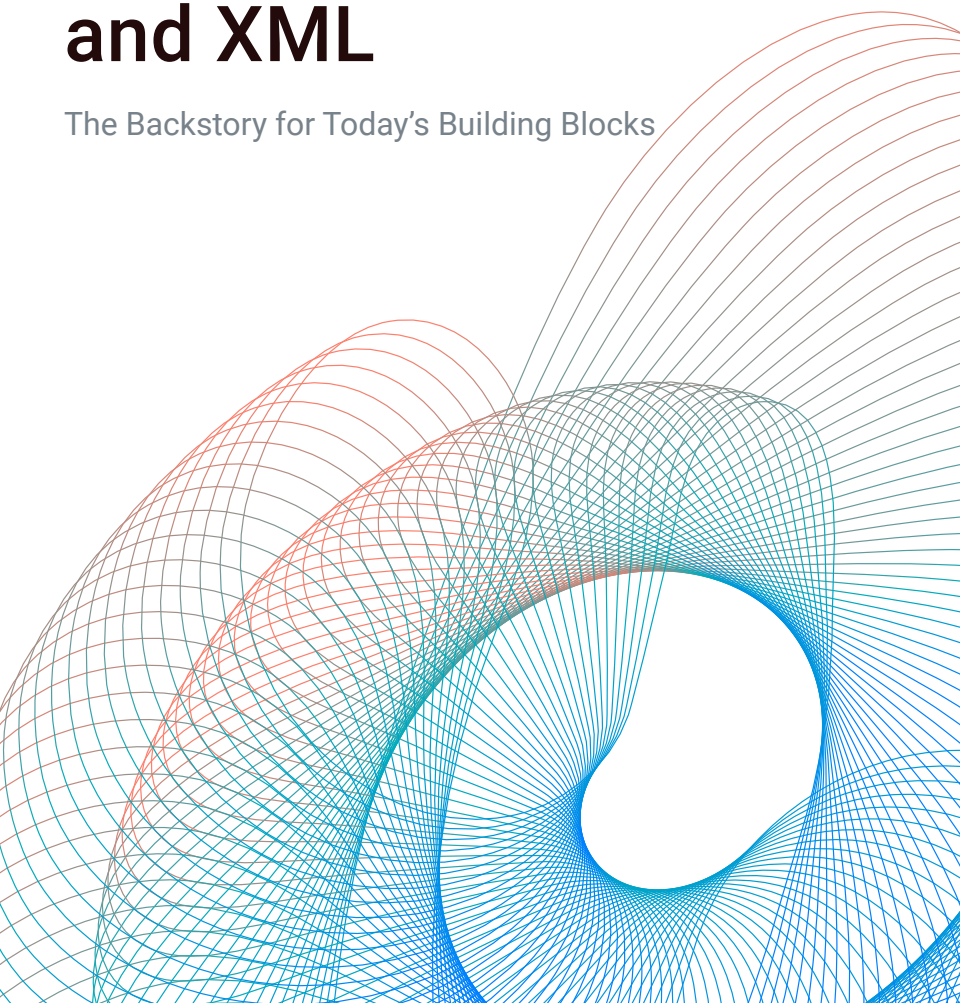
Kong

EBOOK

The Evolution of APIs: From RPC to SOAP and XML

The Backstory for Today's Building Blocks



Content

Foreword 3
APIs: Building Blocks of Modern Applications 4
How APIs Work 4
Common API Features 5
Defined interface 5
Encapsulation 6
Language independence 6
Security checks 6
Why Use APIs? 7
Development velocity 7
Leveraging new technology 7
Offloading responsibilities to third-party providers 7
Business Strategy 8
Examples of Popular APIs 8
Social Media 8
Cloud Services 8
Mobile Applications 9
Automation 9
APIs in the Distant Past 10
The Age of Early Computing 10
Subroutines and Functions 10
Scripting Languages 11
PC Programming Languagesv 11
APIs in the Near Past 12
The Birth of Windows 12
Object-O`riented Programming 12
Client-Server Computing 14
Distributed Computing Technology 14
APIs in the Early Internet Age 16
Enterprise Java Beans 16
.NET 16
XML 17
Web Services 18
Application Service Providers 20
Conclusion 21

Foreword

To work and live in today's digital world, we are unquestionably dependent on interconnected applications. These applications might be massive and highly complex, but they're also often constructed from reusable building blocks. We call each of these building blocks an Application Programming Interface—the API.

We interact with APIs every day[1], whether we realize it or not. When we're ordering a meal on Doordash, getting an Uber, or browsing Netflix, we're using APIs. We use an API when we bring up the mobile phone camera in our Instagram app or log into an app via Google or Facebook.

However, APIs aren't anything new. They were established as a result of the natural evolution of how computer software is written. Understanding what APIs are and how they came about is foundational to your ability to thrive as a software architect, application developer, or IT decision-maker.

This two-part eBook reviews what APIs are, how they work, and why they are so valuable. In Part One, we'll trace the evolution of APIs from early computing up through the early stages of the internet.

In Part Two, we'll look at modern APIs in cloud computing and peek into the future of APIs.

Along the way, we'll touch upon associated core technologies and how APIs have shaped the way services are created and consumed.

You don't need to be an experienced software programmer to read and understand this eBook. However, to get the best value, you should know how computer software works, what TCP/IP communication protocols are, and some basics on how API request and response messages are formatted, i.e., data interchange formats.

Let the journey begin.

[1] https://nordicapis.com/5-examples-of-apis-we-use-in-our-everyday-lives/

APIs: Building Blocks of Modern Applications

An API is software that allows a computer program to communicate with another program to exchange data or consume some type of service. These programs might run on the same computer or different computer networks separated by thousands of miles.

How APIs Work

An API connects a calling application (the client) and a called application (the service). The “service” could be a web server, a database server, a piece of middleware, or even a monolithic application written in COBOL. As far as the client is concerned, it only needs to know how to communicate with the API (endpoint URI, protocol, required parameters, etc.) without concern for the service’s underlying implementation.

The API is program logic to authenticate, validate, verify, and understand the user input. Then it must process the request and respond to the client.

The client starts by sending a request—a query to the API endpoint.

Once the API authenticates the client’s identity, it validates the request message and performs additional processing. Then, the API passes the request to the service (which is sometimes called the “backend” or “backend application”).

The service performs its operation based on the request, likely fetching some piece of requested data. It then returns a response message to the API, which includes the status of the operation (success or fail) and the requested data.

The API takes this message and sends it back to the client.

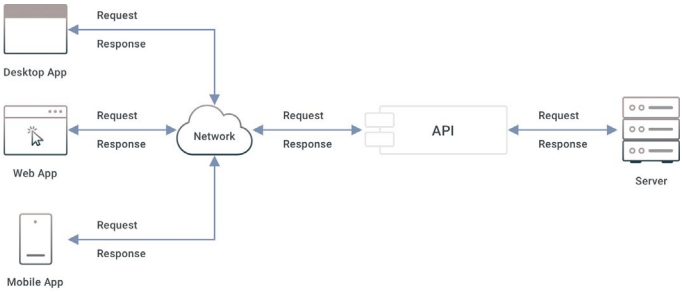


Figure: A Simple API Request-Response Scenario

As you can see, this is similar to a typical client-server application or human-computer interaction.

Common API Features

Regardless of how they are created or consumed, all APIs have some standard features.

Defined interface

An API will have a defined interface. At a minimum, an interface includes the following:

- Where the API is on the network
- What actions the API can perform
- What request and response message formats the client needs to provide to the API for those actions
- Security requirements to communicate with the API

The client can only interact with the API via this interface. Typically, these details are laid out according to a standard specification, such as WSDL, RAML, or OAS.

Message format

For the client and the API to understand each other, they need agreement on request and response messages format. The API provider specifies this format as part of the interface. For example, the format might require request messages to adhere to a specific data format (such as JSON or XML) and use a pre-defined data structure. The response, likewise, will also have a pre-defined format and structure.

URL

Finally, every API has a specific URL to be used by the client and the API to communicate. The URL includes the network protocol (e.g., HTTP or HTTPS), the hostname and the resource path (e.g., /v2/myservice).

Encapsulation

The implementation underneath an API is encapsulated, meaning developers consuming an API don't need to know its inner workings. A developer doesn't know if that API is calling other APIs behind the scenes, or what data that API is using internally, or how the API's program logic processes the data. The API is a black box and all that the API consumer knows about the API in the documented API contract (interface, message format, protocol).

Language independence

Because APIs are black boxes, developers can implement the API in any language without affecting the consumer's ability to invoke the API. For example, if your consumer application is Java-based, it can still call an API written in .NET or Python.

Security checks

Finally, well-written APIs will have built-in security checks. For example, to defend against Distributed Denial of Service (DDoS) attacks, most publicly available APIs will have protections like web application firewalls or API gateways. APIs may also require Secure Socket Layer (SSL) or Transport Layer Security (TLS) to encrypt the data in transit. APIs may also implement input validation or require authentication (username/password or API key) for access.

Why Use APIs?

Though all APIs have core commonalities, they can vary widely in their intent for consumption. Some organizations develop internal APIs accessible only within a company's network. Other organizations build partner APIs for consumption by partnering businesses. Lastly, some companies build public APIs for open or commercial use to provide access to their data.

To go deeper into usage, let's examine why organizations build APIs in the first place and why client applications consume them.

Development velocity

One clear benefit of API usage is reduced time and effort in software development. Developers can use services and data already available, building new features on top. This helps companies to stay competitive and decrease time-to-market.

There are no language barriers when it comes to writing or using APIs. Anyone can develop APIs in any language or framework. Similarly, the client application can be written in any language, independent of the language used for implementing the API.

Leveraging new technology

Companies can use APIs to take advantage of newer technologies. For example, an organization may replace its legacy, monolith application with a distributed, microservice-based architecture. Suppose the backend functionalities are exposed as APIs. In that case, the implementation team can change the back-end architecture to remain faithful to the interface contract.

Offloading responsibilities to third-party providers

Web application developers can completely offload user authentication by using APIs from third-party providers like Google or Facebook. Those developers no longer need to worry about sign-ups or saving passwords.

Another example of API reuse is e-commerce applications that allow payments with PayPal. The user authorizes the e-commerce application to debit their PayPal account. Once approved, the application calls the appropriate PayPal API to initiate a payment request. PayPal takes care of the rest of the transaction, including balance management and transferring payment.

Business Strategy

Companies also publish their APIs for increased brand awareness and monetization. For example, every time a payment is processed through PayPal's APIs, PayPal keeps a transaction fee (similar to banks). Some businesses allow limited access to their APIs for free and full access with a paid subscription.

Examples of Popular APIs

Let's conclude this chapter by considering several popular and publicly available APIs, seeing how they affect our day-to-day lives.

Social Media

The two behemoths of social media, Facebook and Twitter, offer their rich API libraries for third-party application development. Twitter has a suite of APIs for managing content, direct messages, advertising, and more. Facebook provides an equally rich ecosystem, allowing developers to work with Facebook data and applications programmatically. For example, you may want to build an application that helps users automate their Facebook posts. Using the Facebook APIs allows you to do that.

Similarly, social media developers can access APIs from other platforms like YouTube, Pinterest, Reddit, LinkedIn, or Instagram for content crossposting.

Cloud Services

When we use cloud services (such as AWS, Azure, GCP, or DigitalOcean), little do we realize that these are nothing but extensive groups of REST APIs. These APIs help us to access the service provider's backend infrastructure and applications. Cloud APIs are invoked every time you spin up or down a virtual

machine, save data in an S3 bucket, or call a Lambda function. Whether you are using a web interface, working in the command-line, or using an SDK, you send requests to an API provided by the cloud vendor.

Mobile Applications

Two major operating systems (OS) dominate the mobile device market. Each OS has made its APIs publicly available through software development kits (SDKs). For Apple products, the developer documentation is the first stop for iOS developers. Similarly, Android has its developer page.

Automation

Platforms like Zapier or IFTTT allow you to create automated workflows between multiple services, applications, and internet-connected devices. Behind the scenes, these platforms consume the target systems' APIs to build the workflows.

APIs in the Distant Past

Unlike many other developments in information technology, the API was not an invention from any single individual or company, nor did its evolution and adoption happen with the break-neck speed we usually see in today's technology landscape. Instead, the roots of the API predate the modern personal computer era, stretching back to the early days of computing.

The Age of Early Computing

In the pre-PC era of the 1960s and 1970s, computer programs were written for—and run on—large mainframes and minicomputers. Programming and data input started with punch cards. It then moved to dumb terminals that would interface humans with the computer. There was no dedicated network operating system. The internet was still in the making, and programs mostly ran within the boundaries of the physical computer.

At the time, programs were large, monolithic documents of code that you would key into a floppy disk. Having left this and other batch jobs with the computer operator to run overnight, you could only hope it would run without any errors.

Programming languages evolved as operating systems did. They began to incorporate debugging statements and no longer needed to perform low-level tasks like physical memory allocation. Programs could then use libraries of code already compiled and included with the operating system. This was the first step towards code reusability.

Subroutines and Functions

Next came the use of **subroutines**, breaking up code into manageable chunks and calling those chunks from the main program. This development significantly helped clean up existing code and promoted collaboration among developers. People could write a subroutine based on specific criteria, which other programmers could include in their code.

Subroutines soon gave way to their more versatile cousins: functions. A function could accept one or more arguments

(input data) and produce a predictable output. Functions were incorporated into programming languages for everyday tasks (like numerical computations or text manipulation). Developers could call those functions from their code.

Scripting Languages

Operating systems also exposed their scripting capabilities, which system administrators could use to automate most common workflows. These reusable scripts would include a series of operating system commands to perform specific system administration tasks.

Although they were not full-fledged programming languages, scripting languages had all the elements of high-level programming: looping, conditional branching, and user input validation. Since the operating system would run the script, there was no longer a need for compiling and runtimes.

PC Programming Languages

As the PC era started in the early 1980s, more and more programming languages were available on personal computers. One of the first native PC languages was Microsoft BASIC. This interpreted language shipped with the Microsoft DOS operating system. Other programming languages like C, PASCAL, and FORTRAN soon followed suit.

Commercial software development began to emerge as vendors started writing applications for the enterprise market and mass consumers. These applications had graphical user interfaces, made use of newer hardware like mice and graphics cards, and accessed low-level operating system code libraries.

The program packages would be divided into core binaries and supporting files. The binaries would start and load the application into memory. If a requested function weren't part of the application binary, it would also access the supporting file and load it. These were the very seeds of client-server computing.

APIs in the Near Past

A significant change happened in the late-80s to early-90s, when Microsoft brought about the Windows operating system. At the time, Windows was not mainstream and did not rise in popularity until Windows 3.1 launched much later.

The Birth of Windows

Windows was one of the forerunners of the modern, modularized, reusable programming paradigm. It introduced the use of the dynamic link library (DLL) that encapsulated different program modules. Windows—or Windows-hosted applications—could load or unload these DLLs into memory as they ran. For distributing software, custom DLLs had to be packaged with the application installer, but system-provided DLLs would already be present in the target machine.

Meanwhile, Microsoft exposed more operating system functions as part of the Windows Software Development Kit (SDK). The development community could use these functions to access hardware components, perform low-level storage operations, or render advanced graphics capabilities. Even today, SDKs are still offered by modern operating systems like iOS or Android. Many applications also provide their libraries through language-specific SDKs.

Two other significant developments came about during the early-to-mid-90s: the widespread adoption of Object-Oriented Programming (OOP) and the rise of the networked enterprise with client-server computing.

Object-Oriented Programming

The broader adoption of OOP in the latter part of the century helped developers build genuinely reusable components. An object is a piece of software code that encapsulates behavior and data which are exposed via functions. Objects were instantiated from classes, which were the blueprints for objects

For example, a developer could easily create three button objects from the same `CommandButton` class and use those on a dialog box object created from a `DialogBox` class. The programmer

wouldn't have to write any `CommandButton` or `DialogBox` code from scratch because the classes would be already provided by the programming language. In the calling application, the programmer would then write code for events when a user interacted with the dialog box or the buttons. The event code, for example, might call the associated object's methods to save input data to a file.

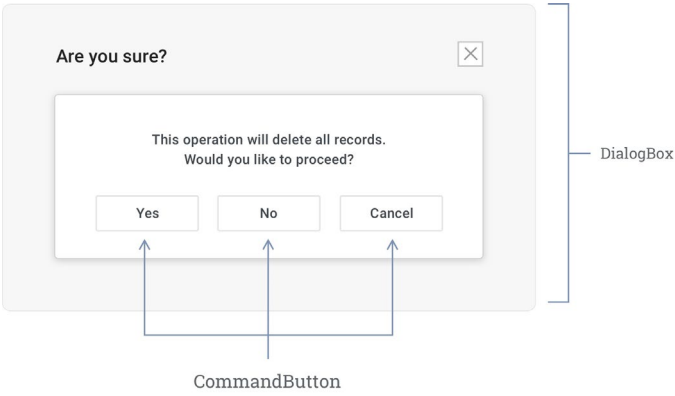


Figure: OOP example

The rise of OOP greatly facilitated corporate software development. Teams would create class libraries that they could use in subsequent projects. Programming languages came out with rich class libraries for interacting with the hardware, graphics, printers, file system, or even the network.

Alongside the rise of OOP, we saw growing popularity in network-capable operating systems, like Netware from Novell, Windows NT from Microsoft, and the first version of Linux in the early 90s. The majority of the internet-connected servers today run on some form of Linux.

The adoption of the internet by the masses was well underway by the mid-90s. However, the average business was still far from running application workloads over the HTTP protocol. They only used HTTP for their public-facing websites. Internally, though, companies were beginning to create network-centric applications using the client-server model.

Client-Server Computing

Early client-server computing used a two-tier architecture in which a server would run in a corporate network. At the same time, user workstations (the clients) would connect to it over TCP/IP or NetBIOS. The server could be a database server, an application server, or even a web server.

By the mid-90s, this two-tier approach soon gave way as Microsoft introduced two new development technologies for client-server programming: Component Object Model (COM) and Distributed COM (DCOM).

Distributed Computing Technology

COM was a specification that enabled inter-process communication (IPC) between software components in the same machine. COM used a client-server model in which a COM client accesses a COM server for its service. The server contains objects with specific interfaces that any client can use.

COM was platform-agnostic and language-independent, so developers could build software using COM components created by third-party vendors. Those components could communicate with one another based on their interfaces.

While COM enabled communication between components in the same machine, DCOM enabled COM components to run across different devices in a network. With DCOM, it was possible to write distributed applications communicating across machine boundaries.

The technique powering DCOM, Remote Procedure Call (RPC), is still available today. RPC allows a computer program to execute a function as local, even though that function is on a remote computer. The local application calls the locally running RPC client, which requests a remote service. The RPC process running on that remote service negotiates the remote function execution and sends back the response. The networking required to connect to a remote service is abstracted away to the locally running application.

Another distributed computing technology was Common Object Request Broker Architecture (CORBA), often seen as a competitor to DCOM for the future of distributed applications.

The CORBA specification deals with distributed objects—the same objects we know from object-oriented programming—running in the same machine or across a network. The object that requests service is known as the client object and the object it calls is known as the service provider object. Each object's functionality and expected inputs and outputs are described in a language specification called the Interface Definition Language (IDL).

Through DCOM and CORBA, we see the evolution of reusable application components and the foreshadowing of their usage across network boundaries. By the mid-to-late 90s, these technologies were roughly providing what modern APIs would one day provide: distributed client-server computing with reusable code, independent of language and platform.

Despite their popularity, neither DCOM nor CORBA adapted well with firewalls, nor could they handle internet-scale network traffic. Other methods and protocols quickly superseded them, as the internet as we know it today gained momentum.

APIs in the Early Internet Age

The third phase of API evolution emerged between the late 90s and the early years of this century. Enterprises realized the potential of the internet to do more than just deliver their public-facing corporate websites; they could use the internet to provide application services.

Java had been in the market for some time as an established product. Interactive Java applets were used heavily as embedded objects in websites. The natural tendency was to host the application in middleware—in this case, a Java Application Server. These applications were known as Enterprise Java Beans (EJB) components.

Enterprise Java Beans

EJB is a Java specification that allows developers to write distributed applications. EJB components are backend applications hosted in an application server like Apache Tomcat or IBM WebSphere. EJB applications can encapsulate business logic, scale with increasing demand, and be accessible with client applications written in Java.

The application server hosting EJBs can also host servlets and Java Server Pages (JSPs). Users would access the application through a browser, communicating with a web server. The web server would interact with the application server to dynamically serve the pages of the application. Since EJB components were portable, they could be hosted in any Java Application Server, thus allowing scalability.

.NET

At the start of the century, Microsoft released one of their revolutionary software development frameworks: .NET. This new framework was a significant departure from how applications would be written. It came with a framework (the .NET framework) which included the Common Language Runtime (CLR). CLR is the engine that takes care of low-level services needed by any application (such as type safety, memory management, garbage collection, exception handling, security, or thread management).

.NET also came with an extensive class library (such as ADO.NET for database connectivity or ASP.NET for web applications). Programming languages other than the ones developed by Microsoft could create .NET applications as long as they were using the framework. Like Enterprise Java, .NET also allowed creating distributed applications.

XML

Another development around that time was eXtensible Markup Language (XML). XML is a data interchange mechanism that allows anyone to create a set of rules to encode data. Data written in XML format is readable by both machines and humans. XML is a tag-based language, similar to HTML. However, unlike HTML, XML tags could be anything—based on the XML schema—used to describe the data. For example, a data interchange use case for customer data could define an XML schema to include first name, last name, email address, and phone. Based on that schema, an XML document could look like this:

```
<customers>
  <customer>
    <firstname>John</firstname>
    <lastname>Doe</lastname>
    <email>john.doe@email.com</email>
    <phone>111-222-3333</phone>
  </customer>
  <customer>
    <firstname>Joe</firstname>
    <lastname>Bloggs</lastname>
    <email>joe@emaiaddessl.com</email>
    <phone>444-555-6666</phone>
  </customer>
</customers>
```

Using XML, exchanging data between applications (and even between companies) suddenly became vastly simplified. All they needed to do was agree upon an XML schema for the data they would exchange for two communicating applications. The calling application would create an XML document to encode the input parameters for the remote application and send it over the network. The receiver would then use the same XML schema to parse the

document and unpack the input parameters. The results would then be returned using another XML document, perhaps even based on a different schema.

Distributed computing has matured, with different technologies and languages available. The development of application servers and platform-independent components shifted the focus of innovation toward finding a common protocol that would allow humans and applications to access remote applications, no matter where they ran. That protocol was HTTP, and the messaging format was XML, thus beginning the journey of web services.

Web Services

A web service is a piece of self-contained software accessible over HTTP. The client application accessing a web service might run on the same server or, more commonly, on a different server communicating over the network. The client and the web service could be written in different languages and run on different operating systems.

The reason clients and web services can be platform and language-independent is because they exchange information in XML. When a client wants to invoke a web service, it sends an XML message. The web service performs the requested operation and sends the result back in XML as well. The communication between the client and the web service happens over HTTP.

Now ask yourself: Are web services modern APIs?

The answer is yes. Web services are APIs because a web service exposes an application's data, performs some type of operation to retrieve that data, and is invoked by a client.

Apart from HTTP and XML, two other components are common to web services: SOAP and WSDL.

The Simple Object Access Protocol (SOAP) is a lightweight messaging framework for clients and services to exchange standardized XML data over a network. It uses XML as its message format. SOAP is relatively rigid in how it stipulates the request and response syntax. For message transfer, SOAP uses the application-

layer protocols HTTP or Simple Mail Transfer Protocol (SMTP). SOAP is still used by some web APIs today.

Web Services Description Language (WSDL) is an XML-based language to describe the functions of a web service. Remember that XML allows you to create custom, tag-based message formats—WSDL is just that. This is a document that a web service uses to describe its functionality and clients to find a web service.

Tight coupling versus loose coupling

In computer software engineering, tight coupling means all parts of an application stack are so dependent on one another that you cannot change one part without making a major change to other parts. For example, to publish an application as a web service, you need to ensure it abides by the SOAP specification. The data is encoded as XML, and uses HTTP for transport.

For example, if you wanted to publish a legacy application as a web service, but that legacy application only exposed its data in CSV format, this would mean extra work. You would either have to change the application code to export its data in XML, or you would have to write a wrapper to convert the CSV data to XML.

We'll return to our legacy application example shortly, but this leads us to the topic of data serialization—an area where distributed applications are still tightly coupled.

Data Serialization

Data serialization is a technique that breaks up the data objects of an application into a stream of bytes for transferring across a network or saving them into memory, disk file, or database. The serialization process ensures that the state of the data object is saved before it's transmitted. Serialization is necessary for complex computer data structures (for example, nested arrays). On the receiving end, a deserializer constructs the data object from the byte stream.

Using our legacy application example, the application needs to return some query results to a client. First, the application serializes the data as a byte stream. On the client end, a deserializer would

need to construct the byte stream response. If both the client and service used an older serialization format, changing any part of the message would mean extra work. In other words, the message format, in this case, is tightly coupled to the application.

Using a text-based, human-readable serialization format like XML (and later JSON) meant the message format could be changed any time. For example, if the web service needs to expose a new data field, this means changing the XML message schema. In other words, message formats were now loosely coupled to the functionality of the application.

The rise of web services also saw a new type of business in those days. These companies were known as Application Service Providers (ASP), not confused with Microsoft's web application authoring language, Active Server Pages. ASPs brought about what we know today as software-as-a-service (SaaS) providers.

Application Service Providers

The ASP model was created to remove the burden of developing, hosting, managing, and upgrading applications from enterprises. Instead, an ASP would develop a business application, host it in its own data center, and provide application access to one or more clients. The ASP would ensure the application and the client's data were available and accessible over the internet. It was responsible for providing security and performing regular backups, upgrades, and patches. The client would not own the application but could rent it for a fee.

ASPs couldn't scale as well as today's SaaS because there was no cloud, and renting data centers was—and still is—expensive. Additionally, the applications were not fully granular API-driven like today's SaaS solutions and often lacked load balancing. One of the most notable ASPs at that time was Salesforce, which later became a SaaS behemoth.

Software development itself was getting into a new era. Different standards and workflows were introduced, the most notable being source control with Git. Source control wasn't something new. Many organizations use systems like Subversion, CVS, or Visual SourceSafe. However, Git made it simpler and standardized. Also,

Git wasn't proprietary, meaning companies could use any Git-compliant source-control system.

Source control with Git would later be part of the Continuous Integration (CI) process. CI would ensure developers could branch from the source code to add new features and then merge it back, triggering a new software build. This enabled a quicker turnaround time for adding new features to applications—and web service APIs.

Conclusion

This brings us to the conclusion of Part One in our journey through the evolution of APIs. We started by looking at standard features for APIs and why they're so integral to software development. Then, we traced the evolution of APIs from early computing through pre-internet modern computing and right up through the early days of the internet.

We'll pick up in the cloud age in "Evolution of APIs: Cloud Age and Beyond (Part Two)", looking at the modern API developers build and work with today. We'll close Part Two by looking at what's next for APIs in the future. We'll see you there!

Konghq.com

Kong Inc.
contact@konghq.com

150 Spear Street, Suite 1600
San Francisco, CA 94105
USA