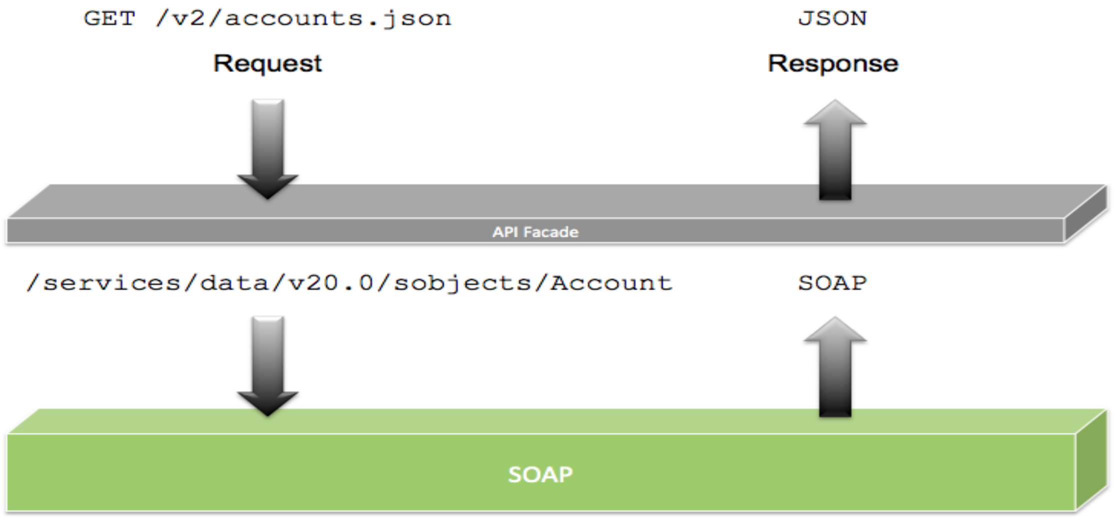
Different developers have different expectations for formats. For example, an HTML5 developer might want JSON responses while a Java developer might depend on libraries to handle SOAP requests and responses.

Let's look at how a facade handles this for a developer who needs SOAP. Take a developer who does a POST to get a collection of accounts. The facade mediates the POST into the more complicated internal system and returns SOAP. This is a simple scenario not unlike the URL mapping scenario. There's no real data format mediation happening here.



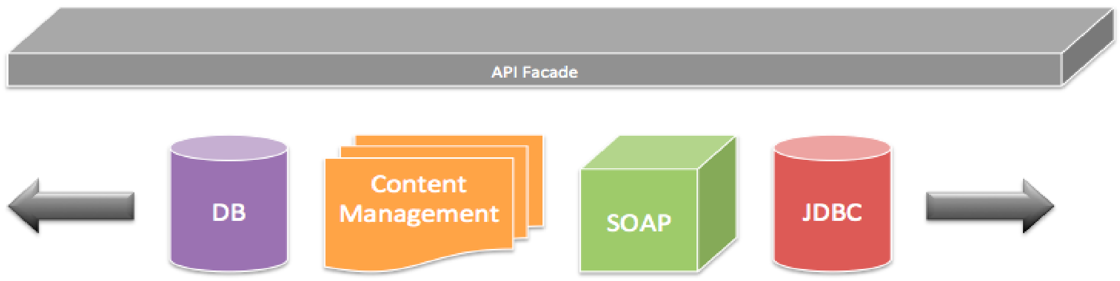
A more complex mediation happens for example when the developer does an HTTP GET and wants JSON in the response. The facade maps SOAP to JSON on the response, probably using XSLT. Note that the developer has no knowledge of the complexity or the mediation and only knows that they are getting the right format data returned for their app.

API Facade Pattern - A Simple Interface to a Complex System



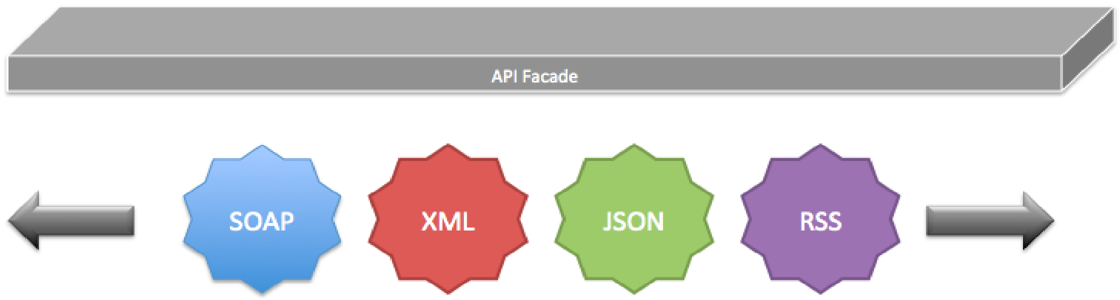
Facades for internal and external systems

So far, we've talked about API facades that make it easy to provide access to internal systems, and a lot of companies use a facade in this mode.



Another use of the same pattern is to easily consume APIs from external systems.

All of the same issues and considerations come into play with internal and external systems. We've seen a number of cases in which core businesses rely on external services. We've also seen cases where those service providers change their pricing models and with that the business that is tied to the service provider can see its profit margins shrink, it's SLAs deteriorate, or other business relationship change. A facade pattern can help mitigate risk in cases like this.



With a facade pattern in place, apps that consume the external services can expect a canonical model to consume the APIs. Also, if implemented through the facade, the external services can easily be plugged and replaced.

All of the same approaches to implementing common patterns come into play for the external scenario - starting with building out the errors facade and the data stubs. You'll quickly create your canonical model of service consumption.

Technology

What are the technologies at the heart of implementing an API façade? You'll need to set up domains and servers, gateways, sub-domain routing. Then you'll need to design for versioning, firewalls, caching and finally orchestration, transformation, compression, and authorization.

Technology for set-up

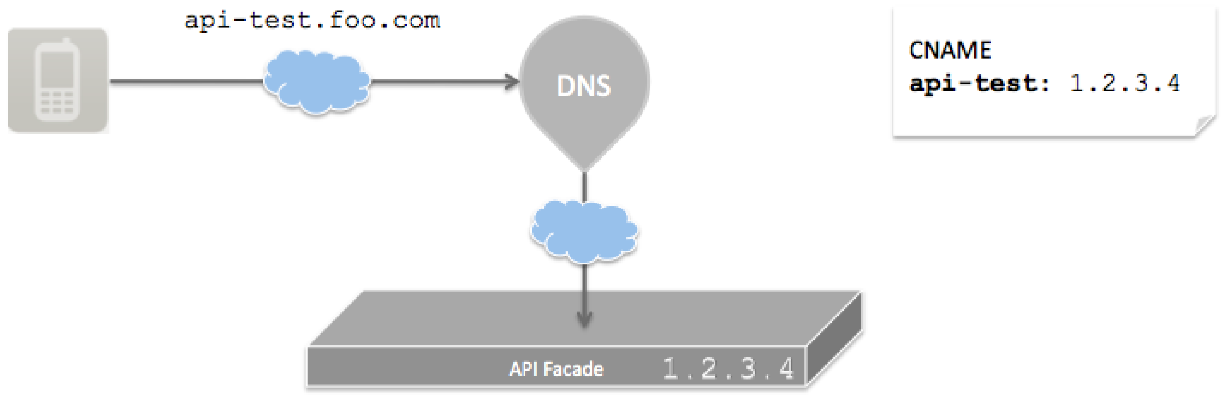
We'll begin with the set up involving DNS, Cloud Platform, Web server, app server, API Gateway and subdomain routing.

DNS provider & cloud platform

In the spirit of test-driven development, the first thing to set up is our test environment. The first piece of technology we'll need is a DNS Provider.

Set up a CNAME entry, which points to our test facade. A good choice for the subdomain is api-test.

For the sake of our example, we're assuming a Cloud Platform technology because it's the most complicated case and will allow us to explore the most options. It's definitely simpler if everything is behind your firewall.

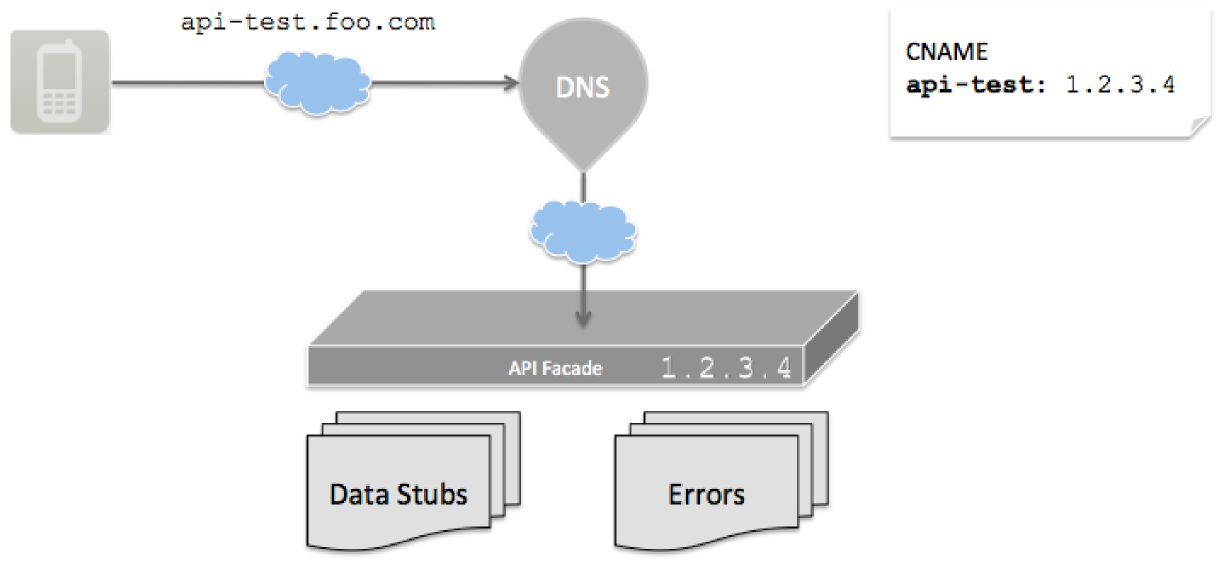


Web server, app server, and API gateway

Once you have the DNS and the Cloud Platform set up, you are ready to implement the first patterns, errors and data stubs.

To ensure that you have a solid foundation for test-driven development, HTTP codes and error responses need to be in place; you need to be able to stub out data to support `mock=true` and `raise={HTTP code}`, and so on.

To do so, you need either a static Web server, or an app server for more dynamic content, or an API Gateway (the Swiss army knife for this scenario) in the Cloud Platform.



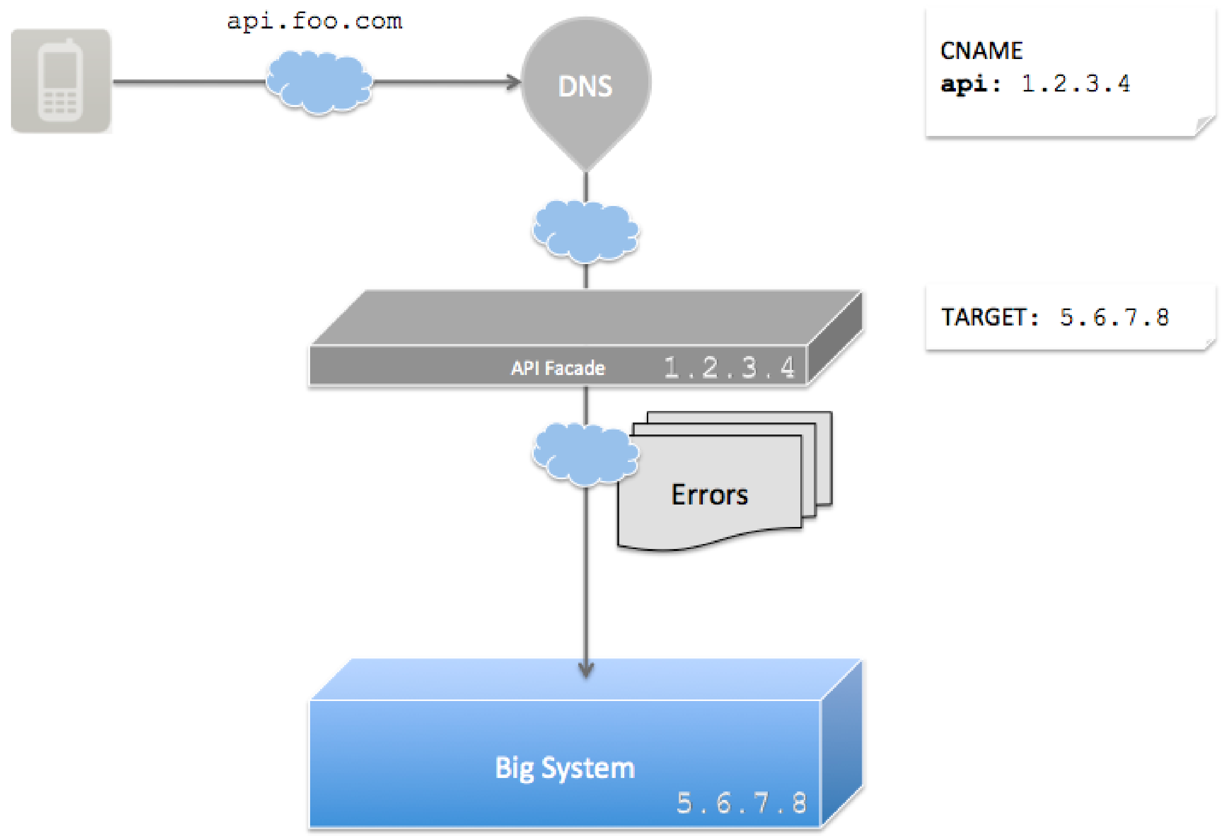
Support for `mock=true` and `raise=404`

Subdomain routing and production environment

The next step is to set up your production environment. You'll add a new CNAME entry in the DNS so that the production subdomain is **api**. In our example below, we've pointed api to the same Cloud Platform as in our test environment, but of course you can have different test and production targets if you like.

In the API facade, you'll specify the IP address of the target system.

Note that the error capability is also here in the production environment. Just as it is important in the test environment, you need to ensure verbose error messages and comprehensive codes in the production environment.



With these pieces in place, you'll have requests coming in to `api-test` and `api`. In addition to the target IP address, the facade has a shunt that knows where the subdomain is and understands where to point - for example to the data stub server for test or to an internal system for production.

Technology for versioning, firewalls, caching

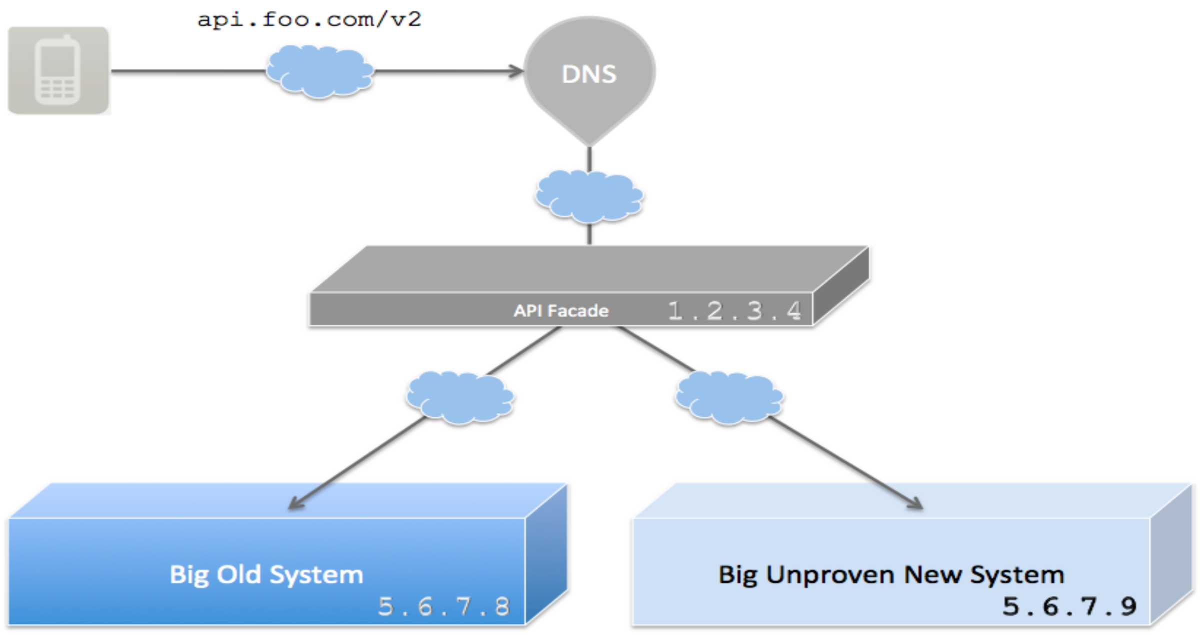
Once you’ve up your test and production environment with DNS settings, a Cloud Platform, Web servers, app servers and an API Gateway, you’re ready to think about the technologies required to handle some of the more common use cases including versioning, caching and securing with a firewall.

Versioning and URL routing

Once you've got subdomain routing taken care of as discussed in our previous section *Technology for Set Up*, you can look at designing to handle multiple versions.

This is a similar scenario to subdomain routing but in this case you're doing URL parsing. Here for example, a request comes in for v2.

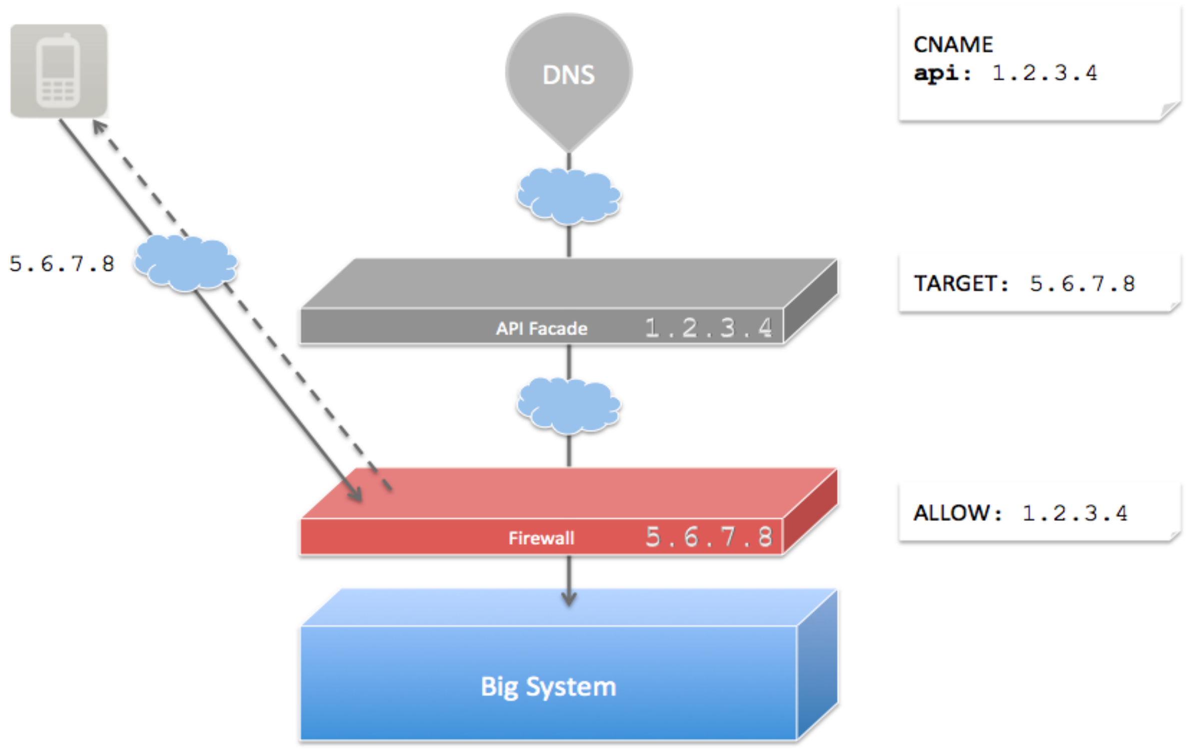
V1 of your facade may point at an old system, while v2 points at a new system. In this way, you have a simple way to shunt between two IP addresses and handle the scenario in which you need multiple versions of your API available.



Firewall

You want app requests coming through the API facade. You don't want anyone figuring out the IP address to which your facade is pointing and bypassing it. If the API facade is bypassed, you'll be unable to track the requests and unable to apply the API design logic you've built into your facade.

To counteract this, you create a firewall to block all the API traffic with the exception of the trusted IP address of the facade. That is, you ALLOW the IP address of the facade in firewall. Your system is then secure with all requests coming through the facade.

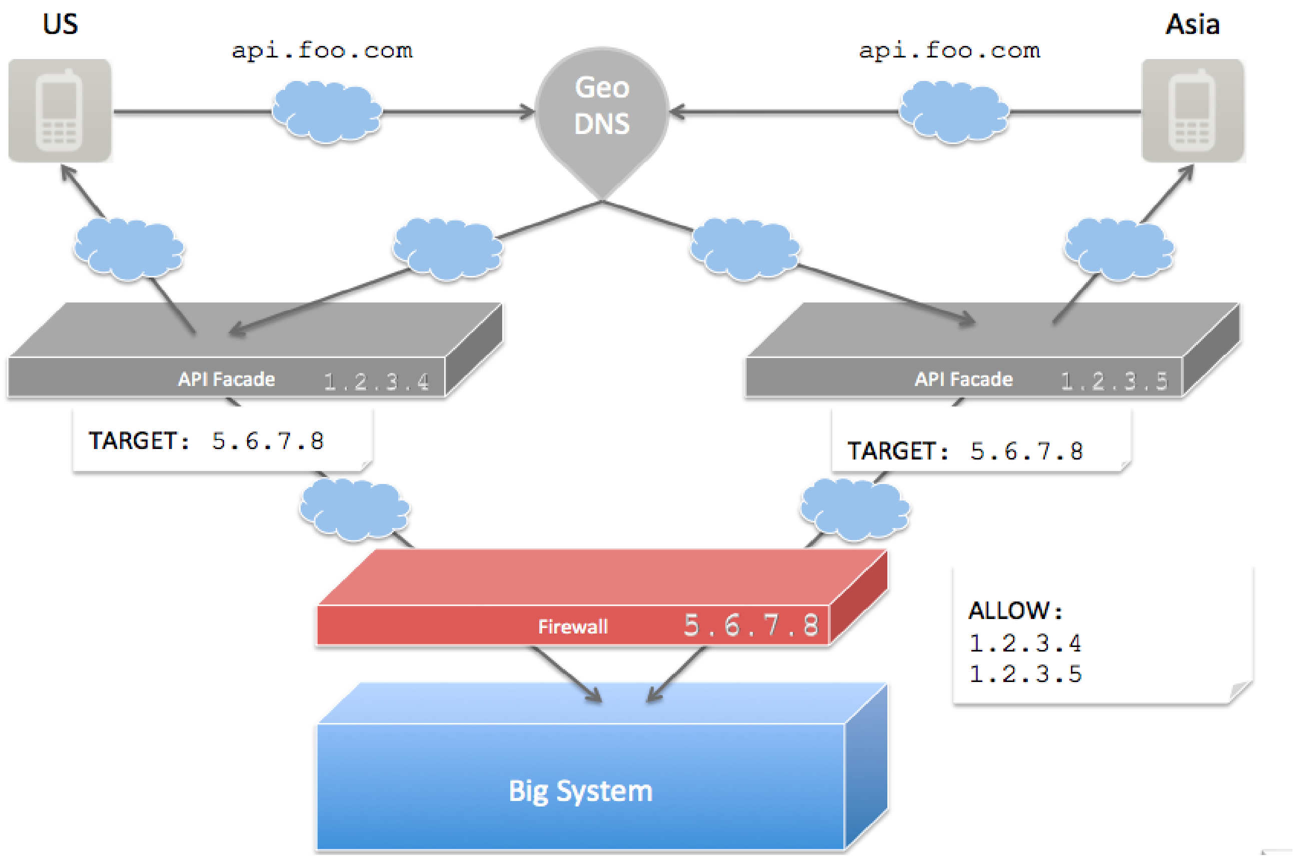


Geo DNS and Caching

In this scenario, we'll build out our facade functionality some more by adding a geo-distributed DNS. The DNS sends the app to a geographically close API facade, based on the source of the request.

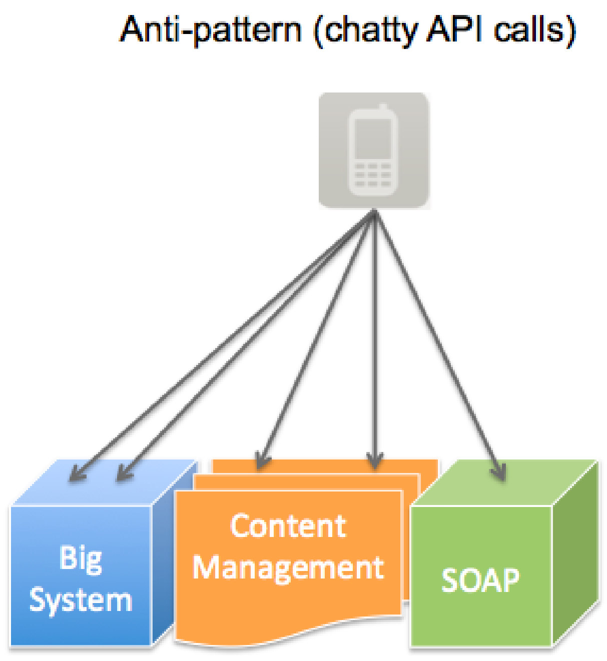
The number one use case for the geo DNS is caching. This is especially important functionality for apps that have a social network element. You can cache information that doesn't cross-regions so that clients enjoy a fast experience because the facade is caching the API responses where the requests originated.

So you've added geo DNS and in the API facade have caching capability. You've told the DNS that based on region; you have 2 different IP addresses to target - either 1.2.3.4 or 1.2.3.5 in our example.

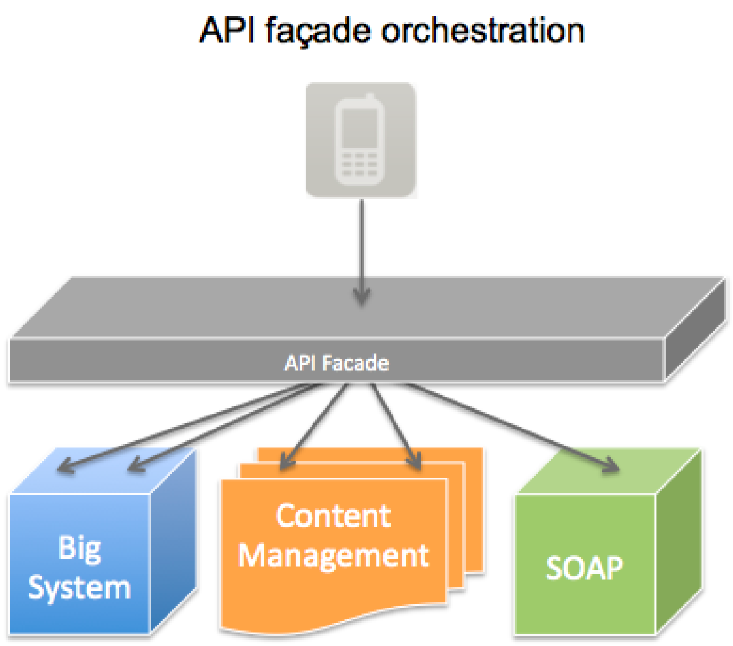


Technology for orchestration

Another common pattern we see is the insertion of a facade to orchestrate across a complex and fine-grained set of API calls. The design represented by the anti-pattern diagram below will likely present a poor and complex design to the app developer.



The facade in this scenario orchestrates across a number of calls. There are configuration- or policy-orientated orchestration technologies available or you can write the orchestration logic in code. Code is often one of the best orchestration tools but the approach you take will depend on the skills and capacity of the team that's implementing the API facade.

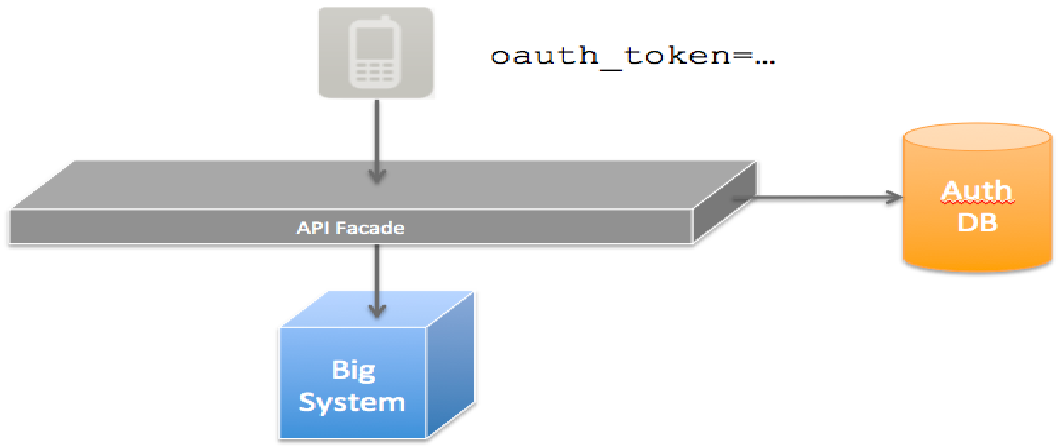


Technology for Authorization - OAuth

Let's look at how to handle authorization through the API facade. A request comes into the facade with an OAuth token or other indications of the authorization scheme.

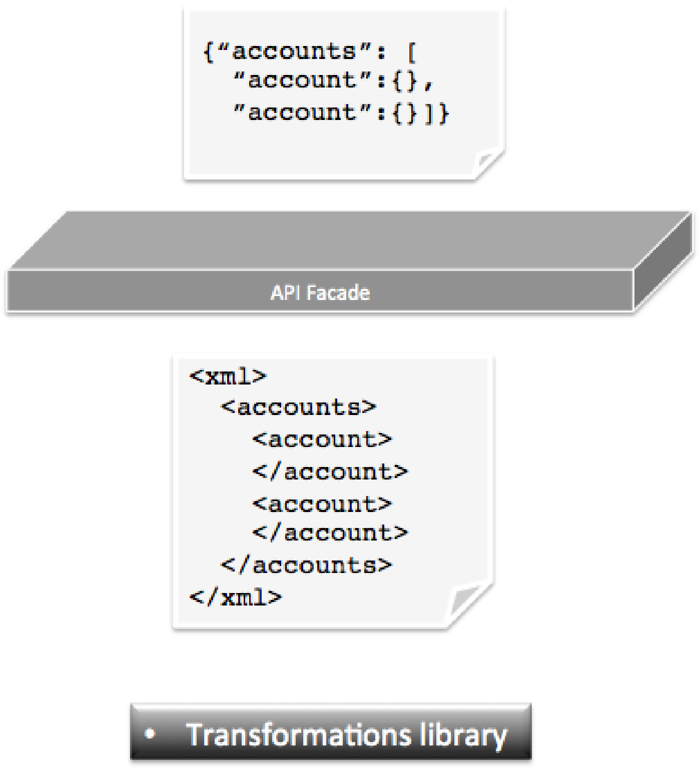
The facade can make the calls to the authorization system of record.

If the request is valid, the facade passes it to the core system; if invalid, the facade returns with an invalid response code.



Technology for transformation and compression

To handle the common use case for apps, in general, of transforming XML to JSON or vice versa, you would use a transformation library (like XSLT) in the facade.

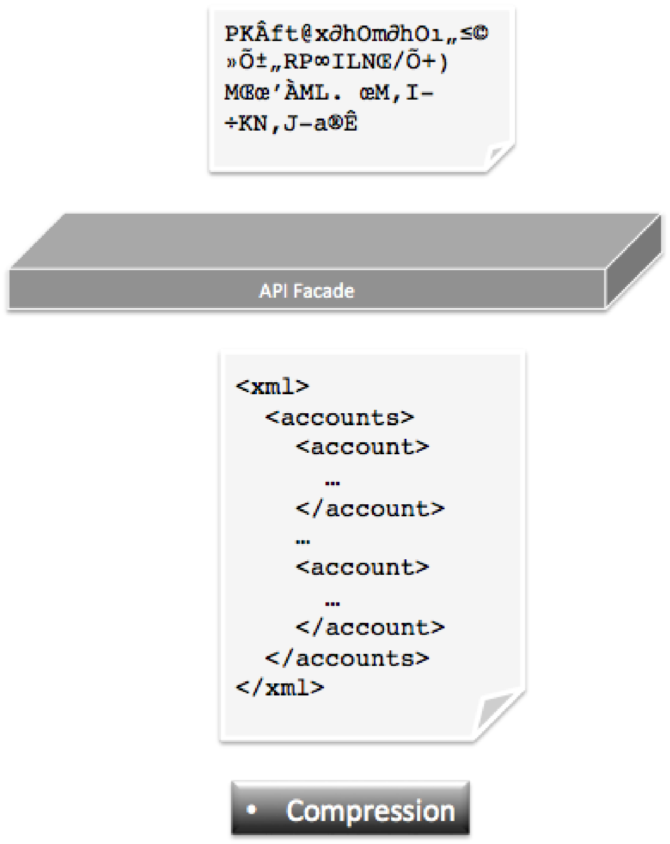


```
{"accounts": [
   "account":{},
   "account":{}]}
```

```
<xml>
   <accounts>
      <account>
      </account>
      <account>
      </account>
   </accounts>
</xml>
```

To handle a common use case for verbose and large XML documents, you add a compression engine to your API facade.

Not clogging the bandwidth is especially important for mobile apps where large payloads become a problem because they impact the cost of users' data plans and use battery powering the radio. Apps that impact the users' bottom line get uninstalled.


```
PKÂft@x∂hOm∂hOı„≤©
»Õ±„RP∞ILNŒ/Õ+)
MŒœ'ÀML. œM,I-
÷KN,J-a®Ê
```
```
<xml>
  <accounts>
    <account>
      …
    </account>
    …
    <account>
      …
    </account>
  </accounts>
</xml>
```

An additional layer of complexity?

One reaction to the facade pattern is that it adds a layer of complexity to the technology stack. However, whether or not an organization uses the facade pattern, there is inherent complexity, which must be addressed between the app developer and the API.

Too often this complexity is buried within individual systems and so is difficult to track. Adding complexity on top of unknowns is bad. The facade pattern actually helps to create order where there was none.

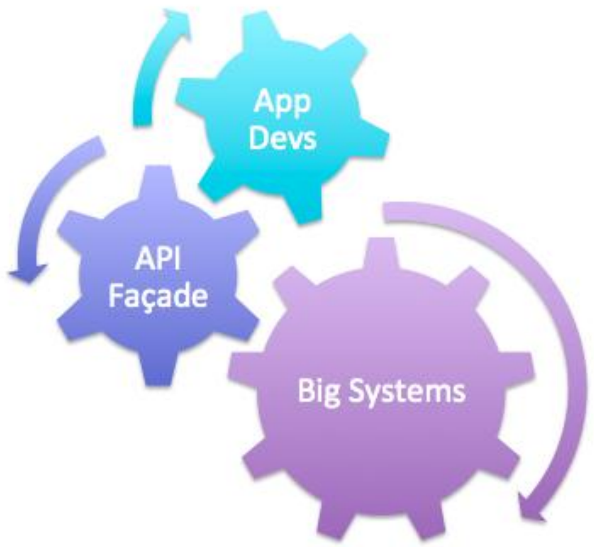
People and agility

It’s easy to focus on the technology but the people involved in your API strategy are key to its success. Although we’re technologists, we should think about the people on the team in the context of the API façade and API program in general. It boils down to the need for agility.

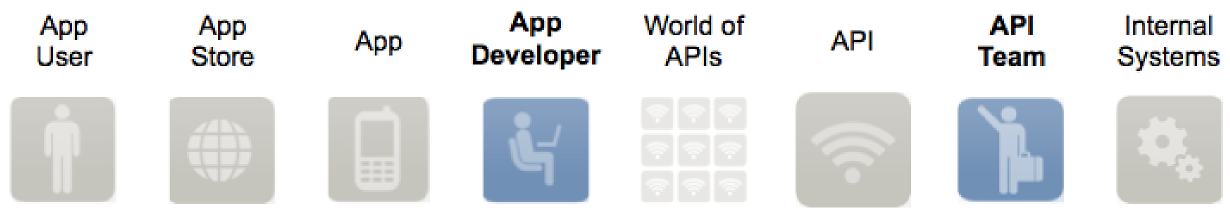
Agility

On the one hand, internal systems tend to be slow moving. On the other, the marketplace – as represented by the app developers who are trying to keep up with trends and shifts in the market – is moving pretty quickly. The API façade does a great job of being the intermediary and doing the frequency matching between the two systems.

It’s not just about wiring the requests and responses. It’s also about having a system in place that allows the API team to adapt the API facade to the needs of the evolving app developer as well as to the changes in internal systems.



Who are the people in that system - in the API value chain? The key players include the API team and the app developers.



The API Team

Building out a solid API team starts with a strong Web team. The core of that team includes **architects**, **software engineers**, **operations professionals**, **QA engineers** and **database administrators**. But don't stop there.

You'll also need API-specific roles on the team in the form of an **API product manager** (the subject matter expert for the company and the domain), an **API Designer** (technical thinker, critical thinker with a strong sense of design and user-interaction), and what we typically refer to as a **Gateway operations** person (.

One person may play multiple roles, but for a successful API program, you'll need to cover all these bases.

The Web team and the API specific team in place, you'll need some additional roles. Depending on the kind of developers who will develop apps that take advantage of your API, you add different people to the API team. The four main persona of app developer we talk about depends on what kind of API strategy you have – whether **internal**, **partner**, **customer**, or **open**.

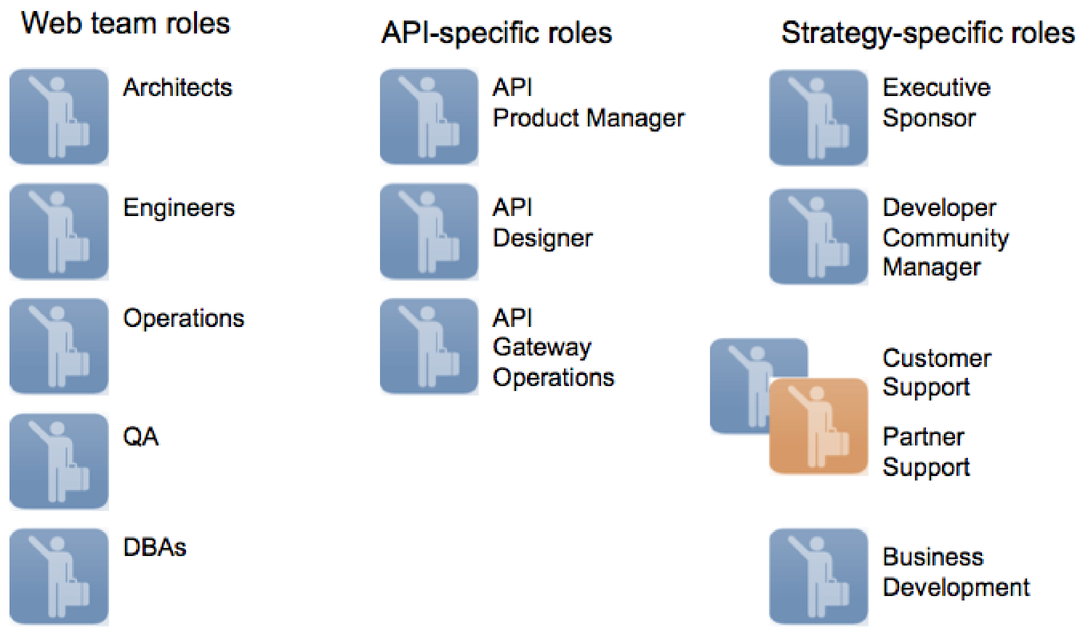
Internal: The app developer using the API works for the company – the API provider.

Partner: The developer works for or is a strategic partner. The go-to-market strategy is one with a shared value proposition.

Customer: The "customer developer" is especially important if your organization is a SaaS provider or if a large amount of revenue comes from B2B customers as opposed to consumer end users.

Open: The developer in this case is any developer who signs up to build apps against your API.

It is our experience that if you target an open scenario you will get things right in the other scenarios. So we finish out this third category of API team member with the open use case in mind. This last set of roles will have only subtle nuances depending on which scenario is your first or primary use case.



One of the most important people is an **executive sponsor** who will not only help with funding for your initiative but who can also set expectations about the business value an open API brings to the marketplace.

The next role is for a **community manager** who will facilitate two core and critical connections. The first is from the API team to the app developers and the second is helping connect developers with other developers to share expertise and collaborate to spur innovation. The community manager builds and forges relationships both online and in real life situations.

While the role for a community manager is clear when you have an open API strategy, even when your strategy is internal, it is valuable to have a person on the team with the mindset of community manager. They will facilitate developer-to-developer interactions and collaborations internally. It's also a great way to learn lessons about running a developer community before you do it externally.

For the customer API scenario, you will likely want to add a **customer support** role. While the developer community manager may play this role, you may need a more typical customer support role that can handle traditional customer tracking systems, follow ups and so on as are expected in typical B2B relationships.

For the partner scenario, you will likely need a **partner support** role. Partner support is very similar to customer support in that there are similar expectations between parties in the business relationship.

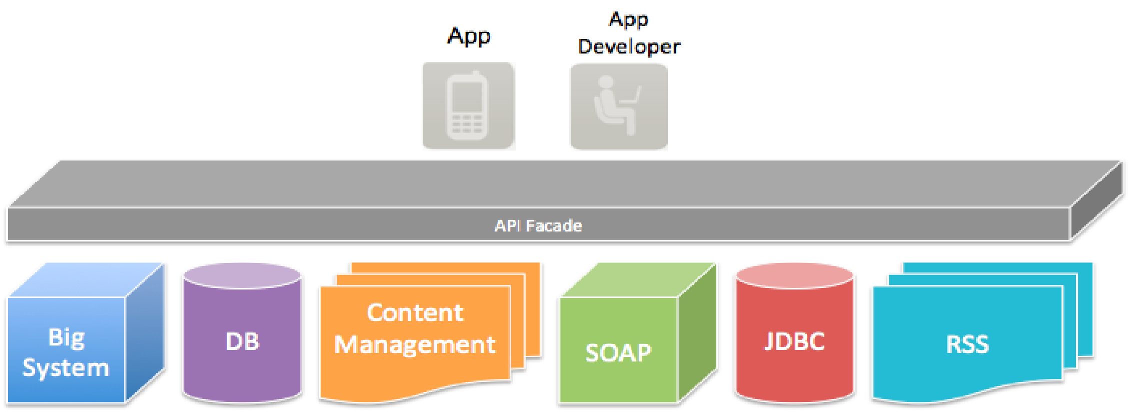
Then finally, you should consider a **business developer**. Perhaps this person doubles as the executive sponsor. They are responsible for pushing the API program in the direction of increased business opportunity.

Summary

Going back to our API value chain, app developers build apps using the APIs provided by the API Team (API provider). Everybody wins if app developers are as productive as possible and adopt your API quickly. Those app developers will help build the value within your organization and to extend that value proposition out beyond the boundaries of your organization into the broader market.

The best way we've found to facilitate this success is to implement an API façade pattern. This pattern gives you a virtual layer between the interface on top and the API implementation on the bottom. It is a comprehensive view of what the API should be. Importantly, it is the view from the perspective of the app developer and end user of the apps they create.

You break one major problem (exposing a set of complex internal systems in such a way as to be beneficial for developers) into three smaller problems (designing the ideal API; implementing the design with data stubs; integrating between the façade and the systems).



Using an API facade allows your organization to keep pace with developers in the real world. Their world changes quickly and frequently and they need to make apps that remain competitive in an ever growing and evolving market place.

It also has benefits internally (behind the façade) in that it enables an API team to create an extensible API that allows new systems to be plugged in, allowing your organization to keep pace with both the marketplace and with changing internal systems.

Resources

Web API Design – Crafting Interfaces that Developers Love (Brian Mulloy)

RESTful API Design Webinar, 2nd edition, Brian Mulloy, 2011

Apigee API Tech and Best Practices Blog

API Craft Google Group

About the author

Brian Mulloy, Products at Apigee

Brian has 15 years of experience ranging from enterprise software to founding a Web startup. He co-founded and was CEO of Swivel, a Website for social data analysis. He was President and General Manager of Grand Central, a cloud-based offering for application infrastructure (before we called it the cloud). And was Director of Product Marketing at BEA Systems. Brian holds a degree in Physics from the University of Michigan.

Brian is a frequent contributor on the Apigee API Tech and best practices blog, the Apigee YouTube channel, Webcasts, and the API Craft Google Group,

About Apigee

Apigee is the leading provider of API products and technology for enterprises and developers. Hundreds of enterprises like Comcast, GameSpy, TransUnion Interactive, Guardian Life and Constant Contact and thousands of developers use Apigee's technology. Enterprises use Apigee for visibility, control and scale of their API strategies. Developers use Apigee to learn, explore and develop API-based applications. Learn more at http://www.apigee.com.