CHAPTER ONE

The Web As a Platform for Building Distributed Systems

THE WEB HAS RADICALLY TRANSFORMED THE WAY we produce and share information. Its international ecosystem of applications and services allows us to search, aggregate, combine, transform, replicate, cache, and archive the information that underpins today's digital society. Successful despite its chaotic growth, it is the largest, least formal integration project ever attempted—all of this, despite having barely entered its teenage years.

Today's Web is in large part the human Web: human users are the direct consumers of the services offered by the majority of today's web applications. Given its success in managing our digital needs at such phenomenal scale, we're now starting to ask how we might apply the Web's underlying architectural principles to building other kinds of distributed systems, particularly the kinds of distributed systems typically implemented by "enterprise application" developers.

Why is the Web such a successful application platform? What are its guiding principles, and how should we apply them when building distributed systems? What technologies can and should we use? Why does the Web model feel familiar, but still different from previous platforms? Conversely, is the Web always the solution to the challenges we face as enterprise application developers?

These are the questions we'll answer in the rest of this book. Our goal throughout is to describe how to build distributed systems based on the Web's architecture. We show how to implement systems that use the Web's predominant application protocol, HyperText Transfer Protocol (HTTP), and which leverage REST's architectural tenets. We explain the Web's fundamental principles in simple terms and discuss their relevance in developing robust distributed applications. And we illustrate all this with challenging examples drawn from representative enterprise scenarios and solutions implemented using Java and .NET.

The remainder of this chapter takes a first, high-level look at the Web's architecture. Here we discuss some key building blocks, touch briefly on the REpresentational State Transfer (REST) architectural style, and explain why the Web can readily be used as a platform for connecting services at global scale. Subsequent chapters dive deeper into the Web's principles and discuss the technologies available for connecting systems in a web-friendly manner.

Architecture of the Web

Tim Berners-Lee designed and built the foundations of the World Wide Web while a research fellow at CERN in the early 1990s. His motivation was to create an easy-to-use, distributed, loosely coupled system for sharing documents. Rather than starting from traditional distributed application middleware stacks, he opted for a small set of technologies and architectural principles. His approach made it simple to implement applications and author content. At the same time, it enabled the nascent Web to scale and evolve globally. Within a few years of the Web's birth, academic and research websites had emerged all over the Internet. Shortly thereafter, the business world started establishing a web presence and extracting web-scale profits from its use. Today the Web is a heady mix of business, research, government, social, and individual interests.

This diverse constituency makes the Web a chaotic place—the only consistency being the consistent variety of the interests represented there; the only unifying factor the seemingly never-ending thread of connections that lead from gaming to commerce, to dating to enterprise administration, as we see in Figure 1-1.

Despite the emergent chaos at global scale, the Web is remarkably simple to understand and easy to use at local scale. As documented by the World Wide Web Consortium (W3C) in its "Architecture of the World Wide Web," the anarchic architecture of today's Web is the culmination of thousands of simple, small-scale interactions between agents and resources that use the founding technologies of HTTP and the URI.*

* "Architecture of the World Wide Web, Volume One," *http://www.w3.org/TR/webarch/*.

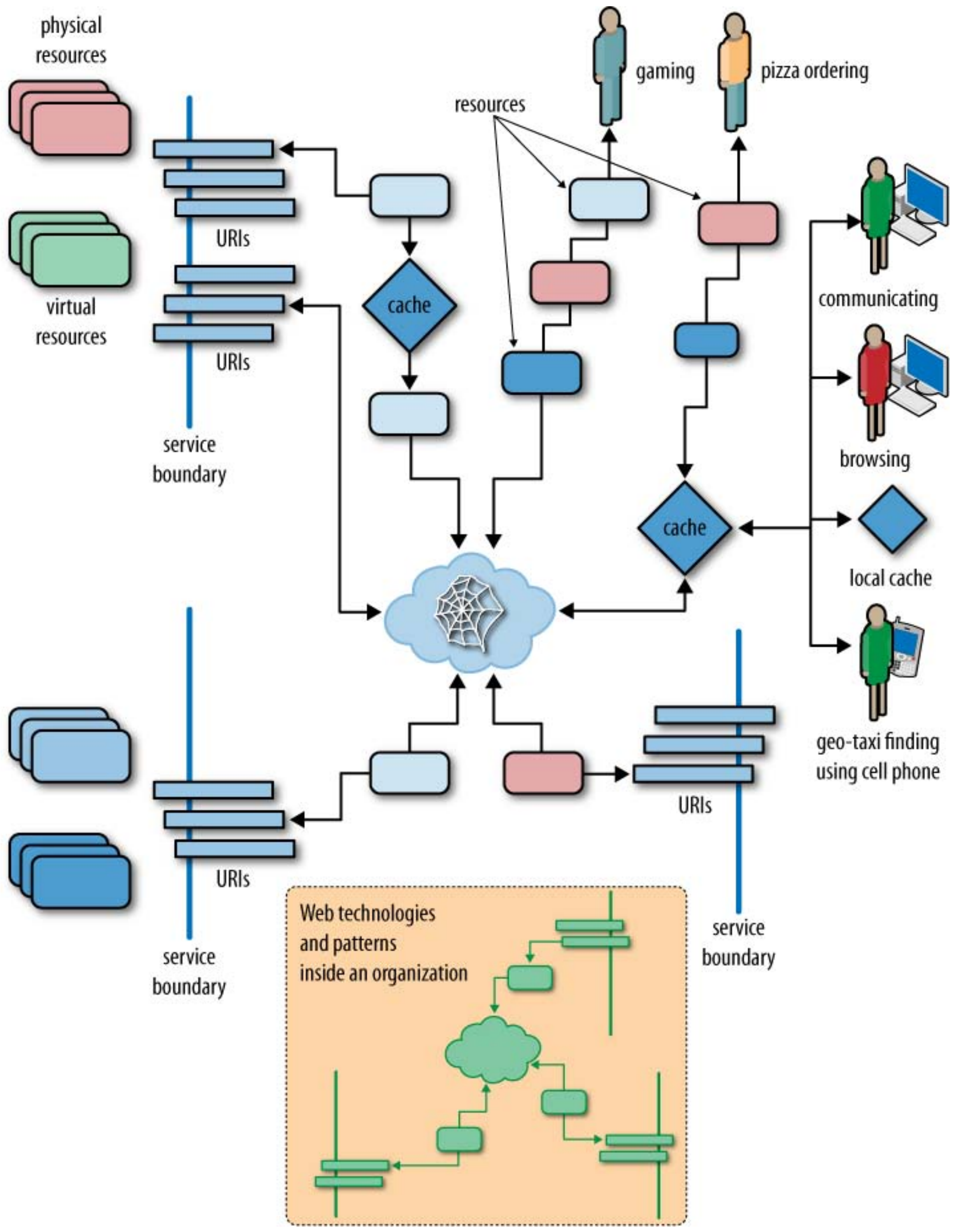


Figure 1-1. *The Web*

The Web's architecture, as portrayed in Figure 1-1, shows URIs and resources playing a leading role, supported by web caches for scalability. Behind the scenes, service boundaries support isolation and independent evolution of functionality, thereby encouraging loose coupling. In the enterprise, the same architectural principles and technology can be applied.

Traditionally we've used middleware to build distributed systems. Despite the amount of research and development that has gone into such platforms, none of them has managed to become as pervasive as the Web is today. Traditional middleware technologies have always focused on the computer science aspects of distributed systems: components, type systems, objects, remote procedure calls, and so on.

The Web's middleware is a set of widely deployed and commoditized servers. From the obvious—web servers that host resources (and the data and computation that back them)—to the hidden: proxies, caches, and content delivery networks, which manage traffic flow. Together, these elements support the deployment of a planetary-scale network of systems without resorting to intricate object models or complex middleware solutions.

This low-ceremony middleware environment has allowed the Web's focus to shift to information and document sharing using hypermedia. While hypermedia itself was not a new idea, its application at Internet scale took a radical turn with the decision to allow broken links. Although we're now nonplussed (though sometimes annoyed) at the classic "404 Page Not Found" error when we use the Web, this modest status code set a new and radical direction for distributed computing: it explicitly acknowledged that we can't be in control of the whole system all the time.

Compared to classic distributed systems thinking, the Web's seeming ambivalence to dangling pointers is heresy. But it is precisely this shift toward a web-centric way of building computer systems that is the focus of this book.

Thinking in Resources

Resources are the fundamental building blocks of web-based systems, to the extent that the Web is often referred to as being "resource-oriented." A resource is anything we expose to the Web, from a document or video clip to a business process or device. From a consumer's point of view, a resource is anything with which that consumer interacts while progressing toward some goal. Many real-world resources might at first appear impossible to project onto the Web. However, their appearance on the Web is a result of our abstracting out their useful *information* aspects and presenting these aspects to the digital world. A flesh-and-blood or bricks-and-mortar resource becomes a web resource by the simple act of making the information associated with it accessible on the Web. The generality of the resource concept makes for a heterogeneous community. Almost anything can be modeled as a resource and then made available for manipulation over the network: "Roy's dissertation," "the movie *Star Wars*," "the invoice for the books Jane just bought," "Paul's poker bot," and "the HR process for dealing with new hires" all happily coexist as resources on the Web.

Resources and Identifiers

To use a resource we need both to be able to identify it on the network and to have some means of manipulating it. The Web provides the Uniform Resource Identifier, or URI, for just these purposes. A URI uniquely identifies a web resource, and at the same time makes it addressable, or capable of being manipulated using an application protocol such as HTTP (which is the predominant protocol on the Web). A resource's URI distinguishes it from any other resource, and it's through its URI that interactions with that resource take place.

The relationship between URIs and resources is many-to-one. A URI identifies only one resource, but a resource can have more than one URI. That is, a resource can be identified in more than one way, much as humans can have multiple email addresses or telephone numbers. This fits well with our frequent need to identify real-world resources in more than one way.

There's no limit on the number of URIs that can refer to a resource, and it is in fact quite common for a resource to be identified by numerous URIs, as shown in Figure 1-2. A resource's URIs may provide different information about the location of the resource, or the protocol that can be used to manipulate it. For example, the Google home page (which is, of course, a resource) can be accessed via both *http://www.google.com* and *http://google.com* URIs.

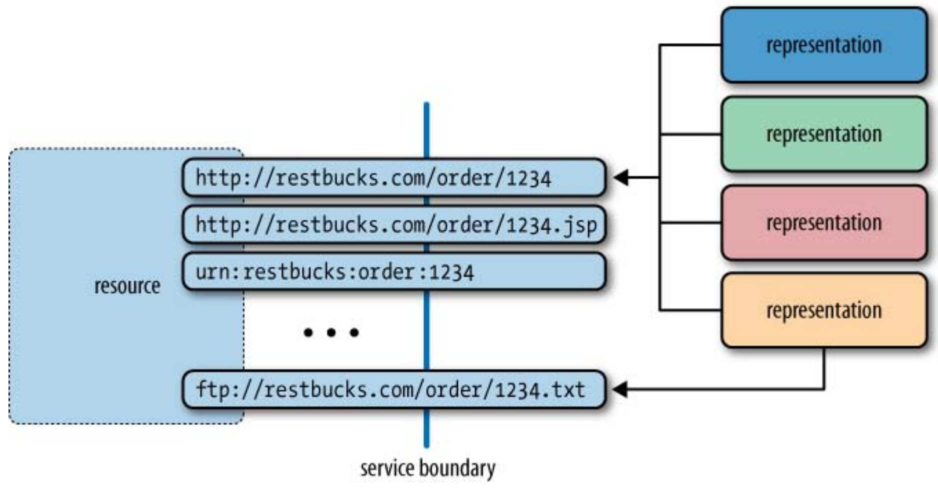


Figure 1-2. *Multiple URIs for a resource*

NOTE

Although several URIs can identify the same resource, the Web doesn't provide any way to compute whether two different URIs actually refer to the same resource. As developers, we should never assume that two URIs refer to different resources based merely on their syntactic differences. Where such comparisons are important, we should draw on Semantic Web technologies, which offer vocabularies for declaring resource identity sameness. We will discuss some useful techniques from semantic computing later in the book.

A URI takes the form *<scheme>:<scheme-specific-structure>*. The *scheme* defines how the rest of the identifier is to be interpreted. For example, the *http* part of a URI such as *http://example.org/reports/book.tar* tells us that the rest of the URI must be interpreted according to the HTTP scheme. Under this scheme, the URI identifies a resource at a machine that is identified by the hostname *example.org* using DNS lookup. It's the responsibility of the machine "listening" at *example.org* to map the remainder of the URI, *reports/book.tar*, to the actual resource. Any authorized software agent that understands the HTTP scheme can interact with this resource by following the rules set out by the HTTP specification (RFC 2616).

NOTE

Although we're mostly familiar with HTTP URIs from browsing the Web, other forms are supported too. For example, the well-known FTP scheme* suggests that a URI such as *ftp://example.org/reports/book.txt* should be interpreted in the following way: *example.org* is the Domain Name System (DNS) name of the computer that knows File Transfer Protocol (FTP), *reports* is interpreted as the argument to the `CWD` (Change Working Directory) command, and *book.txt* is a filename that can be manipulated through FTP commands, such as `RETR` for retrieving the identified file from the FTP server. Similarly, the *mailto* URI scheme is used to identify email addresses: *mailto:enquiries@restbucks.com*.

The mechanism we can use to interact with a resource cannot always be inferred as easily as the HTTP case suggests; the URN scheme, for example, is not associated with a particular interaction protocol.

In addition to *URI*, several other terms are used to refer to web resource identifiers. Table 1-1 presents a few of the more common terms, including *URN* and *URL*, which are specific forms of URIs, and *IRI*, which supports international character sets.

* RFC 1738, Uniform Resource Locators (URLs): *http://www.ietf.org/rfc/rfc1738.txt*.

Table 1-1. *Terms used on the Web to refer to identifiers*

Term	Comments
URI (Uniform Resource Identifier)	This is often incorrectly referred to as a "Universal" or "Unique" Resource Identifier; "Uniform" is the correct expansion.
IRI (International Resource Identifier)	This is an update to the definition of *URI* to allow the use of international characters.
URN (Uniform Resource Name)	This is a URI with "urn" as the scheme, used to convey unique names in a particular "namespace." The namespace is defined as part of the URN's structure. For example, a book's ISBN can be captured as a unique name: *urn:isbn:0131401602*.
URL (Uniform Resource Locator)	This is a URI used to convey information about the way in which one interacts with the identified resource. For example, *http://google.com* identifies a resource on the Web with which communication is possible through HTTP. This term is now obsolete, since not all URIs need to convey interaction-protocol-specific information. However, the term is part of the Web's history and is still widely in use.
Address	Many think of resources as having "addresses" on the Web and, as a result, refer to their identifiers as such.

URI Versus URL Versus URN

URLs and URNs are special forms of URIs. A URI that identifies the mechanism by which a resource may be accessed is usually referred to as a URL. HTTP URIs are examples of URLs.

If the URI has *urn* as its scheme and adheres to the requirements of RFC 2141 and RFC 2611,* it is a URN. The goal of URNs is to provide globally unique names for resources.

* *http://www.ietf.org/rfc/rfc2141.txt* and *http://www.ietf.org/rfc/rfc2611.txt*

Resource Representations

The Web is so pervasive that the HTTP URI scheme is today a common synonym for both identity and address. In the web-based solutions presented in this book, we'll use HTTP URIs exclusively to identify resources, and we'll often refer to these URIs using the shorthand term *address*.

Resources must have at least one identifier to be addressable on the Web, and each identifier is associated with one or more *representations*. A representation is a transformation or a view of a resource's state at an instant in time. This view is encoded in one or more transferable formats, such as XHTML, Atom, XML, JSON, plain text, comma-separated values, MP3, or JPEG.

For real-world resources, such as goods in a warehouse, we can distinguish between the actual object and the logical "information" resource encapsulated by an application or service. It's the information resource that is made available to interested parties through projecting its representations onto the Web. By distinguishing between the "real" and the "information" resource, we recognize that objects in the real world can have properties that are not captured in any of their representations. In this book, we're primarily interested in representations of information resources, and where we talk of a resource or "underlying resource," it's the information resource to which we're referring.

Access to a resource is always mediated by way of its representations. That is, web components *exchange* representations; they never access the underlying resource directly—the Web does not support pointers! URIs relate, connect, and associate representations with their resources on the Web. This separation between a resource and its representations promotes loose coupling between backend systems and consuming applications. It also helps with scalability, since a representation can be cached and replicated.

NOTE

The terms *resource representation* and *resource* are often used interchangeably. It is important to understand, though, that there is a difference, and that there exists a one-to-many relationship between a resource and its representations.

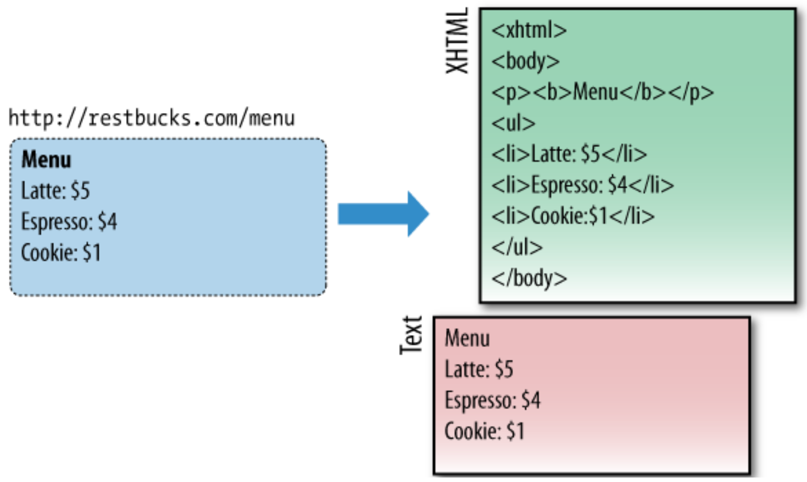
There are other reasons we wouldn't want to directly expose the state of a resource. For example, we may want to serve different views of a resource's state depending on which user or application interacts with it, or we may want to consider different quality-of-service characteristics for individual consumers. Perhaps a legacy application written for a mainframe requires access to invoices in plain text, while a more modern application can cope with an XML or JSON representation of the same information. Each representation is a view onto the same underlying resource, with transfer formats negotiated at runtime through the Web's *content negotiation* mechanism.

The Web doesn't prescribe any particular structure or format for resource representations; representations can just as well take the form of a photograph or a video as they can a text file or an XML or JSON document. Given the range of options for resource representations, it might seem that the Web is far too chaotic a choice for integrating computer systems, which traditionally prefer fewer, more structured formats. However, by carefully choosing a set of appropriate representation formats, we can constrain the Web's chaos so that it supports computer-to-computer interactions.

Resource *representation formats* serve the needs of service consumers. This consumer friendliness, however, does not extend to allowing consumers to control how resources are identified, evolved, modified, and managed. Instead, services control their resources and how their states are represented to the outside world. This encapsulation is one of the key aspects of the Web's loose coupling.

The success of the Web is linked with the proliferation and wide acceptance of common representation formats. This ecosystem of formats (which includes HTML for structured documents, PNG and JPEG for images, MPEG for videos, and XML and JSON for data), combined with the large installed base of software capable of processing them, has been a catalyst in the Web's success. After all, if your web browser couldn't decode JPEG images or HTML documents, the human Web would have been stunted from the start, despite the benefits of a widespread transfer protocol such as HTTP.

To illustrate the importance of representation formats, in Figure 1-3 we've modeled the menu of a new coffee store called Restbucks (which will provide the domain for examples and explanations throughout this book). We have associated this menu with an HTTP URI. The publication of the URI surfaces the resource to the Web, allowing software agents to access the resource's representation(s).



```
<xhtml>
<body>
<p><b>Menu</b></p>
<ul>
<li>Latte: $5</li>
<li>Espresso: $4</li>
<li>Cookie:$1</li>
</ul>
</body>
```

Figure 1-3. *Example of a resource and its representations*

In this example, we have decided to make only XHTML and text-only representations of the resource available. Many more representations of the same announcement could be served using formats such as PDF, JPEG, MPEG video, and so on, but we have made a pragmatic decision to limit our formats to those that are both human- and machine-friendly.

Typically, resource representations such as those in Figure 1-3 are meant for human consumption via a web browser. Browsers are the most common computer agents on the Web today. They understand protocols such as HTTP and FTP, and they know how to render formats such as (X)HTML and JPEG for human consumption. Yet, as we move toward an era of computer systems that span the Web, there is no reason to think of the web browser as the only important software agent, or to think that humans will be the only active consumers of those resources. Take Figure 1-4 as an example. An order resource is exposed on the Web through a URI. Another software agent consumes the XML representation of the order as part of a business-to-business process. Computers interact with one another over the Web, using HTTP, URIs, and representation formats to drive the process forward just as readily as humans.

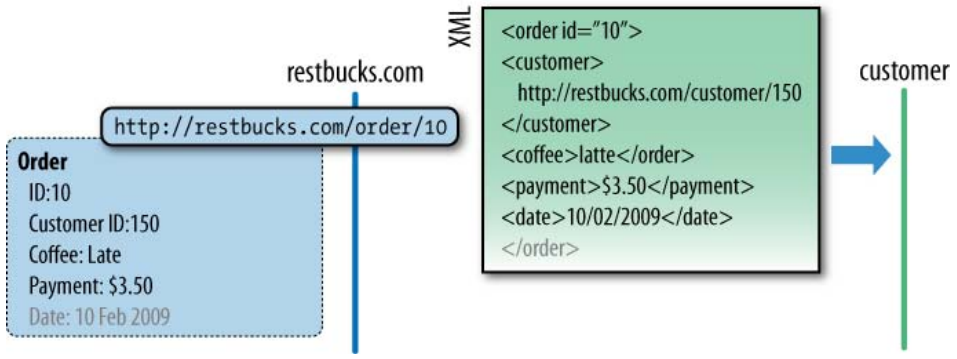


Figure 1-4. *Computer-to-computer communication using the Web*

Representation Formats and URIs

There is a misconception that different resource representations should each have their own URI—a notion that has been popularized by the Rails framework. With this approach, consumers of a resource terminate URIs with *.xml* or *.json* to indicate a preferred format, requesting *http://restbucks.com/order.xml* or *http://example.org/order.json* as they see fit. While such URIs convey intent explicitly, the Web has a means of negotiating representation formats that is a little more sophisticated.

NOTE

URIs should be opaque to consumers. Only the issuer of the URI knows how to interpret and map it to a resource. Using extensions such as *.xml*, *.html*, or *.json* is a historical convention that stems from the time when web servers simply mapped URIs directly to files.

In the example in Figure 1-3, we hinted at the availability of two representation formats: XHTML and plain text. But we didn't specify two separate URIs for the representations. This is because there is a one-to-many association between a URI and its possible resource representations, as Figure 1-5 illustrates.

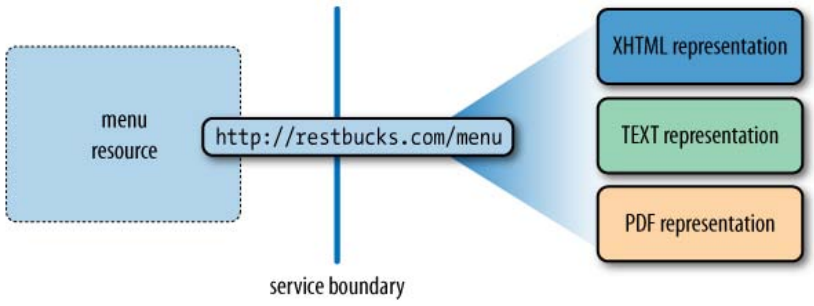


Figure 1-5. *Multiple resource representations addressed by a single URI*

Using *content negotiation*, consumers can negotiate for specific representation formats from a service. They do so by populating the HTTP `Accept` request header with a list of media types they're prepared to process. However, it is ultimately up to the owner of a resource to decide what constitutes a good representation of that resource in the context of the current interaction, and hence which format should be returned.

The Art of Communication

It's time to bring some threads together to see how resources, representation formats, and URIs help us build systems. On the Web, resources provide the subjects and objects with which we want to interact, but how do we act on them? The answer is that we need verbs, and on the Web these verbs are provided by HTTP methods.*

The term *uniform interface* is used to describe how a (small) number of verbs with well-defined and widely accepted semantics are sufficient to meet the requirements of most distributed applications. A collection of verbs is used for communication between systems.

NOTE

In theory, HTTP is just one of the many interaction protocols that can be used to support a web of resources and actions, but given its pervasiveness we will assume that HTTP is *the* protocol of the Web.

In contemporary distributed systems thinking, it's a popular idea that the set of verbs supported by HTTP—`GET`, `POST`, `PUT`, `DELETE`, `OPTIONS`, `HEAD`, `TRACE`, `CONNECT`, and `PATCH`—forms a sufficiently general-purpose protocol to support a wide range of solutions.

NOTE

In reality, these verbs are used with differing frequencies on the Web, suggesting that an even smaller subset is usually enough to satisfy the requirements of many distributed applications.

* Commonly, the term *verb* is used to describe HTTP actions, but in the HTTP specification the term *method* is used instead. We'll stick with *verb* in this book because *method* suggests object-oriented thinking, whereas we tend to think in terms of resources.

In addition to verbs, HTTP also defines a collection of response codes, such as `200 OK`, `201 Created`, and `404 Not Found`, that coordinate the interactions instigated by the use of the verbs. Taken together, verbs and status codes provide a general framework for operating on resources over the network.

Resources, identifiers, and actions are all we need to interact with resources hosted on the Web. For example, Figure 1-6 shows how the XML representation of an order might be requested and then delivered using HTTP, with the overall orchestration of the process governed by HTTP response codes. We'll see much more of all this in later chapters.

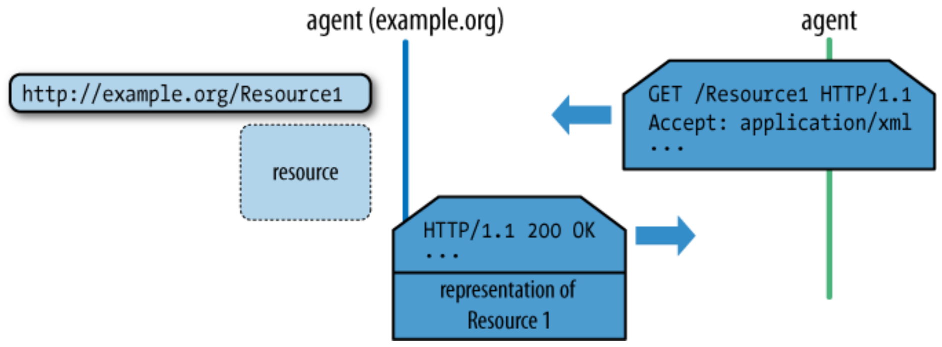


Figure 1-6. *Using HTTP to "GET" the representation of a resource*

From the Web Architecture to the REST Architectural Style

Intrigued by the Web, researchers studied its rapid growth and sought to understand the reasons for its success. In that spirit, the Web's architectural underpinnings were investigated in a seminal work that supports much of our thinking around contemporary web-based systems.

As part of his doctoral work, Roy Fielding generalized the Web's architectural principles and presented them as a framework of constraints, or an *architectural style*. Through this framework, Fielding described how distributed information systems such as the Web are built and operated. He described the interplay between resources, and the role of unique identifiers in such systems. He also talked about using a limited set of operations with uniform semantics to build a ubiquitous infrastructure that can support any type of application.* Fielding referred to this architectural style as *REpresentational State Transfer*, or REST. REST describes the Web as a distributed hypermedia application whose linked resources communicate by exchanging representations of resource state.

* *http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm*

Hypermedia

The description of the Web, as captured in W3C's "Architecture of the World Wide Web"* and other IETF RFC† documents, was heavily influenced by Fielding's work. The architectural abstractions and constraints he established led to the introduction of *hypermedia as the engine of application state.* The latter has given us a new perspective on how the Web can be used for tasks other than information storage and retrieval. His work on REST demonstrated that the Web is an application platform, with the REST architectural style providing guiding principles for building distributed applications that scale well, exhibit loose coupling, and compose functionality across service boundaries.

The idea is simple, and yet very powerful. A distributed application makes forward progress by transitioning from one state to another, just like a state machine. The difference from traditional state machines, however, is that the possible states and the transitions between them are not known in advance. Instead, as the application reaches a new state, the next possible transitions are discovered. It's like a treasure hunt.

NOTE

We're used to this notion on the human Web. In a typical e-commerce solution such as Amazon.com, the server generates web pages with links on them that corral the user through the process of selecting goods, purchasing, and arranging delivery.

This is hypermedia at work, but it doesn't have to be restricted to humans; computers are just as good at following protocols defined by state machines.

In a hypermedia system, application states are communicated through representations of uniquely identifiable resources. The identifiers of the states to which the application can transition are embedded in the representation of the current state in the form of *links*. Figure 1-7 illustrates such a hypermedia state machine.

* *http://www.w3.org/TR/webarch/*

† IETF: Internet Engineering Task Force; RFC: Request for Comments. See *http://www.ietf.org*.

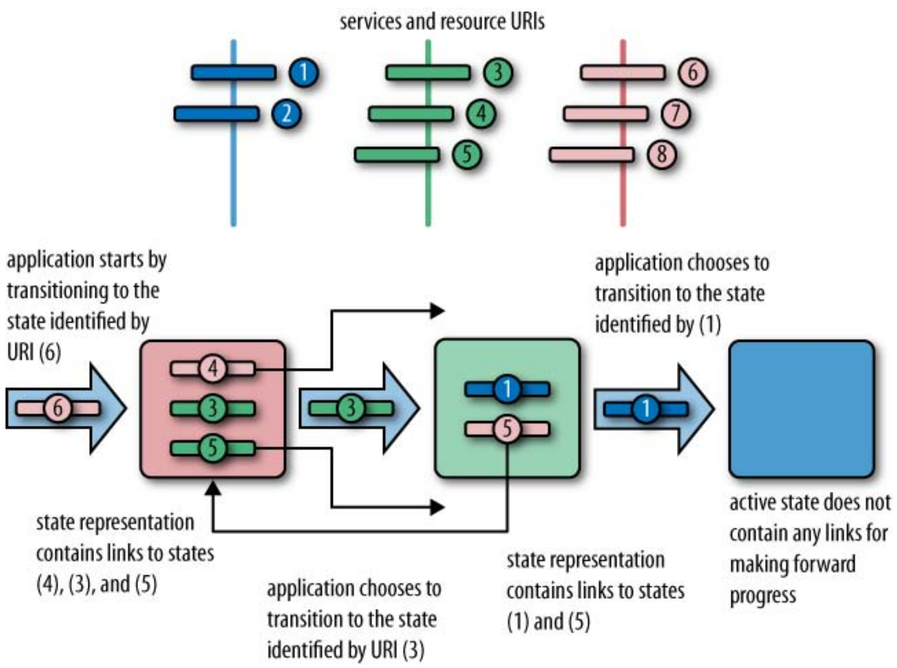

services and resource URIs
1
2
3
4
5
6
7
8
application starts by transitioning to the state identified by URI (6)
6
4
3
5
3
1
5
1
application chooses to transition to the state identified by (1)
state representation contains links to states (4), (3), and (5)
application chooses to transition to the state identified by URI (3)
state representation contains links to states (1) and (5)
active state does not contain any links for making forward progress

Figure 1-7. *Example of hypermedia as the engine for application state in action*

This, in simple terms, is what the famous *hypermedia as the engine of application state* or *HATEOAS* constraint is all about. We see it in action every day on the Web, when we follow the links to other pages within our browsers. In this book, we show how the same principles can be used to enable computer-to-computer interactions.

REST and the Rest of This Book

While REST captures the fundamental principles that underlie the Web, there are still occasions where practice sidesteps theoretical guidance. Even so, the term *REST* has become so popular that it is almost impossible to disassociate it from any approach that uses HTTP.* It's no surprise that the term *REST* is treated as a buzzword these days rather than as an accurate description of the Web's blueprints.

The pervasiveness of HTTP sets it aside as being special among all the Internet protocols. The Web has become a universal "on ramp," providing near-ubiquitous connectivity for billions of software agents across the planet. Correspondingly, the focus of this book is on the Web as it is used in practice—as a distributed application platform rather than as a single large hypermedia system. Although we are highly appreciative of Fielding's research, and of much subsequent work in understanding web-scale systems, we'll use the term *web* throughout this book to depict a warts-'n-all view, reserving the REST terminology to describe solutions that embrace the REST architectural style. We do this

* RFC 2616: *http://www.w3.org/Protocols/rfc2616/rfc2616.html*.

because many of today's distributed applications on the Web do not follow the REST architectural tenets, even though many still refer to these applications as "RESTful."

The Web As an Application Platform

Though the Web began as a publishing platform, it is now emerging as a means of connecting distributed applications. The Web as a platform is the result of its architectural simplicity, the use of a widely implemented and agreed-upon protocol (HTTP), and the pervasiveness of common representation formats. The Web is no longer just a successful large-scale information system, but a platform for an ecosystem of services.

But how can resources, identifiers, document formats, and a protocol make such an impression? Why, even after the dot-com bubble, are we still interested in it? What do enterprises—with their innate tendency toward safe middleware choices from established vendors—see in it? What is new that changes the way we deliver functionality and integrate systems inside and outside the enterprise?

As developers, we build solutions on top of platforms that solve or help with hard distributed computing problems, leaving us free to work on delivering valuable business functionality. Hopefully, this book will give you the information you need in order to make an informed decision on whether the Web fits your problem domain, and whether it will help or hinder delivering your solution. We happen to believe that the Web is a sensible solution for the majority of the distributed computing problems encountered in business computing, and we hope to convince you of this view in the following chapters. But for starters, here are a number of reasons we're such web fans.

Technology Support

An application platform isn't of much use unless it's supported by software libraries and development toolkits. Today, practically all operating systems and development platforms provide some kind of support for web technologies (e.g., .NET, Java, Perl, PHP, Python, and Ruby). Furthermore, the capabilities to process HTTP messages, deal with URIs, and handle XML or JSON payloads are all widely implemented in web frameworks such as Ruby on Rails, Java servlets, PHP Symfony, and ASP.NET MVC. Web servers such as Apache and Internet Information Server provide runtime hosting for services.

Scalability and Performance

Underpinned by HTTP, the web architecture supports a global deployment of networked applications. But the massive volume of blogs, mashups, and news feeds wouldn't have been possible if it wasn't for the way in which the Web and HTTP constrain solutions to a handful of scalable patterns and practices.

Scalability and performance are quite different concerns. Naively, it would seem that if latency and bandwidth are critical success factors for an application, using HTTP is not a good option. We know that there are messaging protocols with far better performance characteristics than HTTP's text-based, synchronous, request-response behavior. Yet this is an inequitable comparison, since HTTP is not just another messaging protocol; it's a protocol that implements some very specific application semantics. The HTTP verbs (and `GET` in particular) support caching, which translates into reduced latency, enabling massive horizontal scaling for large aggregate throughput of work.

NOTE

As developers ourselves, we understand how we can believe that asynchronous message-centric solutions are the most scalable and highest-performing options. However, existing high-performance and highly available services on the Web are proof that a synchronous, text-based request-response protocol can provide good performance and massive scalability when used correctly.

The Web combines a widely shared vision for how to use HTTP efficiently and how to federate load through a network. It may sound incredible, but through the remainder of this book, we hope to demonstrate this paradox beyond doubt.

Loose Coupling

The Web is loosely coupled, and correspondingly scalable. The Web does not try to incorporate in its architecture and technology stack any of the traditional quality-of-service guarantees, such as data consistency, transactionality, referential integrity, statefulness, and so on. This deliberate lack of guarantees means that browsers sometimes try to retrieve nonexistent pages, mashups can't always access information, and business applications can't always make immediate progress. Such failures are part of our everyday lives, and the Web is no different. Just like us, the Web needs to know how to cope with unintended outcomes or outright failures.

A software agent may be given the URI of a resource on the Web, or it might retrieve it from the list of hypermedia links inside an HTML document, or find it after a business-to-business XML message interaction. But a request to retrieve the representation of that resource is never guaranteed to be successful. Unlike other contemporary distributed systems architectures, the Web's blueprints do not provide any explicit mechanisms to support information integrity. For example, if a service on the Web decides that a URI is no longer going to be associated with a particular resource, there is no way to notify all those consumers that depend on the old URI–resource association.

This is an unusual stance, but it does not mean that the Web is neglectful—far from it. HTTP defines response codes that can be used by service providers to indicate what has happened. To communicate that "the resource is now associated with a new URI," a service can use the status code `301 Moved Permanently` or `303 See Other`. The Web always tries to help move us toward a successful conclusion, but without introducing tight coupling.

Business Processes

Although business processes can be modeled and exposed through web resources, HTTP does not provide direct support for such processes. There is a plethora of work on vocabularies to capture business processes (e.g., BPEL,* WS-Choreography†), but none of them has really embraced the Web's architectural principles. Yet the Web—and hypermedia specifically—provides a great platform for modeling business-to-business interactions.

Instead of reaching for extensive XML dialects to construct choreographies, the Web allows us to model state machines using HTTP and hypermedia-friendly formats such as XHTML and Atom. Once we understand that the states of a process can be modeled as resources, it's simply a matter of describing the transitions between those resources and allowing clients to choose among them at runtime.

This isn't exactly new thinking, since HTML does precisely this for the human-readable Web through the `<a href="...">` tag. Although implementing hypermedia-based solutions for computer-to-computer systems is a new step for most developers, we'll show you how to embrace this model in your systems to support loosely coupled business processes (i.e., behavior, not just data) over the Web.

Consistency and Uniformity

To the Web, one representation looks very much like another. The Web doesn't care if a document is encoded as HTML and carries weather information for on-screen human consumption, or as an XML document conveying the same weather data to another application for further processing. Irrespective of the format, they're all just resource representations.

The principle of uniformity and least surprise is a fundamental aspect of the Web. We see this in the way the number of permissible operations is constrained to a small set, the members of which have well-understood semantics. By embracing these constraints, the web community has developed myriad creative ways to build applications and infrastructure that support information exchange and application delivery over the Web.

Caches and proxy servers work precisely because of the widely understood caching semantics of some of the HTTP verbs—in particular, `GET`. The Web's underlying infrastructure enables reuse of software tools and development libraries to provide an ecosystem of middleware services, such as caches, that support performance and scaling. With plumbing that understands the application model baked right into the network, the Web allows innovation to flourish at the edges, with the heavy lifting being carried out in the cloud.

* *http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html*

† *http://www.w3.org/TR/2004/WD-ws-cdl-10-20041217/*

Simplicity, Architectural Pervasiveness, and Reach

This focus on resources, identifiers, HTTP, and formats as the building blocks of the world's largest distributed information system might sound strange to those of us who are used to building distributed applications around remote method invocations, message-oriented middleware platforms, interface description languages, and shared type systems. We have been told that distributed application development is difficult and requires specialist software and skills. And yet web proponents constantly talk about simpler approaches.

Traditionally, distributed systems development has focused on exposing custom behavior in the form of application-specific interfaces and interaction protocols. Conversely, the Web focuses on a few well-known network actions (those now-familiar HTTP verbs) and the application-specific interpretation of resource representations. URIs, HTTP, and common representation formats give us reach—straightforward connectivity and ubiquitous support from mobile phones and embedded devices to entire server farms, all sharing a common application infrastructure.

Web Friendliness and the Richardson Maturity Model

As with any other technology, the Web will not automatically solve a business's application and integration problems. But good design practices and adoption of good, well-tested, and widely deployed patterns will take us a long way in our journey to build great web services.

You'll often hear the term *web friendliness* used to characterize good application of web technologies. For example, a service would be considered "web-friendly" if it correctly implemented the semantics of HTTP `GET` when exposing resources through URIs. Since `GET` doesn't make any service-side state changes that a consumer can be held accountable for, representations generated as responses to `GET` *may* be cached to increase performance and decrease latency.

Leonard Richardson proposed a classification for services on the Web that we'll use in this book to quantify discussions on service maturity.* Leonard's model promotes three levels of service maturity based on a service's support for URIs, HTTP, and hypermedia (and a fourth level where no support is present). We believe this taxonomy is important because it allows us to ascribe general architectural patterns to services in a manner that is easily understood by service implementers.

The diagram in Figure 1-8 shows the three core technologies with which Richardson evaluates service maturity. Each layer builds on the concepts and technologies of the

* Richardson presented this taxonomy during his talk "Justice Will Take Us Millions Of Intricate Moves" at QCon San Francisco 2008; see *http://www.crummy.com/writing/speaking/2008-QCon/*.

layers below. Generally speaking, the higher up the stack an application sits, and the more it employs instances of the technology in each layer, the more mature it is.

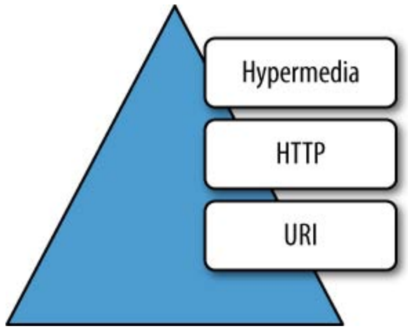


Figure 1-8. *The levels of maturity according to Richardson's model*

Level Zero Services

The most basic level of service maturity is characterized by those services that have a single URI, and which use a single HTTP method (typically `POST`). For example, most Web Services (WS-*)-based services use a single URI to identify an endpoint, and HTTP `POST` to transfer SOAP-based payloads, effectively ignoring the rest of the HTTP verbs.*

NOTE

We can do wonderful, sophisticated things with WS-*, and it is not our intention to imply that its level zero status is a criticism. We merely observe that WS-* services do not use many web features to help achieve their goals.†

XML-RPC and Plain Old XML (POX) employ similar methods: HTTP `POST` requests with XML payloads transmitted to a single URI endpoint, with replies delivered in XML as part of the HTTP response. We will examine the details of these patterns, and show where they can be effective, in Chapter 3.

Level One Services

The next level of service maturity employs many URIs but only a single HTTP verb. The key dividing feature between these kinds of rudimentary services and level zero services is that level one services expose numerous logical resources, while level zero services tunnel all interactions through a single (large, complex) resource. In level one services,

* The suite of SOAP-based specifications and technologies, such as WSDL, WS-Transfer, WS-MetadataExchange, and so forth. Refer to *http://www.w3.org/2002/ws/* as a starting point. We'll discuss Web Services and their relationship to the Web in Chapter 12.

† The report of the "Web of Services" workshop is a great source of information on this topic: *http://www.w3.org/2006/10/wos-ec-cfp.html*.

however, operations are tunneled by inserting operation names and parameters into a URI, and then transmitting that URI to a remote service, typically via HTTP `GET`.

NOTE

Richardson claims that most services that describe themselves as "RESTful" today are in reality often level one services. Level one services can be useful, even though they don't strictly adhere to RESTful constraints, and so it's possible to accidentally destroy data by using a verb (`GET`) that should not have such side effects.

Level Two Services

Level two services host numerous URI-addressable resources. Such services support several of the HTTP verbs on each exposed resource. Included in this level are Create Read Update Delete (CRUD) services, which we cover in Chapter 4, where the state of resources, typically representing business entities, can be manipulated over the network. A prominent example of such a service is Amazon's S3 storage system.

NOTE

Importantly, level two services use HTTP verbs and status codes to coordinate interactions. This suggests that they make use of the Web for robustness.

Level Three Services

The most web-aware level of service supports the notion of hypermedia as the engine of application state. That is, representations contain URI links to other resources that might be of interest to consumers. The service leads consumers through a trail of resources, causing application state transitions as a result.

NOTE

The phrase *hypermedia as the engine of application state* comes from Fielding's work on the REST architectural style. In this book, we'll tend to use the term *hypermedia constraint* instead because it's shorter and it conveys that using hypermedia to manage application state is a beneficial aspect of large-scale computing systems.

GET on Board

Can the same principles that drive the Web today be used to connect systems? Can we follow the same principles driving the human Web for computer-to-computer scenarios? In the remainder of this book, we will try to show why it makes sense to do exactly that, but first we'll need to introduce our business domain: a simple coffee shop called Restbucks.