API Strategy and Architecture: A Coordinated Approach

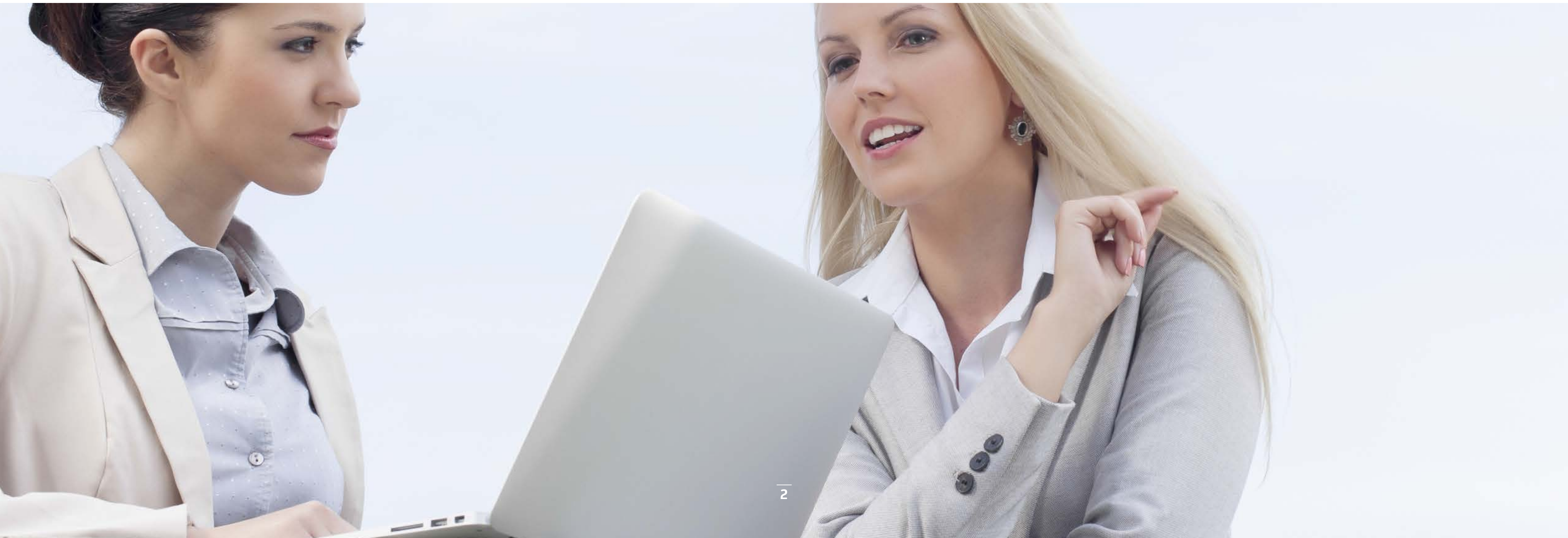
Introduction

The rise of the application programming interface (API) represents a business opportunity and a technical challenge. For business leaders, APIs present the opportunity to open new revenue streams and maximize customer value. But enterprise architects are the ones charged with creating the APIs that make backend systems available for reuse in new Web and mobile apps.

It is vital that all stakeholders understand that the business goals and technical challenges of an API program are intimately related. Program managers must take responsibility for clearly communicating the key business goals of a proposed API to the architects who will actually build the interface.

Architects, meanwhile, must take responsibility for maintaining a clear focus on these goals throughout the process of deploying an API infrastructure and designing the interface itself. All technical decisions should contribute to the creation of an interface that empowers developers to build client apps that end users will really value.

This eBook outlines best practices for designing results-focused APIs that will form the cornerstone of your API program's success.



Part 1: From SOA to API

Enterprise IT in the 21st Century has been characterized by a move towards opening up previously siloed databases and applications, so that data and functionality can be accessed across organizational boundaries or reused in new systems. The initial manifestation of this trend came with service oriented architecture (SOA) and the most recent has been the explosion of web-oriented APIs.

On one level, the "Web services" central to SOA represent the same thing as Web APIs. Both are interfaces used to open up backend systems. However, there are some fundamental differences between the two technologies, which are highly relevant to basic design decisions:

- The core technical difference is that SOA programs are focused on creating Web services to facilitate internal, server-to-server integrations, whereas Web APIs exist to speed the creation of Web and mobile-based apps, often of a customer-facing nature.
- SOA programs are generally driven by IT departments and focused on cost savings, but API programs more commonly originate with business development organizations and focus on generating new revenues.
- Most SOA projects are created by and for enterprise architects to help them more easily integrate heterogeneous systems and deliver new IT services. API programs, by contrast, should be focused on meeting the needs of application developers.

Figure 1: SOA vs APIs

	SOA	**APIs**
Integration Goal	Internal or to partners	External, often to customers
$$$ **Project Driver**	IT costs	Business revenues
Interface Consumer	Enterprise architects	App developers

Goals of API Design

Nevertheless, many API programs are growing out of previous SOA initiatives. Web services focused on internal or partner integrations are being opened up to developers—both within and outside the enterprise. During this process, it is important for API designers to remember that an API program has drivers and requirements quite different from the ones that initially led enterprises to open their IT assets via Web services.

With this in mind, the broad goals of API design in general can be defined as:

- Enabling self-service for app developers and app users alike
- Reducing barriers to accessing valuable enterprise resources
- Prioritizing the needs and preferences of client app developers
- Encouraging collaboration between and among internal and external resources
- Addressing the security and scaling issues of exposing IT assets to the open market

Above all, API design must be focused on maximizing the business value of the interface. In part two, we will take a closer look at how APIs add value to the business.

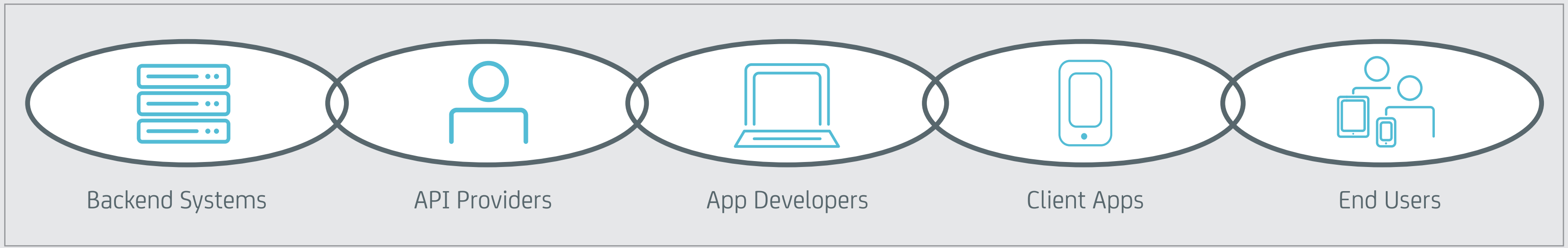
Part 2: The API Value Chain

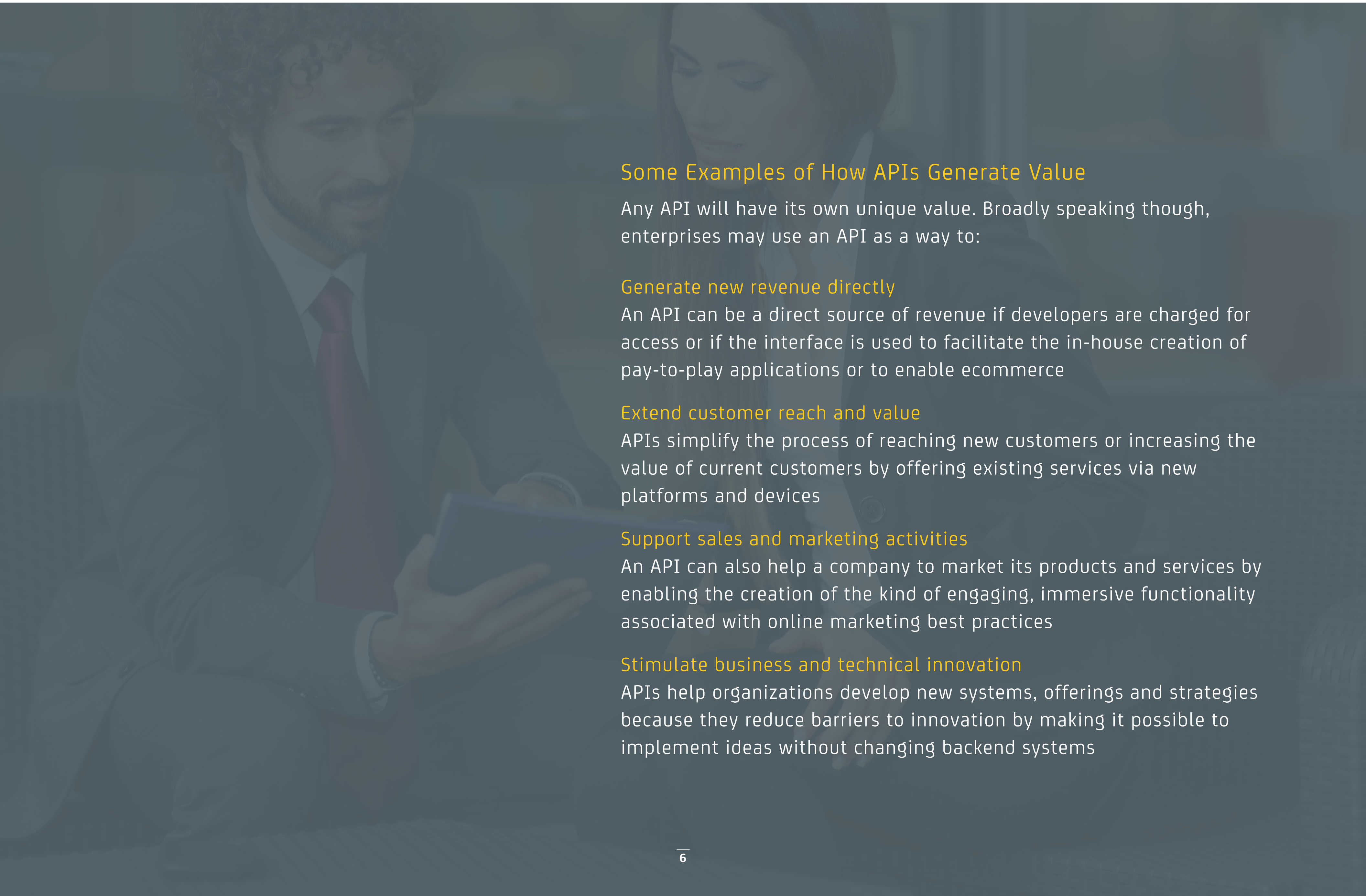
APIs may have no intrinsic value, but they do bring enormous value to the business. They do so through their backend data and the application functionality the interface enables. In this view, the API is simply a facilitator that allows systems with great organizational value to be reused in applications more likely to yield direct business value.

While this is a useful perspective, when looked at more closely, it becomes clear that a well-designed API is, in fact, a complex and powerful connector. It joins a wide variety of business assets—IT systems, internal and external personnel, client applications and customers—in order to more effectively realize the potential value of those assets. We can refer to this state of affairs as “the API value chain.”

It is important to understand that an API delivers value in this relatively complex way because it is otherwise quite easy to lose sight of the fact that APIs exist to deliver business value, not technical efficiencies. But while APIs deliver value more directly than SOA, they do so less directly than the browser-based Web—where a site can deliver actual sales leads or sales. APIs generate revenue in a more subtle way, by linking the various assets outlined below.

Figure 2: The API Value Chain





Some Examples of How APIs Generate Value

Any API will have its own unique value. Broadly speaking though, enterprises may use an API as a way to:

Generate new revenue directly

An API can be a direct source of revenue if developers are charged for access or if the interface is used to facilitate the in-house creation of pay-to-play applications or to enable ecommerce

Extend customer reach and value

APIs simplify the process of reaching new customers or increasing the value of current customers by offering existing services via new platforms and devices

Support sales and marketing activities

An API can also help a company to market its products and services by enabling the creation of the kind of engaging, immersive functionality associated with online marketing best practices

Stimulate business and technical innovation

APIs help organizations develop new systems, offerings and strategies because they reduce barriers to innovation by making it possible to implement ideas without changing backend systems

Making Design Decisions

API design decisions should be driven by what precisely the API will link—what will be on either side of the interface, both inside the organizational IT infrastructure and outside the enterprise firewall. Specifically, it is vital to answer these two questions:

- What systems are being exposed and where (and with whom) do they reside?
- Who are the target developers and what kind of apps will they build?

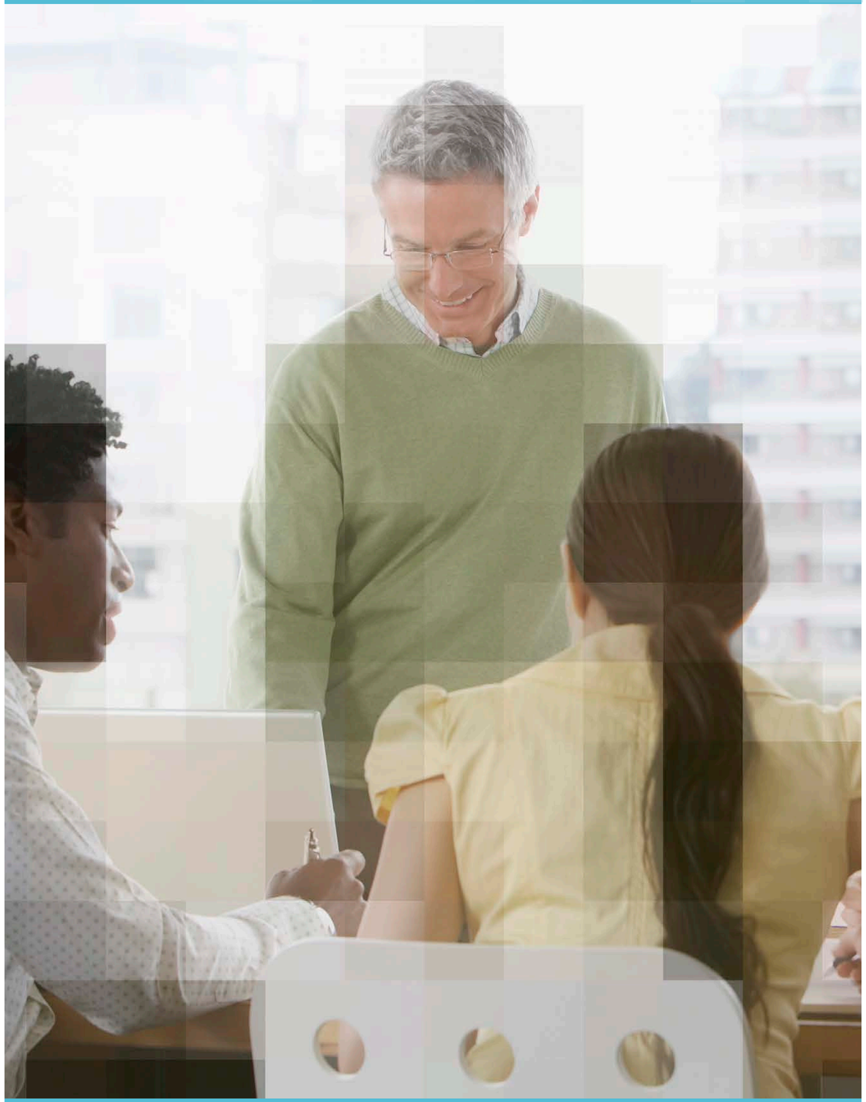
"Who are the target developers?" is a particularly important question and one that is relevant to the most fundamental way APIs are categorized—as "private" or "open." Private APIs are for use only within the enterprise or, in some cases, by partner organizations. Open APIs are made available to the wider community of external developers, who are free to create their own apps using the enterprise's backend resources.

Private APIs are closer in spirit to Web services. Typically, the goal of a private API will be to help internal developers, contactors or partners more efficiently create apps for use internally or externally. As with Web services, cost savings often represent the key driver as APIs allow new applications to be developed in a cost-effective manner. However, many private APIs are used to create public-facing Web and mobile apps that generate new business value more directly.

Open API programs tend to focus on adoption. By allowing third-party developers to access their APIs, enterprises aim to make their IT assets available to the widest possible user base. Therefore, developer adoption is a key metric for measuring the success of an open API. While there are fewer open APIs than private APIs, it is with the open APIs that both the greatest business opportunities and the most significant design challenges/technical risks lie.

In fact, not only do open APIs create a range of completely new integration design challenges (for example, how to open backend systems to external developers without exposing these systems to hackers), they also create new business risks. A poorly conceptualized open API program can lead an enterprise to cannibalize its own core business and potentially expose the enterprise's critical business assets to competitors.

Business considerations like these must drive technical design decisions. We will discuss how to align business considerations with technical decisions further in part three.



Part 3: Aligning API Design with Business Goals

Whereas SOA has historically sought to improve organizational processes, API programs seek to increase business revenues. Therefore, API design decisions must focus clearly on the core strategic business aims of the company's API program. Before starting to design an API, you must be clear about what problems the API program aims to solve, which opportunities it aims to realize and how it is going to do so. Specifically, it is important to answers these questions:

- What assets will be made available?
- How should the API make those assets available?
- What kind of applications could be built against the API?
- How can developers be motivated to use the API?
- How will the applications create value for the business?

Communication and collaboration are the keys to designing an API that addresses these challenges and opportunities. Throughout the process of designing, deploying and managing an interface, program managers and API architects must work closely to ensure they agree on their core strategic goals, what they will do to achieve these goals and how they will evaluate the outcomes of their efforts. Specifically, business and technical roles must be in agreement on:

- The objective and ideal end-state of the program
- The initial tasks that will allow the organization to work towards these objectives
- The key metrics that will be used to measure success
- The ongoing day-to-day tasks that will allow the program to keep hitting its targets

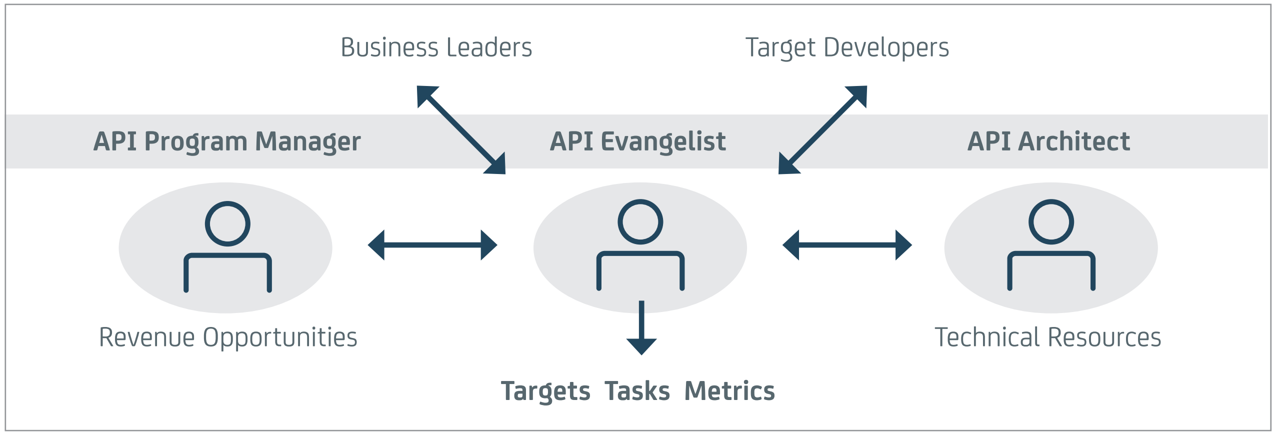
Assigning a Sponsor

To ensure business managers and architects stay on the same page, the program must have a “sponsor” who is able to span the divide that often appears between technical departments, business managers and app developers. Organizations often make the mistake of assigning this role to a non-technical marketing manager, but this “API evangelist” must be able to understand the organization’s architectural constraints and share the enthusiasms of app developers.

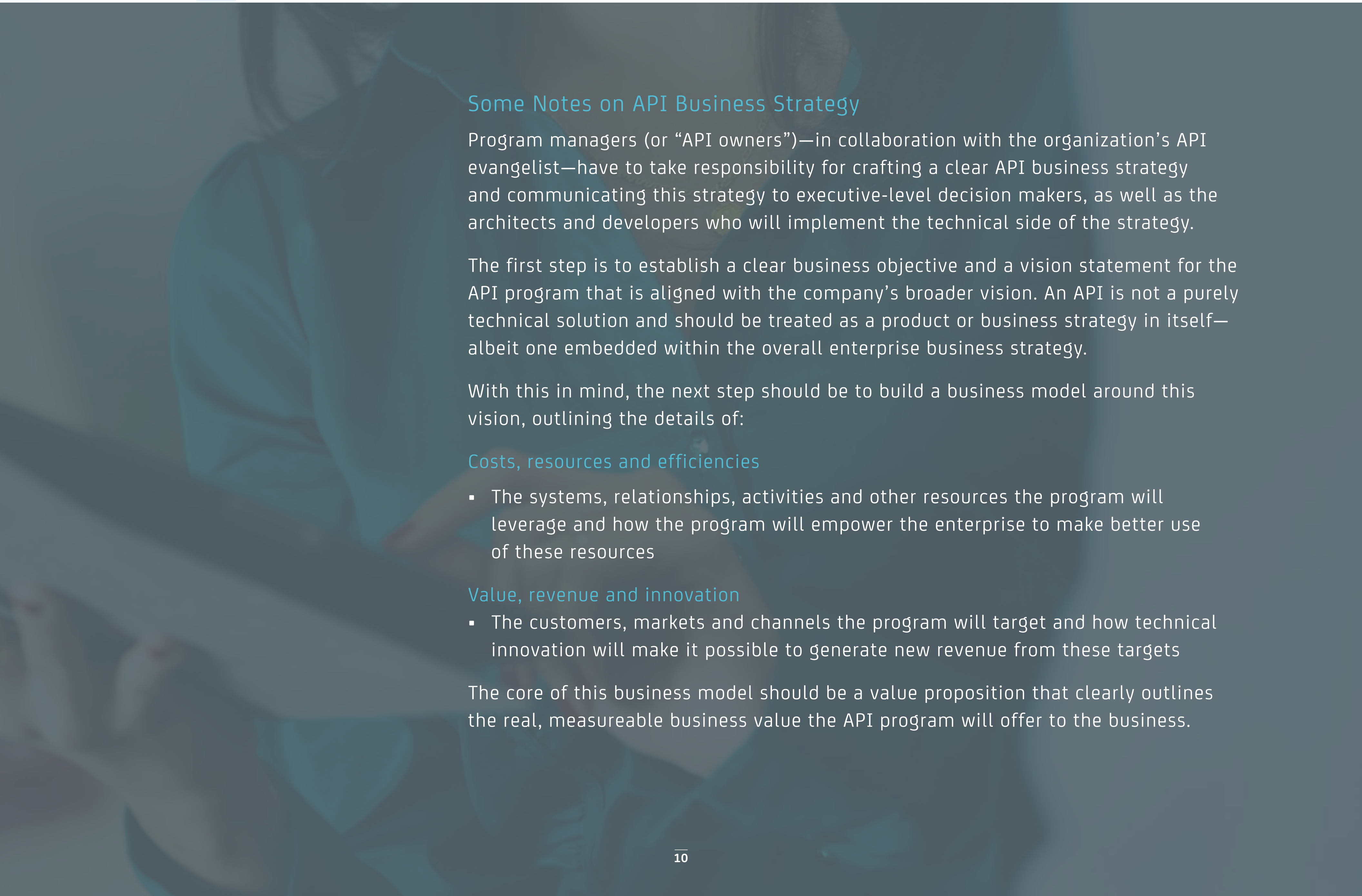
The evangelist’s role is to establish clear communication with all stakeholders, specifically:

- “Selling” the API program to executives and other senior decision makers
- Ensuring API architects understand program managers’ business goals
- Helping program managers understand architects’ technical resources and constraints
- Gathering information on target developers’ preferences and requirements

Figure 3: Aligning API Goals



Once communication has been established, and objectives, tasks and metrics have been agreed upon, the real work of API design can begin, which is what we shall discuss in part four.



Some Notes on API Business Strategy

Program managers (or “API owners”)—in collaboration with the organization’s API evangelist—have to take responsibility for crafting a clear API business strategy and communicating this strategy to executive-level decision makers, as well as the architects and developers who will implement the technical side of the strategy.

The first step is to establish a clear business objective and a vision statement for the API program that is aligned with the company’s broader vision. An API is not a purely technical solution and should be treated as a product or business strategy in itself—albeit one embedded within the overall enterprise business strategy.

With this in mind, the next step should be to build a business model around this vision, outlining the details of:

Costs, resources and efficiencies

- The systems, relationships, activities and other resources the program will leverage and how the program will empower the enterprise to make better use of these resources

Value, revenue and innovation

- The customers, markets and channels the program will target and how technical innovation will make it possible to generate new revenue from these targets

The core of this business model should be a value proposition that clearly outlines the real, measureable business value the API program will offer to the business.

Part 4: Designing a Usable API

From a purely technical perspective, designing an API is relatively easy. But, designing one that contributes real value to the business can complicate matters. Beyond functionality, enterprise architects must also consider business goals and the end-user experience.

This may be particularly challenging for anyone who is extending a SOA project into the API realm. In SOA, it is the architect's needs that are central, and user adoption is assumed. Consequently, architects with SOA backgrounds will commonly approach API design decisions with the assumption that interface and app users will have the same needs and biases they have. This almost always leads to bad design decisions.

With APIs, the design focus should not be on functionality, but on user experience. The key question is not "What functionality do I need to expose?" but "How will developers use this interface?" If developers do not want to use your API, then it has no value. Therefore, design must be developer-centric and focused on providing the lowest possible barrier to entry for the target developer audience.

Whether an API is published privately or openly, a good developer experience (DX) will be essential to its success. DX is significantly harder to quantify than exposed functionality. While it can be defined as the sum of interactions between the API provider and the developer, the result of this sum is less a number and more of a feeling: how does the interface make developers feel?

Obviously, this is a rather nebulous metric, but there are certainly practical steps you can take in the real world to understand how your developers are likely to feel about the different approaches you might take to designing your API. Specifically, you should:

- Create developer profiles
- Prototype and test your API in the field

Developer Profiles

You cannot create a usable API unless you know the needs and preferences of your target developer. There is a tendency to assume that developers who build client applications against APIs are young self-described “hackers,” obsessed with the latest languages and protocols. But, in many cases—particularly in private API scenarios—developers of enterprise services are still loyal to more ingrained ways of doing things.

The point is that every API project will need to address a particular developer audience in order to be successful. In some cases, this may be a very homogenous group with shared needs. In others, you may need to address a wide variety of preferences. Regardless, you must understand who will be using your API and how you can define the interface to ensure these developers can quickly and effectively use your backend resources.

So, the first step is to draw up a persona (or set of personas) to define the type (or types) of developer you are targeting with your APIs. This should include information on:

- Who they work for (and in what department) and why they are developing an app
- Programming skills, technical constraints and language/protocol preferences
- Personal temperament and in what context they work best

Prototyping

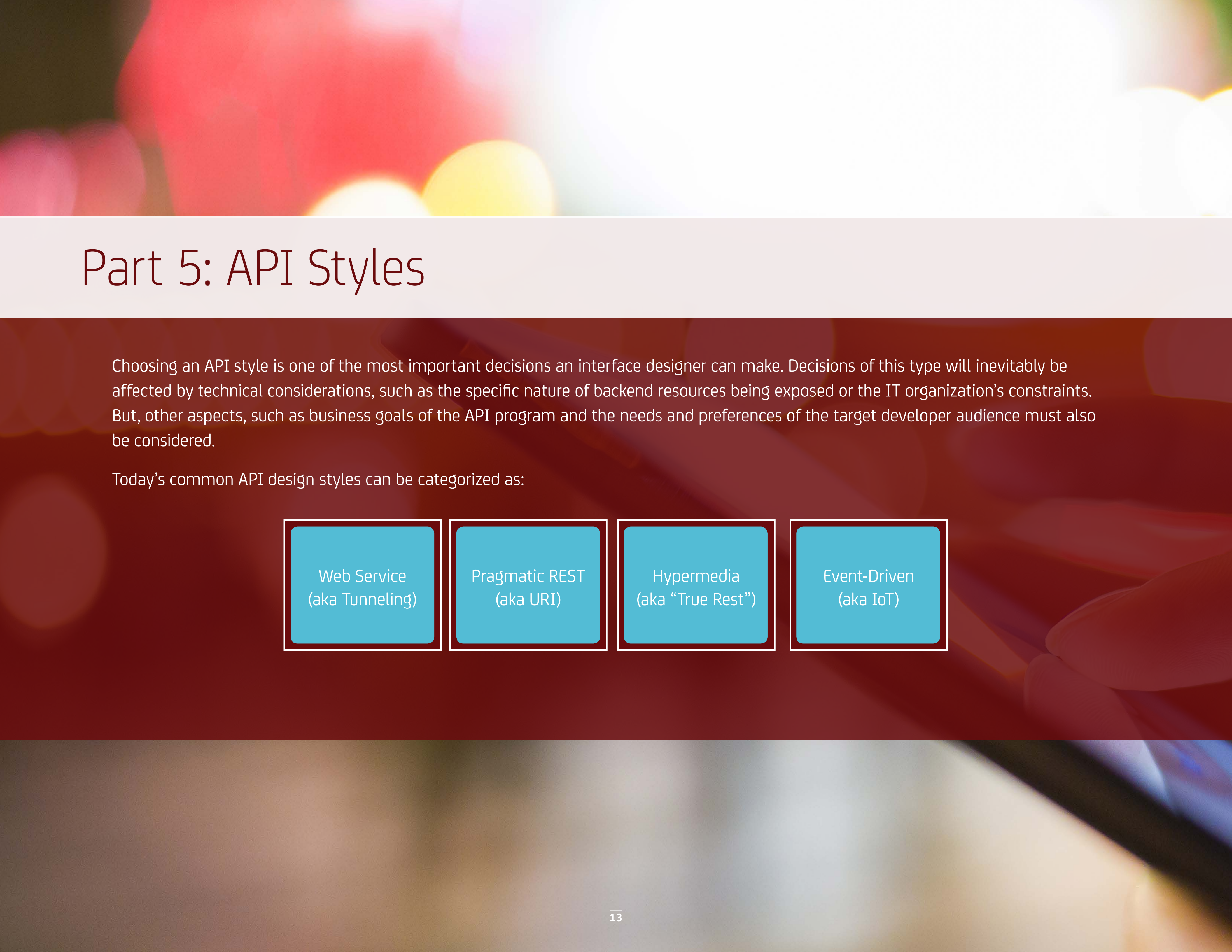
Once you have an understanding of the work goals, technical requirements and personal preferences of your target developers, you can start building an interface that addresses these criteria. However, before creating a production API bound to real data or backend systems, you should build a lightweight prototype that can more easily be changed. This prototype will allow you to test the design assumptions you have made based on your target persona.

Figure 4: Useful API Prototyping Tools

Various online tools exist that can simplify the process of building and testing lightweight API prototypes. Popular examples include...	1	Apiary aplary.lo	A design tool that makes it possible to quickly build an API prototype, without writing any code.
	2	RAML raml.org	API description languages that can help developers discover and begin to use your prototype interface.
	3	SWAGGER swagger.io	

One of the advantages of building a lightweight prototype based on “throwaway” data or functionality is it allows you to apply minimal security and provide the lowest possible barrier to entry for developers. This will make it possible to engage your target developers early on. They will write light apps to test your API design and provide feedback. Then, you can make changes to the interface and test again. After a couple of iterations, you should be on the right track.

Of course, none of this addresses how you will make fundamental, real-world decisions about interface design. In part five, we begin to discuss the actual API design options.



Part 5: API Styles

Choosing an API style is one of the most important decisions an interface designer can make. Decisions of this type will inevitably be affected by technical considerations, such as the specific nature of backend resources being exposed or the IT organization's constraints. But, other aspects, such as business goals of the API program and the needs and preferences of the target developer audience must also be considered.

Today's common API design styles can be categorized as:

- Web Service (aka Tunneling)
- Pragmatic REST (aka URI)
- Hypermedia (aka "True Rest")
- Event-Driven (aka IoT)

Web Service

The Web Service style is a transport-agnostic, operation-based approach to API design, which uses Web Services Description Language (WSDL) to describe interfaces. It comes from the SOA world, where Web Service interfaces were used to integrate heterogeneous networks. Therefore, this may be a good choice of style if your program involves extending SOA interfaces. The large amount of tooling that exists for Web Services also means that client applications can often be built quickly and easily.

However, there are serious limitations to using this style. First of all, while this transport-agnostic style can use Hypertext Transfer Protocol (HTTP), it is very inefficient in this context. Therefore, it is not the best choice if your services are being extended to the open Web.

Furthermore, it is only practical if your target developers are familiar with SOA standards like WSDL, Simple Open Access Protocol (SOAP) and Remote Procedure Call (RPC). For most client developers, the learning curve is likely to be steep.

This is particularly true in open API scenarios and especially those focused on mobile. As a rule, app developers don't like SOAP as a programming language and the tooling available for building Web Service clients tends not to support mobile. Practical considerations aside, there is a problem of perception: using the Web Service style could make your organization seem like a slow-moving "dinosaur," which is bound to decrease adoption among mobile app developers.

Pragmatic REST

The Pragmatic Representational State Transfer (REST) style is a simpler, more Web-centric approach to designing integration interfaces. This style, which uses URI instead of WSDL and is transport-specific (it exclusively supports HTTP), has largely taken over from the Web Service style in enterprise API design. Indeed, the term "Web API" is commonly used interchangeably with "RESTful API" and achieving "RESTfulness" is often considered to be a key goal of any interface design project.

In fact, most REST APIs in use today do not fully meet the REST criteria outlined in Roy Fielding's defining Ph.D thesis from 2000. Whereas, REST was defined to formally describe the kind of dynamic, hyperlinked interactions that power the Web, most Web APIs deal in the exchange of static data. Therefore, for the sake of argument, it is more accurate to refer to this design style as "Pragmatic REST."

It is easy to see why the Pragmatic REST style has become so popular. Because URI is intuitive and Web and mobile developers are mostly familiar with RESTful interfaces, developer adoption and productivity are likely to be high. Furthermore, the concentration on HTTP makes Pragmatic REST APIs ideal for developing today's Web and mobile applications. Right now, this is likely to be the go-to style for the majority of projects.

However, the Pragmatic REST style is not perfect for every context and future developments seem likely to challenge its dominance. There are definite tradeoffs with this style: it is limited to four methods, it can be "chatty" and URI design is not standard. Furthermore, with the Internet of Things (IoT) and Big Data greatly expanding and altering online networking, there are likely to be challenges to this specifically web-centric approach.

Hypermedia

The Hypermedia API design style is a task-based approach that aims to provide a more sustainable alternative to Pragmatic REST. Like Pragmatic REST, Hypermedia APIs are focused on URI, HTTP and RESTful standards generally. But in a sense, the Hypermedia Style represents a more faithful application of RESTful architecture, according to Fielding, which describes why the Web has proven to be so scalable.

As such, the Hypermedia approach is even more Web-centric: the hyperlinks and forms of the Web are mirrored in the way a Hypermedia API provides links to navigate workflow and template input to request information. Just as the RESTful architecture of the Web has proven to be highly scalable and evolvable, a well-designed Hypermedia API can continue to support new applications for years.

While this architectural approach is clearly an attractive option for enterprises seeking to create scalable APIs that reliably support Web and mobile applications over the long term, it is still an emerging design style with a notable lack of associated tooling. This may impact developer adoption rates and make it harder for those developers that do adopt the API to quickly create powerful client apps.

Event-Driven

While HTTP-focused styles like Pragmatic REST and Hypermedia may be ideal for the Web and mobile apps as we know them, the arrival of HTML5 and IoT is changing things—creating the possibility of more dynamic apps, but also demanding more lightweight interfaces.
In this context, the Event-Driven style has appeared as a transport-agnostic alternative, ideal for enabling apps to use WebSocket and other emerging alternatives to HTTP.

This style, which focuses on server- or client-initiated events, provides a low-overhead option, able to deliver better performance in scenarios where a large number of small messages are passing between the backend and the app. Therefore, it is ideal for IoT and a range of mobile use cases—especially instant messaging, video chat, multi-player games and so forth. It is also likely to appeal to the most cutting-edge developers.

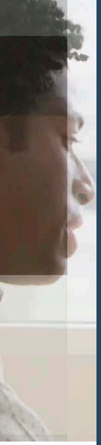
Of course, not all developers are that obsessed with being edgy and there are plenty of use cases where a conventionally RESTful approach will be more appropriate. HTTP is still the transport protocol that powers the Web and it does not accommodate client-sent events particularly well. Furthermore, the request-reply model this style is built upon makes building client apps more complex for developers.

Figure 5: Architectural Styles for API Design

Web Service	Pragmatic REST	Hypermedia	Event-Driven
SOA-Related	Ideal for Web and mobile apps	Highly web-centric	Appropriate for IoT and devices
Lots of tooling available	Familiar to most app devs	Scalable and evolvable	Lightweight and dynamic
Not suitable for mobile	May not be adaptable over time	Not familiar to many devs	Not suitable for standard scenarios

Your chosen style will depend on your technical constraints, business goals and developer preferences. Be careful not to fall into the trap of adopting a “fashionable” style if it is not appropriate for your specific context. At the same time, try to pick a style that will prove scalable and adaptable over the long term, as your resources change, your user audience grows and the very nature of online networking evolves.

No matter what style you choose, there are certain architectural components you will want your API to include. In part six, we will outline these components and how they will be organized.



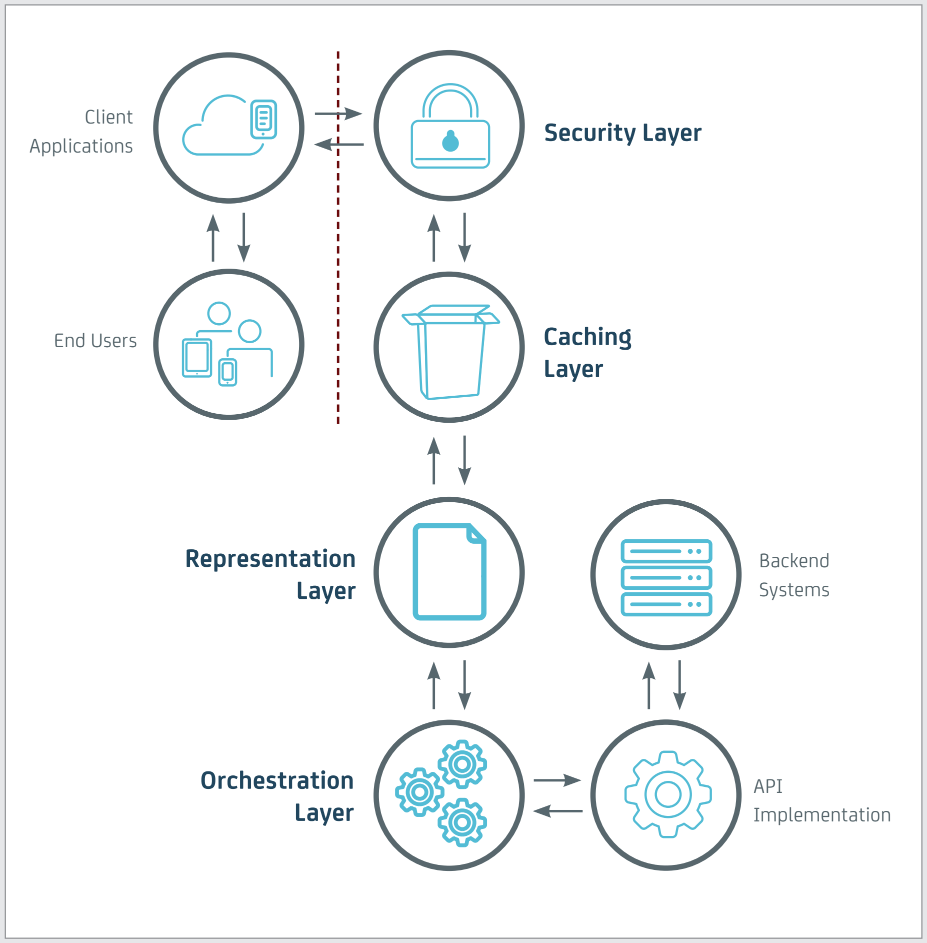
Part 6: API Architecture

The architectural design styles previously outlined should provide a model for how you design the architectural framework that enables the unique functionality of your API implementation. Certain use cases will call for the implementation of specific design styles. It is also important to note, however, that there are a number of components that should be included in any API architecture, no matter what the use case.

These common architectural components should not be built into the implementation of any given API. Instead, they should be deployed into a core API infrastructure that will sit between the organization's APIs and the client apps that leverage these APIs. Abstracting out these components makes it quicker and easier to design additional APIs, to update a range of APIs in unison and to ensure the smooth running of APIs, backend systems and client applications.

For maximum effectiveness, these components should be architected in a layered manner, so that all data traffic must pass through each of the layers named to the right, in the specified order.

Figure 6: Architectural Layers



The Security Layer

As well as opening up a world of business opportunities, APIs have the potential to open the enterprise to serious new security threats, by exposing sensitive backend systems and data to the outside world. APIs are vulnerable to many of the security threats that have plagued the Web plus a range of new API-specific threats. Therefore, it is vital to deploy strong, API-specific security at the edge of your API architecture.

This need for strong security can conflict with a basic goal of API design—a well-designed API makes it easy for developers to create apps that provide seamless access to enterprise resources. Strong security is likely to impact this ease of access. Deploying security in a centralized API architecture (rather than in the API implementation) will help mitigate this impact, as will enabling the use of flexible access management technologies like OAuth and OpenID Connect.

The Caching Layer

Interface efficiency will prove essential to providing the frictionless developer and end-user experiences necessary for meeting your API program's adoption and retention goals. One way to maximize API efficiency is by placing a caching layer near the edge of the API architecture. This layer should allow cached responses to be delivered for common requests, reducing pressure placed on the actual API implementations and backend resources.

The Representation Layer

Clearly, the presentation of your API should be as developer-friendly as possible. By abstracting this element away from the implementation, you can focus on centrally creating a welcoming way into your APIs, without impacting the APIs or backed resources themselves. This makes it significantly easier to present complex backend systems as Web and mobile-centric interfaces that developers can quickly understand and leverage to make powerful, user-friendly apps.

The Orchestration Layer

While some apps may be able to deliver value by accessing a single resource via a single API, the possibilities grow exponentially when you combine data from multiple APIs (including ones from other enterprises) and backend resources. Deploying an orchestration layer next to the interfaces themselves can enable such combinations, as well as simplify the process of composing new implementations from multiple backend resources.

The most efficient way to create a centralized API architecture is by deploying an API Management solution. In part seven, we will outline key API Management components.

Part 7: API Management

Building an infrastructure that centralizes common architectural components of secure, developer-centric APIs can significantly simplify the process of implementing APIs that add real value to your business. But, building such an infrastructure internally can be a significant challenge. Thankfully, a range of enterprise software vendors now offer "API Management" solutions that remove the need to develop this critical infrastructure in-house.

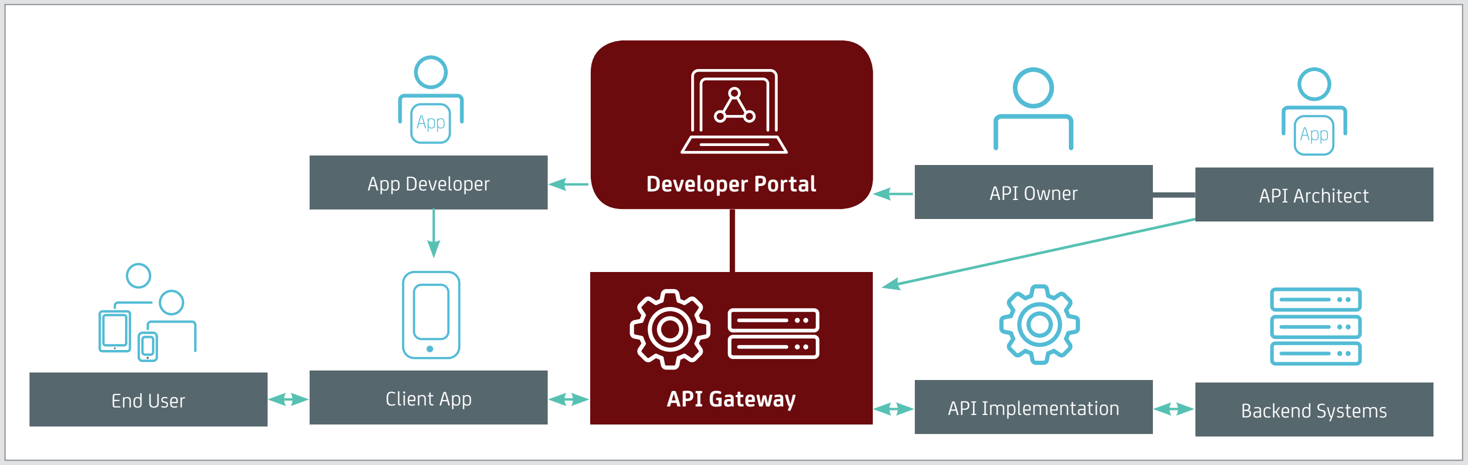
Furthermore, as the name suggests, API Management solutions also include functionality for managing and optimizing the performance of APIs over the long term. And the most powerful solutions also have features for building a Web-based interface through which developers can discover, learn about and access APIs—an absolutely vital part of presenting a developer-centric API, which cannot be built into the implementation itself.

API Management Components

An enterprise-level API Management solution will have two key components:

- API Gateway – Delivers the security, caching and orchestration functionality needed to deploy a core API architecture
- Developer Portal – Provides a customizable interface, through which developers access the APIs as well as documentation, community forums and other useful content

Figure 7: API Management Components



It is important to note that API Management is not simply a technical requirement. It will inevitably play a role in the business success of any enterprise API program. Managing the composition, performance and security of enterprise APIs is essential to ensuring the organization gets a good return on its investment in an API program. Likewise, it is vital to actively engage and manage developers to ensure they build apps that create business value.

For most enterprises, an API Management infrastructure will prove essential to designing, deploying and maintaining APIs that developers will use to create truly powerful new apps.

Discover API Management Essentials with the 5 Pillars of API Management eBook

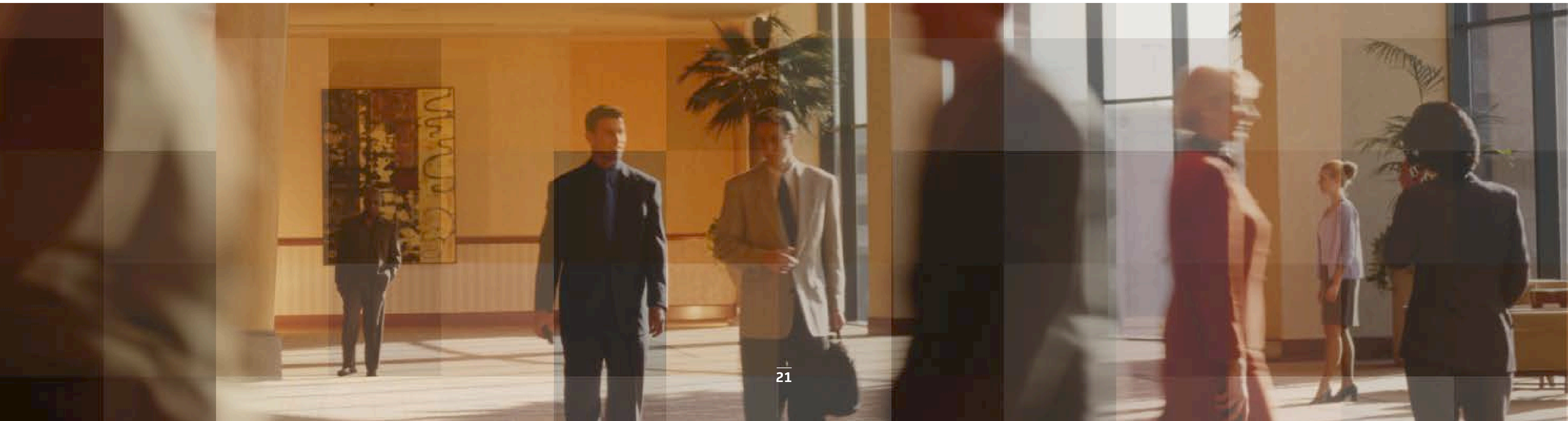
Conclusion

From an architectural standpoint, APIs represent an extension of SOA. Just as SOA created interfaces to open up legacy systems for reuse in new services that might span organizational boundaries, APIs are used to open the enterprise backend to developers building applications for mobile devices and the public Web. This is a significant extension and the design requirements for a Web API are likely to be very different from those for a SOA Web service.

Whereas, SOA programs are generally driven by the need for IT cost savings, API programs focus on generating new revenue streams. A Web API connects a range of existing business assets in order to create value in previously unforeseen ways. Good API design is always focused on business results. Therefore, API design and architecture practices must be aligned with the organization's business strategy, from the ground up.

API owners and architects must communicate to ensure they agree on key goals, how they aim to achieve these and how they will measure their success. To ensure communication is effective, an API evangelist who is able to bridge the gap between business and technical roles should parse the needs of business leaders, API owners, app developers and enterprise architects in order to negotiate an appropriate set of targets, tasks and metrics.

In practice, designing an API for business success usually means creating an interface that developers actually want to use. Therefore, before you build anything, it is vital to systematically research your developer audience in order to understand who your target developers are and what they want from an API. It can also be helpful to test any assumptions about developer preferences by offering lightweight prototype APIs.



Once you are ready to design your actual API implementation, you will have to choose the design style that best suits your project. Web Service APIs will suit internal programs aimed at developers with experience in SOA. Pragmatic REST APIs are more suitable for open API projects focused on mobile devices and the Web. The Hypermedia and Event-Driven styles are emerging as approaches that might prove more sustainable in the mobile and IoT-driven future.

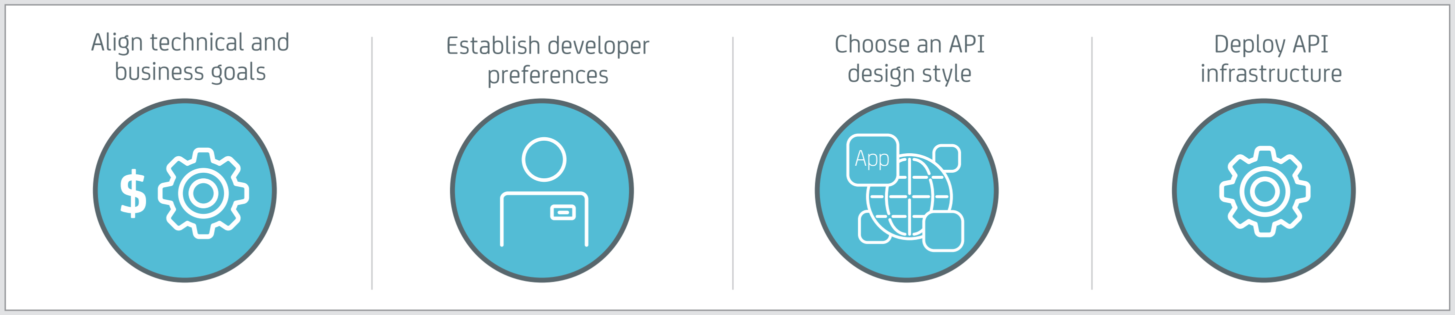
Whatever the style, there are certain architectural elements that all APIs must include—namely security, caching, representation and orchestration. For maximum efficiency and manageability, these elements should not be built into the individual API implementations. Instead, all of the APIs should leverage a central, layered API architecture that sits between the edge of the enterprise and the APIs themselves.

The most efficient and effective way to deploy a central API architecture—and ensure the API program remains successful over the long term—is to adopt an API Management solution. There are a variety of solutions on the market, but most include two common components:

- An API Gateway that provides security functionality and other key infrastructure
- A Developer Portal that simplifies the process of engaging and enabling developers

There is a lot at stake in today's enterprise API projects—huge business opportunities, significant security risks and much more. It is vital that you do your preparation before starting to build an API: align design goals with business goals; establish the preferences of your target developers; choose an appropriate implementation style; and deploy an API Management infrastructure. Then you will be ready to build a truly valuable API.

Figure 8: Prerequisites for Good Design


Align technical and business goals
$
Establish developer preferences
Choose an API design style
App
Deploy API infrastructure

Only CA API Management enables organizations to integrate systems, simplify app development and monetize data with the level of API security and protection enterprises need today. Learn about CA API Management at **ca.com/api**

About CA API Management

With over 300 API Management customers across sectors as diverse as communications, financial services, government and retail, CA Technologies offers industry-leading technology and know-how that helps organizations deliver value through APIs. CA provides a complete API Management solution, including a full-functioned API Gateway with military-grade security features, plus a developer portal offered in on-premises and SaaS versions. Learn about CA API Management at **ca.com/api**.

API Academy

API Strategy, Architecture and Design Services

The API Academy team consists of industry experts who have been brought together by CA Technologies, to develop free resources for the community and provide expert consulting services for organizations that want to take their API programs to the next level. To learn how the API Academy can help your organization with API strategy, architecture and design, visit **apiacademy.com**.

CA Technologies (NASDAQ: CA) creates software that fuels transformation for companies and enables them to seize the opportunities of the application economy. Software is at the heart of every business, in every industry. From planning to development to management and security, CA is working with companies worldwide to change the way we live, transact and communicate – across mobile, private and public cloud, distributed and mainframe environments. Learn more at **ca.com**.

CS200-131275