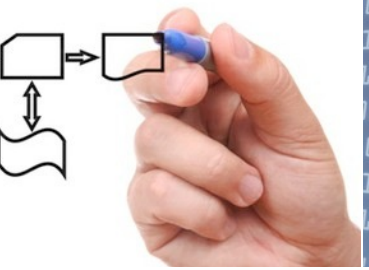
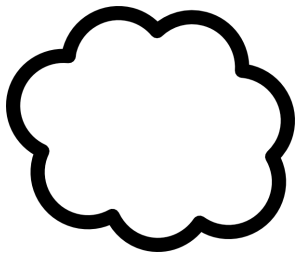
San Jose State University

CMPE 281

Cloud Technologies

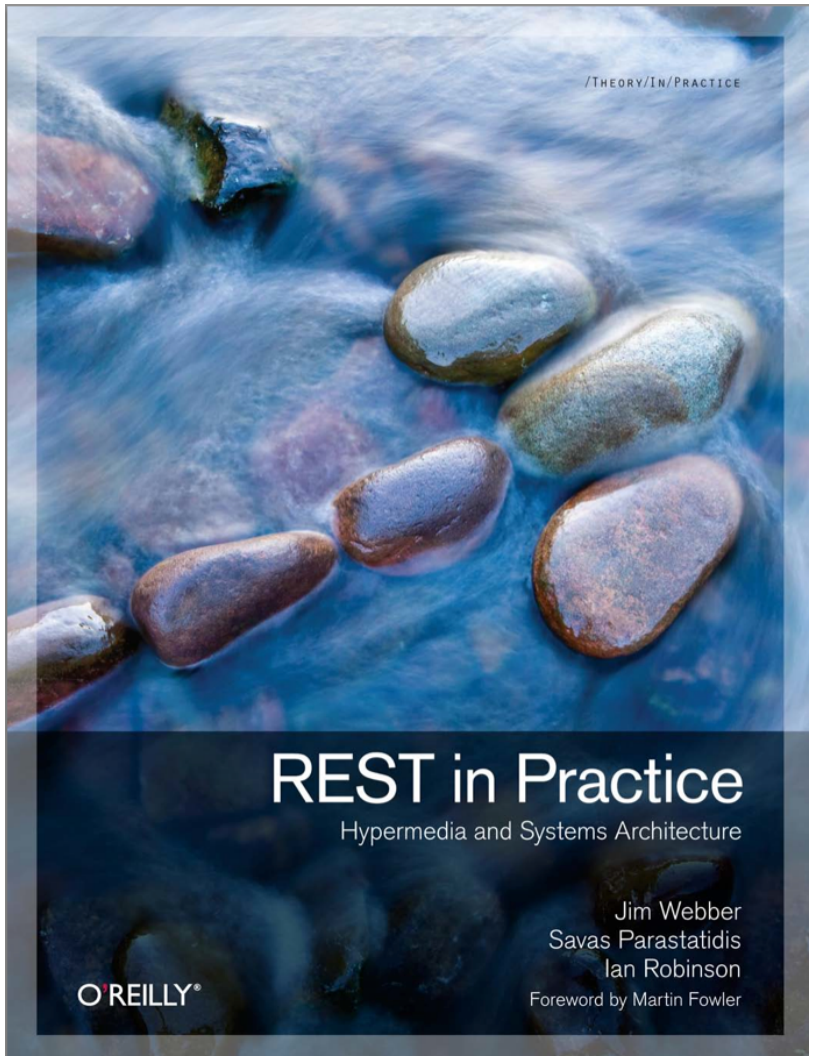
RESTbucks





REST Overview

(*Excerpts from the "REST in Practice" book on the REST Architecture Style*)



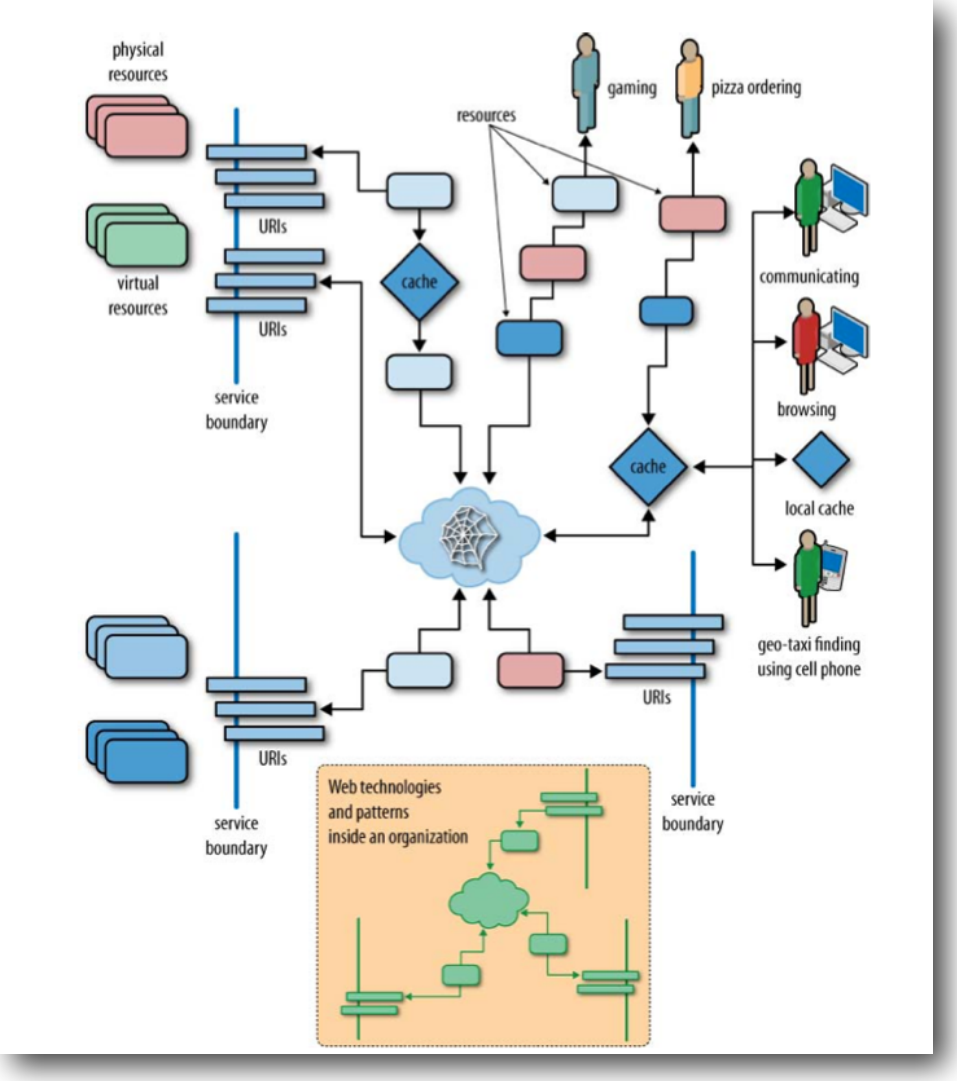


Figure 1-1. *The Web*

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.

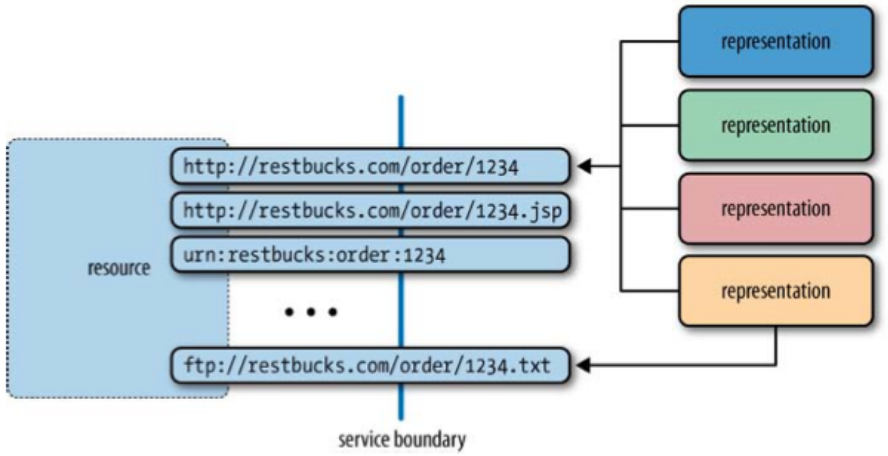


Figure 1-2. *Multiple URIs for a resource*

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.

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.

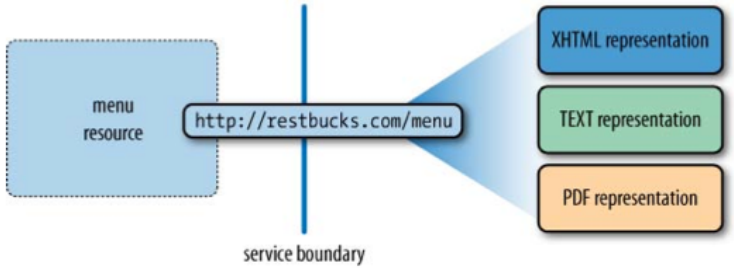


Figure 1-5. *Multiple resource representations addressed by a single URI*

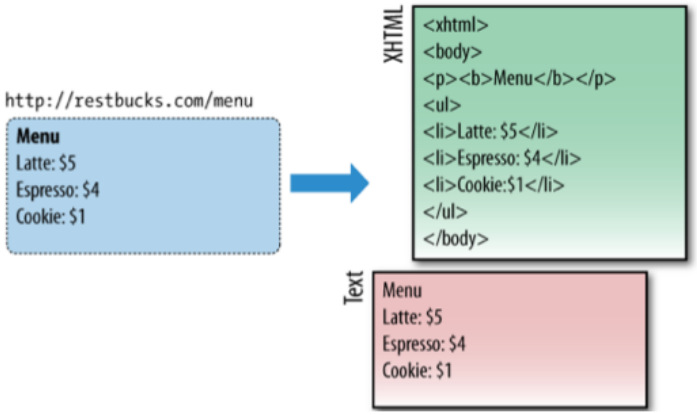


Figure 1-3. *Example of a resource and its representations*

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.

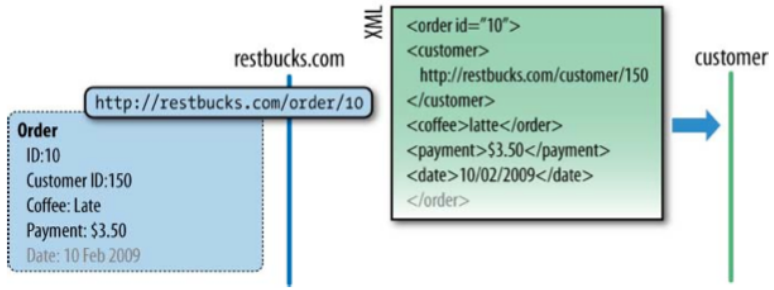


Figure 1-4. *Computer-to-computer communication using the Web*

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.

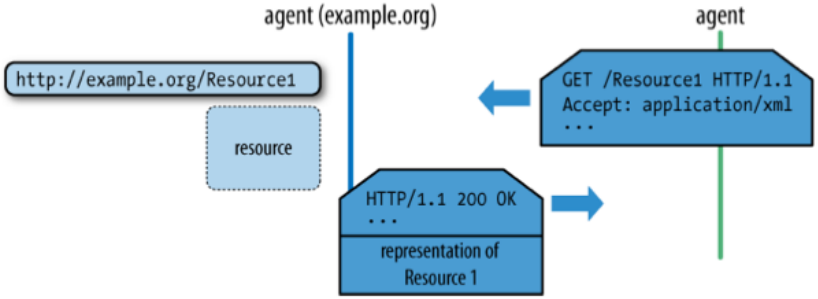


Figure 1-6. *Using HTTP to "GET" the representation of a resource*

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.

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.

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.

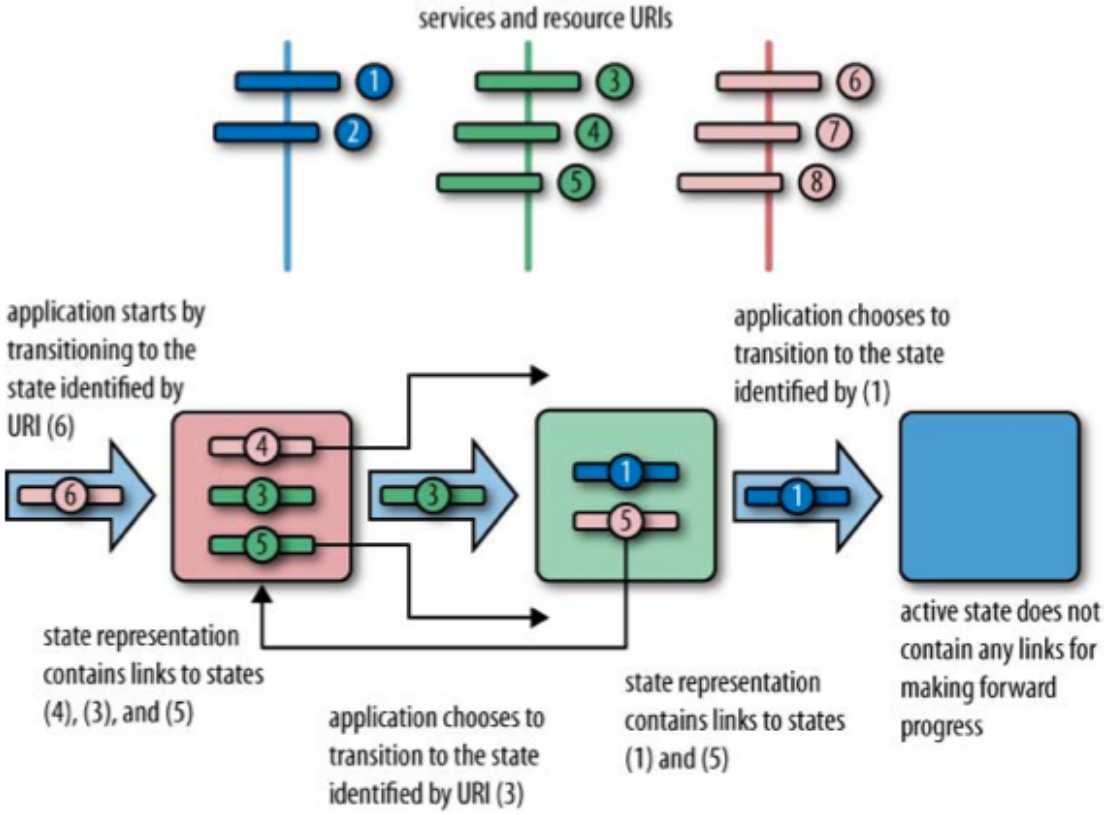


Figure 1-7. *Example of hypermedia as the engine for application state in action*

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.

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.

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.

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.

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.

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

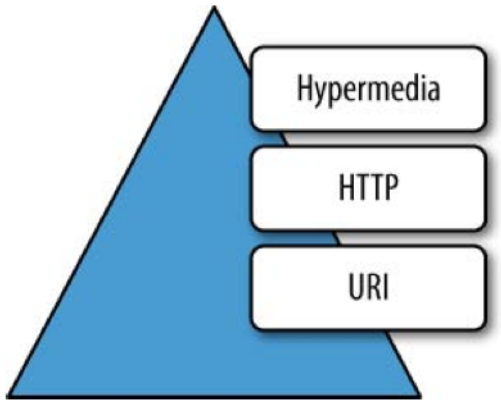


Figure 1-8. *The levels of maturity according to Richardson's model*

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.

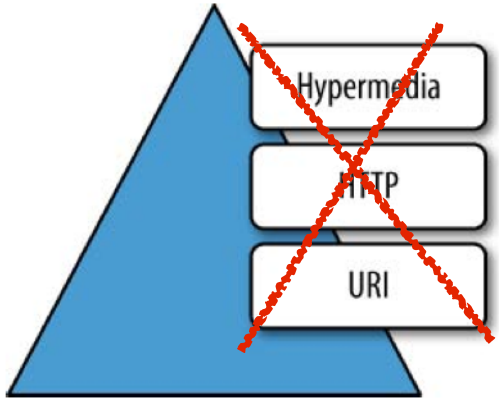


Figure 1-8. *The levels of maturity according to Richardson's model*

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, 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.

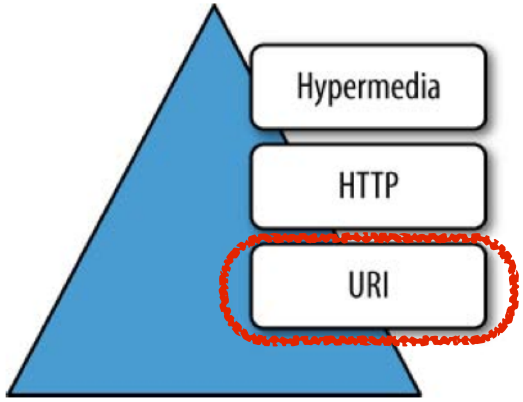


Figure 1-8. *The levels of maturity according to Richardson's model*

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.

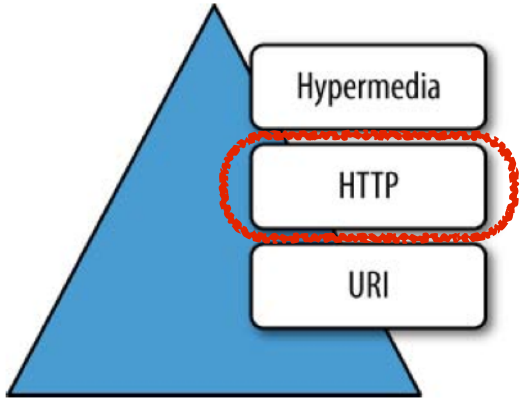


Figure 1-8. *The levels of maturity according to Richardson's model*

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.

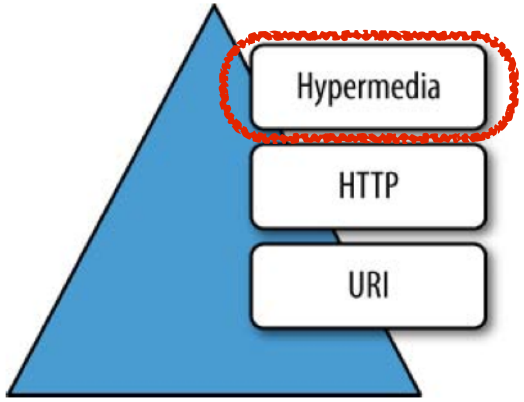
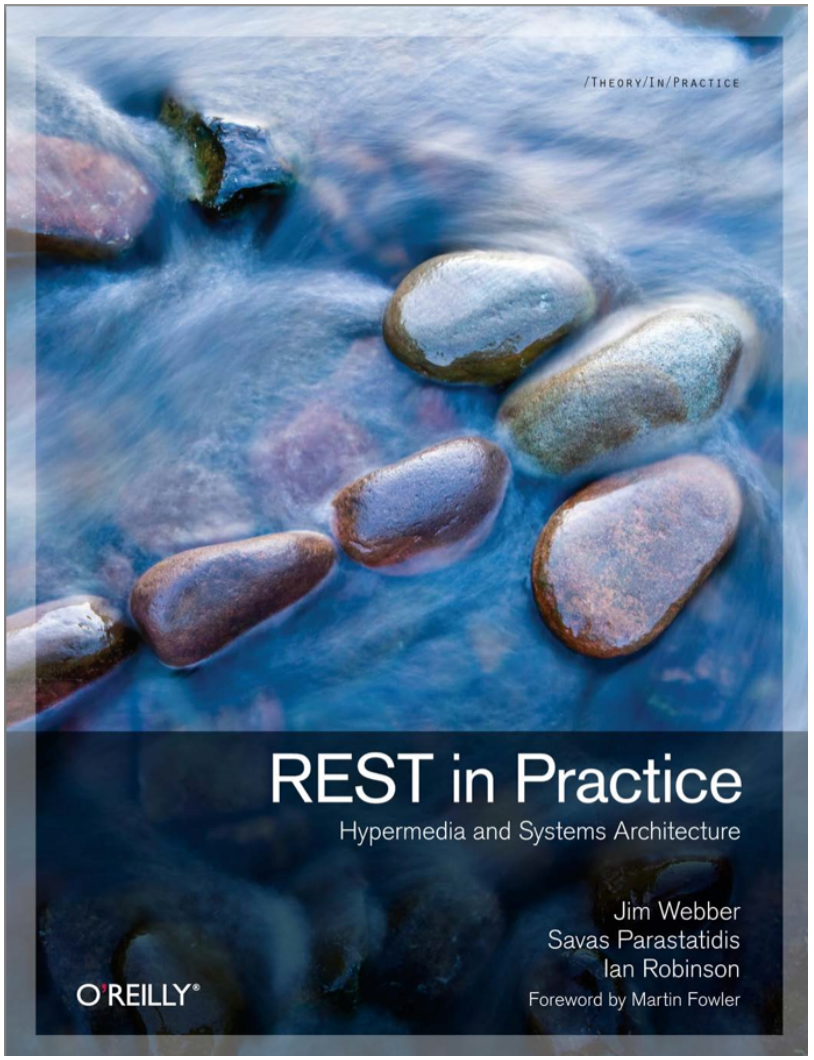


Figure 1-8. *The levels of maturity according to Richardson's model*

Restbucks

(*Excerpts from the "REST in Practice" book on the REST Architecture Style*)



design

Editor: Martin Fowler ■ ThoughtWorks ■ fowler@acm.org

Your Coffee Shop Doesn't Use Two-Phase Commit

Gregor Hohpe

You know you're a geek when going to the coffee shop gets you thinking about interaction patterns between loosely coupled systems. This happened to me on a recent trip to Japan. One of the more familiar sights in Tokyo is the numerous Starbucks coffee shops, especially around Shinjuku and Roppongi. While waiting for my "Hotto Cocoa," I started thinking about how a coffee shop processes customer orders. As a business, the coffee shop is naturally interested in maximizing order throughput, because more fulfilled orders mean more revenue.

Interestingly, the optimization for throughput results in a concurrent and asynchronous processing model: when you place your order, the cashier marks a coffee cup with your order and places it into a queue. This queue is literally a line of coffee cups on top of the espresso machine. The queue decouples the cashier and barista, letting the cashier continue to take orders even when the barista is backed up. It also allows multiple baristas to start servicing the queue if the store gets busy, without impacting the cashier.

Asynchronous processing models can be highly efficient but are not without challenges. If the real world writes the best stories, then maybe we can learn something from Starbucks about designing successful asynchronous messaging solutions.

Correlation

For example, the asynchronous processing model means that drink orders aren't necessarily completed in the same sequence in which they were placed. This can happen for two different reasons. First, multiple baristas might be processing orders using different equipment. Blended drinks usually take longer to make than drip coffee, so a drip coffee ordered last might be delivered first. Second, baristas can make multiple drinks in one batch to optimize processing time.

As a result, Starbucks has a correlation problem. Drinks are delivered out of sequence and must be matched up with the correct customer. Starbucks solves the problem with the same "pattern" we use in messaging architectures—they use a *correlation identifier*.[1] In the US, most Starbucks use an explicit correlation identifier by writing your name on the cup and calling it out when the drink is ready. In other countries, they often correlate by drink type. The correlation issue became very apparent in Japan, where I had difficulties understanding the baristas calling out the drinks. My approach was to order extra large "venti" drinks because they're uncommon and therefore easily identifiable—that is, "correlatable."

Exception handling

Exception handling in asynchronous-messaging scenarios is naturally difficult. For example, if the receiver of a message is truly decoupled from the sender, what's the receiver to do when something goes wrong? What does Starbucks do if they've already placed your drink order into

0740-7459/05/$20.00 © 2005 IEEE

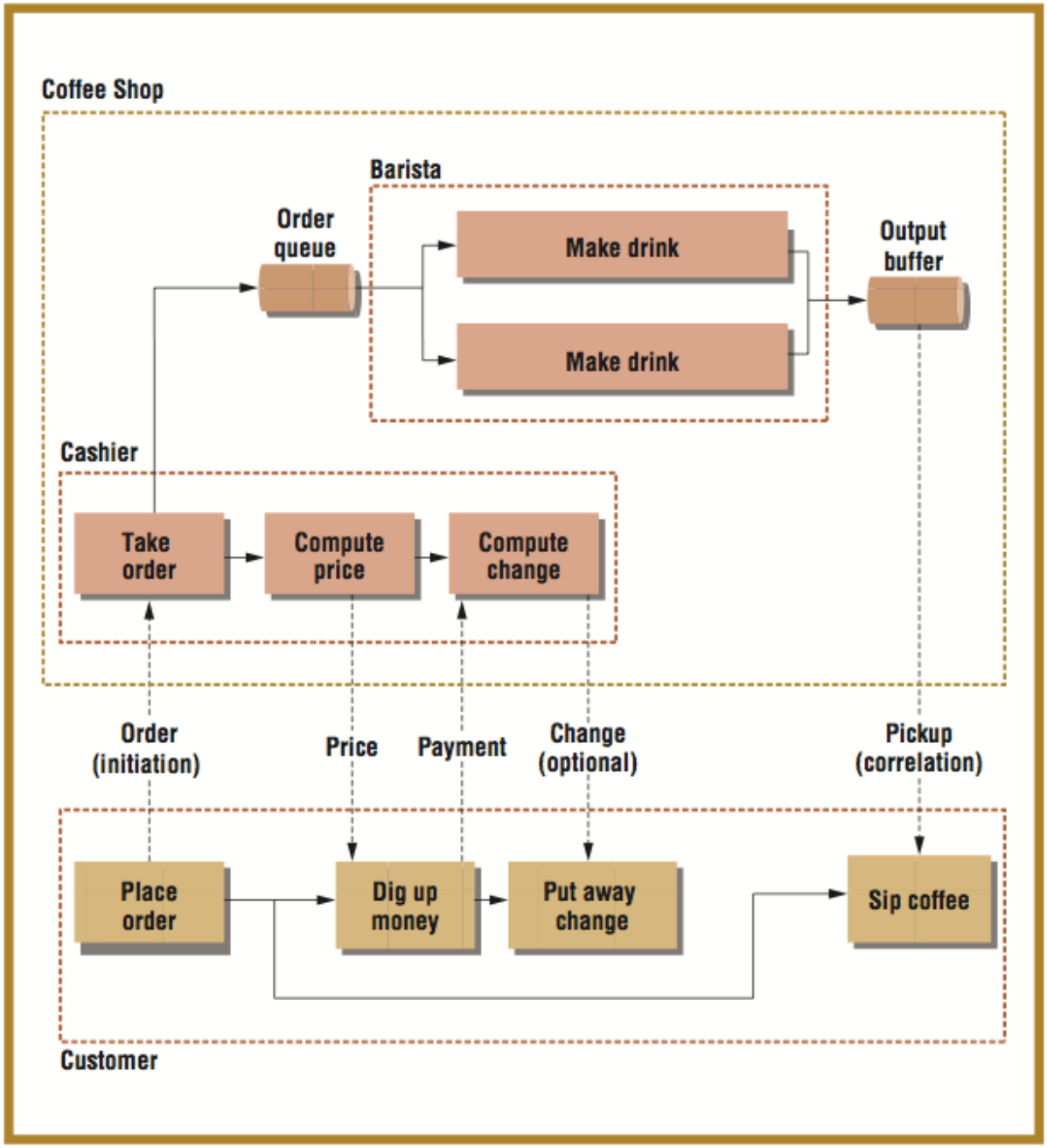


Figure 2. A conversation pattern.

Restbucks: A Little Coffee Shop with Global Ambitions

Throughout this book, we'll frame our problems and web-based solutions in terms of a coffee shop called Restbucks, which grows from modest beginnings to become a global enterprise. As Restbucks grows, so do its needs for better organization and more efficient use of resources for operating at larger scale. We'll show how Restbucks operations can be implemented with web technologies and patterns to support all stages of the company's growth.

While nothing can replace the actual experience of waiting in line, ordering, and then tasting the coffee, our intention is to use our coffee shop to showcase common problems and demonstrate how web technologies and patterns can help solve them, within both Restbucks and systems development in general. The Restbucks analogy does not describe every aspect of the coffee shop business; we chose to highlight only those problems that help support the technical discussion.

Actors and Conversations

The Restbucks service and the resources that it exposes form the core of our discussion. Restbucks has actors such as customers, cashiers, baristas, managers, and suppliers who must interact to keep the coffee flowing.

In all of the examples in this book, computers replace human-to-human interactions. Each actor is a program that interacts through the Web to drive business processes hosted by Restbucks services. Even so, our business goals remain: we want to serve good coffee, take payments, keep the supply chain moving, and ultimately keep the business alive.

Interactions occur through HTTP using formats that are commonly found on the Web. We chose to use XML since it's widely supported and it's relatively easy for humans to parse, as we can see in Figure 2-1. Of course, XML isn't our only option for integration; others exist, such as plain text, HTML forms, and JSON. As our problem domain becomes more sophisticated in later chapters, we'll evolve our formats to meet the new requirements.

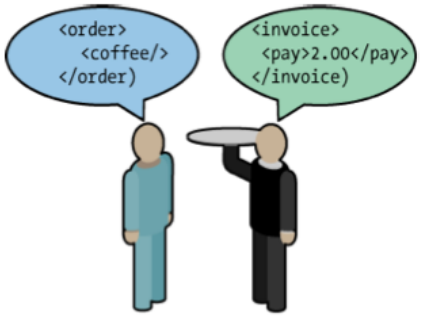


Figure 2-1. *XML-based exchange between a customer and a waiter*

Boundaries

In Restbucks, we draw boundaries around the actors involved in the system to encapsulate implementation details and emphasize the interactions between systems. When we order a coffee, we don't usually care about the mechanics of the supply chain or the details of the shop's internal coffee-making processes. Composition of functionality and the introduction of façades with which we can interact are common practices in system design, and web-based systems are no different in that respect. For example, in Figure 2-2 the customer doesn't need to know about the waiter–cashier and cashier–barista interactions when he orders a cup of coffee from the waiter.

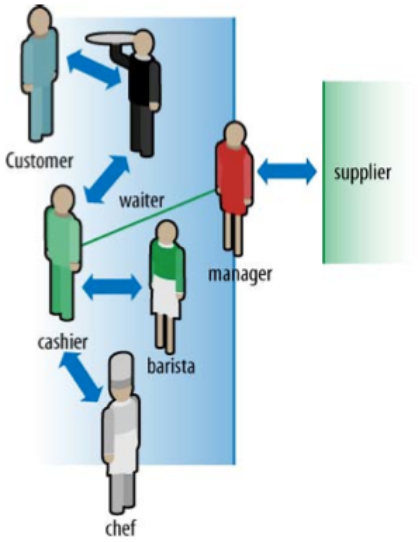


Figure 2-2. *Boundaries help decompose groups of interactions*

The Menu

Restbucks prides itself on the variety of products it serves and allows customers to customize their coffee with several options. Table 2-1 shows some of the products and options offered. Throughout the book, we'll see how these options manifest themselves in service interactions and the design decisions regarding their representation.

Table 2-1. *Sample catalog of products offered by Restbucks*

Product name	Customization option
Latte Cappuccino Espresso Tea	Milk: skim, semi, whole Size: small, medium, large Shots: single, double, triple
Hot chocolate	Milk: skim, semi, whole Size: small, medium, large Whipped cream: yes, no
Cookie	Kind: Chocolate chip, ginger
All	Consume location: take away, in shop

Sample Interactions

Let's set the scene for the remainder of the book by examining some of the typical interactions between the main actors. Subsequent chapters build on these scenarios, expand them further, and introduce new ones.

Customer–Barista

Restbucks takes its first steps as a small, neighborhood coffee shop. A barista is responsible for everything: taking orders, preparing the coffee, receiving payment, and giving receipts. Figure 2-3 shows the interaction between a customer and a barista.

Figure 2-3. *A simple interaction between a customer and a barista*

Customer–Cashier–Barista

Although Restbucks stems from modest roots, its coffee quality and increasingly positive reputation help it to continue to grow. To help scale the business, Restbucks decides to hire a cashier to speed things up. With a cashier busy handling the financial aspects of the operation, the barista can concentrate on making coffee. The customer's interactions aren't affected, but Restbucks now needs to coordinate the cashier's and barista's tasks with a low-ceremony approach using sticky notes. The interactions (or protocol) between the cashier and the barista remain hidden from customers. Now that we've got two moving parts in our coffee shop, we need to think about how to encapsulate them, which leads to the scenario shown in Figure 2-5.

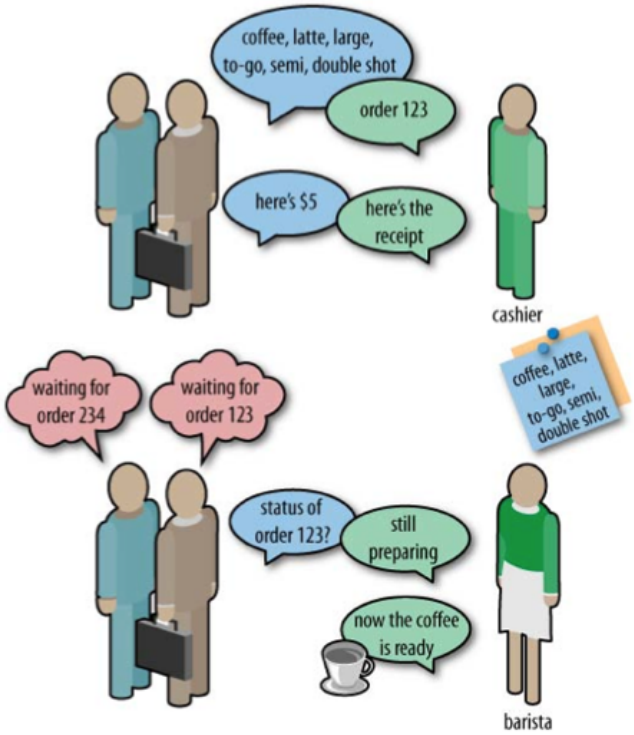


Figure 2-5. *A cashier helps the barista*

By implementing this scheme, Restbucks decouples ordering and payment from the coffee preparation. In turn, it is possible for Restbucks to abstract the inner workings of the shop through a façade. While the customer gets the same good coffee, Restbucks is free to innovate and evolve its implementation behind the scenes.

Restbucks Formats

We discussed formats for resource representations in general terms in Chapter 1, but here we'll introduce formats that Restbucks uses in its business. All Restbucks resources are represented by XML documents defined in the `http://restbucks.com` namespace and identified on the Web as the media types `application/xml` and `application/vnd.restbucks+xml` for standard XML processing and Restbucks-specific processing, respectively.*

NOTE

We've chosen XML-based formats deliberately for this book since they're easily understood and readable by humans. However, we shouldn't see XML as the only option. As we discussed in Chapter 1, real web services use myriad other formats, depending on the application.

Example 2-1. *A Restbucks order resource represented in XML format*

```
POST /order HTTP/1.1
Host: restbucks.com
Content-Type:application/vnd.restbucks+xml
Content-Length: 243

<order xmlns="http://schemas.restbucks.com/order">
  <location>takeAway</location>
  <item>
    <name>latte</name>
    <quantity>1</quantity>
    <milk>whole</milk>
    <size>small</size>
  </item>
</order>
```

A cashier service receiving the order in Example 2-1 will lodge the order and respond with the XML in Example 2-2, which contains the representation of a newly created order resource.

Example 2-2. *Acknowledging the order with a custom XML format*

```
HTTP/1.1 200 OK
Content-Length: 421
Content-Type: application/vnd.restbucks+xml
Date: Sun, 3 May 2009 18:22:11 GMT
<order xmlns="http://restbucks.com" xlmns:atom="http://www.w3.org/2005/Atom">
  <location>takeAway</location>
  <item>
    <name>latte</name>
    <quantity>1</quantity>
    <milk>whole</milk>
    <size>small</size>
  </item>
  <cost>3.00</cost>
  <currency>GBP</currency>
  <atom:link rel="payment" type="application/xml"
href="http://restbucks.com/order/1234/payment"/>
</order>
```

A customer receiving a response such as that in Example 2-2 can be assured that its order has been received and accepted by the Restbucks service. The order details are confirmed in the reply and some additional information is contained, such as the payment amount and currency, a timestamp for when the order was received, and a `<link>` element that identifies another resource with which the customer is expected to interact to make a payment.

Modeling Protocols and State Transitions

Using `<atom:link>` elements to describe possible next steps through a service protocol should feel familiar; after all, we're quite used to links and forms being used to guide us through HTML pages on the Web. In particular, we're comfortable with e-commerce sites guiding us through selecting products, confirming delivery addresses, and taking payment by stringing together a set of pages into a workflow. Unwittingly, we have all been driving a business protocol via HTTP using a web browser!

It's remarkable that the Web has managed to turn us humans into robots who follow protocols, but we take it for granted nowadays. We even think the concept of computers driving protocols through the same mechanism is new, yet this is the very essence of building distributed systems on the Web: using hypermedia to describe protocol state machines that govern interactions between systems.

> **NOTE**
>
> Protocols described in hypermedia are not binding contracts. If a Restbucks consumer decides not to drive the protocol through to a successful end state where coffee is paid for and served, the service has to deal with that. Fortunately, HTTP helps by providing useful status codes to help deal with such situations, as we shall see in the coming chapters.

Although hypermedia-based protocols are useful in their own right, they can be strengthened using microformats, such as hCard.[†] If we embed semantic information about the next permissible steps in a protocol inside the hypermedia links, we raise the level of abstraction. This allows us to change the underlying URIs as the system evolves without breaking any consumers, as well as to declare and even change a protocol dynamically without breaking existing consumers.

Of course, we *can* break existing consumers, but only if we remove or redefine something on which they rely. We're safe to add new, optional protocol steps or to change the URIs contained within the links, provided we keep the microformat vocabulary consistent.

Figure 2-6 shows an example of a protocol state machine as it evolves through the interactions between the customer, cashier, and barista. The state machine will not generally show the total set of permissible states, only those choices that are available during any given interaction to take the consumer down a particular path. If the customer cancels its order, it will not be presented with the option to pay the bill or add specialties to its coffee. The description of an application's state machine might be exposed in its entirety as part of metadata associated with a service, if the service provider chooses to do so. However, a state machine might change over time, even as a customer interacts with the service.

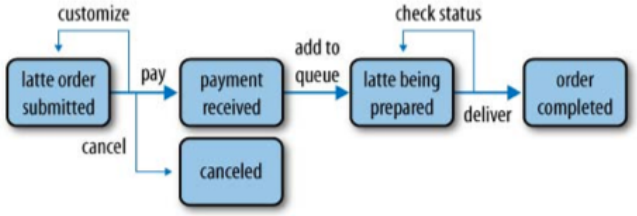


Figure 2-6. *Modeling state machines dynamically*

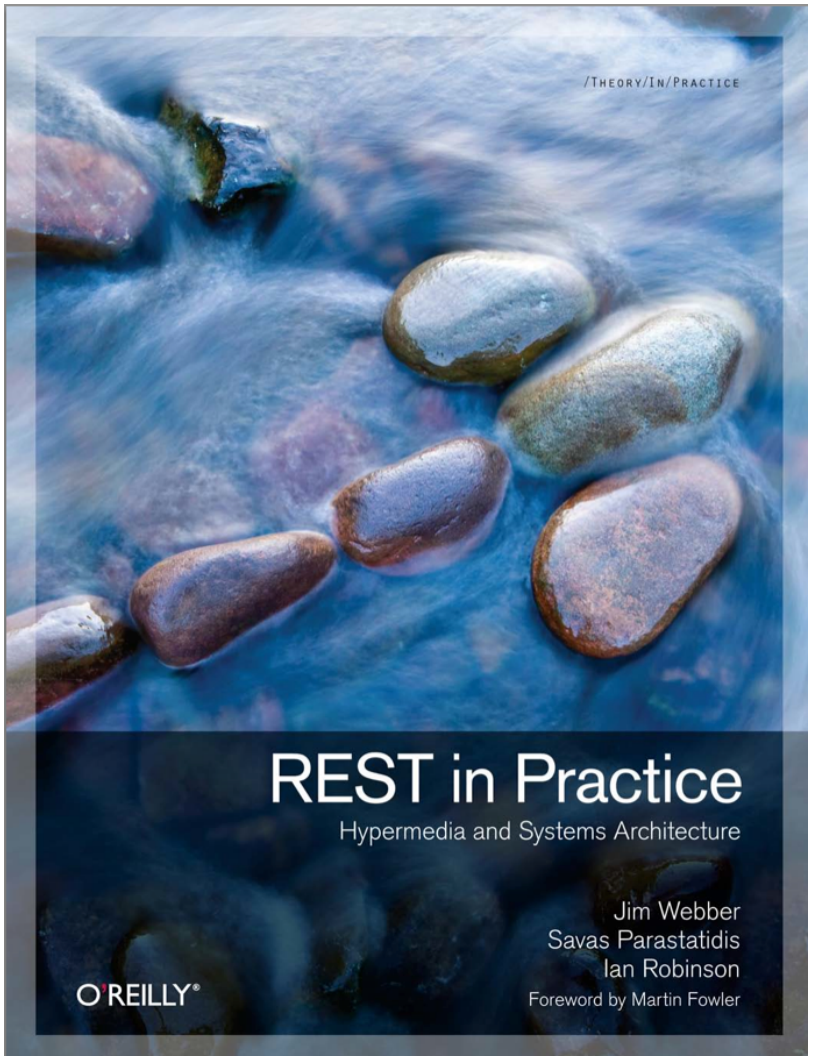
Here Comes the Web

Restbucks provides a domain that allows us to think of the Web as a platform for building distributed systems. We'll continue to expand Restbucks' domain throughout the book as more ambitious scenarios are introduced. Expect to see the addition of third-party services, security, more coordination of interactions, and scalability measures. Along the way, we'll dip into topics as diverse as manageability, semantics, notifications, queuing, caching, and load balancing, all neatly tied together by the Web.

But to start with, we're going to see how we can integrate systems using the bedrock of web technologies: the humble URI.

URI Tunneling

(*Excerpts from the "REST in Practice" book on the REST Architecture Style*)



URI Tunneling

When we order coffee from Restbucks, we first select the drinks we'd like, then we customize those drinks in terms of size, type of milk (if any), and other specialties such as flavorings. Once we've decided, we can convey our order to the cashier who handles all incoming orders. Of course, we have numerous options for how to convey our order to a cashier, and on the Web, URI tunneling is the simplest.

URI tunneling uses URIs as a means of transferring information across system boundaries by encoding the information within the URI itself.* This can be a useful technique, because URIs are well understood by web servers (of course!) and web client software. Since web servers can host code, this allows us to trigger program execution by sending a simple HTTP GET or POST request to a web server, and gives us the ability to parameterize the execution of that code using the content of the URI. Whether we choose GET or POST depends on our intentions: retrieving information should be tunneled through GET, while changing state really ought to use POST.

On the Web, we use GET in situations where we want to *retrieve* a resource's state representation, rather than deliberately *modify* that state. When used properly, GET is both *safe* and *idempotent*.

By safe, we mean a GET request generates no server-side side effects for which the client can be held responsible. There may still be side effects, but any that do occur are the sole responsibility of the service. For example, many services log GET requests, thereby changing some part of their state. But GET is still safe. Server-side logging is a private affair; clients don't ask for something to be logged when they issue a GET request.

An idempotent operation is one that generates absolute side effects. Invoking an idempotent operation repeatedly has the same effect as invoking it once. Because GET exhibits no side effects for which the consumer can be held responsible, it is naturally idempotent. Multiple GETs of the same URI generate the same *result*: they retrieve the state of the resource associated with that URI at the moment the request was received, even if they return different *data* (which can occur if the resource's state changes in between requests).

When developing services we *must* preserve the semantics of GET. Consumers of our resources expect our service to behave according to the HTTP specification (RFC 2616). Using a GET request to do something other than retrieve a resource representation—such as delete a resource, for example—is simply an abuse of the Web and its conventions.

POST is much less strict than GET; in fact, it's often used as a wildcard operation on the Web. When we use POST to tunnel information through URIs, it is expected that changes to resource state will occur. To illustrate, let's look at Figure 3-3.

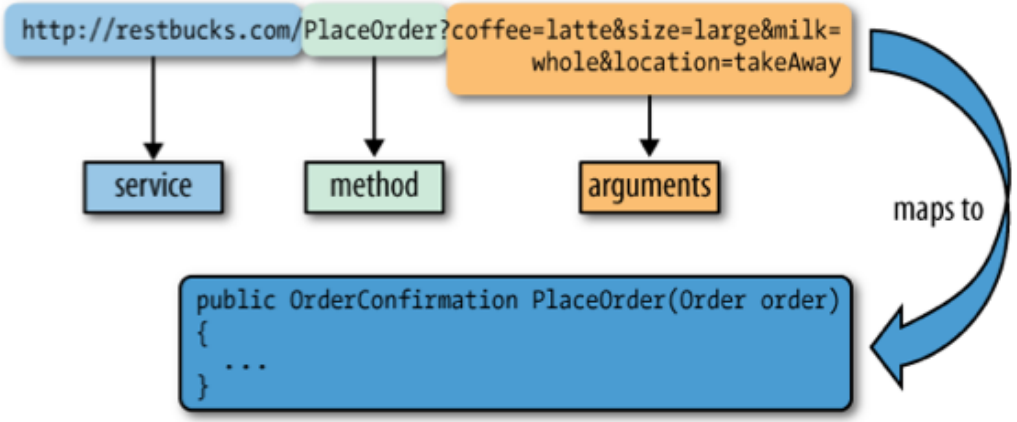


Figure 3-3. *Mapping method calls to URIs*

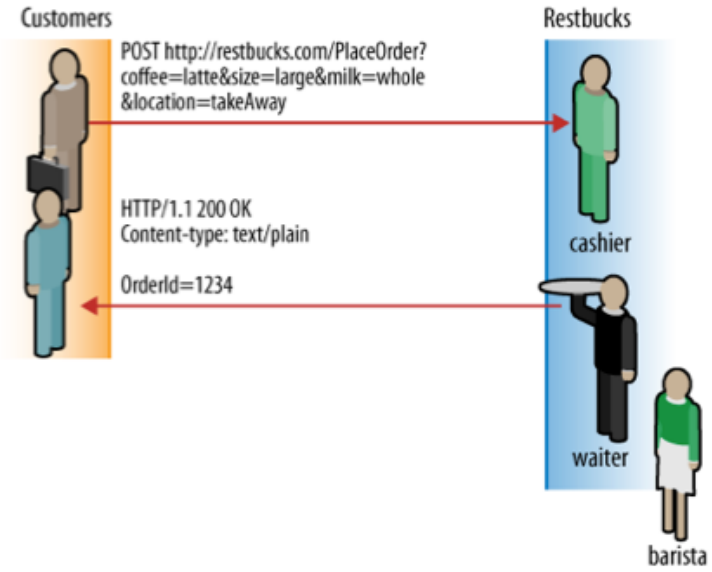


Figure 3-4. *HTTP request/response for URI tunneling*

Is URI Tunneling a Good Idea?

Services that use URI tunneling are categorized as level one services by Richardson's maturity model. Figure 3-5 highlights that URIs are a key concept in such services, but no other web technologies are embraced. Even HTTP is only used as a transport protocol for moving URIs over the Web.

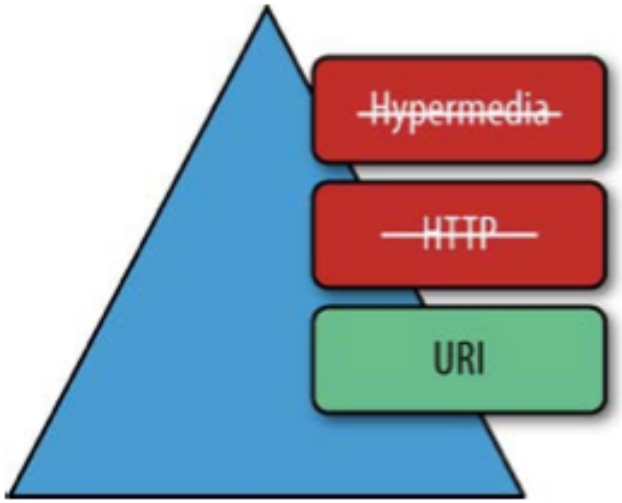
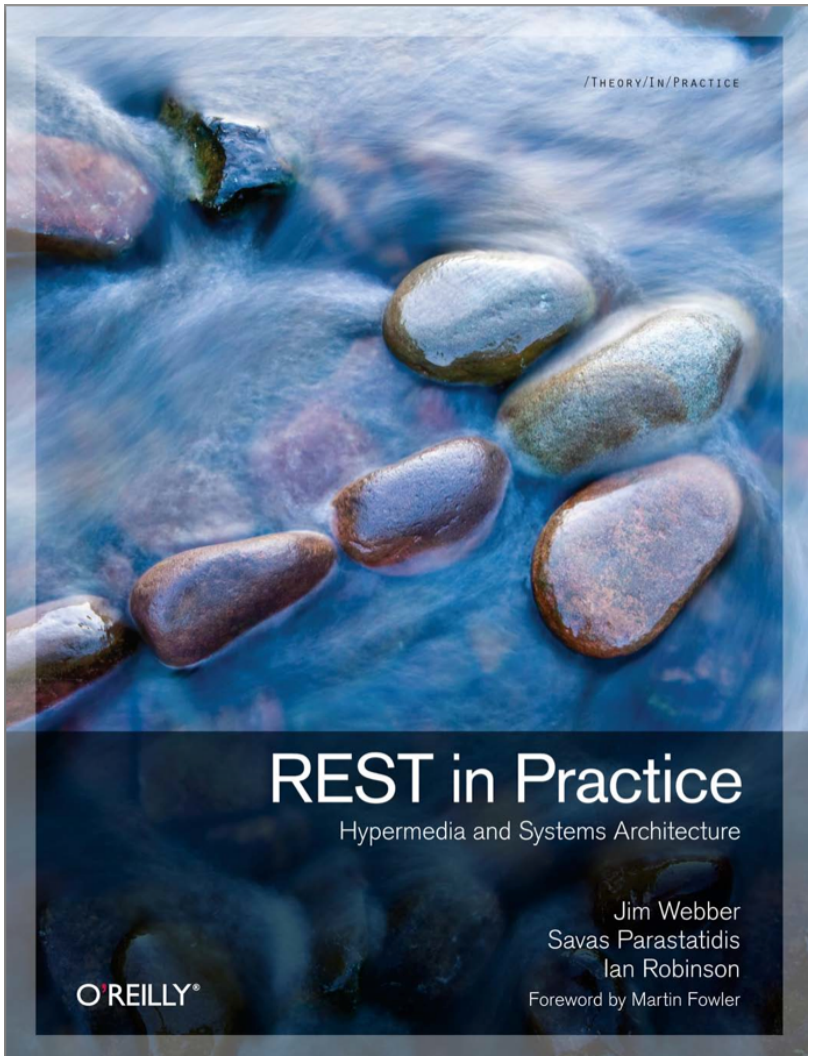


Figure 3-5. *URI tunneling is only at level one in Richardson's maturity model*

Plain Old XML

(*Excerpts from the "REST in Practice" book on the REST Architecture Style*)



POX: Plain Old XML over HTTP

For all its ingenuity (and potential drawbacks too), URI tunneling is a little out of the ordinary for enterprise integration—using addresses to convey business intent is, after all, strange. Our second web-based approach to lightweight integration puts us squarely back in familiar territory: messaging. The Plain Old XML (POX) web-style approach to application integration uses HTTP requests and responses as the means to transfer documents, encoded in regular XML, between a client and a server. It's a lot like SOAP, but without the SOAP envelope or any of the other baggage.

POX is appealing as an approach because XML gives us platform independence, while the use of HTTP gives us practically ubiquitous connectivity between systems. Furthermore, compared to the URI tunneling approach, dealing with XML allows us to use more complex data structures than can be encoded in a URI, which supports more sophisticated integration scenarios.

That's not to say that POX is on a par with enterprise message-oriented middleware, because clearly it isn't. We have to remember that POX is a pattern, not a platform, and POX can't handle transacted or reliable message delivery in a standard way.

Using XML and HTTP for Remote Procedure Calls

POX uses HTTP POST to transfer XML documents between systems. On both sides of the message exchange, the information contained in the XML payload is converted into a format suitable for making local method calls.

It's often said of POX that, like URI tunneling, it too tunnels through the Web. Since POX uses HTTP as a transport protocol, all application semantics reside inside the XML payload and much of the metadata contained in the HTTP envelope is ignored. In fact, POX uses HTTP merely as a synchronous, firewall-friendly transport protocol for convenience. POX would work just as well over a TCP connection, message queues, or even SOAP as it does over HTTP.

While POX isn't rocket science, it can form the basic pattern for constructing distributed systems that are relatively simple to build and easy to deploy, as shown in Figure 3-6.

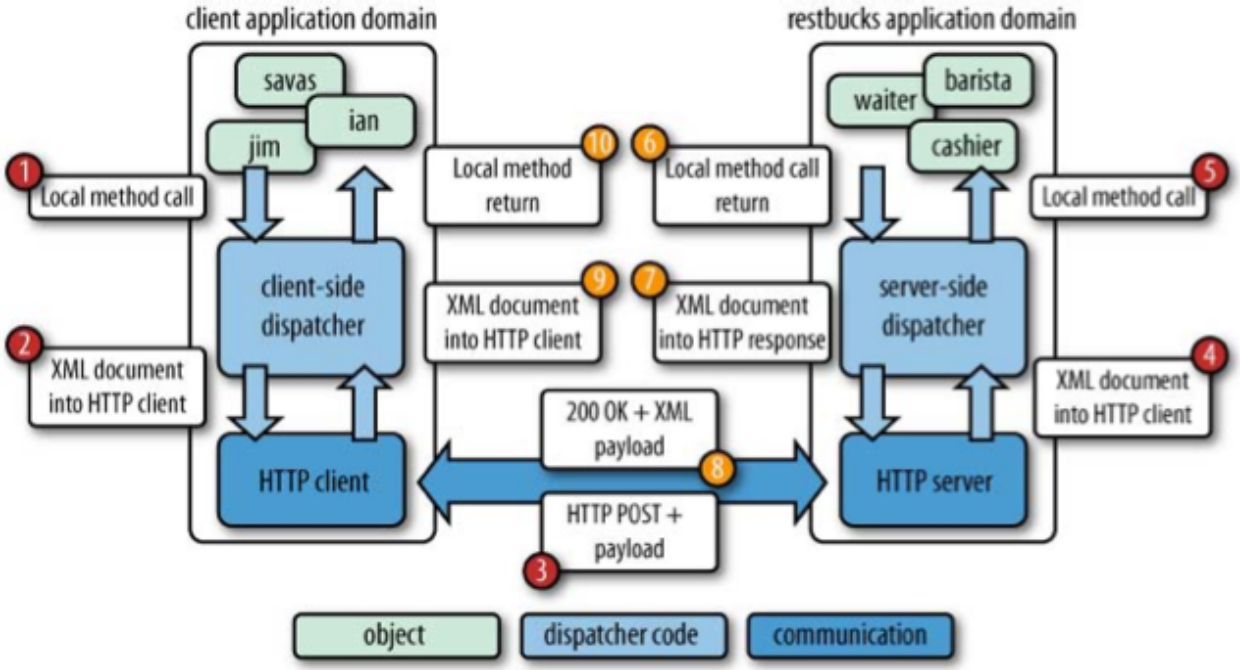


Figure 3-6. *Canonical POX interaction*

Figure 3-6 shows the typical execution model of a POX-based solution:

1. A POX invocation begins with an object in the customer's system calling into a dispatcher that presents a local interface to the remote Restbucks service.
2. The dispatcher converts the values of parameters it receives from the application-level object into an XML document. It then calls into an HTTP client library to pass the information over the network.
3. The HTTP client `POST`s the XML payload from the dispatcher to the remote service.
4. The web server hosting the ordering service accepts the incoming `POST` and passes the request's context to the server-side dispatcher.
5. The server-side dispatcher translates the XML document into a local method call on an object in the Restbucks service.
6. When the method call returns, any returned values are passed to the dispatcher.
7. The dispatcher creates an XML document from the returned values and pushes it back into the web server.
8. The web server generates an HTTP response (habitually using a `200` status code) with the XML document from the dispatcher as the entity body. It sends the response over the same HTTP connection used for the original request.
9. The HTTP client on the customer's system receives the response and makes the XML payload available to the client-side dispatcher.
10. The client-side dispatcher extracts values from the XML response into a return value, which is then returned to the original calling object, completing the remote call.

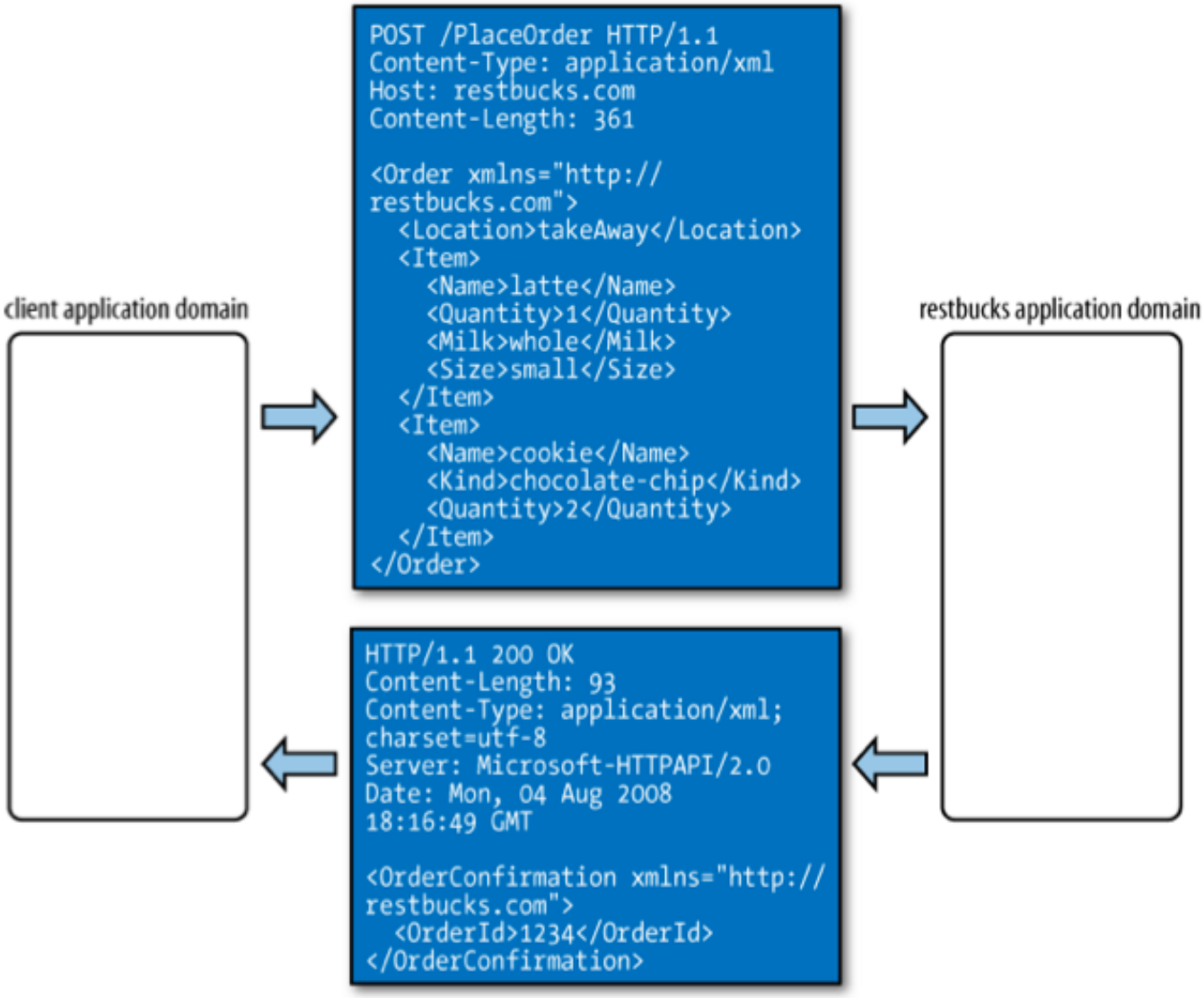


Figure 3-7. *POX wire-level protocol*

The POX approach is popular because it's lightweight and almost universally interoperable, but it is not an especially robust pattern.

> **NOTE**
>
> It's perhaps ironic that POX services are ranked more lowly on Richardson's model than URI tunneling, since they seem less harmful. Nonetheless, we believe arguing over which is the poorest approach is uninteresting; it simply shows that neither model is especially suited for web-scale computing.

POX services are given a level zero rating by Richardson's maturity model. Figure 3-8 highlights that none of the fundamental web technologies is prevalent in such services. Instead, level zero services use HTTP for a transport protocol and a single URI as a well-known endpoint through which XML messages can be exchanged.

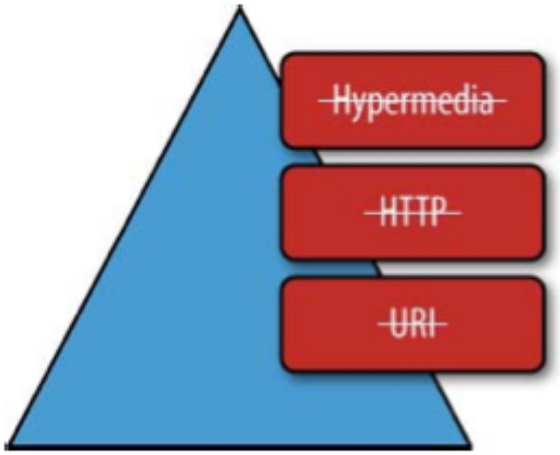
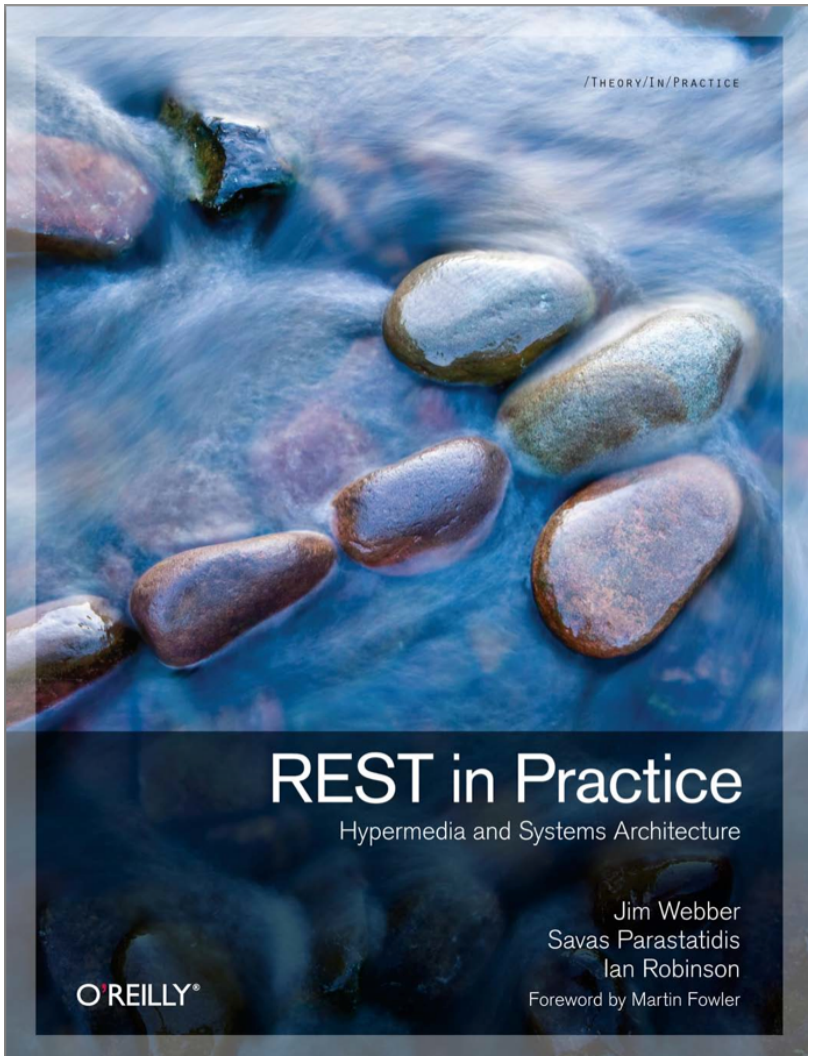


Figure 3-8. *POX services are level zero in Richardson's maturity model*

CRUD Web Services

(*Excerpts from the "REST in Practice" book on the REST Architecture Style*)



Modeling Orders As Resources

In Restbucks, orders are core business entities, and as such, their life cycles are of real interest to us from a CRUD perspective. For the ordering parts of the Restbucks business process, we want to create, read, update, and delete order resources like so:

- Orders are created when a customer makes a purchase.
- Orders are frequently read, particularly when their preparation status is inquired.
- Under certain conditions, it may be possible for orders to be updated (e.g., in cases where customers change their minds or add specialties to their drinks).
- Finally, if an order is still pending, a customer may be allowed to cancel it (or delete it).

Within the ordering service, these actions (which collectively constitute a protocol) move orders through specific life-cycle phases, as shown in Figure 4-1.

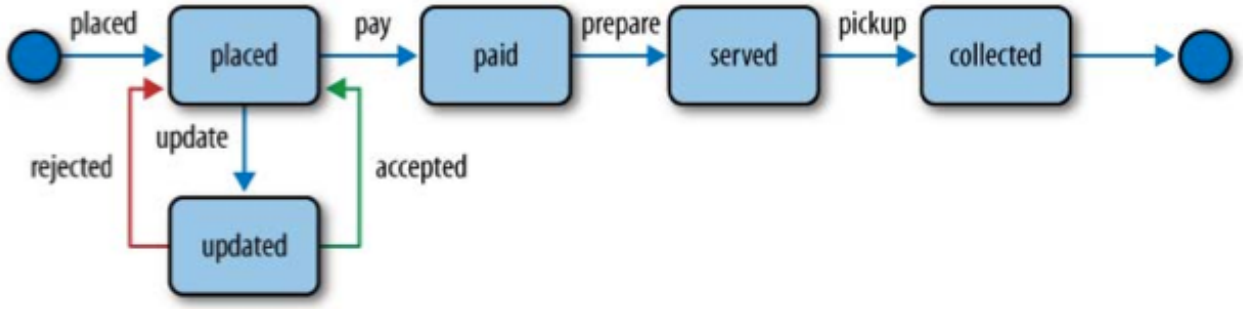


Figure 4-1. *Possible states for an order*

Each operation on an order can be mapped onto one of the HTTP verbs. For example, we use POST for creating a new order, GET for retrieving its details, PUT for updating it, and DELETE for, well, deleting it. When mixed with appropriate status codes and some commonsense patterns, HTTP can provide a good platform for CRUD domains, resulting in really simple architectures, as shown in Figure 4-2.

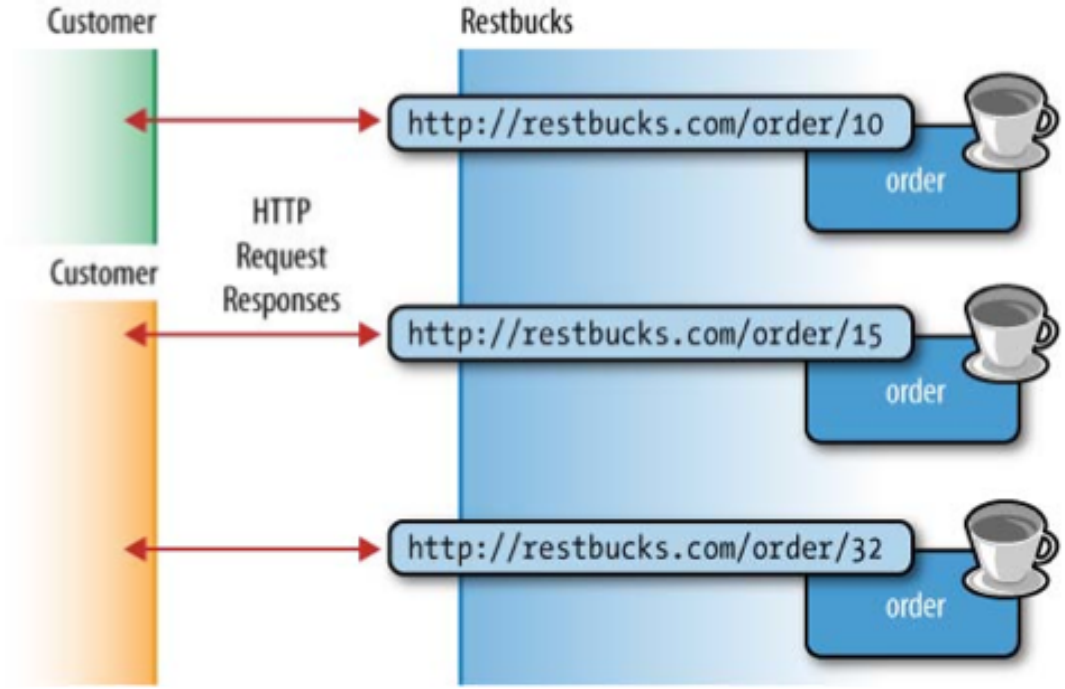


Figure 4-2. *CRUD ordering service high-level architecture*

While Figure 4-2 exemplifies a very simple architectural style, it actually marks a significant rite of passage toward embracing the Web's architecture. In particular, it highlights the use of URIs to identify and address orders at Restbucks, and in turn it supports HTTP-based interactions between the customers and their orders.

Since CRUD services embrace both HTTP and URIs, they are considered to be at level two in Richardson's maturity model. Figure 4-3 shows how CRUD services embrace URIs to identify resources such as coffee orders and HTTP to govern the interactions with those resources.

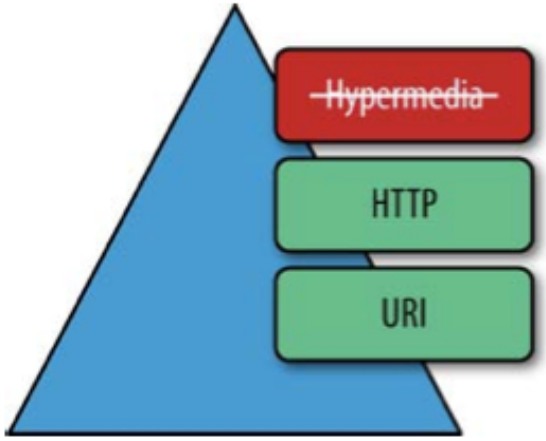


Figure 4-3. *CRUD services reach level two on Richardson's maturity model*

Level two is a significant milestone in our understanding. Many successful distributed systems have been built using level two services. For example, Amazon's S3 product is a classic level two service that has enabled the delivery of many successful systems built to consume its functionality over the Web. And like the consumers of Amazon S3, we'd like to build systems around level two services too!

Building CRUD Services

When you're building a service, it helps to think in terms of the behaviors that the service will implement. In turn, this leads us to think in terms of the contract that the service will expose to its consumers. Unlike other distributed system approaches, the contract that CRUD services such as Restbucks exposes to customers is straightforward, as it involves only a single concrete URI, a single URI template, and four HTTP verbs. In fact, it's so compact that we can provide an overview in just a few lines, as shown in Table 4-1.

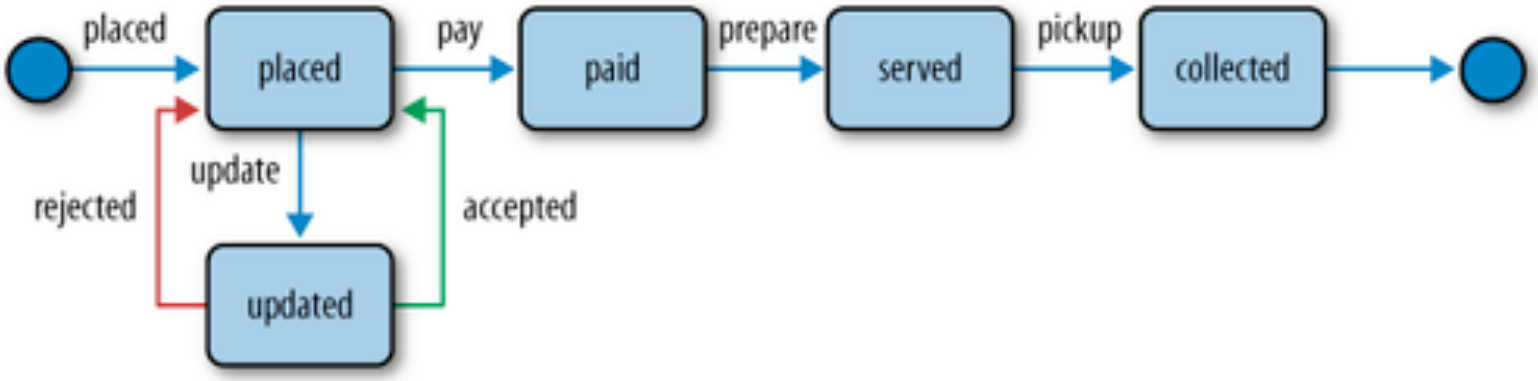
Table 4-1. *The ordering service contract overview*

Verb	URI or template	Use
`POST`	`/order`	Create a new order, and upon success, receive a `Location` header specifying the new order's URI.
`GET`	`/order/{orderId}`	Request the current state of the order specified by the URI.
`PUT`	`/order/{orderId}`	Update an order at the given URI with new information, providing the full representation.
`DELETE`	`/order/{orderId}`	Logically remove the order identified by the given URI.

The contract in Table 4-1 provides an understanding of the overall life cycle of an order. Using that contract, we can design a protocol to allow consumers to create, read, update, and delete orders. Better still, we can implement it in code and host it as a service.

NOTE

What constitutes a good format for your resource representations may vary depending on your problem domain. For Restbucks, we've chosen XML, though the Web is able to work with any reasonable format, such as JSON or YAML.



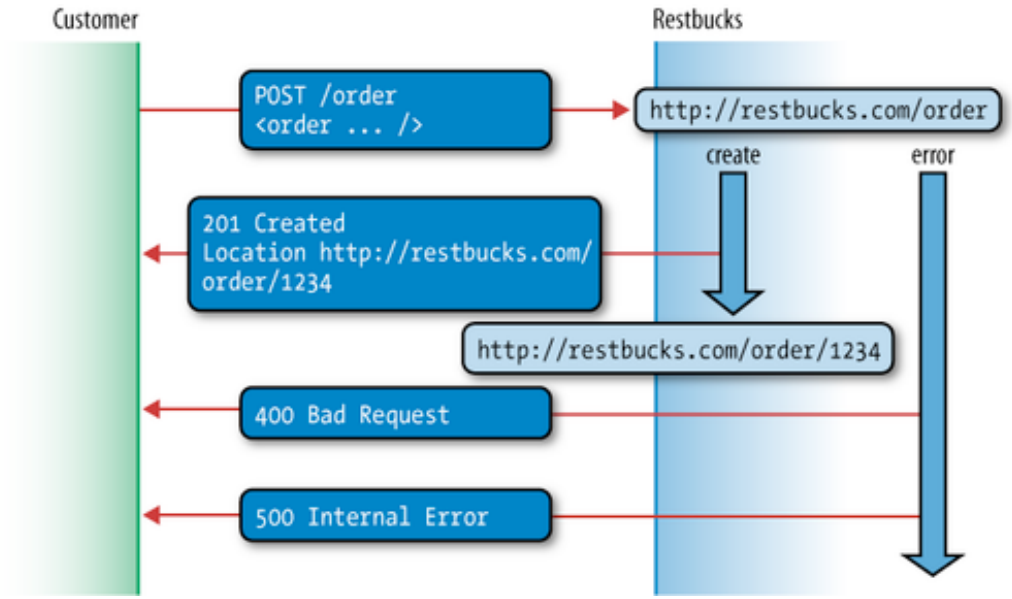
Each operation on an order can be mapped onto one of the HTTP verbs. For example, we use `POST` for creating a new order, `GET` for retrieving its details, `PUT` for updating it, and `DELETE` for, well, deleting it. When mixed with appropriate status codes and some commonsense patterns, HTTP can provide a good platform for CRUD domains, resulting in really simple architectures

Verb	URI or template	Use
`POST`	`/order`	Create a new order, and upon success, receive a `Location` header specifying the new order's URI.
`GET`	`/order/{orderId}`	Request the current state of the order specified by the URI.
`PUT`	`/order/{orderId}`	Update an order at the given URI with new information, providing the full representation.
`DELETE`	`/order/{orderId}`	Logically remove the order identified by the given URI.

POST

```
POST /order HTTP/1.1
Host: restbucks.com
Content-Type: application/xml
Content-Length: 239

<order xmlns="http://schemas.restbucks.com/order">
  <location>takeAway</location>
  <items>
    <item>
      <name>latte</name>
      <quantity>1</quantity>
      <milk>whole</milk>
      <size>small</size>
    </item>
  </items>
</order>
```

```
HTTP/1.1 201 Created
Content-Length: 267
Content-Type: application/xml
Date: Wed, 19 Nov 2008 21:45:03 GMT
Location: http://restbucks.com/order/1234

<order xmlns="http://schemas.restbucks.com/order">
  <location>takeAway</location>
  <items>
    <item>
      <name>latte</name>
      <quantity>1</quantity>
      <milk>whole</milk>
      <size>small</size>
    </item>
  </items>
  <status>pending</status>
</order>
```

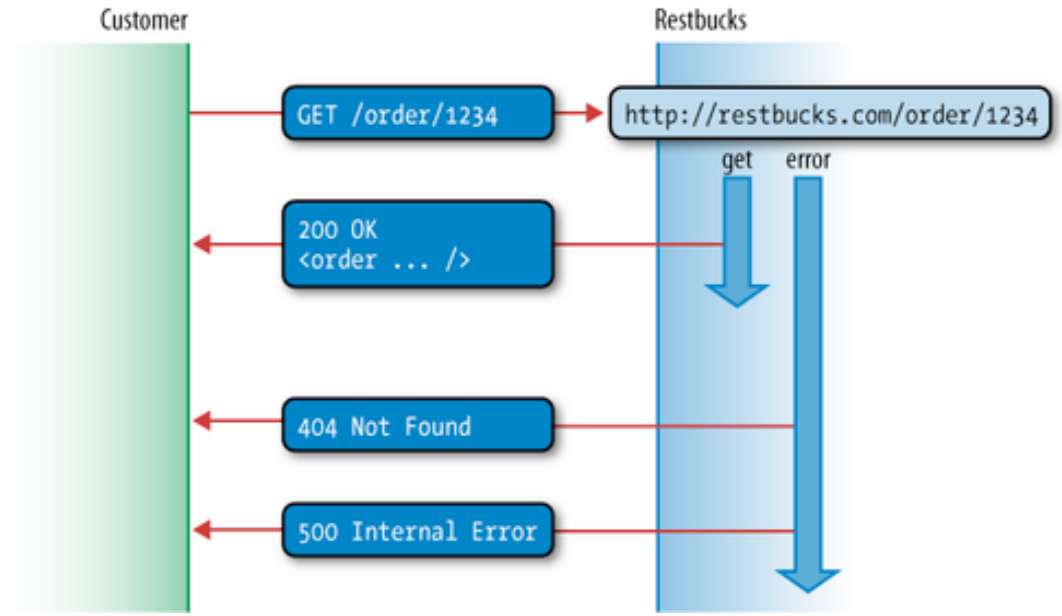



```
HTTP/1.1 400 Bad Request
Content-Length: 250
Content-Type: application/xml
Date: Wed, 19 Nov 2008 21:48:11 GMT

<order xmlns="http://schemas.restbucks.com/order">
  <location>takeAway</location>
  <items>
    <item>
      <!-- Missing drink type -->
      <quantity>1</quantity>
      <milk>whole</milk>
      <size>small</size>
    </item>
  </items>
</order>
```

GET

```
GET /order/1234 HTTP/1.1
Host: restbucks.com
```

```
HTTP/1.1 200 OK
Content-Length: 241
Content-Type: application/xml
Date: Wed, 19 Nov 2008 21:48:10 GMT

<order xmlns="http://schemas.restbucks.com/order">
  <location>takeAway</location>
  <items>
    <item>
      <name>latte</name>
      <quantity>1</quantity>
      <milk>whole</milk>
      <size>small</size>
    </item>
  </items>
</order>
```

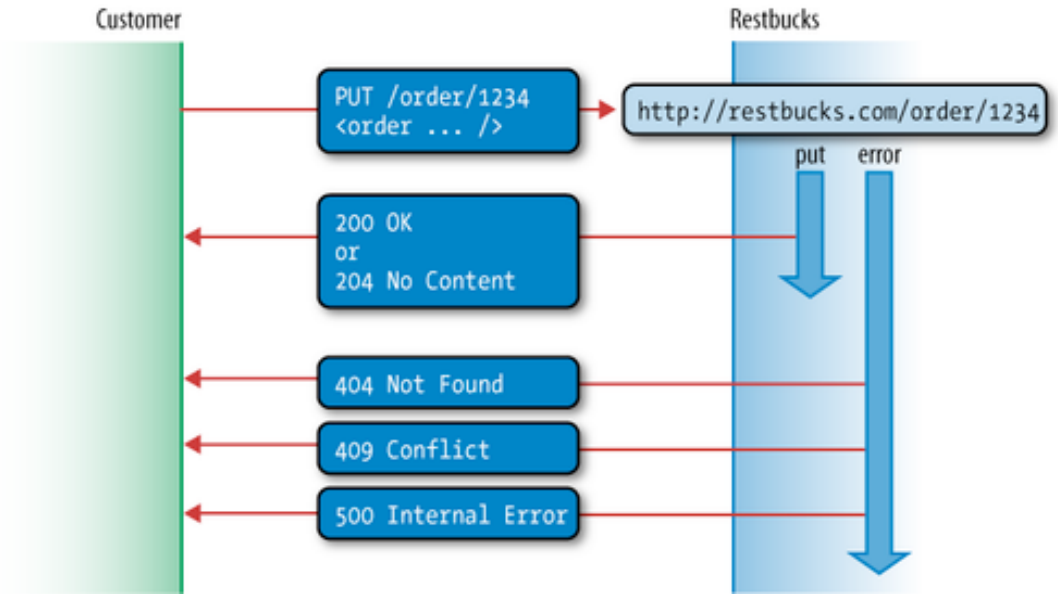



```
HTTP/1.1 404 Not Found
Date: Sat, 20 Dec 2008 19:01:33 GMT
```

PUT

```
PUT /order/1234 HTTP/1.1
Host: restbucks.com
Content-Type: application/xml
Content-Length: 246

<order xmlns="http://schemas.restbucks.com/order">
  <location>takeAway</location>
  <items>
    <item>
      <milk>skim</milk>
      <name>cappuccino</name>
      <quantity>1</quantity>
      <size>large</size>
    </item>
  </items>
</order>
```

```
HTTP/1.1 200 OK
Content-Length: 275
Content-Type: application/xml
Date: Sun, 30 Nov 2008 21:47:34 GMT

<order xmlns="http://schemas.restbucks.com/order">
  <location>takeAway</location>
  <items>
    <item>
      <milk>skim</milk>
      <name>cappuccino</name>
      <quantity>1</quantity>
      <size>large</size>
    </item>
  </items>
  <status>preparing</status>
</order>
```

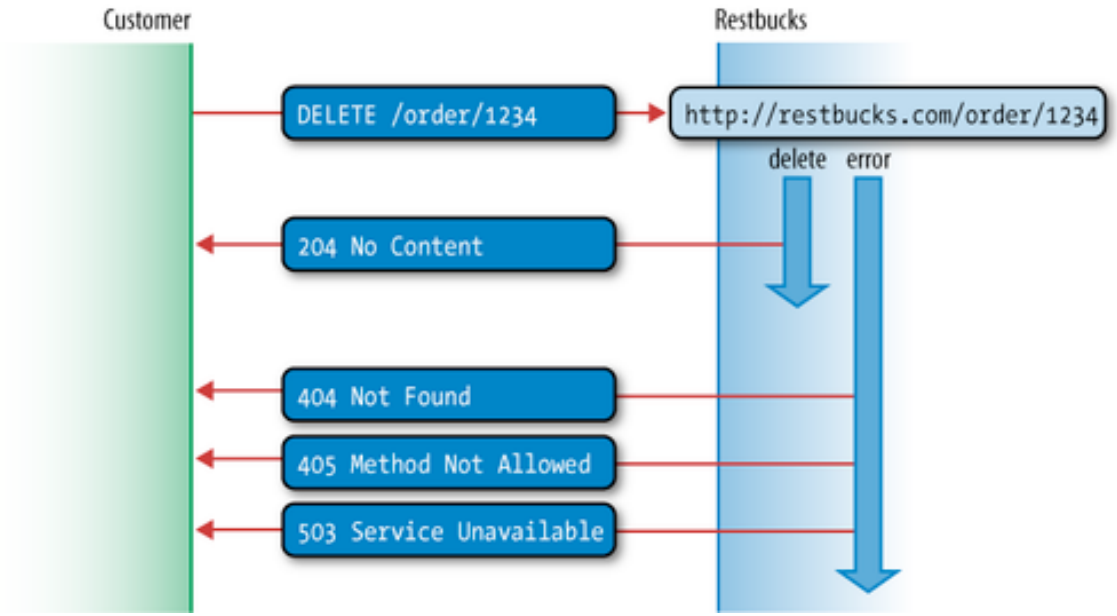



```
HTTP/1.1 204 No Content
Date: Sun, 30 Nov 2008 21:47:34 GMT
```

DELETE

```
DELETE /order/1234 HTTP/1.1
Host: restbucks.com
```

```
HTTP/1.1 204 No Content
Date: Tue, 16 Dec 2008 17:40:11 GMT
```




```
HTTP/1.1 404 Not Found
Content-Length: 0
Date: Tue, 16 Dec 2008 17:42:12 GMT
```

Safety and Idempotency

We saw in Chapter 3 that GET is special since it has the properties of being both safe and idempotent. PUT and DELETE are both idempotent, but neither is safe, while POST is neither safe nor idempotent. Only GET returns the same result with repeated invocations and has no side effects for which the consumer is responsible.

With GET, failed requests can be repeated without changing the overall behavior of an application. For example, if any part of a distributed application crashes in the midst of a GET operation, or the network goes down before a response to a GET is received, the client can just reissue the same request without changing the semantics of its interaction with the server.

In broad terms, the same applies to both PUT and DELETE requests. Making an absolute update to a resource's state or deleting it outright has the same outcome whether the operation is attempted once or many times. Should PUT or DELETE fail because of a transient network or server error (e.g., a 503 response), the operation can be safely repeated.

However, since both PUT and DELETE introduce side effects (because they are not safe), it may not always be possible to simply repeat an operation if the server refuses it at first. For instance, we have already seen how a 409 response is generated when the consumer and service's view of resource state is inconsistent—merely replaying the interaction is unlikely to help. However, HTTP offers other useful features to help us when state changes abound.

Aligning Resource State

In a distributed application, it's often the case that several consumers might interact with a single resource, with each consumer oblivious to changes made by the others. As well as these consumer-driven changes, internal service behaviors can also lead to a resource's state changing without consumers knowing. In both cases, a consumer's understanding of resource state can become misaligned with the service's resource state. Without some way of realigning expectations, changes requested by a consumer based on an out-of-date understanding of resource state can have undesired effects, from repeating computationally expensive requests to overwriting and losing another consumer's changes.

HTTP provides a simple but powerful mechanism for aligning resource state expectations (and preventing race conditions) in the form of *entity tags* and *conditional request headers*. An entity tag value, or `ETag`, is an opaque string token that a server associates with a resource to uniquely identify the state of the resource over its lifetime. When the resource changes—that is, when one or more of its headers, or its entity body, changes—the entity tag changes accordingly, highlighting that state has been modified.

`ETags` are used to compare entities from the same resource. By supplying an entity tag value in a conditional request header—either an `If-Match` or an `If-None-Match` request header—a consumer can require the server to test a precondition related to the current resource state before applying the method supplied in the request.

> **NOTE**
>
> `ETags` are also used for cache control purposes, as we'll see in Chapter 6.

To illustrate how `ETags` can be used to align resource state in a multiconsumer scenario, imagine a situation in which a party of two consumers places an order for a single coffee. Shortly after placing the order, the first consumer decides it wants whole milk instead of skim milk. Around the same time, the second consumer decides it, too, would like a coffee. Neither consumer consults the other before trying to amend the order.

To begin, both consumers `GET` the current state of the order independently of each other. Example 4-25 shows one of the consumer's requests.

Example 4-25. *Consumer GETs the order*

```
GET /order/1234 HTTP/1.1
Host: restbucks.com
```

The service's response contains an ETag header whose value is a hash of the returned representation (Example 4-26).

Example 4-26. *Service generates a response with an ETag header*

```
HTTP/1.1 200 OK
Content-Type: application/xml
Content-Length: 275
ETag: "72232bd0daafa12f7e2d1561c81cd082"

<order xmlns="http://schemas.restbucks.com/order">
  <location>takeAway</location>
  <items>
    <item>
      <milk>skim</milk>
      <name>cappuccino</name>
      <quantity>1</quantity>
      <size>large</size>
    </item>
  </items>
  <status>pending</preparing>
</order>
```

NOTE

The service computes the entity tag and supplies it as a quoted string in the ETag header prior to returning a response. Entity tag values can be based on anything that uniquely identifies an entity: a version number associated with a resource in persistent storage, one or more file attributes, or a checksum of the entity headers and body, for example. Some methods of generating entity tag values are more computationally expensive than others. ETags are often computed by applying a hash function to the resource's state, but if hashes are too computationally expensive, any other scheme that produces unique values can be used. Whichever method is used, we recommend attaching ETag headers to responses wherever possible.

When a consumer receives a response containing an ETag, it can (and should) use the value in any subsequent requests it directs to the same resource. Such requests are called *conditional requests*. By supplying the received entity tag as the value of an If-Match or If-None-Match conditional header, the consumer can instruct the service to process its request only if the precondition in the conditional header holds true.

Of course, consumers aren't obliged to retransmit ETags they've received, and so services can't expect to receive them just because they've been generated. However, consumers that don't take advantage of ETags are disadvantaged in two ways. First, consumers will encounter increased response times as services have to perform more computation on their behalf. Second, consumers will discover their state has become out of sync with service state through status codes such as 409 Conflict at inconvenient and (because they're not using ETags) unexpected times. Both of these failings are easily rectified by diligent use of ETags.

An If-Match request header instructs the service to apply the consumer's request only if the resource to which the request is directed *hasn't changed* since the consumer last retrieved a representation of it. The service determines whether the resource has changed by comparing the resource's current entity tag value with the value supplied in the If-Match header. If the values are equal, the resource hasn't changed. The service then applies the method supplied in the request and returns a 2xx response. If the entity tag values don't match, the server concludes that the resource has changed since the consumer last accessed it, and responds with 412 Precondition Failed.

NOTE

Services are strict about processing the If-Match header. A service can't (and shouldn't) do clever merges of resource state where one coffee is removed and another, independent coffee in the same order is changed to decaf. If two parts of a resource are independently updatable, they should be separately addressable resources. For example, if fine-grained control over an order is useful, each cup of coffee could be modeled as a separate resource.

Continuing with our example, the first consumer does a conditional PUT to update the order from skim to whole milk. As Example 4-27 shows, the conditional PUT includes an If-Match header containing the ETag value from the previous GET.

Example 4-27. *The first consumer conditionally PUTs an updated order*

```
PUT /order/1234 HTTP/1.1
Host: restbucks.com
If-Match: "72232bd0daafa12f7e2d1561c81cd082"

<order xmlns="http://schemas.restbucks.com/order">
  <location>takeAway</location>
  <items>
    <item>
      <milk>whole</milk>
      <name>cappuccino</name>
      <quantity>1</quantity>
      <size>large</size>
    </item>
  </items>
  <status>pending</preparing>
</order>
```

Because the order hadn't been modified since the first consumer last saw it, the PUT succeeds, as shown in Example 4-28.

Example 4-28. *The conditional PUT succeeds*

```
HTTP/1.1 204 No Content
ETag: "6e87391fdb5ab218c9f445d61ee781c1"
```

Notice that while the response doesn't include an entity body, it does include an updated ETag header. This new entity tag value reflects the new state of the order resource held on the server (the result of the successful PUT).

Oblivious to the change that has just taken place, the second consumer attempts to add its order, as shown in Example 4-29. This request again uses a conditional PUT, but with an entity tag value that is now out of date (as a result of the first consumer's modification).

Example 4-29. *The second consumer conditionally PUTs an updated order*

```
PUT /order/1234 HTTP/1.1
Host: restbucks.com
If-Match: "72232bd0daafa12f7e2d1561c81cd082"

<order xmlns="http://schemas.restbucks.com/order">
  <location>takeAway</location>
  <items>
    <item>
      <milk>skim</milk>
      <name>cappuccino</name>
      <quantity>2</quantity>
      <size>large</size>
    </item>
  </items>
  <status>pending</preparing>
</order>
```

The service determines that the second consumer is trying to modify the order based on an out-of-date understanding of resource state, and so rejects the request, as shown in Example 4-30.

Example 4-30. *The response indicates a precondition has failed*

```
HTTP/1.1 412 Precondition Failed
```

When a consumer receives a `412 Precondition Failed` status code, the correct thing to do is to `GET` a fresh representation of the current state of the resource, and then use the `ETag` header value supplied in this response to retry the original request, which is what the second consumer does in this case. Having done a fresh `GET`, the consumer sees that the original order had been modified. The second consumer is now in a position to `PUT` a revised order that reflects both its and the first consumer's wishes.

Our example used the `If-Match` header to prevent the second consumer from overwriting the first consumer's changes. Besides `If-Match`, consumers can also use `If-None-Match`. An `If-None-Match` header instructs the service to process the request only if the associated resource *has changed* since the consumer last accessed it. The primary use of `If-None-Match` is to save valuable computing resources on the service side. For example, it may be far cheaper for a service to compare `ETag` values than to perform computation to generate a representation.

NOTE

`If-None-Match` is mainly used with conditional `GET`s, whereas `If-Match` is typically used with the other request methods, where race conditions between multiple consumers can lead to unpredictable side effects unless properly coordinated.

Both `If-Match` and `If-None-Match` allow the use of a wildcard character, *, instead of a normal entity tag value. An `If-None-Match` conditional request that takes a wildcard entity tag value instructs the service to apply the request method only if the resource doesn't currently exist. Wildcard `If-None-Match` requests help to prevent race conditions in situations where multiple consumers compete to `PUT` a new resource to a well-known URI. In contrast, an `If-Match` conditional request containing a wildcard value instructs the service to apply the request only if the resource does exist. Wildcard `If-Match` requests are useful in situations where the consumer wishes to modify an existing resource using a `PUT`, but only if the resource hasn't already been deleted.

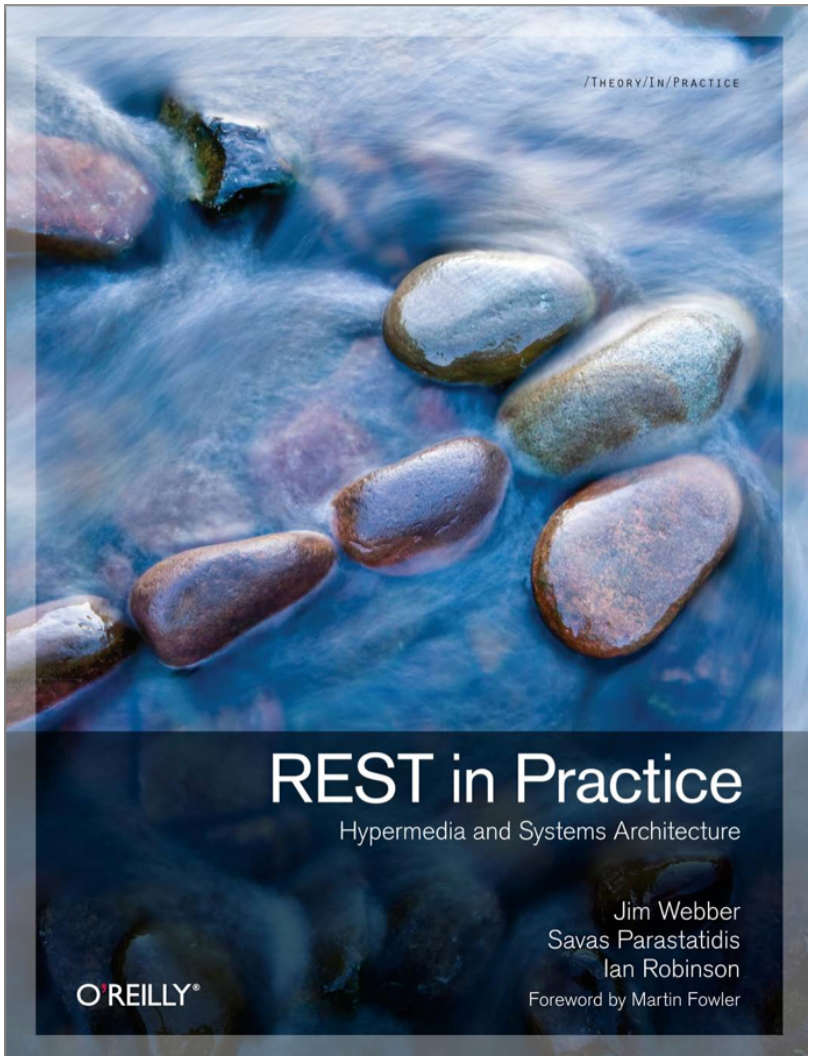
NOTE

As well as `ETag` and its associated `If-Match` and `If-None-Match` headers, HTTP supports a timestamp-based `Last-Modified` header and its two associated conditional headers: `If-Modified-Since` and `If-Unmodified-Since`. These timestamp-based conditional headers act in exactly the same way as the `If-Match` and `If-None-Match` headers, but the conditional mechanism they implement is accurate only to the nearest second—the limit of the timestamp format used by HTTP. Because timestamps are often cheaper than hashes, `If-Modified-Since` and `If-Unmodified-Since` may be preferable in solutions where resources don't change more often than once per second.

In practice, we tend to use timestamps as cheap `ETag` header values, rather than as `Last-Modified` values. By using `ETags` from the outset, we ensure that the upgrade path to finer-grained `ETags` is entirely at the discretion of the service. The service can switch from using timestamps to using hashes without upsetting clients.

Hypermedia

(*Excerpts from the "REST in Practice" book on the REST Architecture Style*)



EMBRACING HTTP AS AN APPLICATION PROTOCOL puts the Web at the heart of distributed systems development. But that's just a start. In this chapter, we will go further, building RESTful services that use hypermedia to model state transitions and describe business protocols.

The Hypermedia Tenet

When browsing the Web, we're used to navigating between pages by clicking links or completing and submitting forms. Although we may not realize it, these interlinked pages describe a protocol—a series of steps we take to achieve a goal, whether that's buying books, searching for information, creating a blog post, or even ordering a coffee. This is the very essence of hypermedia: by transiting links between resources, we change the state of an application.

Hypermedia is an everyday part of our online activities, but despite this familiarity, it's rarely used in computer-to-computer interactions. Although Fielding's thesis on REST highlighted its role in networked systems, hypermedia has yet to figure significantly in contemporary enterprise solutions.

Hypermedia As the Engine of Application State

The phrase *hypermedia as the engine of application state*, sometimes abbreviated to HATEOAS, was coined to describe a core tenet of the REST architectural style. In this book, we tend to refer to the *hypermedia tenet* or just *hypermedia*. Put simply, the tenet says that hypermedia systems transform application state.

A hypermedia system is characterized by the transfer of links in the resource representations exchanged by the participants in an application protocol. Such links advertise other resources participating in the application protocol. The links are often enhanced with semantic markup to give domain meanings to the resources they identify.

For example, in a consumer-service interaction, the consumer submits an initial request to the entry point of the service. The service handles the request and responds with a resource representation populated with links. The consumer chooses one of these links to transition to the next step in the interaction. Over the course of several such interactions, the consumer progresses toward its goal. In other words, the distributed application's state changes. Transformation of application state is the result of the systemic behavior of the whole: the service, the consumer, the exchange of hypermedia-enabled resource representations, and the advertisement and selection of links.

On each interaction, the service and consumer exchange representations of *resource* state, not *application* state. A transferred representation includes links that reflect the state of the application. These links advertise legitimate application state transitions. But the application state isn't recorded explicitly in the representation received by the consumer; it's inferred by the consumer based on the state of all the resources—potentially distributed across many services—with which the consumer is currently interacting.

The current state of a resource is a combination of:

- The values of information items belonging to that resource
- Links to related resources
- Links that represent a transition to a possible future state of the current resource
- The results of evaluating any business rules that relate the resource to other local resources

A service enforces a protocol—a *domain application protocol*, or *DAP*—by advertising legitimate interactions with relevant resources. When a consumer follows links embedded in resource representations and subsequently interacts with the linked resources, the application's overall state changes, as illustrated in Figure 5-1.

NOTE

Domain application protocols (DAPs) specify the legal interactions between a consumer and a set of resources involved in a business process. DAPs sit atop HTTP and narrow HTTP's broad application protocol to support specific business goals. As we shall see, services implement DAPs by adding hypermedia links to resource representations. These links highlight other resources with which a consumer can interact to make progress through a business transaction.

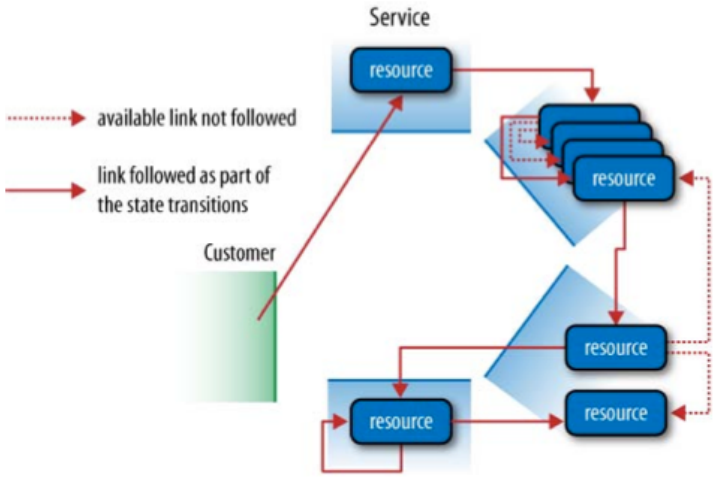


Figure 5-1. *Resources plus hypermedia describe protocols*

NOTE

Services should ensure that any changes they introduce do not violate contracts with existing consumers. While it's fine for a service to make structural changes to the relationships between its resources, semantic changes to the DAP or to the media types and link relations used may change the contract and break existing consumers.

Consumers in a hypermedia system cause state transitions by visiting and manipulating resource state. Interestingly, the application state changes that result from a consumer driving a hypermedia system resemble the execution of a business process. This suggests that our services can advertise workflows using hypermedia. Hypermedia makes it easy to implement business protocols in ways that reduce coupling between services and consumers. Rather than understand a specific URI structure, a consumer need only understand the semantic or business context in which a link appears. This reduces an application's dependency on static metadata such as URI templates or Web Application Description Language (WADL). As a consequence, services gain a great deal of freedom to evolve without having to worry whether existing consumers will break.

Creating a domain-specific hypermedia format isn't as difficult as it might seem. In Restbucks' case, we can build on the XML schemas we've already created. All we have to do is to introduce additional elements into the representations. We can add hypermedia controls to these schemas by defining both a link and the necessary semantic markup. Example 5-3 shows a first attempt at adding an application-specific hypermedia control to an order.

Example 5-3. *A coffee order with a custom hypermedia link*

```
<order xmlns="http://schemas.restbucks.com">
  <location>takeAway</location>
  <item>
    <name>latte</name>
    <quantity>1</quantity>
    <milk>whole</milk>
    <size>small</size>
  </item>
  <cost>2.0</cost>
  <status>payment-expected</status>
  <payment>https://restbucks.com/payment/1234</payment>
</order>
```

In this format, the order representation contains a proprietary <payment> element. <payment> is a hypermedia control.

Example 5-4. *A coffee order with hypermedia*

```
<order xmlns="http://schemas.restbucks.com">
  <location>takeAway</location>
  <item>
    <name>latte</name>
    <quantity>1</quantity>
    <milk>whole</milk>
    <size>small</size>
  </item>
  <cost>2.0</cost>
  <status>payment-expected</status>
  <link rel="http://relations.restbucks.com/payment"
    href="https://restbucks.com/payment/1234" />
</order>
```

The Restbucks hypermedia format specification would have to document the meaning of the `rel` attribute's `payment` value so that consumers understand the role of the linked resource in relation to the current resource.

NOTE

By incorporating reusable hypermedia controls in our format, we can minimize how much of our representation we need to document and explain to consumers. If we can construct our business documents solely from widely understood and reusable building blocks, so much the better. Indeed, we'll have a closer look at a collection of such building blocks when we discuss the Atom format in Chapter 7 and semantics in Chapter 10.

Media types

Grafting hypermedia controls onto XML documents is easy, but it's only half the story. What we really need to do is to create a *media type*. A media type specifies an interpretative scheme for resource representations. This scheme is described in terms of encodings, schemas, and processing models, and is a key step in creating a contract for a service.

A media type name (such as `application/xml`) is a key into a media type's interpretative scheme. Media types are declared in `Content-Type` HTTP headers. Because they're situated outside the payload body to which they're related, consumers can use them to determine how to process a representation without first having to crack open the payload and deeply inspect its content—something that might require relatively heavy processing, such as a decrypt operation, for example.

> **NOTE**
>
> Media type names may also appear in some inline links and forms, where they indicate the *likely* representation format of the linked resource.

A media type value indicates the service's *preferred* scheme for interpreting a representation: consumers are free to ignore this recommendation and process a representation as they see fit.

A media type for Restbucks

The media type declaration used in the `Content-Type` header for interactions with Restbucks is `application/vnd.restbucks+xml`. Breaking it down, the media type name tells us that the payload of the HTTP request or response is to be treated as part of an `application`-specific interaction. The `vnd.restbucks` part of the media type name declares that the media type is vendor-specific (`vnd`), and that the owner is `restbucks`. The `+xml` part declares XML is used for the document formatting.

More specifically, the `vnd.restbucks` part of the media type name marks the payload as being part of Restbucks' DAP. Consumers who know how to interact with a Restbucks service can identify the media type and interpret the payloads accordingly.*

In the Restbucks application domain, we assume that consumers who understand the `application/vnd.restbucks+xml` media type are capable of dealing with everything defined by it. However, it occasionally happens that some consumers want to handle only a subset of the representation formats defined in a media type. While there is no standard solution to this issue on the Web, there is a popular convention defined by the Atom community. The `application/atom+xml` media type defines both the *feed* and the *entry* resource representation formats.* While the vast majority of consumers can handle both, there is a small subset wishing only to deal with standalone entries. In recognition of this need, Atom Publishing Protocol (AtomPub) added a type parameter to the media type value (resulting in `Content-Type` headers such as `application/atom+xml;type=entry`). With such a value for the `ContentType` header, it is now possible to include "entry" resource representations as payloads in HTTP requests or responses without requiring that the processing software agents have a complete understanding of all the Atom-defined formats.

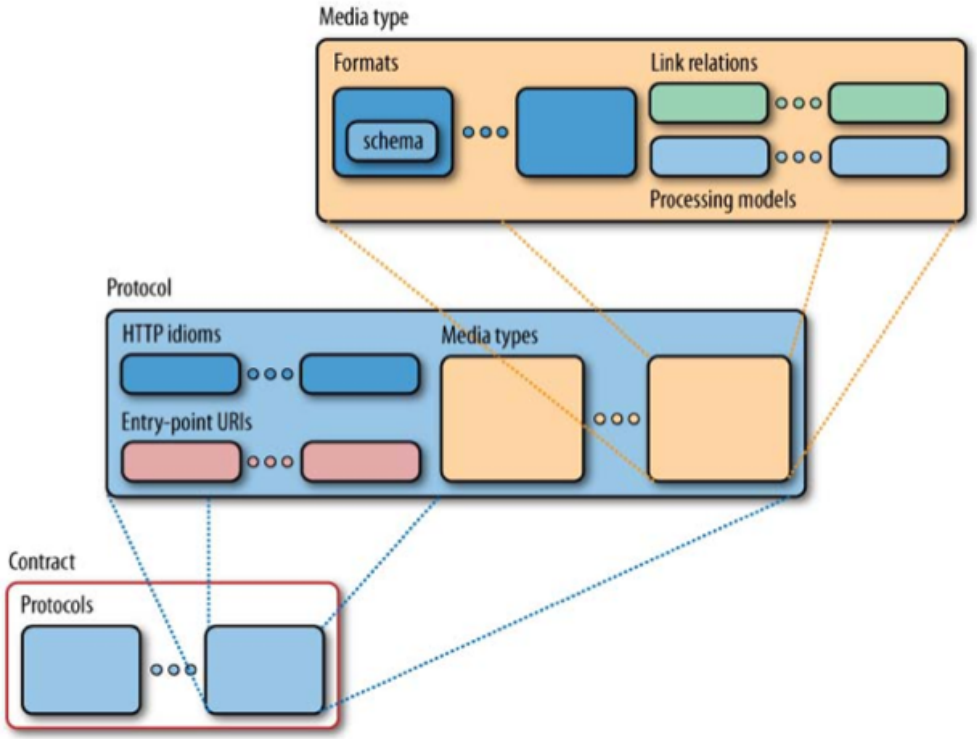


Figure 5-2. *Contracts are a composition of media types and protocols*

Hypermedia Protocols

REST introduces a set of tenets that, when applied to distributed systems design, yield the desirable characteristics of scalability, uniformity, performance, and encapsulation. Using HTTP, URIs, and hypermedia, we can build systems that exhibit exactly the same characteristics. These three building blocks also allow us to implement application protocols tailored to the business needs of our solutions.

The Restbucks Domain Application Protocol

As a web-based system, Restbucks supports a DAP for ordering and payment. Figure 5-4 summarizes the HTTP requests that the ordering service supports and the associated workflow logic each request will trigger.

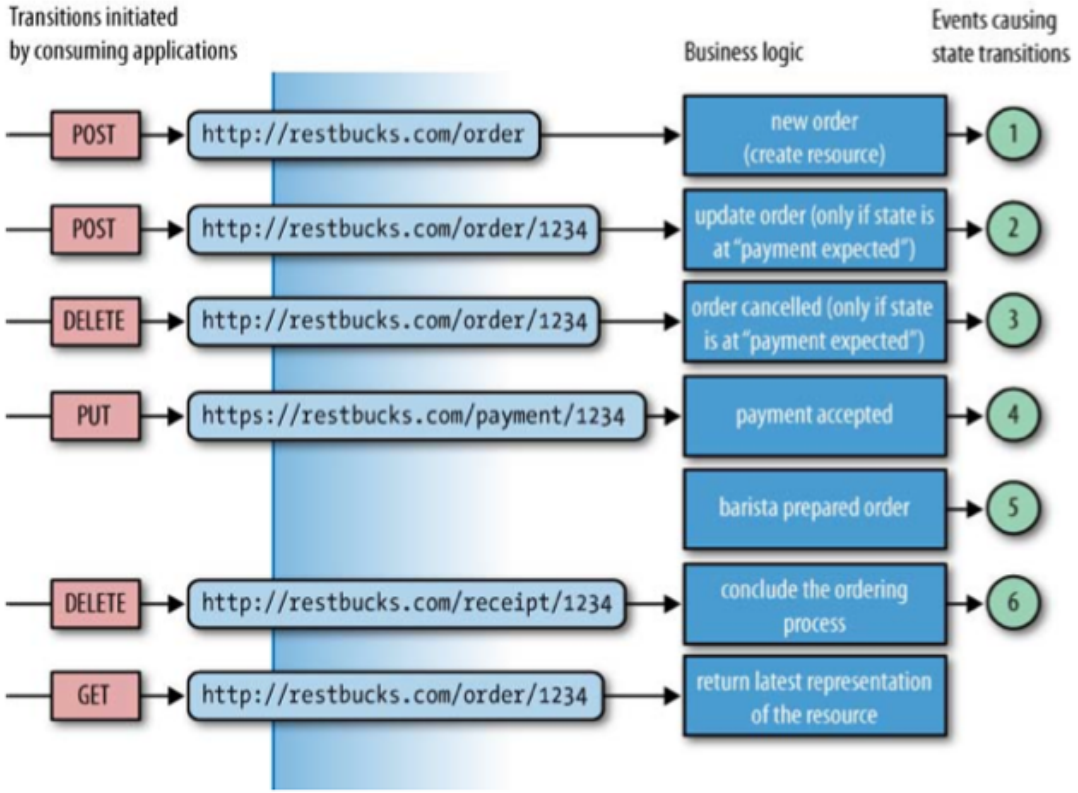


Figure 5-4. *Possible HTTP requests for the Restbucks ordering service*

Figure 5-5 shows the resource state machine for an order as implemented in the service. From this diagram and Figure 5-4, we can derive the DAP:

- POST creates an order.
- Any number of POSTs updates the order.
- A single DELETE cancels the order, or a single PUT to a payment resource pays for the order.
- And finally, a single DELETE confirms that the order has been received.

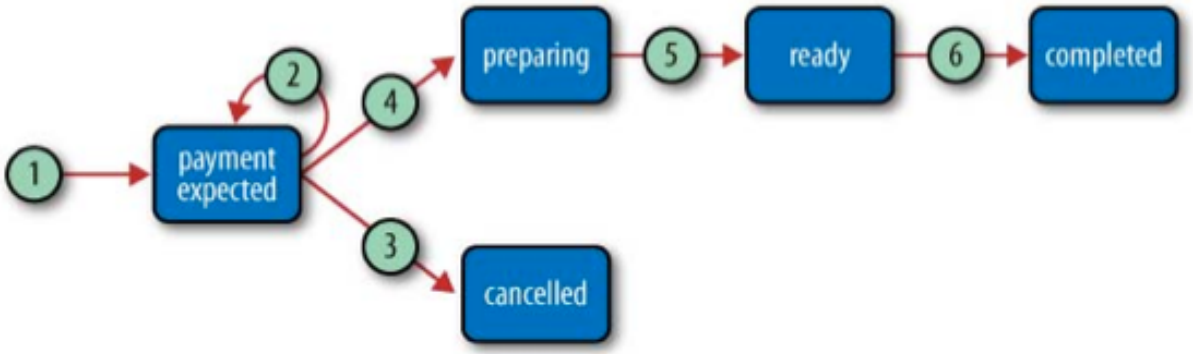


Figure 5-5. *State transitions for the order resource from Figure 5-4*

The state machine diagram in Figure 5-5 is a useful design artifact, but it isn't a good representation format for sharing protocols over the Web. For that, we use hypermedia, which starts with a single, well-known entry point URI.

Resources Updates: PUT Versus POST

In Chapter 4, our CRUD service used PUT to update the state of a resource, whereas in our hypermedia service, we're using POST—a verb we'd normally associate with resource creation.

We've made this change because of the strict semantics of PUT. With PUT, the state encapsulated by the incoming representation will, if legal, wholly replace the state of the resource hosted by the service. This obliges a client to PUT *all* the resource state, including any links, as part of the representation it sends. But since, in this example, clients have no business sending links to a service, we can't expect them to use PUT.

At the time of this writing, a new verb called PATCH has become an Internet RFC under the auspices of the IETF.* Unlike PUT, PATCH is neither safe nor idempotent, but it can be used to send *patch documents* (diffs) over the wire for partial resource modification. Using PATCH, a consumer could legally transmit the business information portion of the order to the Restbucks service, which would then apply the information to an existing order resource and update links as necessary.

PATCH has only recently become an Internet standard (as RFC5789) and is not yet widely supported. Until that situation changes, we will continue to send partial updates to the service using POST.

* *http://tools.ietf.org/html/rfc5789*

Implementing a Hypermedia Service

Implementing a hypermedia service might seem at first to be an intimidating prospect, but in practice, the overhead of building a hypermedia system is modest compared to the effort of building a CRUD system. Moreover, the effort generally has a positive return on investment in the longer term as a service grows and evolves. Although the implementation details will differ from project to project, there are three activities that every service delivery team will undertake throughout the lifetime of a service: designing protocols, choosing formats, and writing software.

We've been describing Restbucks' DAP and formats throughout this chapter, so we're already one step toward a working implementation.

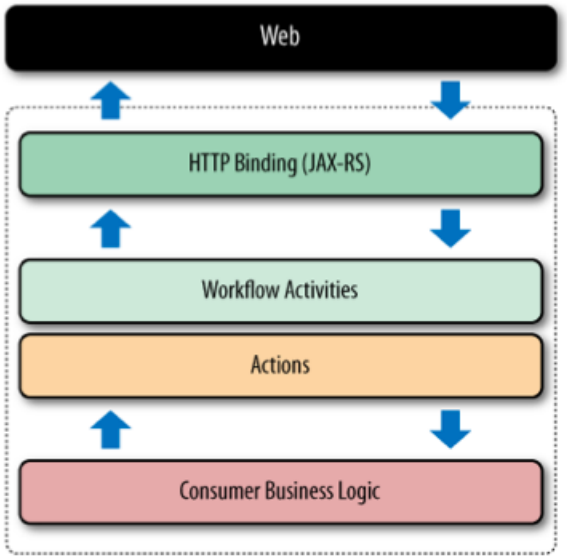


Figure 5-11. *Consumer-side architecture*

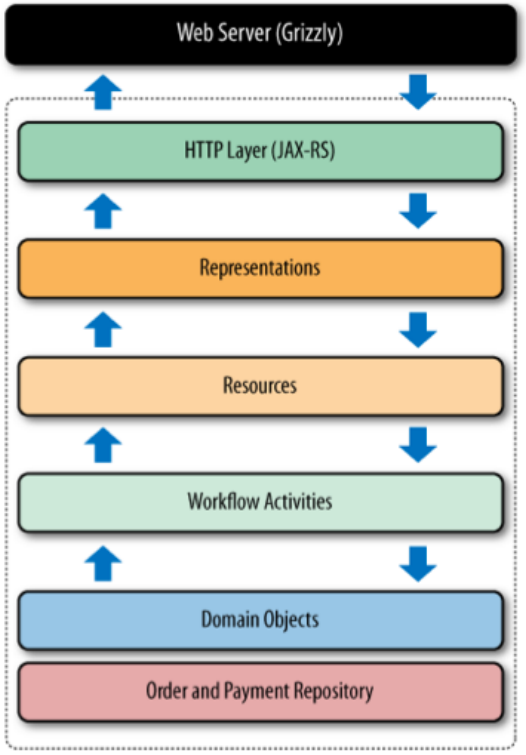


Figure 5-8. *Server-side Java architecture*

The hypermedia controls that the service makes available to the consumer describe the parts of the DAP the consumer can use to drive the service through the next stages of its business workflow, as we see in Figure 5-9.

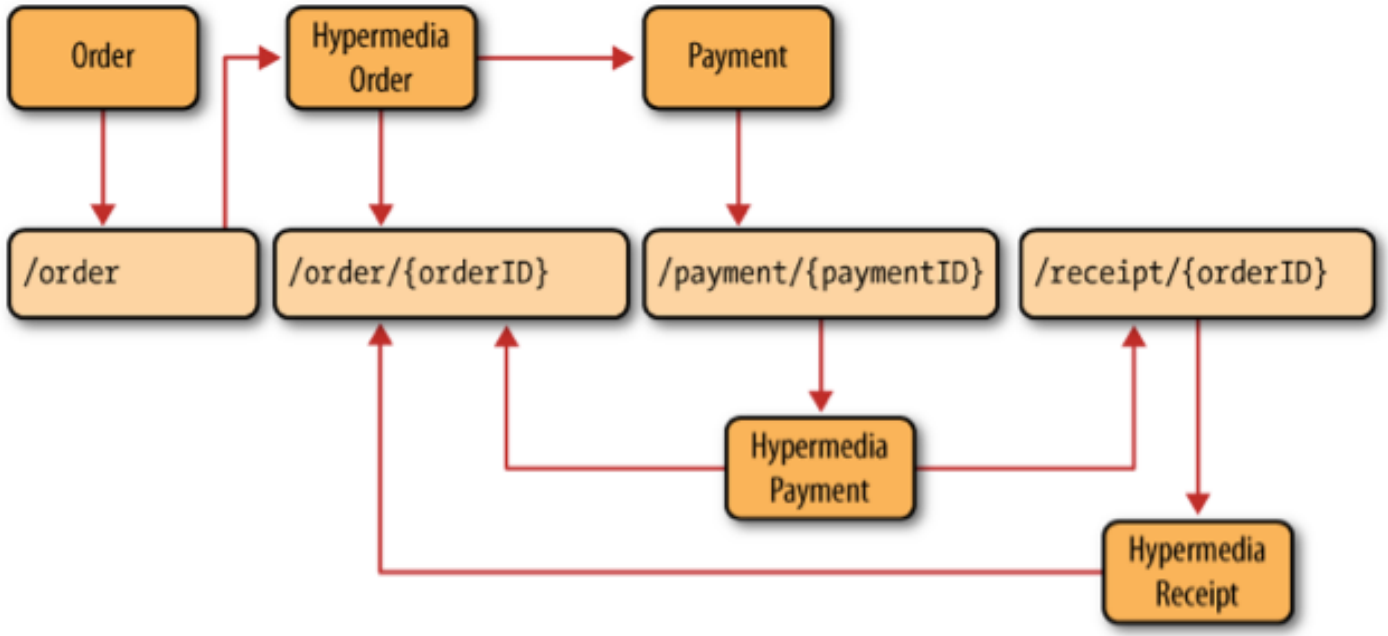


Figure 5-9. *Hypermedia resources describe the ordering and payment protocol to consumers*

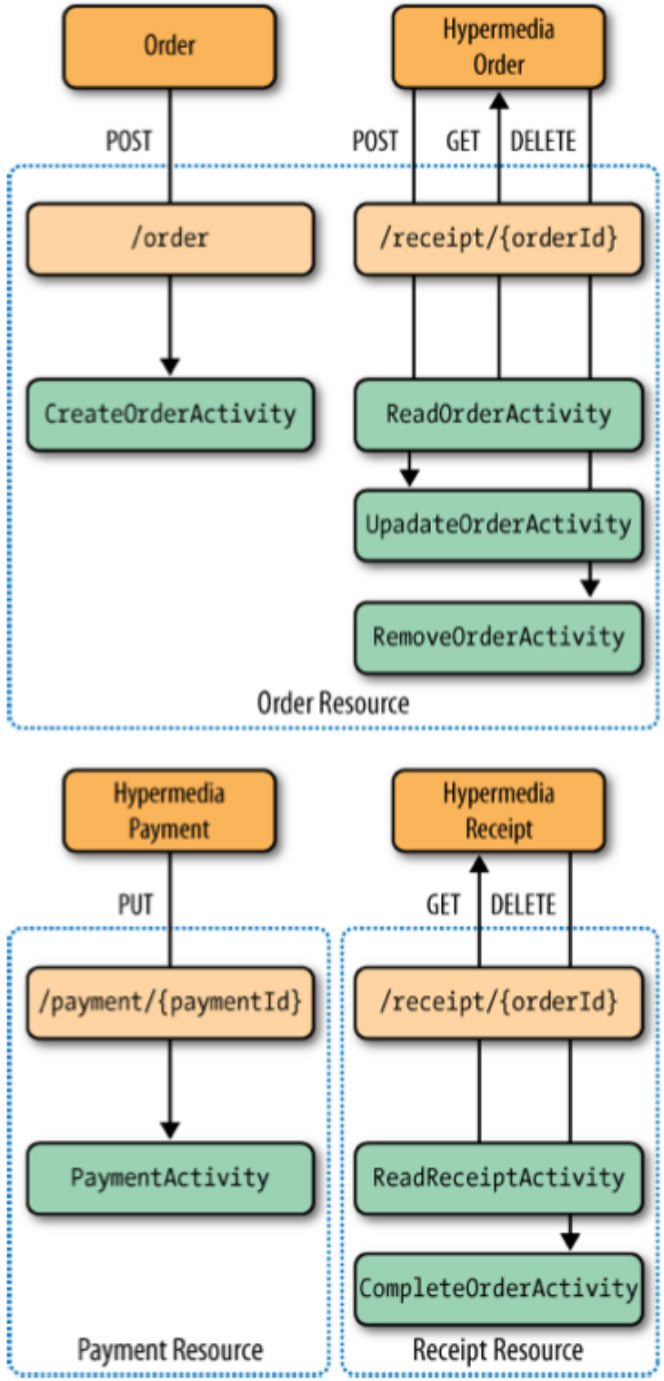


Figure 5-10. *Resources are implemented with workflow activities*

Figure 5-12 shows how activities are again at the core of our architecture. Activities take business objects and use them to create representations to transfer to the service. For responses with representations, activities provide access to the business payload in the received data.

Importantly, activities also surface actions to the business logic—abstractions that correspond to future legal interactions with the service. Actions encapsulate the hypermedia controls and associated semantic context in the underlying representation, allowing the consumer business logic to select the next activity in the workflow.

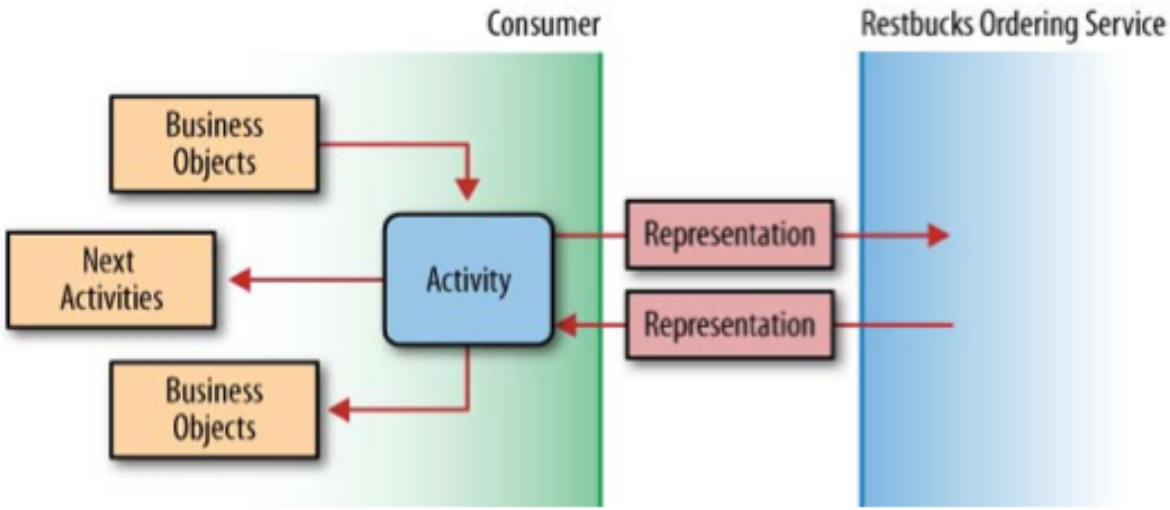


Figure 5-12. *Activities are the key abstraction on the consumer side*

An External DSL for Hypermedia Interactions

The DSL used for the Restbucks hypermedia framework is a state machine description markup language. This language includes constructs representing hypermedia controls, which are injected into representations as XML `<dap:link>` elements.

The DSL in Example 5-26 declares the possible states of a Restbucks order resource and the supported transitions from each of those life-cycle states. It's easy to see how tooling could be used to produce such DSL documents, but we think it's easy enough to write by hand too. While the primary focus of the DSL is to describe a resource's state transitions, it also includes those DAP-specific transitions that relate to the particular resource.

Ready, Set, Action

With the addition of hypermedia, we've reached the pinnacle of Richardson's service maturity model in Figure 5-15. While we still haven't completed our journey—we have plenty more to learn about using the Web as a platform for building distributed systems—we now have all the elements at our disposal to build RESTful services.

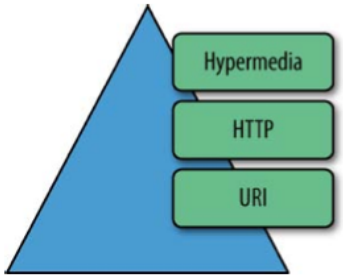


Figure 5-15. *Hypermedia services are level three on the Richardson maturity model*

There's more to the Web than REST, but this milestone is important because of the significant benefits, in terms of loose coupling, self-description, scalability, and maintainability, conferred by the tenets of the REST architectural style.

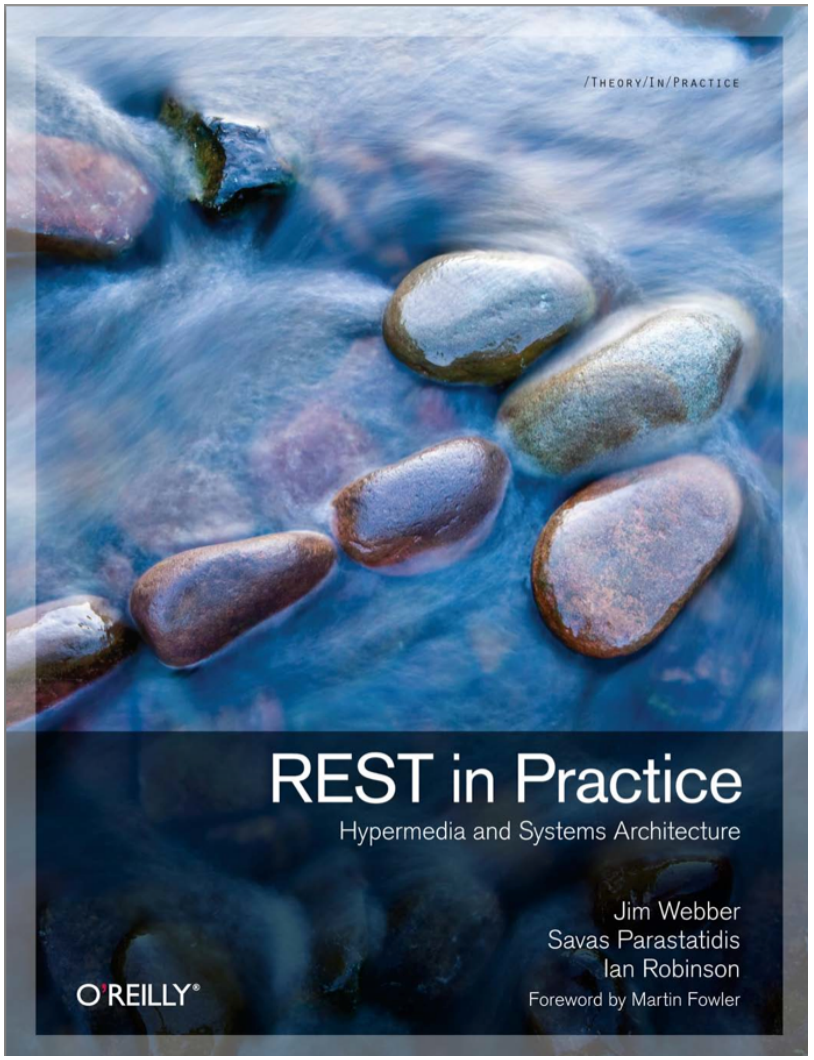
Example 5-26. *Representing a resource state machine and the supported DAP transitions using the Restbucks hypermedia DSL*

```
StateMachine
  UriTemplate http://restbucks.com/order/{id}
  Namespace Restbucks.OrderingService
  MediaType application/vnd.restbucks+xml
  RelationsIn http://relations.restbucks.com

State OrderCreated
   POST NewOrder 201 => Unpaid
     When NotValidOrder 400
End

State Unpaid
   GET GetOrderStatus 200
     When NoSuchOrder 404
   POST UpdateOrder 200
   DELETE CancelOrder 200 => Cancelled
   Links
     latest # When the URI is missing, the active resource's one is assumed
     update
     payment http://restbucks.com/payment/{id}
     cancel
End

State Preparing
   GET GetOrderStatus 200
   Links
     latest
End

State Ready
   GET GetOrderStatus 200
   Links
     latest
     receipt http://restbucks.com/receipt/{id}
End

Final State Delivered End
Final State Cancelled End
```

The state machine description in Example 5-26 consists of a declaration of a set of global properties and a series of states through which the order resource may transition. The first state lexically is considered to be the initial resource state.

Scaling Out

(*Excerpts from the "REST in Practice" book on the REST Architecture Style*)



Scaling Out

THE WEB IS THE WORLD'S LARGEST ONLINE INFORMATION SYSTEM, scaling to billions of devices and users. Hypermedia documents connect a near limitless number of resources, most of which are designed to be read, not modified. At a grand scale, structural hypermedia rules, helped by the safe, idempotent properties of the ubiquitous GET method, and to a lesser extent some of the other cacheable verbs.

From a programmatic web perspective, the infrastructure that has evolved on the Web—particularly around information retrieval—solves many integration challenges. In this chapter, we look at how we can use that infrastructure and some associated patterns to build scalable, fault-tolerant enterprise applications.

GET Back to Basics

According to the HTTP specification, GET is used to retrieve the representation of a resource. Example 6-1 shows a consumer retrieving a representation of an order resource from a Restbucks service by sending an HTTP GET request to the server where the resource is located.

Example 6-1. *A GET request using a relative URI*

```
GET /order/1234 HTTP/1.1
Connection: keep-alive
Host: restbucks.com
```

The value of the Host header plus the relative path that follows GET together give the complete URI of the resource being requested—in this case, *http://restbucks.com/order/1234*. In HTTP 1.1, servers must also support absolute URIs, in which case the Host header is not necessary, as shown in Example 6-2.

Example 6-2. *A GET request using an absolute URI*

```
GET http://restbucks.com/order/1234 HTTP/1.1
```

The response to either of these two requests is shown in Example 6-3.

Example 6-3. *A response to a GET request*

```
HTTP/1.1 200 OK
Content-Length: ...
Content-Type: application/vnd.restbucks+xml
Date: Fri, 26 Mar 2010 10:01:22 GMT
Last-Modified: Fri, 26 Mar 2010 09:55:15 GMT
Cache-Control: max-age=3600
ETag: "74f4be4b"

<order xmlns="http://schemas.restbucks.com">
  <location>takeaway</location>
  <item>
    <drink>latte</drink>
    <milk>whole</milk>
    <size>large</size>
  </item>
</order>
```

As well as the payload, the response includes some headers, which help consumers and any intermediaries on the network process the response. Importantly, we can use some of these headers to control the caching behavior of the order representation.

As we discussed in Chapter 3, GET is both safe and idempotent. We use GET simply to *retrieve* a resource's state representation, rather than deliberately *modify* that state.

NOTE

If we don't want the entire representation of a resource, but just want to inspect the HTTP headers, we can use the HEAD verb. HEAD allows us to decide how to make forward progress based on the processing context of the identified resource, without having to pay the penalty of transferring its entire representation over the network.

Because GET has no impact on resource state, it is possible to optimize the network to take advantage of its safe and idempotent characteristics. If we see a GET request, we immediately understand that the requestor doesn't want to modify anything. For these requests, it makes sense to store responses closer to consumers, where they can be reused to satisfy subsequent requests. This optimization is baked into the Web through caching.

NOTE

GET isn't the only HTTP verb to yield cacheable responses, though it is by far the most prevalent and useful. We'll focus on GET for now because it's so widespread, but later in the chapter we'll look at caching in the context of other verbs too.

Benefits of Caching

Optimizing the network using caching improves the overall quality-of-service characteristics of a distributed application. Caching significantly benefits four areas of systems operation, allowing us to:

Reduce bandwidth
: By reducing the number of network hops required to retrieve a representation, caching reduces network traffic and conserves bandwidth.

Reduce latency
: Because caches store copies of frequently accessed information nearer to where the information is used, caching reduces the time it takes to satisfy a request.

Reduce load on servers
: Because they are able to serve a percentage of requests from their own stores, caches reduce the number of requests that reach an origin server.

Hide network failures
: Caches can continue to serve cached content even if the origin server that issued the content is currently unavailable or committed to an expensive processing task that prevents it from generating a response. In this way, caches provide fault tolerance by masking intermittent failures and delays from consumers.

Ordinarily, we'd have to make a substantial investment in development effort and middleware in order to achieve these benefits. However, the Web's existing caching infrastructure means we don't have to; the capability is already globally deployed.

Caching and the Statelessness Constraint

One of the Web's key architectural tenets is that servers and services should not preserve application state. The statelessness constraint helps make distributed applications fault-tolerant and horizontally scalable. But it also has its downsides. First, because application state is not persisted on the server, consumers and services must exchange application state information with each request and response, which adds to the size of the message and the bandwidth consumed by the interaction. Second, because the constraint requires services to forget about clients between requests, it prevents the use of the classical publish-subscribe pattern (which requires the service to retain subscriber lists). To receive notifications, consumers must instead frequently poll services to determine whether a resource has changed, adding to the load on the server.

Caching helps mitigate the consequences of applying the statelessness constraint.* It reduces the amount of data sent over the network by storing representations closer to where they are needed, and it reduces the load on origin servers by having caches satisfy repeated requests for the same data.

* Benjamin Carlyle discusses this topic in more detail here: *http://soundadvice.id.au/blog/2010/01/17/.*

Response Headers Used for Caching

There are two main HTTP response headers that we can use to control caching behavior:

`Expires`
: The `Expires` HTTP header specifies an absolute expiry time for a cached representation. Beyond that time, a cached representation is considered stale and must be revalidated with the origin server. A service can indicate that a representation has already expired by including an `Expires` value equal to the `Date` header value (the representation expires *now*), or a value of `0`. To indicate that a representation never expires, a service can include a time up to one year in the future.

`Cache-Control`
: The `Cache-Control` header can be used in both requests and responses to control the caching behavior of the response. The header value comprises one or more comma-separated directives. These directives determine whether a response is cacheable, and if so, by whom, and for how long.

If we can determine an absolute expiry time for a cached response, we should use an `Expires` header. If it's more appropriate to indicate how long the response can be considered fresh once it has left the origin server, we should use a `Cache-Control` header, adding a `max-age` or `s-maxage` directive to specify a relative *Time to Live* (TTL).

Cacheable responses (whether to a `GET` or to a `POST` request) should also include a *validator*—either an `ETag` or a `Last-Modified` header:

`ETag`
: In Chapter 4, we said that an `ETag` value is an opaque string token that a server associates with a resource to uniquely identify the state of the resource over its lifetime. When the resource changes, the entity tag changes accordingly. Though we used `ETag` values for concurrency control in Chapter 4, they are just as useful for validating the freshness of cached representations.

`Last-Modified`
: Whereas a response's `Date` header indicates when the response was generated, the `Last-Modified` header indicates when the associated resource last changed. The `Last-Modified` value cannot be later than the `Date` value.

Example 6-4. *A response with an absolute expiry time*

```
Request:
GET /product-catalog/9876
Host: restbucks.com

Response:
HTTP/1.1 200 OK
Content-Length: ...
Content-Type: application/vnd.restbucks+xml
Date: Fri, 26 Mar 2010 09:33:49 GMT
Expires: Sat, 27 Mar 2010 09:33:49 GMT
Last-Modified: Fri, 26 Mar 2010 09:33:49 GMT
ETag: "cde893c4"

<product xmlns="http://schemas.restbucks.com/product">
  <name>Sumatra Organic Beans</name>
  <size>1kg</size>
  <price>12</price>
</product>
```

This response can be cached and will remain fresh until the date and time specified in the `Expires` header. To revalidate a response, a cache uses the `ETag` header value or the `Last-Modified` header value to do a conditional `GET`.* If a consumer wants to revalidate a response, it should include a `Cache-Control: no-cache` directive in its request. This ensures that the conditional request travels all the way to the origin server, rather than being satisfied by an intermediary.

Example 6-5 shows a response containing a `Last-Modified` header, an `ETag` header, and a `Cache-Control` header with a `max-age` directive.

* Choosing between `ETag` values and `Last-Modified` timestamps depends on the granularity of updates to the resource. `Last-Modified` is only as accurate as a timestamp (to the nearest second), while `ETags` can be generated at any frequency. Typically, however, timestamps are cheaper to generate.

Example 6-5. *A response with a relative expiry time*

```
Request:
GET /product-catalog/1234
Host: restbucks.com

Response:
HTTP/1.1 200 OK
Content-Length: ...
Content-Type: application/vnd.restbucks+xml
Date: Fri, 26 Mar 2010 12:07:22 GMT
Cache-Control: max-age=3600
Last-Modified: Fri, 26 Mar 2010 11:45:00 GMT
ETag: "59c6dd9f"

<product xmlns="http://schemas.restbucks.com/product">
  <name>Fairtrade Roma Coffee Beans</name>
  <size>1kg</size>
  <price>12</price>
</product>
```

This response is cacheable and will remain fresh for up to one hour. As with the previous example, a cache can revalidate the representation using either the `Last-Modified` value or the `ETag` value.