Building Successful APIs

SOA | software™

Table of Contents

Introduction

We are in the midst of an API revolution. The pace of change is so rapid that it doesn't even make sense to list the reasons for the explosive growth in the development of Application Programming Interfaces (APIs). By the time we release this document, the list will have grown and new uses for APIs will already have been created. Given the amazing growth and diversity of uses for APIs, it makes sense to have a resource that you can use to help understand the best way to go about creating APIs.

The purpose of this document is to describe general best practices for providing an API for internal users, external users, or a combination of both. We discuss specific technology examples as well as broad business-level issues that affect API development. Our assumption is that you are involved in the development or management of APIs. Or, if you are not an API Owner, at the very least you are considering creating APIs and wish to understand the subject in more depth.

This document covers the following conceptual topics:

- Creating awareness of the challenges and opportunities inherent in APIs.
- Determining the business requirements that drive API development.
- Understanding what the API does.
- Creating an infrastructure and platform to host APIs.
- Describing changes needed by IT and business to successfully launch and operate an API.
- Supporting a development community for the API.

1 Creating an API Platform

When first faced with the need to build an API, you will probably create a diagram similar to the one in Figure 1.

As an API owner you should be building the API from the top down. That is, you should think about your customers' needs, the apps that will meet those needs, and the APIs that will connect those apps to your backend systems. However, as you go through this process, you will likely notice a mismatch between your requirements and the capabilities of the backend applications, data sources, integration platforms and security systems. To fill this void, you will need to create an API platform with the following capabilities:

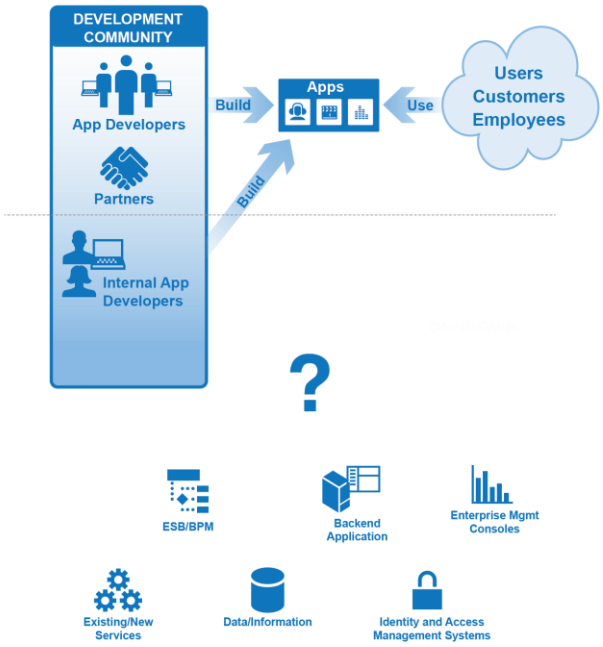

Figure 1 - Your API Ecosystem

1. Developer portal
2. Sandbox
3. API Framework and Server Platform
4. API Security and Management
5. Option for on-premise deployment, or hybrid cloud/on-premise deployment
6. API Platform-as-a-Service (PaaS)

You will also need to ensure that the lifecycle processes around API development conform with or extend your existing processes for product and software development.

1.1 Developer Portal

The Developer Portal is the focal point for much of the activity around the API. It provides a set of capabilities critical for attracting and interacting with a community of developers, including:

- API promotional and marketing material that describes the value of the API to a developer community.
- API document management to host and support the technical and legal documentation relating to the API and underlying data.
- Forums, blogs and ticket management to support the developers.
- App registration, security credential provisioning and certification services to manage the lifecycle of the Apps being built.

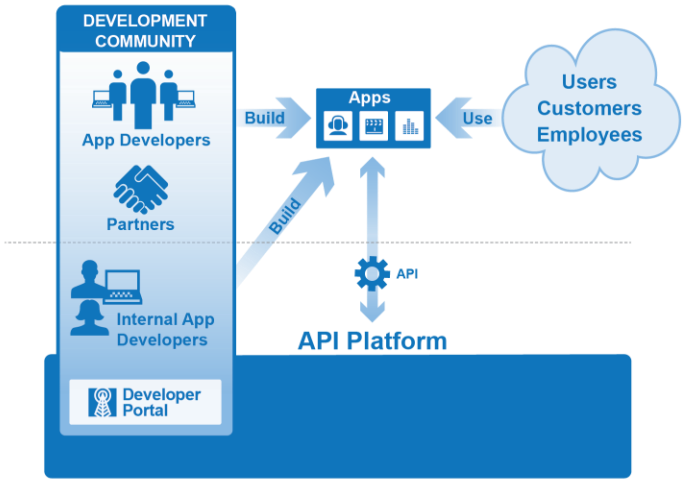

Figure 2 - The Developer Portal

The better and more interactive the Developer Portal is, the more engaged and loyal the developers will be to your API. This is the goal of the Developer Portal – to inform and empower developers.

The more loyal the developers are to your API, the more Apps they'll produce with it. The more Apps produced means a greater likelihood of more customers transacting through your API, and it also means a greater return on investment for the API.

1.2 API Framework and Server Platform

There are a number of ways to expose an API, from custom code on a variety of platforms to more complex Business Process Management (BPM) and mashup engines. Choosing the right platform should be based on two key factors:

1. The number, type and abilities of development resources at your disposal.
2. The presence of, and complexity of, the backend services and data sources with which you will need to integrate.

If your organization has a team of developers and a lightweight backend (or green field environment) then the obvious choice would be to pick up a REST framework for the platform of choice and simply develop your API.

If, on the other hand, you need to compose an API from a number of backend systems then you may want to consider a more declarative option such as a mashup engine. These tools attempt to lighten the integration burden and allow you to remain as agile as possible with a set of dependencies. There is a danger of getting bogged down in an integration effort, so make sure that you use the platform to stub out the API to allow for parallel development. See section 3.6 for more information on API development.

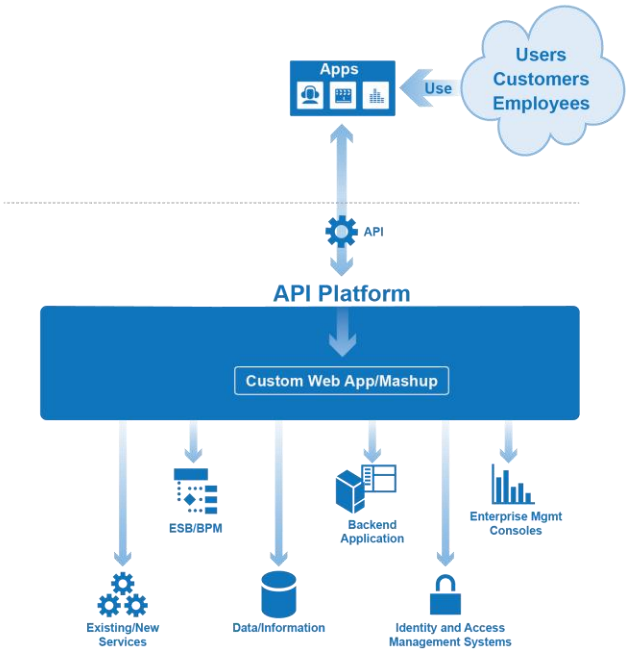

Figure 3 - API Composition

1.3 Sandbox

The Sandbox is the testing environment for your App developers. Just as the default for IT is to test against test data, you should set up an ecosystem where this occurs for the App developers using your API.

A Sandbox is a test platform that allows developers to make API calls and observe the results. The Sandbox API can be connected to a set of test services that you have in a test environment, or to Virtual Test Services offered by leading test products vendors.

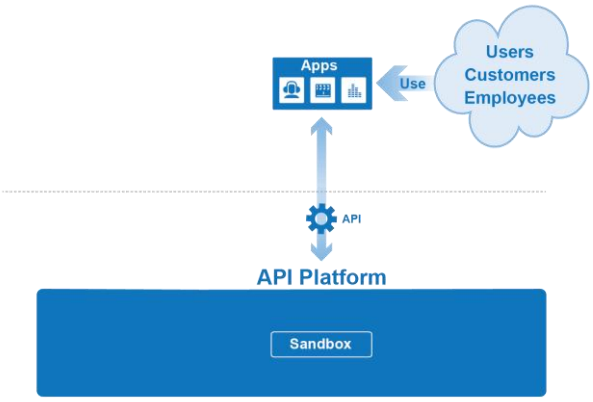

Figure 4 - API Sandbox

App Verification is another benefit that a Sandbox offers both an organization and its App developers. Once the App has been built and tested by the App developer the organization needs to be assured that the App is a good Corporate IT citizen. That is, that it behaves the way it is supposed to. The Sandbox allows for the verification testing of the App and recording of the results.

Just as the quality of the Developer Portal is a major determinant of API popularity, so will the Sandbox. Ease of use and configuration is very important. If it is too hard, or takes too long, to configure then it won't be used. If it won't be used, then you cannot expect any good quality Apps to be produced.

1.4 API Security and Management

Being able to secure and manage the API declaratively is of paramount importance. Security, in particular, is important in an API world, as an APIs is usually exposed to the Internet with all the risks this brings with it.

The API platform must support non-functional requirements. This allows API developers to focus on implementing business logic without being slowed down by implementing capabilities such as logging and security for each API. Ideally, your API Platform should support a declarative, policy-based mechanism to attach a non-functional capability to an API, either in total, or at the operation level.

The policy may describe a particular authentication mechanism, authorization rule, logging level, or fail-over process to be followed. Security is always a hot topic. It becomes even more so when enterprises begin to discuss APIs, so let's spend a bit more time on this topic.

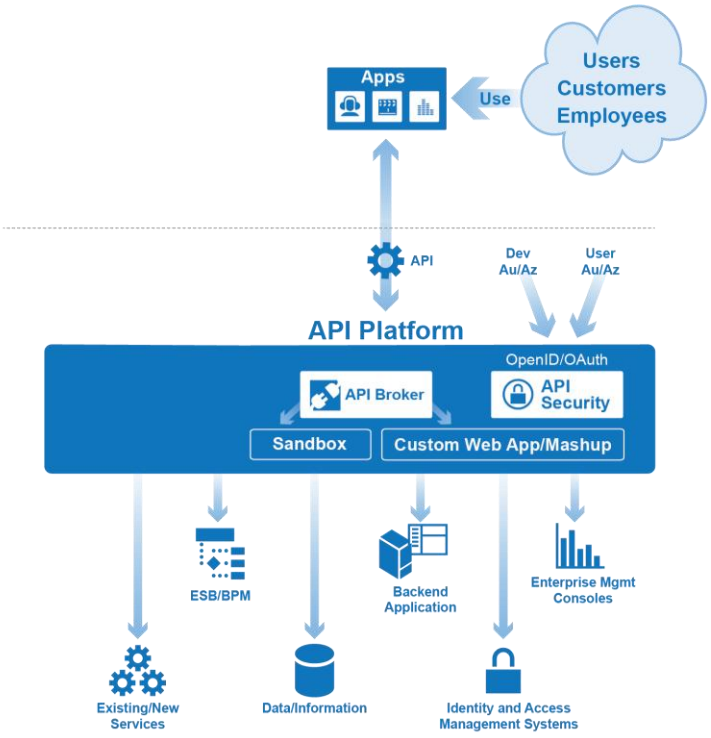

Figure 5- API Security and Management

What all this means is that an API Platform must have the ability to secure and manage an API quickly, easily, and correctly.

1.5 API Lifecycle Management

In addition to the important technical capabilities of the API Platform, you must also consider how it will integrate with your existing SDLC processes. The API Platform must give an organization the ability to guide and define its roles, processes, and procedures to ensure that the defining, building and running of an API is always consistent and correct.

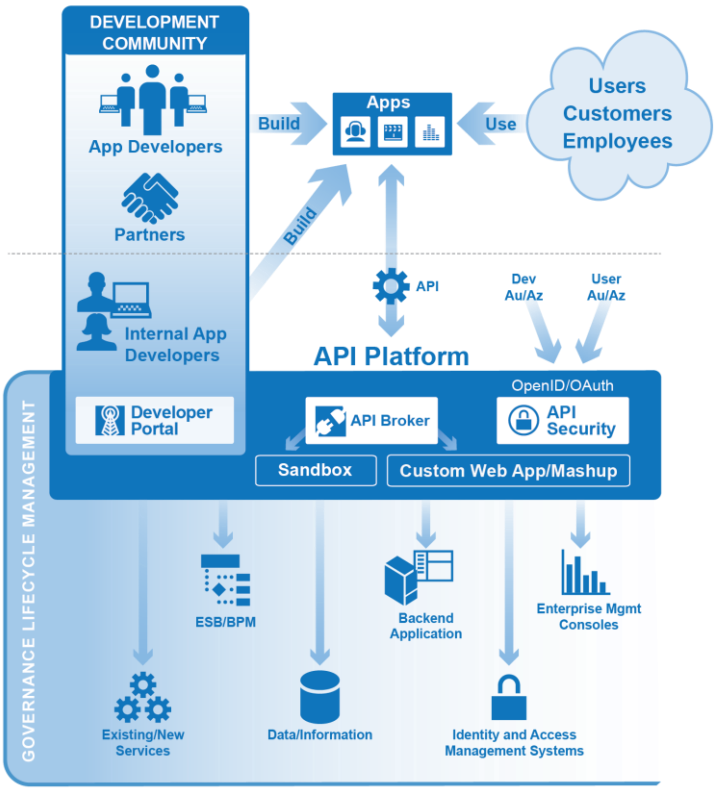

Figure 6- API Lifecycle Management

Figure 6 shows what the complete API platform looks like. Compare this with Figure 1. An organization could opt to build or buy these capabilities.

Please note that while Figure 6 implies that the API Platform is deployed on-premise, the Platform components can be either:

1. Bound together on premises
2. Bound together off premises as Platform-as-a-Service (PaaS) offering
3. A hybrid, where some components (such as the Developer Portal) may be off premises, and the remainder on premises. See Figure 7.

Choosing one topology over another is usually a combination of security, performance, and cost.

An organization may choose to host the whole API Platform on premises because of regulatory or security concerns, the need to control the implementation and access to the components, or simply because all of the API consumers are internal.

Conversely, an organization may choose to utilize a PaaS offering for the API Platform because it provides a quicker time-to-market, is more cost effective and reduces the management load.

Lastly, an organization may choose to have a Hybrid topology where the Developer Portal is provided as a service, but the remaining components are kept internal. This keeps costs and management overhead down for the most naturally externally facing component, while securing and optimizing API performance by retaining the remainder on-premises.

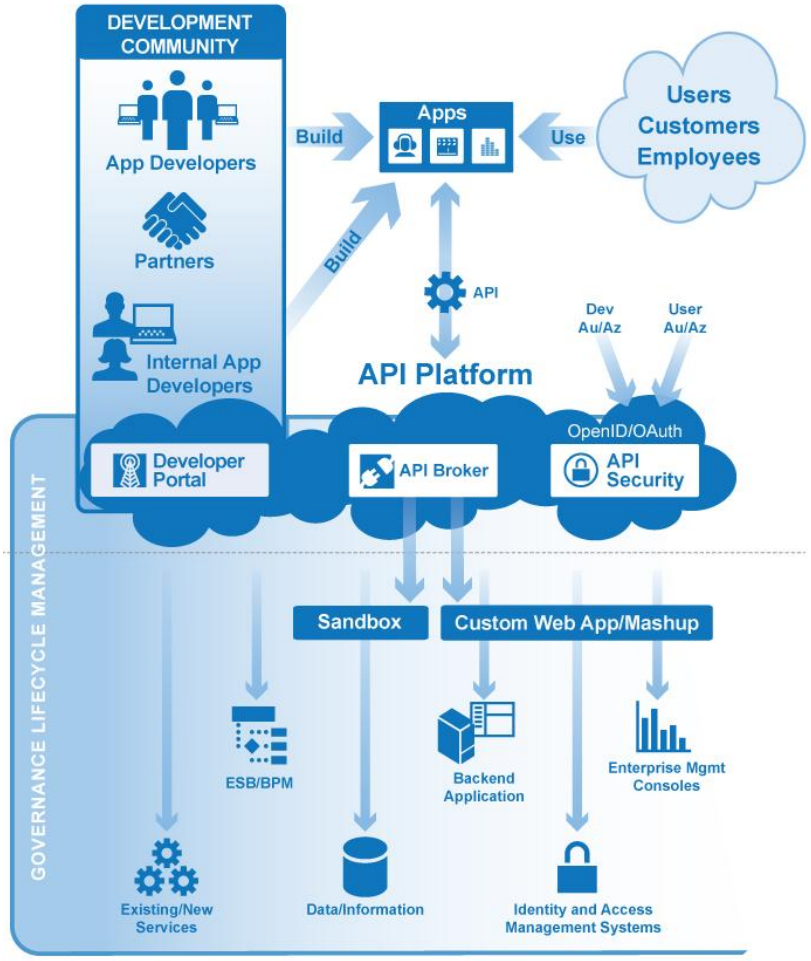

Figure 7 - API Platform-as-a-Service

The remainder of this document will describe the detail of the Plan, Build, Run, Share lifecycle and how these activities would be supported by the existence of the API Platform.

2 Planning the API

Few things in life or business get done very well without a plan. APIs are no different. An effective API that delivers business value needs to be planned thoroughly. At the same time, we encourage you to evolve a planning process that remains flexible enough to manage the inevitable changes that will occur in the fast-paced, unpredictable world of applications and APIs.

There are some major goals you should be driving for when planning your API:

1. Determine the business purpose for the API.
2. Understand the cost/benefit outcomes for the business and intended users.
3. Agree on the priority and delivery schedule for the API.
4. Structure your business to support and manage the API.

2.1 The Business Purpose for the API

The fundamental premise of all API planning is that an API exists to serve a business purpose. There should never be an API "just because." While it may be useful to hold exercises in creative thinking by API developers, any API that gets serious resource investment must answer some business need. Even an API that is designed to serve a distinct technical purpose should ideally map to, or support, a business need. An API can generate new opportunities, streamline business functions, or facilitate new partnerships. An API may be created for any number of reasons, but the form and function of the API should be driven by the requirements of the business.

Business purposes for an API may include:

- Driving more traffic to a company Website or portal.
- Building a community around a brand or campaign.
- Providing self-service options for customers.
- Providing access to online catalogs and inventory.
- Expanding the customer base to users of mobile devices.
- Exposing core functions to internal users in a controlled manner.
- Exposing core functions to external users in a controlled manner.

- Refactoring business processes so they can be delivered as services.
- Refactoring the technology landscape within an enterprise, solving the issues the Bring Your Own Device (BYOD) trend is creating.
- Facilitating a merger or acquisition of another business.

The business reasons for developing a specific API should be part of a controlled list defined by senior executives at the very beginning of the project, so that the team never loses sight of these goals. The list can be revised as needed based on changing needs and conditions.

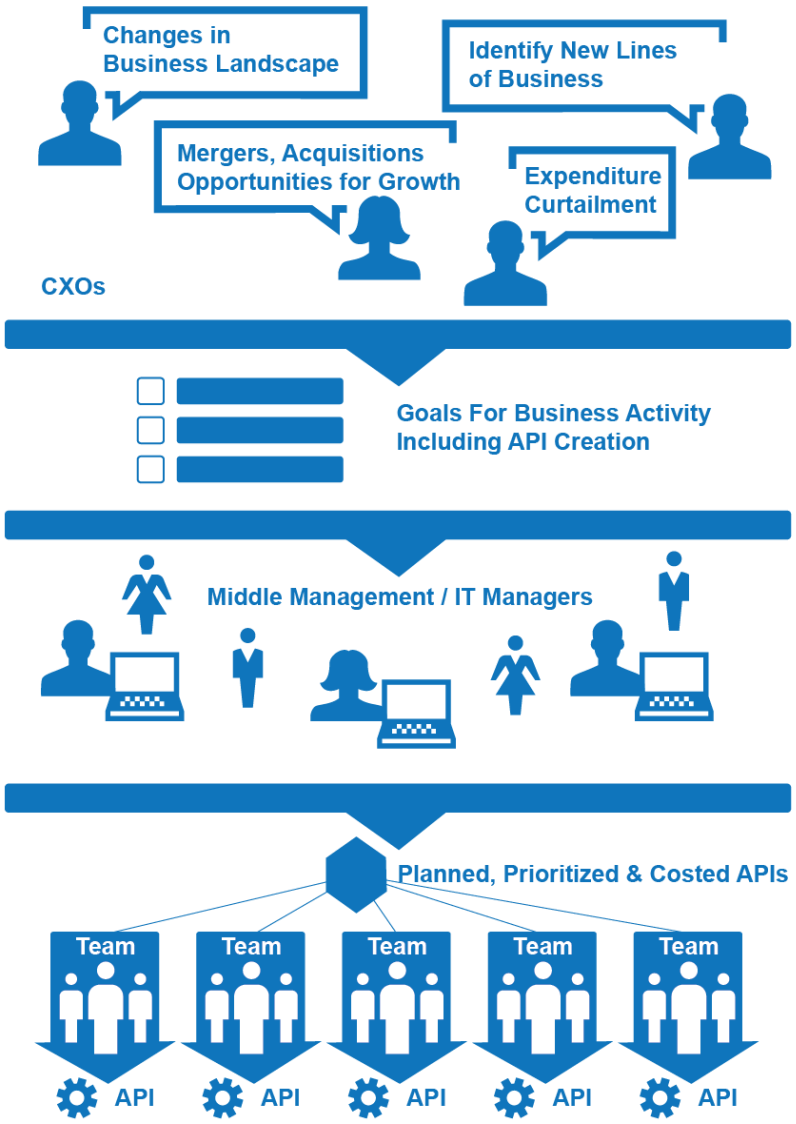

***Figure 8**-the importance of aligning your API delivery to your Enterprise Strategy*

Once these purposes have been identified for the API, they should be used to drive the overall API strategy. This will help identify how new requirements should be viewed and prioritized over time.

2.1.1 Addressing the Conflict between IT and Business

The business-driven nature of APIs creates a clear tension between IT and line of business (LOB) managers. This is because APIs add a powerful new element to the IT mix, which is the active involvement of outsiders in IT strategy and planning. An API is a technical construct that IT must create and support, even though the API project is being driven by non-IT people for use, sometimes free of charge, by outside technologists.

This can seem to be a difficult and unrewarding challenge for IT. However, the popularity and success of APIs in general indicates that the challenges are not insurmountable and the rewards can be great. The truth is that APIs are the perfect vehicle for marrying the discipline and professionalism of the IT staff with the free-thinking and agile world of the Mobile App developer or freelance developer. IT has become the home of professionals who understand risk and can be relied on to identify and manage that risk on behalf of the enterprise. There is an abundance of skilled developers who make it their business to create new solutions by integrating software from many different businesses and sources. When you share your APIs with these developers, and they share their apps with your customers, everyone wins.

This developer pool can make your enterprise's products more valuable without asking for time from product managers and marketing experts. Your products become more valuable because when you share your API it opens the door to new capabilities. The developers' products become more viable because they can tie into enterprise products. The most efficient and effective way to do this is with APIs. APIs have been used for quite a while, but with the advent of more ever-present Web services and cloud-based applications, they truly have become one of the most crucial elements in the process of getting your product into the hands of more customers. An API makes your product more attractive, not just to your customers, but to their customers as well. That's the network effect in action, and it builds a more interesting marketplace for your products.

2.1.2 Your API Strategy

An API isn't worth much unless you've figured out its purpose as well as how it fits into your overall API strategy which, of course, should flow from your business strategy.

The best API strategy is one that frees an enterprise to focus on what it is best at, while at the same time allowing its ecosystem of partners and developers to contribute their own strengths. These developers are people who see your product as an opportunity. You provide the framework and foundation. They provide their insight and skills. From there, the world benefits from better products, and the organization benefits from an expanded bottom line.

Your API strategy is driven by the business strategy and the business purpose list. If the business strategy is to retain more customers, then an API set that generates "stickiness" should be your first priority. On the other hand, if your strategy is to generate better market awareness of your products or services, your APIs should be aimed at creating new sales channels for your business.

As well as senior IT and business staff, you may well have representatives from the organization's most prolific or successful App developers on your team (this implies a mature organization that has gone through one or two API delivery cycles).

In the end there should be a document that clearly defines the APIs that will be needed over the strategy lifetime as well as a clear definition of what the strategy lifetime is. These should be mapped to either existing or new business functions, which are, in turn, mapped to existing or new IT functions.

2.1.3 Writing Your API Strategy Statement

You may want to pull together an *API Strategy Statement* that encapsulates your thinking and vision to accompany the API strategy document, but we suggest you go through the exercise of putting it together.

Making sure key players agree on a clear, simple strategy statement, and making sure everyone on the team is on board with it, helps ensure that the project starts and continues on the right track. One side benefit is that differences in vision and direction come to light and can be resolved before the project starts.

As an example, the API Strategy Statement for an ERP platform that is focused on growing its reseller channel into different verticals might read:

> *"Our API strategy is based on developing APIs that enable reseller partners to create industry vertical add-on solutions, fulfilling the business objective of revenue growth by building a closed, highly professional developer community that creates applications that we monitor and govern closely."*

2.2 The Costs and Benefits of the API

Having decided on the "why" in respect to the business drivers, let's focus on the "how."

At this point, more detail is required. You might need to do more analysis of some or all of the following:

1. Existing APIs, if any and what they do.
2. Planned APIs that haven't yet been delivered.
3. The urgency of delivery and the possible staging of the delivery.
4. Whether the requirements require an API at all.
5. Whether this is a purely externally facing API, internally facing, or both.
6. Whether existing Services in your service-oriented architecture (SOA) can be leveraged either partially or wholly.

You'll notice if you have an SOA, that these are the types of analyses that you carry out when planning your SOA services. There is no difference here – an API at this stage is just another asset.

2.3 Prioritizing Your API

Once you've decided on the "why" and "how" in respect to your API program, you can consider the "when."

At this point, you should have a clear understanding of the business reasons for an API, an alignment with the business strategy, and a new or updated API strategy. You should also have analyzed the costs, benefits, and target markets both for users and consumer developers, leading to defining your API specification.

The last piece in your API plan is to provide a roadmap that clearly prioritizes the delivery of each API within the growing list of business-dependent APIs. Communicating the roadmap is a critical step - the information provided has to be sufficient to allow any Manager to understand what API needs to be designed, why it is being built, what its importance is and how it will be funded.

2.3.1 Create an API Roadmap

When creating a roadmap, it is important to phase the delivery of each API and prioritize each phase separately based on its relevance to the business strategy and available resources.

Given the amount of information collected in the past planning phases, a roadmap usually includes references to the API Specification and Strategies.

For example, if you have an ERP solution and you want developers to add industry vertical capabilities through your API, your roadmap might look something like this:

API Program Factor	Phase I	Phase II	Phase III
API feature set	Access general ledger	Add access to logistics management	Complete application access
Priority	1	3	7
Platform support	REST	REST and SOAP	REST, SOAP, POX
Developer community	Strictly by invitation only.	Enable applicants to sign up online and be approved for access after IT department review.	Limited open community.
API Specification	http://docs.org.com?doc=blah	http://docs.org.com?doc=blah1	http://docs.org.com?doc=blah2
API Strategy	http://docs.org.com?doc=stat-blah	http://docs.org.com?doc=stat-blah1	http://docs.org.com?doc=stat-blah2

Table 1- API Roadmap Template

2.4 Structure Your Business to Support and Manage the API

In line with just about everything in this document, there will be changes to the roles required to deliver a successful API. The following is a list of the roles that are significant to a successful API development project. You can either create new positions within your organization or assign additional responsibility to existing roles:

2.4.1 Product Manager

It's important to realize that the API, unlike previous IT activities within the enterprise, is a first class product of the organization—something as critical as a cellular offering in a Telco, or a loan or other type of product in a financial organization. Because of this, there must be an API Product Manager, especially if the Enterprise has, or plans to have, more than one API.

This person in the API Product Manager role would manage the team, most likely as a combination of direct reports and matrix managed reports from other parts of the enterprise, through the Plan, Build, Run, Share lifecycle described in this document. In addition, the API Product Manager is responsible for the branding and marketing of the APIs, as well as costing, tracking, and billing for API usage.

2.4.2 Community Manager

The Community Manager directly supports the community of App developers. He or she is responsible for making sure that the developers derive value from, and are valued by, the company and also by other community members. This role is all about the notions introduced in Section 5 - Sharing the API. The Community Manager should:

- Ensure that the Forum interactions are frequent and lively, but not caustic as some developer interactions can be. Timely, diligent, but not intrusive moderation is key.
- Remove all illegal content and spam.
- Ensure that software development kits (SDKs) remain relevant to the needs of the developers.
- Ensure that the documentation is in line with the version of APIs being offered.
- Coordinate support activities with the API owner and IT.

2.4.3 Technical Writer

Documentation is critically important to the success of the API. A professional technical writer should be engaged to ensure that the documentation is accurate, complete, and accessible, and is delivered in coordination with the version release schedule.

This may not be a full-time role depending on the size and number of versions of the APIs being delivered, but it is a very important role given that developers cannot use your API without documentation.

This role would also have to liaise closely with the support staff to ensure that any reported issues with the documentation are remedied.

2.4.4 API Support Staff

The API Support staff is responsible for the timely support of the API. Unlike traditional IT support roles, these people would be more like front-line software product support staff.

The API support staff should not only be responsible for the correct and smooth running of the API in the normal IT sense, but should also manage and respond to a trouble ticketing system driven by the Community. They would report to the Community Manager.

API support personnel should also be scheduled to monitor and respond to any API-specific questions in the forums of the community. Lastly, they are responsible for the API in the Sandbox. The Sandbox is a set of test APIs with corresponding dummy data and optional UI that App developers can use to test their apps. For more information see Section 3.1.

2.4.5 Developer

While this would seem the most obvious role, it still needs to be defined.

The line between developer and support staff may be quite blurred, especially during the early phases of delivery of an API, or in a small organization. In any case there should be a close relationship between the two roles.

As well as developing the API, the developer would be responsible for the following areas of development:

1. SDKs that would be available via the community.
2. The API Sandbox.
3. Non-functional capabilities, such as security, logging, and monitoring for the APIs that are not provided by the API Platform – see Section 3.1.

2.4.6 Operations

The API will be a 24x7, 365 day, business asset. As with all of these asset types, IT operations are asked to manage the asset and the support of all the API's non-functional requirements. Once the API goes live, continuous operational support is a must.

3 Designing and Building the API

As discussed in the previous section, it's important to have an API Strategy and Roadmap in place before beginning development. From this roadmap you can structure a development team to deliver the next API, or the next version of an existing API.

So, how is an API best designed and built? What goals must be achieved in order for your organization to confirm that it has built the API according to best practices?

1. Understand and clearly articulate the detailed requirements for the API. Make sure there is agreement between key players before development starts.
2. Separate functional from non-functional requirements and develop only to the functional requirements.
3. Iterate through the API development process.
4. Utilize an existing SOA investment.

3.1 Defining the API Requirements

Defining "what" the API does should be separated from "how" the API does it. At this point, you should be defining functional characteristics of the API, specifying the interface, and developing model use cases and user stories. Implementation details are purposely deferred until later.

There are several artifacts that can help refine the requirements. They include:

- **Use case models** that show the relationship between actors and use cases.
- **Use cases** that describe the scenario—the interaction between API and the user—as well as the expected result.
- **User stories** that are small narrative descriptions that describe a specific interaction between the API and the user of the system.

Not every requirement will need to be described in this much detail, and a team might choose to slightly modify the artifacts they use to define a requirement. However, the set described here provides a best practices approach for elaborating a requirement.

You might find that you choose only one approach. Which one is right for you and your organization really depends on your organizational culture. Use Cases and Use Case Models tend to be more prescriptive and detailed, whereas User Stories are more compact and lend themselves better to an iterative approach.

Whatever method or methods you choose, the important thing is to explain the requirements clearly, with sufficient detail, so that the API design and development can proceed smoothly to a successful completion. Use case models, use cases, and user stories help to illustrate the requirements in a very practical and realistic way, and can be invaluable in helping keep the project focused on real-world user needs.

3.1.1 Identify How the API will be Used

The way in which the API will be used affects several issues such as the technology choices, regulatory issues, and security. For example, an API that's being used to perform financial transactions will have more constraints than one delivering advertisements.

Some of the ways an API might be used include:

- Within a mobile application
- Delivery of banner ads on a Web page
- As part of a mashup
- Servicing financial transactions
- Providing a self-serve portal
- Enabling the connection of a new business to the existing enterprise

3.1.2 API Usage Patterns (B2B, B2E, B2D)

The users of the API will also help determine technology issues and may determine the content of each development phase as the API is being delivered.

Consumers of the API could include:

- Internal consumers
- External unregistered users
- External registered users
- Business partners

In general it is best to plan for the least controlled set of users when identifying API function and content. For example, if an API is intended for internal users, but there is a chance that in the future the API will be made available to business partners or customers, the designers should focus on the external users when determining API function and content. It may be a good idea to develop an API initially for internal consumption before making it available externally, but the design should not factor out external requirements in future releases. Internal deployment may help identify new requirements that should be added to a list of possible features for future development phases. This allows an API provider to gain experience with the design before making it public. This is important because when an API is provided to external users, it signals a long-term commitment to the API, and it implies that the API will be stable.

3.1.3 Regulatory Issues

Almost any business activity is affected by some form of regulatory activity. When a business decides to provide an API to consumers, there will probably be a requirement for some type of regulatory compliance. This is an activity where the business domain expert must identify the applicable guidelines and make sure they are included in the requirements list for the first release. It is usually easier for the product owner to plan for these types of issues in the earlier development phases than it is to retrofit the requirement. Examples of regulations that might affect API development in the USA include the PCI DSS payment card data security standard, HIPAA which governs privacy of healthcare information, and FERPA (the Family Education Rights and Privacy act). There are many others in the USA, and each country has its unique regulations. Key factors to consider are a) the industry in which the API is deployed and b) the deployment areas.

3.2 Usage, Content or Functionality Constraints

Not all enterprises are happy "showing off the family jewels" or even part of them. For an API to be successful, it must offer value to those who are going to use it, but it also must be seen as a valuable, core part of the enterprise offering it. This is not likely to be the case if senior management is unhappy about the risk being created by the exposure of content and functionality and is worried about undesirable use of the API. It's important to think through, discuss, and get agreement on what is going to be constrained in the API, as well as the phasing of the relaxing of those constraints (if any).

3.3 Leveraging the SOA Investment

If your organization has been through the expense, trials, and effort of creating a successful SOA infrastructure, it makes sense to leverage that investment. The design constraints for the modern API are similar but not identical to those of Services and an SOA. The Design of the API will be different and there are no short cuts. However, if your organization has a successful SOA then the build of your API becomes largely an exercise in mapping rather than fully blown coding.

3.4 Separating Functional from Non-Functional Requirements

Not all requirements are functional in nature: availability, scalability, logging, security, and performance are all critical to the successful use of an API, but none of them have anything to do with the business process or domain of the API's resource.

Much of the information gathered during this phase may be well understood before the project begins. However, documenting the non-functional requirements might identify new requirements that were not originally considered and must be added to the project backlog. This can affect the cost and viability of the project.

Lastly, most Services in SOA are rendered non-reusable because of non-functional incompatibilities. This is certainly true for APIs. For example, the fact that a particular security mechanism was assumed and hard-coded into the service, but other consumers required a different security mechanism, means that the service/API is not reusable. This is true for any non-functional capability, including logging and fail-over.

There are two big differences between functional and non-functional requirements:

1. Non-functional requirements are much more variable than functional requirements.
2. Implementing non-functional requirements by declarative policies is much quicker and easier than implementing functional requirements.

Because of the differences between them, it's important to separate out these two types of requirements.

3.5 Defining Your API Specification

Once you've worked through the points in this chapter, you should have your strategic groundwork done. Once you've identified and mapped your business strategy to APIs, which are really your API requirements, it's time to put together the API Specification. This specification adds the detail needed to sensibly understand what an API is intended to do and its importance in respect to the business strategy.

3.6 Designing Your API

APIs are lightweight, self-describing, and agile interfaces that lend themselves to easy access and re-use. They are not SOA Services by another name. You must keep this in mind when designing and building your APIs.

That being said, the process of designing and developing APIs is comparable to that required for any other software development projects, although there are some notable differences and these are spelled out in the remainder of this section.

3.6.1 Guidelines to Apply to All Build Stages

The remainder of this chapter is a set of sections that collectively define the steps to follow when designing and building your API. Throughout all of these sections, keep in mind the following guidelines:

- Begin with a short section of the already defined specification that covers the functionality to be built, bearing in mind that you should stay flexible about your API definition, even if it means omitting detail.
- Code the API as you become more confident that you are moving in the right direction.
- Be realistic:
    - You can't please every stakeholder.
    - Many API designs suffer from excessive constraints.
    - You will definitely make mistakes. The real world will reveal them and you will be able to correct them.
    - Your API will evolve.

3.6.2 Develop a Simple API Structure

- Define the object model. One of the major differences between an API design and a SOA Service design is that API's are Data, or Object Oriented, while SOA Services tended to be Process Oriented. Objects map to resources, and a resource is a first class citizen in the HTTP protocol. Therefore, before you do anything else you must have the object model defined for the API. As stated in Section 3.3, if you already have a successfully running SOA this Object model most likely already exists.
- Determine the resource relationships based on the object model – represent these relationships in the resource. While relationship definitions are a consequence of good modeling what is being stated here is that the resource definition must include the explicit definition of the relationships to ensure that when the resource is returned, as a result of the API request, the relationships are also returned. The greatly enhances support for HATEOAS (see later definition).

3.6.3 Build the API

- Serve the resources and collections with stubbed out code to get started. This will help with unit tests and allow for parallel development. If the API is supporting a Web client, a stubbed out backend will greatly speed up UI development. Conversely, if you are composing an aggregate API using a composition tool or mashup engine then create it one operation at a time.
- Write to the API frequently, starting early in the process. Even if the API is not complete, start programming to it. The learning from this process will save you time later as you avoid unnecessary implementation issues. This applies to either approach, both custom code and mashup.
- Include resource relationships to support HATEOAS. HATEOAS stands for "**H**ypermedia **A**s **T**he **E**ngine **O**f **A**pplication **S**tate". This is referring to the same "state" as **Re**presentational **S**tate **T**ransfer (REST). In short HATEOAS means that when a resource representation is returned as a results of an API request the representation not only includes the information that defines that resource, it contains all the valid relationships that the resource has with other resources for that state, as well as all the valid states the resource could transition to. In other words the resource is self-describing.
- Determine a strategy for retrieving and updating partial data. As per the statement made at the very start of this chapter, API's are lightweight. They

need to be because of the latency introduced by the fact that they live in the Internet. Because of this, it is not often sensible to return every resource that meets the requirements of the API request. An API must have a designed and consistent scheme to allow the App to request a partial set of satisfying resources. Consequently, the modification of this result set needs also to be supported – proper adherence to HATEOAS should resolve this requirement.

- Add validation. Once the following steps have been implemented successfully (remember this can be partially – per resource – as per the general guidelines) Validation any resource that is the subject of a PUT, POST, or DELETE call (Usually JSON, but may also be XML) should now be developed. Use an HTTP error code to define and transfer exception information. This will enable you to handle those exceptions on the client side.

3.6.4 Build a Client

- Keep it simple. Use the technology you are most familiar with that supports the API technology set. This could even be a tool such as SOAPUI. This client will become invaluable in both development and testing.
- Make sure it can be automated for testing. This is an important point. Once it is automatable it can be incorporated into the API build and test process. This will ensure consistent delivery of new versions of the API.

3.6.5 Add the Non-functional Requirements

- Mime type
- Security – authentication and authorization
- Caching
- Paging
- Load balancing
- Fail-over
- Disaster Recovery
- Logging/Auditing

The list above is a good representation of the likely non-functional requirements that will have to be met for an API, although this list is not exhaustive. Please be aware that it is not meant that you should code the solution for these requirements. As stated in section 3.4, quite the opposite is suggested. If you have a good API Platform available to your organization then it should supply the components to

support the solutions declaratively. This is important. Maximized re-use and flexibility is achieved when non-functional requirements are supported by this mechanism.

This being said, adding the support for the non-functional requirements at this late stage does not imply that they are less important than the other steps mentioned. It's just more practical to ensure that the required functionality is supported before adding the requisite complications of the non-functional support.

Apart from security, there are two broad types of non-functional requirements that this list represents. The first is performance, and the second is resilience. Both of these types are solved by a combination of Network Architecture and API Platform. The Platform gives the ability to support Load Balancing, Fail-over, Paging, Logging and Auditing, and Disaster Recovery, while the Network Architecture provides the Caching, Load Balancing and Fail-over. Notice that Load Balancing and Fail-over are mentioned under both categories. This is because they should be supported by both and an organization has the choice of using one over the other, or in concert with each other. The ultimate solution is dependent on the organization's needs and capabilities.

3.6.6 Standardized Content

Section 3.1.3 covers the requirements elicitation for business and technical regulatory requirements. Here is where those requirements are defined in the build to support these requirements.

First, match these to the capabilities of your API Platform to ensure that it gives you the maximum coverage possible and then incorporate the remaining non-functional requirements into your API design. A good API Platform should support as many of the non-functional capabilities as possible, as stated in section 1.4.

3.6.7 Technical Environment

Your API will face constraints that flow from its technical environment. These include:

- Server functionality
- Server capacity
- Client technology
- Security

- Network capability and design
- Scalability
- High Availability and Disaster Recovery

The amount of traffic that will flow through the API will help identify the required server capacity and functionality. A dedicated API server may be required in order to provide the required capacity and to isolate the core business processes. In addition, the type of user should be well understood and documented during the charting of technical environment constraints. For example, your user base could include anonymous users, registered clients, or a combination of both.

4 Running the API

APIs that serve businesses needs also need to function in accordance with business expectations. This is true of all systems run in a business; however, APIs are different because they touch so many external players. In this chapter we discuss what it takes to make a production API a successful asset and not a problem for an enterprise. We apply traditional IT values to the distinct task of running an API.

When you launch your API, you may feel a corporate chill run down your spine. Is this really happening? Is your enterprise really going to offer up the ability to drive its internal processes from anywhere in the Internet without the ability to decide who it wants to do this? Are you actually offering up the ability to drive your databases and transactional services without the ability to decide how much of those expensive assets are going to be used by a specific external user at any one time? Are you seriously going to create external consumers that will be associated with your brand and cherished customers without first being able to test and approve these consumers?

You'd think the answer to these questions would be "No!" and you might be tempted to switch that API off right away. The truth is many organizations leave these questions unanswered until after the API is built.

This chapter includes the following recommendations for running your API:

- Support non-functional requirements in the API Platform, not each API.
- Implement contracts with API consumers to enforce SLAs.
- Provide support for API monitoring.

4.1 Declaratively Support API Non-Functional Requirements

As discussed in section 1.4, the API Platform should provide this ability. Therefore, at this point, the development team should engage the runtime governance team to have them declare the requisite non-functional capabilities to the API as part of the API Platform. Section 3.4 discussed the need to separate the function and non-functional requirements which lead, in turn, to the function requirements being

implemented in section 3.6. This is where the non-functional requirements are implemented, declaratively via the API platform.

This is what makes such a platform so powerful, there should be no coding, just quickly adjustable definitions that are referenced when the API is called.

4.1.1 Authorization and Authentication

The table below shows common authorization and authentication options for APIs.

Authorization and Authentication options	**Comparisons**
API keys (Yahoo and Google Maps APIs)	Fairly open, An API key gives the API provider a way to know the identity of each caller. The API provider can use this information to maintain a log and establish quotas by user.
Usernames and passwords /OAuth (Twitter)	Provide greater security; used more sensitive data, these technologies work well for application-to-application communications.
Session-based authentication	Much more complex when associated with an API.
OAuth	OAuth was designed to solve application-to-application security problems. This technology allows a user to give an API or site access to his or her account on another site, in whole or, importantly, in part, without sharing a password. Instead of sharing a password, the user logs in to grant access; an OAuth "token" provides continuing access to the requesting app. An additional benefit is that the user can revoke the OAuth access at any time.
Two-Way SSL, X.509, SAML, WS-Security...	An API that will primarily be used by "enterprise" customers.

For API security, we recommend OAuth, the new and much improved authentication and authorization standard. OAuth takes the lessons learned from XACML and SAML in the SOA world and applies them to the problem that exists in the current crop of Web Security practices. That is, the current security solutions assume only two actors, the consumer and the provider.

In the traditional client-server authentication model, the client uses its credentials to access its resources hosted by the server. OAuth introduces a third role to this model: the resource owner. In the OAuth model, the client (which is not the resource owner, but is acting on its behalf) requests access to resources controlled by the resource owner, but hosted by the server. In order for the client to access resources, it first has to obtain permission from the resource owner. This permission is expressed in the form of a token and matching shared secret. The purpose of the token is to make it unnecessary for the resource owner to share its credentials with the client. Unlike the resource owner credentials, tokens can be issued with a restricted scope and limited lifetime and revoked independently.

Why is this important? Because, unlike the Web's traditional two-party, client-server model, the API can extract more value from mashups of APIs by a third party such as a mobile App. The problem is, in order for these applications to access user data on other sites, they ask for usernames and passwords. Not only does this require exposing user passwords to someone else – often the same passwords used for online banking and other sites – but also it provides these applications with unlimited access. They can do anything, including changing the passwords and locking users out. OAuth provides a method for users to grant third-party access to their resources without sharing their passwords. It also provides a way to grant limited access (in scope, duration, etc.). For example, a Facebook user can give another app access, via OAuth, to his or her contacts list without allowing the app to create posts.

There are now three OAuth versions; 1.0, 1.0a, and the newly released 2.0. The most common implementation is 1.0a (which fixed a security vulnerability in OAuth 1.0) , although with 2.0 being newly released and having a simpler handshaking protocol and better support for JavaScript, 2.0 may become the more common version implemented within a short period of time.

If you are looking for a platform that implements OAuth to support your API's then make sure it supports at least 1.0a and has a delivery date for support for 2.0.

4.1.2 Protecting Data and Detecting Threats

Because APIs use Web technologies over the open Internet, an API developer is going to encounter the security threats commonplace in this ecosystem. Security is a non-trivial issue for APIs, especially considering how they can expose your internal systems to unknown outsider users. Consider the following API security risks, and determine steps you might need to take for threat prevention, detection, and handling:

- **Malicious Code Injection -** manipulates security design flaws in technologies to send valid code to services using SQL, LDAP, XPATH, or XQuery statements to open up the interface for any user to take control or to cause harm.
- **Denial of Service (DoS) Attacks -**cripples an API by overwhelming it with requests (e.g. a "Request Burst").
- **Service Information Leakage–**An API inadvertently leaking data about its configuration, resulting in the ability to take control or expose private data.
- **Broken Session IDs, Keys and Authentication**- Exposure to unauthorized access through authentication factors that are not functioning because of poor security design or technology bugs.
- **API Data Eavesdropping–** When non-secure API communications expose the data to access while in transit.
- **Tampering with API Requests and Responses**- An attack that manipulates the API request and response parameters exchanged between client and services with the goal of modification of data. An Example of this is the "Man in the Middle" type of attacks.

4.2 Manage the API Quality of Service (QoS)

OAuth enables an enterprise to resolve who can use what in the API. However, what about how much or how many times they can use it? Excessive API use can be even more damaging and costly to the API provider than no use at all. But there is another side to this discussion. Good, well-built and designed APIs can be monetized. This monetization is not going to be countenanced by management, nor accepted by API users, unless the SLA agreements can be managed, automatically, by the API. The best way to do this is with easily definable, modifiable, and actionable QoS policies that are based on reliable metrics.

First, look at the API provider. The API is an interface, a door into an organization's business, as defined and allowed by that organization. No IT representative in any organization would be allowed to implement a critical business system without the ability to monitor and control critical factors such as who was accessing it, usage levels, and performance. Therefore, it doesn't make sense to implement an API, which exposes critical business capability over the Internet, without similar controls in place. Conversely, an API consumer is not likely to use your API if you don't guarantee the availability and performance of the API, since interruptions in service will adversely affect the consumer's ability to deliver its own business services, whether monetary or technical.

The best practice for addressing these issues is to define and manage service level agreements (SLAs) through contracts that are standards-based and are declarative. With these agreements in place, an API manager can define the absolute performance and availability constraints for the API, and can specify when and how such alerts will be issued if the constraints are violated. Over-use can be treated with throttling, while under-capacity can be treated with dynamic provisioning of additional capacity when needed. Thus, an organization and its IT department can be assured that the API will not perform any worse than expected, and will not consume any resource other than as expected. The API consumer is guaranteed that it will get the availability and performance it requires, whether it is paying for this guarantee or not, in addition, the consumer guarantees that it will not require, and use, anything more than has been established.

Please note that any guarantee that leads to SLAs and Quota policies must be backed up with legally enforceable agreements between the consumer entity and the enterprise, otherwise you are just defining these policies for monitoring purposes only.

The process of defining the contract between an App and an API can be summarized as follows:

1. Determine the absolute maximum capabilities of your API (max supportable throughput, response time, availability, etc.)
2. Define an SLA policy for the API that warns at 50% of these criteria, and alerts at between 75%-80%, depending on the criteria.

3. Analyze the SOA Services that support the API (or systems, if you do not have an SOA infrastructure) and apply SLA policies to the services that match the SLAs defined for your API.
4. Determine the capabilities required by the App and codify these into an SLA policy for the App, with warnings and alerts as agreed.
5. Define quota policies that cease allowing access to the API, based on the alert levels of the SLA policy defined in step 4.
6. Define a contract that specifies the identity of the consumer and the API and then includes the app SLA and quota policies defined in steps 4 and 5.

In summary, the contract defines which app can access which API, when, and with what constraints (specified by the Plan). An API owner will know when the API has to have additional resources allocated to it, or load balancing and load sharing strategies defined for it, when the limits of its SLA policy are exceeded.

It's worth paying particular attention to step 3 in the above process. In most enterprises, the implementation of the API is not likely to reside in code in a supporting App Server. Rather it will be in services that already exist in the organization's SOA (or backend systems if the organization has not yet created its SOA). In this case, defining SLA policies and managing the services using them is as critical as managing the API itself, if for no other reason than the fact that if something is causing an SLA transgression the next step in the root-cause analysis is to go to the services supporting that API anyway.

Having a process like this in place, which utilizes Plans and Contracts, allows simpler consideration for charging an App owner based on the agreed Plan.

4.3 Monitoring API Usage

It is not possible to implement the ability to manage an API's quality of service (QoS) without the ability to monitor the activities and usage of the API. Your API management solution should enable you to monitor the performance and availability of an API and its consumers. It is also a best practice to monitor aspects of the usage of the API, such as most popular consumers, most popular operation, or operation consumption per consumer. Analytics from the running of the API can be very useful in the planning of extensions to the API, or in understanding the actual popularity and money making ability of the API in comparison to what was projected.

Furthermore, if your API is involved in financial transactions, you might well be required to make the API metrics part of your audit process. Law or other compliance policies could require you to adhere to specific security practices. For instance, you might need to provide auditors with verifiable logs of API requests and responses to enable the detection of unauthorized users.

Lastly, root cause analysis heavily relies on logging as a means of providing a window into the actions of the API at the time of error. Without some deep, but easily variable, logging capability, API issue resolution would become a horribly hit-and-miss activity.

API monitoring must meet the needs of these three key roles:

1. App Developer/provider
2. API Developer
3. API Operations staff

First, the App developer will need the ability to see what the requests and responses to the API look like and how they are treated by the API. They will also want the ability to be able to gauge the app's impact on the API via some common usage and performance metrics. This is not the same as the formalized SLAs mentioned in the previous section, but these metrics will help determine the SLA conformance.

Second, the API developer will need logging to help debug the code while building the API, and will need to be able to see metrics on the API performance during stress testing.

Lastly, once the API is in production, operations personnel will need logging to help in the root cause analysis of any usage issues that might arise, and will also need to monitor metrics on app usage and the API's performance in relation to their app.

5 Sharing the API

Once you've put the hard work and investment of resources into the creation of your production API, the equally hard work of sharing and popularizing it begins. Supporting the App developer community is a critical factor in the successful adoption of an API. App developers will be drawn to the API if it is obvious that they are being supported by the API provider as well as their colleagues. Without a vibrant and satisfied developer community, an API is unlikely to be used by a wide audience.

Ensuring profitable engagement with developers involves such considerations as interactive forums, excellent documentation, ability to test against an API, and a Software Development Kit (SDK) that makes it easier to deliver the API to the external developers.

Here are some best practices for supporting an API once it's been released:

1. Interact with and recognize your API Developers.
2. Create SDKs to make life easier for your API Developers.
3. Create great documentation about your API and how to use it.
4. Make testing against your API as easy as possible.
5. Monetize your API to assist in future cost benefit analysis.

5.1 Interact With and Recognize Your Developers

Developers typically like to communicate with other developers. App Developers in particular may work on geographically dispersed teams but still want to share thoughts and ideas. An API provider should provide an environment that makes that easy. Your API management tool should include the following features to help give developers the information and interaction they need:

- **FAQ:** A frequently asked question section can easily and quickly solve many of the questions a new developer will have. A good FAQ is expected and can greatly enhance the initial developer experience.
- **Blogs:** Provide a blog where your chief technical designers can discuss options, opinions, and possible enhancements for the API. Developers like to know how an API may be evolving. There are several good examples that

can be used as a model. For example, this is a good place to share implementation tips, and even information on upcoming releases. Even if there are issues with your API, developers will appreciate being notified of pitfalls and, ideally, workarounds.

- **Forums**: This is perhaps one of the most valuable communication tools available. It allows developers to share problems, solutions, and suggestions with each other. The forums should be constantly monitored by the API provider for proper content, but in most cases the API provider should not be a major participant. This is a medium for the developers.
- **Access to the API developers**: Providing a way for key external developers to communicate directly with the API development team can be very effective. This needs to be controlled to prevent overloading the internal team; however, providing an e-mail address (with the appropriate privacy measure implemented, of course) where external developers can ask specific questions or make suggestions is another way to foster a community of developers.

Your developers will also likely thrive if their efforts are recognized. The best champion you can have for your API is a happy, well informed and knowledgeable developer constantly using your community to help out other App Developers. Some of the ways you can recognize developers include:

- **Monetary**: Establish monetary incentives for higher levels of adoption or usage of the API.
- **Peer Recognition**: Establish superstar criteria that feature outstanding developers who have shown new and innovative ways to use the API. Peer recognition within the developer community is a strong incentive.
- **Involvement in API design or strategy:** When a developer is invited to help work through the future of the API, he or she will bring unique insight to the API owner. The developer will also be reverently treated by the App Developer community, an even more powerful advocate for the use of the API.

5.2 Create SDKs

Making it as easy as possible for App developers to use your API may necessitate the creation of a software development kit (SDK).For instance, on some platforms/devices support for OAuth or even REST and JSON are not as simple as it is on others. Consider the following reasons you might want to build an SDK:

- **Speed adoption** on a specific platform - for example, with an Objective-C SDK for iPhone. Lots of experienced developers are just starting off with Objective-C, so an SDK might be helpful.
- **Simplify the integration effort** required to work with your API - If key use cases are complex or need to be complemented by standard on-client processing.
- **Reduce bad or inefficient code** that might slow down service for everyone.
- **Create a developer resource** - Good SDKs help users write good code by providing good source code examples.
- **Market your API** to a specific community - you can upload the SDK to a samples or plug-in page for the platform's existing developer community.
- **Foster adoption across all devices—**you might produce APIs for all the currently popular platforms; if so, providing corresponding SDKs for each device will help encourage developers across the boards to use your API.

That said, you may not want to provide an SDK. For example, if your API is designed for adoption, is well-documented and is built using a standards-based approach, developers may be able to get rolling without a client SDK. Below are some reasons you might decide against providing an SDK:

- **Resources** - it's tough to provide an SDK for each platform you target. You'll have to choose carefully. Also, it can be time-consuming and expensive to create SDKs. (Tip: Packaging up internal samples or test cycles can provide a good starting point for SDK development.)
- **Maintenance and versioning -** convincing clients to upgrade to the newest version can be rough. Also, capturing the "local flavor" of each language you support can be a challenge.
- **Complexity** - On each platform there might be use cases you don't expect, such as keeping application-level secrets off clients, debugging, and so forth.

- **More is not necessarily better** - a few well-documented code-level samples can often be the best resource. Facebook provides Android, C#, iPhone, JavaScript, PHP, and Python libraries (all documented differently), but Twitter supplies none. Facebook has more resources but it is still difficult, at times, to understand how to use the Facebook APIs.

It is worth acknowledging that delivering all these SDKs will be as big a job as building the API. As discussed earlier, your job is not done by just building the API. There is no "Field of Dreams" in the API economy, so building the API is only half the job. Getting App developers excited about your API and telling others because of great documentation and SDKs is what will ultimately make that fantastic API successful.

5.3 Document Your API

You're going to need well-written documentation if you want to be successful at sharing your API. Though live interaction may be a better way of communication, there's just no substitute for non-ambiguous documentation when explaining complex or critical notions. Ideally, your documentation will have the quality of a small O'Reilly textbook or IBM Redbook. Anything else will not be used and may detract from the use of the API. Best practices suggest that the delivery of your documentation be via HTML, not MS Word or PDF. The HTML documents may be served by the organization's document or content management systems, but HTML is the best means of delivery. Why? Because all that content is then searchable, especially by people seeking information via Google, Bing, or any other Internet searching mechanism. This also acts as a marketing tool, letting prospective API developers know how great your API is. Another benefit is that online documentation can be immediately updated, so your users will always be referring to the latest and greatest version.

5.4 Make Testing Against Your API Easy

Everybody dislikes testing (except those who make a living out of it). However, testing is a critical factor in any software development cycle. This applies to your API, and applies equally to the apps that use your API. Untested or poorly tested API consumers will wreak havoc in your API ecosystem. Not only will your organization suffer from costly support, as will your "traditional" systems, but your other App

Developers and users will also be impacted. Providing a means of making testing easier for App developers is a must. Remember, a significant number of them are from Gen Y – they will not do anything that they don't need to do or can do in a more compelling way.

5.4.1 Provide a UI to Allow for API Interaction

A Test Console gives the user the ability to define an API call and to see the result. This is just as important in testing and debugging the output of an App in development as it is in understanding what the API does. Nothing is more attractive to a developer than to be able to do something easily and see an immediate result.

If the API consumer is a large "dev shop" or a business partner, being able to embed that functionality into their own ecosystem is an even more striking attraction.

5.4.2 Utilize the Sandbox

Unlike many other development activities within an organization, API development involves parties that are not part of a single organizational entity. This means that creating a number of replicated environments with different data is not practical. A sandbox provides a means of testing against sample data without creating another instance of your API. A sandbox simplifies the management, and hence the cost, to an API provider. More importantly, it gives the App developer an Internet-accessible place to test the functionality of the App. Once the Developer considers the App ready for use, i.e. ready for Production, you can use the Sandbox for acceptance testing. Moving the App into Production should be a simple promotion process within the API Platform. No actual configuration or code movement is necessary, as far as the API is concerned, if a production API definition already exists. It's just that the App is now authorized to access the Production URLs for the API.

5.5 Monetizing Your API

Very few internal IT initiatives make money. Fewer still were built with that purpose in mind. However, the best practice with APIs is to do exactly that. In contrast to the SOA, which could make money through internal charges in theory but rarely did in practice, API monetization actually works. It's a cultural, not technical, issue. The parties involved in creating and using an IT service are part of the culture where work is paid for by one entity and the result can then be used at no charge by any other part of that organization. From a traditional IT perspective, the idea of selling

access to an API might go against the grain of an existing IT culture. This is not the case with APIs. In the API Economy, Apps are considered a commercial quantity. So, too, is the data that they consume. Some organizations, such as Netflix, are valued based on the capability of their APIs.

Not every organization should build APIs to make money. However, once a successful API is built, App developers invariably want to do one of the following:

1. Enter into a business agreement to use the API because it offers an ability to make money that the App developer has realized, but the API provider organization just did not imagine.
2. Use the API to make money for them.

In scenario 1 the API **is being monetized** by way of the new business opportunity. In scenario 2 the API **should be monetized** by way of selling access to the API to the App Developer. In both of these scenarios, the notion of using the API for free is just not entertained. Culturally, this just does not happen. An enterprise that builds a best practice API should consider the monetizing of the API in its newly evolving business model, if for no other reason than because it makes future cost benefit analysis much easier and much more accurate.

About SOA Software

SOA Software is a leading provider of Enterprise API Management and SOA governance products that enable customers to plan, build, run and share enterprise services and APIs. The world's largest companies including Bank of America, Pfizer, and Verizon use SOA Software products to harness the power of their technology and transform their businesses. Gartner placed SOA Software in the Leaders Quadrant for the 2011 "Magic Quadrant for SOA Governance Technologies." The company is also recognized as a "Leader" by the Forrester Research Waves for Integrated SOA Governance , SOA Management, and SOA Life Cycle Management.

SOA Software's comprehensive suite of products helps enterprises drive developer adoption, reach new channels and gain business advantage, addressing all stages of the API lifecycle.

- Community Manager™ - a sophisticated developer community product to help enterprises attract, manage, and support the developers that build Apps using their APIs
- Portfolio Manager™ - provides planning capabilities to help ensure the alignment of APIs with strategic IT investments and business opportunities
- Lifecycle Manager™ - provides API and App lifecycle management capabilities to help customers build APIs that meet current and future business requirements
- Policy Manager™ - provides the services and APIs that support the rest of the product family. It is required by API Manager and Community Manager, and is a recommended option for Portfolio Manager and Repository Manager
- API Manager™ - an API proxy server providing security, monitoring, mediation and other runtime capabilities
- Enterprise API Platform-as-a-Service™ - Enterprise API Management as-a-Service - get all the benefits of the SOA Software Enterprise API Platform delivered as a cloud service.

For more information, please visit http://www.soa.com.

# # #

SOA Software, Inc.

12100 Wilshire Blvd, Suite 1800

Los Angeles, CA 90025

866-SOA-9876

www.soa.com

info@soa.com

http://www.soa.com/solutions/enterprise-api-management