CHAPTER 11

REST: Representational State Transfer

Until now our focus has been XML-RPC and SOAP-based web services. However, recent developments in this field show that these methods don't cover all available web services. One alternative is Representational State Transfer, or REST. Despite the long history (in Internet time) and large number of applications that run the Internet, this approach for developing distributed applications wasn't popular until recently. Some of the more vocal proponents of the REST style include Roy Fielding, Mark Baker, Jeff Bone, Paul Prescod, and Roger Costello. The information presented in this chapter is based largely on their ideas.

Think of a distributed system* in which every resource has a standard way to reference it (URI), uses a standard method of access (HTTP), has well-defined document structure of its metadata and representation (XML), supports multiple formats (through content negotiation and transformations), refers to other related resources in which the client can dig for more information (XLink, XPointer), has a standard way to describe how resources can be accessed (WSDL, WRDL), advertises its presence in a standard manner (UDDI, WS-Inspection), and uses a common mechanism for authentication and authorization. It isn't expected that all the pieces will be in place at once; this vision can be implemented incrementally. What's more, it can be done for little additional cost if the simple principles described in this chapter are followed.

Defining REST

REST stands for Representational State Transfer, a term coined by Roy Fielding in his doctoral dissertation. (Check it out at *http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm.)* The author introduces the REST architectural style, developed

* This is a restatement of Phillip J. Windley's vision outlined in *http://www.das.state.ut.us/cc/aug2002/ciomessage.html*.

as an abstract model of the web architecture to guide the redesign and definition of the HTTP and URIs.

REST encompasses a simple philosophy for modeling problem domains: "give a URI to everything that can be manipulated by a limited set of operations and let the client software determine the usage metaphor." This pattern is ubiquitous: a small number of methods applied to diverse kinds of data.

The main questions that motivated the development of the REST architectural style were: how to introduce a new set of functionality to an architecture that is already widely deployed and how to ensure that its introduction doesn't adversely impact, or even destroy, the architectural properties that have enabled the Web to succeed. This chapter discusses one possible answer to that question. Some examples will be provided, although the main focus of the discussion is on architecture and design of applications in accordance with REST's principles; examples are added purely for illustrative purposes and will provide you some guidelines.

The Scientific Definition

If you aren't comfortable with definitions derived from Ph.D. dissertations, you can skip this section (it will be kept short).

The REST style is an abstraction of the architectural elements within a distributed system. The key aspects of REST are the nature and state of data elements (compare this approach to the distributed object style, in which the key aspect is the encapsulation of all data within the processing components). REST identifies six data elements: a resource, resource identifier, resource metadata, representation, representation metadata, and control data, shown in Table 11-1.

Table 11-1. REST data elements

Data element	Examples
Resource	The intended conceptual target; "the home page of the O'Reilly web site," "the latest version of the `SOAP::Lite` toolkit"
resource identifier	Name, URI; *http://soaplite.com/download/SOAP-Lite-latest.zip*
resource metadata	`Vary`
Representation	Sequence of bytes, HTML document, archive document, image document
representation metadata	`Last-Modified`, media type; `text/xml`
control data	`If-Modified-Since`, `If-Match`

The key abstraction of information in REST is a *resource*. Any information that can be *named* can be a resource: a document, a home page of a weblog, or a search result. A resource is a conceptual object that has identity, state, and behavior. A *resource identifier* (in the form of a URI) identifies the particular resource that might be observable via its *representations*. A representation (for example, a web page or a document) is something that you get from a resource and not the resource itself. Some resources are static; their representations don't change. When queried, they always correspond to the same sequence of bytes. Other resources are dynamic: their value changes over time, but the semantics stay the same. The semantics are unique (one concept, one identity) and distinguish one resource from another.

Representation metadata describes the representation and *resource metadata* describes the resource. *Control data* defines the purpose of a message between components, such as the action being requested. If the value set of a resource consists of multiple representations, content negotiation may be used to select the appropriate representation depending on the message control data.

The Practical Definition

The Web is full of resources that are named by URLs and have different representations. For example, CPAN (the Comprehensive Perl Archive Network) defines a resource that may be described as "Distribution of `SOAP::Lite` module v0.55." Clients may access this resource using this URL:

http://search.cpan.org/author/KULCHENKO/SOAP-Lite-0.55/

A representation of the resource is returned. This might be a HTML page that provides information about the module. This representation places the client application in a *state* (another representation, for example a XML document, may place the client in a different state). The client traverses the returned representation and accesses another resource using a link in the document:

http://search.cpan.org/src/KULCHENKO/SOAP-Lite-0.55/Changes

The client application changes the state based on the returned representation. As a result, the transfer of state occurs with each resource representation; hence Representational State Transfer.

REST is an architecture style that separates a server's implementation from a client's perception of resources. It enables transfer of data in streams of unlimited size and type, supports intermediaries as data transformation and caching components, and concentrates the application state within the user agent components. It leverages HTTP and the URI namespace for all types of applications and allows for independent evolution of clients, servers, and intermediaries. In the future it may allow accessing individual data objects as resources.

Figure 11-1 gives a summary of REST in one diagram.

All these services that have been used in past years—search engines (Google), book ordering services (Amazon), CPAN—are services created according to REST principles.

Figure 11-1. REST definition

REST Principles

There are 12 principles, shown in Table 11-2. Most are either self-explanatory or will be covered later in this chapter.

Table 11-2. REST principles

Group	Principle
Resources	(1) A resource is anything that has identity.
	(2) Every resource has a URI.
	(3) A URI is "opaque," exposes no details of its implementation.
Protocol	(4) `GET` operations are "idempotent," free of side effects.
	(5) Any request that doesn't have side effects should use `GET`.
	(6) All interactions are stateless.
Representations	(7) Data and metadata formats are documented.
	(8) Data is available in multiple flavors.
	(9) Representations include links to other resources.
Style	(10) Document and advertise your service API.
	(11) Use available standards and technology.
	(12) Refine and extend architecture, standards and tools.

The first group of principles relates to standardized addressing (URI), which describes how to address resources. The second group of principles relates to standard application protocol (HTTP), which describes what operations can be applied to the resources. Stateless interaction in this context means that each request from client to server must contain all the information necessary to execute the request, and can't take advantage of any stored context on the server. The third group of principles relates to standard resource representation (HTML, GIF, JPG, XML-based vocabularies), which describes what data is passed to resources or accepted from them. Standardization of all these key elements of the REST approach allows for the achievement of widespread interoperability, and an ability to rapidly and independently evolve clients, servers and systems as a whole.

Aesthetics of URI Design

Uniform resource locators (URLs), like uniform resource names (URNs), are a subset of the general class of uniform resource identifiers (URIs). The distinction between URIs, URLs, URNs, and the rest of resource identifiers is often more confusing than useful.[*] In this chapter, the term "URI" refers to all of them.

URI syntax

Before going into detail about resource modeling, let's first review and elaborate on URIs as discussed in Chapter 2. An absolute URI reference consists of three parts: a *scheme*, a *scheme-specific* part, and a f*ragment identifier*. Some schemes are hierarchical, allowing for both relative and absolute URIs (`http:`) and some aren't, allowing only absolute URIs (such as `mailto:`). For hierarchical namespaces, a scheme-specific part is broken down into *authority*, *path*, and *query* components. Relative URI references can be created by omitting the scheme and authority components (they are implied by the context of the URI reference). These forms of URI reference syntax are summarized as follows:

```
<scheme>:<scheme-specific-part>#<fragment>
<scheme>://<authority><path>?<query>#<fragment>
<path>?<query>#<fragment>
```

Resource modeling

The strength and flexibility of REST comes from the pervasive use of URIs. REST is about exposing resources through URIs, not services through messaging interfaces. On this view, URIs are acted on by HTTP methods, and the result of those actions is to transfer representations of some resource from the origin server to the client that

[*] If you're interested, the paper "URIs, URLs, and URNs: Clarifications and Recommendations" describes the relationship among the concepts and the reason for the confusion. Check it out at *http://www.w3.org/TR/2001/NOTE-uri-clarification-20010921/*.

initiated the request. Some REST advocates call the process of bringing an application into this model "resource modeling," i.e., how to model a given problem domain as a set of resources. Each component of the URI should be a resource for which operations make sense. One guiding principle is to ask "does it make sense to use `GET`, `PUT`, `POST`, or `DELETE` operations on this named resource?"

Most URIs are opaque to client software most of the time. In other words, a public API shouldn't depend on the structure of the URIs; the structure of the URI is irrelevant to client software. URI opacity opens the door to new URI schemas and different interpretations of URI spaces. It isn't possible to satisfy opacity requirements in all cases: URIs that include fragment identifiers aren't opaque to the client because the fragment identifier is evaluated on the client. For HTML, the fragment ID is an ID of an element within the HTML object (anchor). For XML, if it is just a word, it is the XML ID of an element in the document.

Putting any kind of method name, action, or process in a URI is typically considered poor style in REST (because resources are objects rather than processes or services, URI parts should be named for nouns rather than verbs). Using a name to identify a process is RPC-like; using a name to identify a processor is REST-like. The difference is important, because a processor can conceivably have its own state. It would seem that this is an academic debate about naming styles and conventions, but it is actually more important than that: at its core, this debate is about what a resource is or can be, and what underlying models and design processes should be, to achieve REST-fullness. Decomposing a problem domain into a set of resource representations with a generic interface isn't the same as decomposing the same domain into a set of objects and type-specific operations on those objects. There is a mismatch between REST and current process of modeling problem domains as system of objects.

Imagine a description of a simple model with two domain entities: mailbox and message. Even when the model is restricted to representation as a hierarchical collection of resources, there are still a number of options to explore (*pavelkulchenko* is the name of mailbox and *567* is the unique number of the message in the mailbox):

/mailbox=pavelkulchenko/message=567
/pavelkulchenko/567
/mailboxes/pavelkulchenko/messages/567
/mail?mailbox=pavelkulchenko;message=567

Even though the choice may not be immediately clear, for a number of reasons that will be covered later, this approach is the favored one:

/mailboxes/<mailboxId>/messages/<msgId>/

Using it, you can model resources. The following part refers to a collection of mailboxes:

/mailboxes/

This part refers to an instance, i.e., a particular mailbox:

> */mailboxes/<mailboxId>/*

This refers to a collection of messages:

> */mailboxes/<mailboxId>/messages/*

This refers to an instance, i.e., a particular message:

> */mailboxes/<mailboxId>/messages/<msgId>/*

Extending into the message themselves, you'd use the following to refer to a fragment of the message:

> */mailboxes/<mailboxId>/messages/<msgId>/#from*

This refers to a set of messages filtered from a collection:

> */mailboxes/<mailboxId>/messages/<msgId>/?after=20020831*

You can continue and include *parts/<partId>* to address MIME parts of the message or include *first/last* instead of *<msgId>* to address the first or last messages in a mailbox.

The reason that the slashes have been instituted as the common universal syntax for a hierarchical boundary is their familiarity. Hierarchical schemes are common, and the relative naming within hierarchical space has many advantages. Relative naming allows small groups of documents that are located closely within a tree to refer to each other without being aware of their absolute position within any absolute tree.

It's easy to say that URIs change because of the lack of forethought (and it's probably true in many cases), but it is important to realize that many properties of URIs are social rather than technical in nature. Creating URIs that can be used in one year or ten from now requires thought, organization, and commitment on the part of authorities assigning URIs.

There is nothing about HTTP that makes URIs unstable. URIs change when there is some information in them that changes—authors' names, status, corporate structure, database design, and project name are all things that can change over the course of time and thus change a URI. We're in the bind that we need information in the URI to access the resource it identifies, but the very act of putting information in the URI makes it fragile. The only way out of the bind is to include the minimum information that is enough to identify a resource.

It is important to emphasize that a URI points to a resource as a concept, rather than a document. Don't expose the mechanism of how a server runs (`cgi-bin`, `servlet`, or `mod_perl` URIs). Make a change to the mechanism (even though the content stays the same), and all the URIs change. It's highly unlikely that, if you follow these principles, you'll come up with a URL such as *http://www.foo.org/cgi-bin/articles.pl?newsid=1234567*. Instead, it is more likely that if you follow REST principles you'll get something more like *http://www.foo.org/articles/1234567*.

This much space has been devoted to URIs because it's extremely important to properly understand and implement resource names.

Methods

The HTTP specification provides a number of generic methods (the HEAD, GET, and POST methods were discussed in Chapter 2), but only five are relevant to this discussion: GET, HEAD, POST, PUT, and DELETE. Table 11-3 recalls the operations from Chapter 2 and defines the new ones.

Table 11-3. REST methods

Method	Meaning
GET	Retrieves a resource
HEAD	Retrieves representation and resource metadata
POST	Inserts, updates, or extends a resource; may change the state of other resources
PUT	Creates, updates, or replaces a resource
DELETE	Deletes a resource

Unifying the method vocabulary provides tremendous opportunities for simplifying interactions. It is precisely because HTTP has few methods that HTTP clients and servers can grow and be extended independently without confusing each other. Essentially, what are needed are methods that correspond to the "CRUD" concept: Create, Retrieve, Update, and Delete. In HTTP they are called GET, POST, DELETE, and PUT.

GET method

To get a representation from a resource, a client uses the HTTP method GET. This operation is *idempotent*, which means that a client may use the result of a previous operation instead of repeating it; the state of the server shouldn't be changed in ways that are visible to the client. GET is restricted to a single URL line, which enforces one of the design principles: everything interesting should be URL-addressable. Wanting to create a system in which resources aren't URL-addressable means needing to justify that decision.

POST method

Modifying a resource uses the POST operation. This meaning of POST can be ambiguous. It can mean append, create, remove, or modify a portion of resource, or something else entirely. This ambiguity leads to a number of misuses. For example, in violation of one of the REST principles, it can be used for data queries without side effects.

A web-based address book from a well-known vendor illustrates this point. Users can submit the query with an arbitrary string using the POST method and submit the query with a predefined unique identifier using the GET method. If a client wants to link to the address book using the name of a given person, it isn't directly possible (a client can use a scripting language to submit the form, but this isn't always feasible). This is an example of design that violates REST Principle 5: any request that doesn't have side effects should use GET.

Here's a good test of REST-fulness: can an application do a GET on the same URLs it POSTs to, and if so, does it get something that in some way represents the state of what it has been building with the POST operations?

DELETE method

To delete a resource, use the DELETE method on a valid URL.*

PUT method

To change the representation of a resource, a client uses the PUT method. There is a subtle different between the POST and PUT methods. To distinguish between the two, consider a simple example that will be described in greater detail later in this chapter. To get an image of a cover page, a technical editor may use a service that creates the image from a template based on a number of parameters (such as title, cover, or animal); whereas a graphical designer may create the same image in a program such as Photoshop and submit it.

The editor uses POST to submit parameters; the server application modifies a resource based on the submitted parameters and returns the URL of the modified resource (which might be different from the URL the request was submitted to). The designer uses PUT to submit the image; the server modifies the representation. Both can update a resource or create a resource if there is none, but the difference is that PUT (and DELETE) operates on a representation as a whole, where POST submits information that causes a server to modify, create, or delete the resource (or multiple resources), often with a URL that is different from that submitted.

A PUT request may also include a Content-Range header to request modification of only a portion of the entity. It may also include the If-Match and If-None-Match headers to indicate which various entity versions to modify. Thus, including a header, If-None-Match: *, in a PUT request allows a new entity to be created only if it doesn't already exist.

* You may note that the DELETE method is supposed to be used to delete the resource. However, the POST method can delete resources too. Consider a situation in which a mailbox has to be deleted along with all messages. That operation would be executed using POST, rather than DELETE, because more than one resource is involved. At the same time, it is possible to DELETE a resource that has multiple representations. For instance, a DELETE /image operation may delete GIF, PNG, and JPG representations of the resource.

Security

REST greatly simplifies security and benefits it in a sociological manner. Where RPC protocols try as hard as possible to make the network look as if it isn't there (such as the `SOAP::Lite` module's support of the `autodispatch` mode, which tries to make remote calls look local), REST requires that the software developer design a network interface in terms of methods and resources. Whereas RPC interfaces encourage clients to view incoming messages as method parameters to be passed directly and automatically to programs, REST requires a certain disconnect between the interface (which is REST-oriented) and the implementation (which is usually object-oriented).

Resource-centric web services are inherently firewall-friendly: only four main operations are permitted (not counting `HEAD`). Server configurations can apply the four basic permissions to each data object (`GET`, `POST`, `PUT`, and `DELETE`), and they mean what they say: `GET` means get, and `DELETE` means delete. Many REST proponents believe that it will be impossible to securely and widely deploy a protocol across administrative boundaries without knowing the precise meaning of the methods. Using the hierarchical nature of the resource URI, servers may allow or disallow specific operations on subresources.

HTTP authentication and authorization are topics that most web developers are already familiar with. It's easy to assume that a service can't hide on the Internet, but the sole fact of making a resource addressable doesn't make it accessible. For instance, servers can create cryptographically unguessable URIs or associate unguessable signatures with each URI to insure that the resource gets accessed only from a legitimate source. Unguessable URIs are essentially a form of security mechanism known as a *capability*. Capability security is simple, but it requires software and end-users to adhere to a discipline that sharing a URI is equivalent to sharing the resource (a widely used service that employs this method is QuickTopic, available at *http://quicktopic.com/*).

Programming REST

You may notice the absence of code examples in this section. Why doesn't the REST programming section include any code? It is largely because REST is more about a mindset rather than code, more about design than implementation.

It may look simple (because it is), but there are some important points to keep in mind while developing applications in the REST style. This section describes them briefly and links them back to the REST principles (see Table 11-2). The section that follows this scrutinizes some of the points and provides code examples.

Think about the business problem in terms of resource manipulation rather than API design. Enabling web services requires making data available for use by applications without knowing ahead of time exactly how that data will be used. Start by modeling the persistent resources that is to be exposed. Follow the DRY (Don't Repeat Yourself) principle: every piece of knowledge must have a single, unambiguous, authoritative identity within a system (yet may have multiple representations) [Principles 1, 2, and 8]. The key is to identify all the conceptual entities that the system exposes as resources and to assign a unique URL to each of them. Be careful with names even if you plan to use them only as a temporary resource; an old saying goes: nothing is more permanent than "temporary."

Sort publicly exposed resources into those that are immutable by the client (retrieved by the `GET` method), and those that are mutable (modified by the `PUT`, `POST` or `DELETE` methods) [Principles 4 and 5]. Develop a habit of `PUT`ing and `DELETE`ing resources when appropriate. Implement methods that will allow both sender and receiver to make the absolute minimum of assumptions as to the other's state [Principle 6]. Multiple requests should not be required to implement a single logical operation.

Document the format that the application expects, accepts, and returns [Principle 7]. Provide an XML representation for each resource when possible [Principle 8]. Specify the representational schema of both mutable and immutable resources with a formal mechanism (for example, XML Schema, DTD, Schematron, or RelaxNG). Describe and document how the resources can be accessed [Principle 10].

Use a "gradual unfolding methodology" to expose data for clients. Include links to other related resources in (almost) every representation to enable clients to drill down for more information [Principle 9]. Otherwise, the retrieved representation (while valuable by itself) will be the end of story (the decision about the next step to take will have to be made out of hand).

Know and use the software. Set up a server to do content negotiation, authentication, authorization, compression, caching, vacuuming, and house cleaning [Principles 11 and 12].

Invest in abstractions today and in implementations tomorrow. Abstractions can survive the storm of changes from different implementations and new technologies. Realize that things believed in strongly today will get abstracted away in the future.

REST Primer

The REST architectural style is best explained with an example. Therefore, the rest of this section will present a hypothetical example that depicts O'Reilly deploying services using the REST architectural style.

O'Reilly plans to deploy several web services to enable its customers to:

- Get a list of books
- Get detailed information about a particular book
- Get detailed information about authors

- Submit a purchase order
- Update or cancel a purchase order

O'Reilly also wants to enable their authors to:

- Add and update author biographical information

For internal purposes, O'Reilly also wants to enable their editors and designers to:

- Create or update a cover for the book

The next step is to consider how these services can be implemented in a REST style.

Modeling Resources

Before you can model resources, you must create a domain model, which describes entities and their relationships and provides the information necessary for resource modelling (see Figure 11-2).

Figure 11-2. Book service domain model

Based on this domain model, it is possible to model the resources and design URIs to access them. The web service makes available a specific URL to represent a books list resource, for example, */books/*.

The absolute URL for this resource may look like *http://www.oreilly.com/books/*, but because the hostname and protocol will not change, there is no need to include them in the examples.

Note that the process the web service uses to generate the response is completely transparent to the client. A list of books may be available as a physical document or may be created by the server side application at run-time. All the client has to do is submit a request, and a document containing the book list is returned.

The XML document that a client receives may look like this:

```
<?xml version="1.0"?>
<Books xmlns="http://schema.oreilly.com/book"
       xmlns:xlink="http://www.w3.org/1999/xlink">
  <Book id="progwebsoap" xlink:href="progwebsoap/">
    <Title>Programming Web Services with SOAP</Title>
    <Authors xlink:href="progwebsoap/authors/"/>
    <ISBN>0596000952</ISBN>
    <ListPrice>34.95</ListPrice>
  </Book>
  <Book id="progxmlrpc" xlink:href="progxmlrpc/">
    <Title>Programming Web Services with XML-RPC</Title>
    <Authors xlink:href="progxmlrpc/authors/"/>
    <ISBN>0596001193</ISBN>
    <ListPrice>34.95</ListPrice>
  </Book>
</Books>
```

There are several important things to notice. XML is only one of the possible representations that a client may receive as a result of content negotiation (which will be discussed later). The example also includes `xlink:href` attributes that point to the resources that contain related information (book and author details in this case). This is a key feature of REST. A client changes its state by accessing the alternative resources addressed by URLs in response document.

An `xlink:href` attribute is described in XLink specification that defines an attribute-based syntax for attaching links to XML documents. The most current version of the XLink specification can be found at *http://www.w3.org/TR/xlink/*.

Note that, even though the server can make these URLs absolute (*/books/progwebsoap/* or even *http://www.oreilly.com/books/progwebsoap/*), for now it is kept relative. The benefit of doing this will be clearer when discussing the “cover page” service. The model can use the ISBN (as Amazon does, for example) instead of using unique book identifier (as O’Reilly does), but it wouldn’t change anything in this design and implementation.

It is important to decide what information to include in this representation and what to keep for subsequent requests. It was decided to present `Title`, `ISBN`, and `ListPrice` elements and provide links to other resources. While it is possible to include all relevant information, it’s not always feasible or practical. For example, book information may include a cover page; author information may include the author’s bio and pictures as well as links to articles. There is no reason to include all this information in the book list, so the line must be drawn somewhere.

At this stage, the server has implemented a service that returns a list of books. Work can now focus on the service that returns specific information about a particular book. This service will be available through the URL */books/progwebsoap/*.

The XML that the client receives (only as one of several representations) may look like this:

```
<?xml version="1.0"?>
<Book id="progwebsoap"
      xmlns="http://schema.oreilly.com/book"
      xmlns:xlink="http://www.w3.org/1999/xlink">
  <Title>Programming Web Services with SOAP</Title>
  <Authors xlink:href="authors/">
    <Author id="jamessnell" xlink:href="authors/jamessnell/">
      <Name>James Snell</Name>
    </Author>
    <Author id="dougtidwell" xlink:href="authors/dougtidwell/">
      <Name>Doug Tidwell</Name>
    </Author>
    <Author id="pavelkulchenko" xlink:href="authors/pavelkulchenko/">
      <Name>Pavel Kulchenko</Name>
    </Author>
  </Authors>
  <ISBN id="isbn">0596000952</ISBN>
  <ListPrice>34.95</ListPrice>
  <CoverPage xlink:href="coverpage"/>
</Book>
```

Again, observe how information in this document is linked to other resources: list of authors, detailed information about each of the authors, and the cover page. It allows the client to drill down to get more detailed information.

Now look at what happens when a client follows a link. Relative URLs that are being used have only a locally different part of the URI in them (*authors/pavelkulchenko/* in this example). The application that follows the link combines it with the absolute URI of the resource that the application has remembered (perhaps something like *http://www.oreilly.com/books/progwebsoap/*) and generates a new absolute URL of the resource, *http://www.oreilly.com/books/progwebsoap/authors/pavelkulchenko/*.

In response to this request, a client may receive this document:

```
<?xml version="1.0"?>
<Author id="pavelkulchenko"
        xmlns="http://schema.oreilly.com/book"
        xmlns:xlink="http://www.w3.org/1999/xlink">
  <Name>Pavel Kulchenko</Name>
  <Bio xlink:href="bio"/>
  <Books xlink:href="books/"/>
  <Articles xlink:href="articles/"/>
</Author>
```

Here's a good place to step back and take a look at what has been done thus far. The design has modeled four resources and created four services (and the list of authors for a particular book was the bonus service; it wasn't in the initial list). These are shown in Table 11-4.

Table 11-4. The resources and services thus far

Resource	Service
/books/	A list of books
/books/progwebsoap/	Information about a particular book
/books/progwebsoap/authors/	A list of authors for a particular book
/books/progwebsoap/authors/pavelkulchenko/	Information about a particular author

Now it is easy to model other URLs, shown in Table 11-5.

Table 11-5. Other resources using the model

Resource	Service
/authors/	List of authors
/authors/pavelkulchenko/	Information about a particular author
/authors/pavelkulchenko/books/	Books by a particular author
/authors/pavelkulchenko/articles/	Articles by a particular author
/authors/pavelkulchenko/bio	The author's bio

The */authors/* resource represents the list of all O'Reilly authors, while the resource */authors/pavelkulchenko/* represents information about a particular author (which is the same as */books/progwebsoap/authors/pavelkulchenko/*). The resource represented by the sequence, */authors/pavelkulchenko/books/*, represents the list of books written by a particular author and in this case, */authors/pavelkulchenko/books/progwebsoap/* links to the same resource as */books/progwebsoap/*.

Note that structure of the URI is irrelevant to client software: it always operates on a URI as a whole. Even when relative URIs are used, they are processed according to the algorithm described in the URI specification, without an application knowing anything about their internal structure.

Creating Multiple Representations

Only one representation has been dealt with for now (XML), but in a real world environment, it will definitely be a good idea to provide a broad range of representations to satisfy the needs of different clients. Imagine that the service makes the */books/* resource available in RSS format; a user can then subscribe (using one of the many available RSS readers) to that feed and see when new books come out. In a similar fashion, a RSS representation for the */books/progwebsoap/* resource may return news about a book, and for */authors/*, may return information about new authors. Following that line of reasoning, a representation for */authors/pavelkulchenko/* may return a news feed about the particular author (books, articles and everything related). Some of these resources may even be requested in PDF format.

All this variety of formats and representations brings up an interesting question: how does the server know what format to return for a particular request?

It was easier in the good old days. Most available resources were documents, so there was no need for a server to choose what format to use (most resources had only one representation). Even in the case of Common Gateway Interface (CGI) scripts, it was the rare situation when one resource had multiple formats, and yet it was mostly the client's responsibility to figure out what to do with the response based on the `Content-Type` or other representation metadata. As a result, browsers (which represented the vast majority of client software) became greedy: "give me everything; I am smart enough to figure out how to deal with it." In many cases that was true, but now servers want to accommodate nonbrowser clients and keep the compatibility with browser-based clients when possible.

There are three primary ways to convey desired format information:

- File extensions ("`GET /image.gif`")
- Query string ("`GET /image?type=gif`")
- "Accept" header ("`GET /image`" with "`Accept: image/gif`")

File extensions are bad for several reasons: they circumvent the content negotiation, they trick client software into thinking that a file extension has something to do with the content type, and they don't work with resources that are collections of other resources (A */books.rss/* resource would be clumsy at best). Imagine a server using the URL */images/camel.gif* and needing to upgrade it to */images/camel.png* for some reason. This is a problem, because of the need to update all links that point to that resource (note that only the representation of the resource has changed; the meaning of it stays the same). Using */images/camel* as a link, the client and server can do a content negotiation based on values of `Accept`, `Accept-Encoding`, `Accept-Language`, and `UserAgent` headers, customer preferences, or even legal obligations, without affecting other resources or client software. For example, a PDF version of a purchase order can be returned in different languages based on client preferences. However, file extensions aren't easy to avoid: many resources mirror local filesystems, and extensions are used on the server side to provide a hint to the web server about the content type it should return.

Query strings are only slightly better and still share the main weakness: a server always returns the format a client asks for without leaving a room for negotiation.

A more appropriate role for the query string is to carry additional information that may change the representation of a resource or collection of resources without changing semantics. For instance, it can filter or sort entities in a collection:

```
.../books/?number=10;orderby=date
.../books/?status=outofprint
```

The Accept header solution is better for a number of reasons. It allows a generic resource to exist and support multiple dimensions (formats, languages, client requirements, preferences, etc.) that may result in many different representations. It releases the technology from the constraint of being tied to a particular data format and allows independent evolution on client and server sides: a server may add a new format without a client even knowing it, and a client may request specific formats from a server, allowing resolution with minimum round trips.

The major drawback of this approach is that the client application has to know not only about URLs to get a proper format, but also about headers; there is client software in use that doesn't have this capability. Another issue with this approach is that it doesn't work well for different formats that share the same content type (which might be important for XML-based documents). Although it may be possible to specify this information in parameters on the Accept header, this approach isn't standardized.

Doing this also means designing and developing server software in a way that will allow content negotiation and make it aware of content negotiation headers and related responsibilities. If done right, a client can request PDF representation using a `Accept: application/pdf` header, XML representation using `Accept: text/xml, application/xml`, and SOAP representation using `Accept: application/soap+xml`. Multiple preferences and different weights can be assigned.

Developing REST Applications

Returning to the client side, Example 11-1 is a sample application from the client's perspective. It traverses the list of books searching for a particular book, gets the list of authors, and then gets the resource that describes an individual author. Finally, it prints the bio for each author.

Example 11-1. A REST-style client using SOAP::Lite deserialization

```
#!perl -w

use strict;

my $startURL = 'http://www.soaplite.com/oreilly/books/';

# get list of O'Reilly books
foreach my $book (parse(get($startURL))->Book) {

  # look for Programming web services with SOAP
  next unless $book->ISBN eq '0596000952';
  print $book->Title, "\n";

  # get link to Authors for the book
  my $authorsURL = absurl(href($book->Authors), $startURL);
  foreach my $author (parse(get($authorsURL))->Author) {
    print $author->Name, "\n";
```

Example 11-1. A REST-style client using SOAP::Lite deserialization (continued)

```
    # get link to particular Author
    my $authorURL = absurl(href($author), $authorsURL);
    my $bio = parse(get($authorURL))->Bio;

    # get link to Author's bio
    my $bioURL = absurl(href($bio), $authorURL);
    print parse(get($bioURL)), "\n";
  }
}

sub href {
  return shift->attr->{'{http://www.w3.org/1999/xlink}href'};
}

sub get {
  use LWP::UserAgent;
  my $url = shift;
  my $req = HTTP::Request->new(GET => $url, HTTP::Headers->new);
  $req->header(Accept => 'text/xml');
  my $res = LWP::UserAgent->new->request($req);
  die $res->status_line unless $res->is_success;
  return $res->content;
}

sub parse {
  use SOAP::Lite;
  return SOAP::Custom::XML::Deserializer->deserialize(shift)->root;
}

sub absurl {
  use URI;
  my($url, $baseurl) = @_;
  return $url unless $baseurl;
  URI->new_abs($url, $baseurl)->as_string;
}
```

There is one little detail that is worth mentioning: when modeling the URI for a resource that points to a collection of other resources (such as */books/* or */authors/*), be careful to end the URI with a slash character (/). It serves as a visual aid for users and developers (resources that point to collections have a slash at the end, and resources that point to instance documents don't). More importantly, the algorithm that converts a relative URL into an absolute one is written in such a way that it produces */books/authors/jamessnell/* from the pair */books/progwebsoap* (no trailing slash) and *authors/jamessnell/*. The expected result from this production would be */books/progwebsoap/authors/jamessnell*. Using a final slash (as in */books/progwebsoap/*) brings the expected result.

This code used the custom deserializer that is included with the `SOAP::Lite` package, but applications are free to use any other code for parsing XML documents. For example, you can use the simple code in Example 11-2 (based on the `XML::Parser` module).

Example 11-2. A simple XML parsing using XML::Parser

```
#!/usr/bin/perl -w

use strict;
use XML::Parser;

sub parse {
  XML::Parser->new(
    Namespaces => 1, Handlers => {Start => \&start},
  )->parse(shift);
}

sub start {
  my($xp, $el) = (shift, shift);

  while (@_) {
    my($nm, $v) = (shift, shift);
    my $ns = $xp->namespace($nm) || '';
    print $el, " => ", $v, "\n"
      if $nm eq 'href' &&
         $ns eq 'http://www.w3.org/1999/xlink';
  }
}

use LWP::Simple;

parse(get('http://www.soaplite.com/oreilly/progwebsoap/'));
```

Using this to process the XML document that was returned in the response for accessing the */books/progwebsoap/* resource, yields this output:

```
Authors => authors/
Author => authors/jamessnell/
Author => authors/dougtidwell/
Author => authors/pavelkulchenko/
CoverPage => coverpage
```

Working with POST, PUT, and DELETE Methods

That was probably the easiest part: four resources, one method (`GET`), and a fairly simple XML payload. Let's now extend the services, and model and implement other operations.

The first step is to implement the service that allows authors to add or update their bio information. This example will be using a simple format and only one operation, `PUT`. The choice between the `POST` and `PUT` methods isn't always obviously clear. The rationale here is that the URL of the resource (*/authors/pavelkulchenko/bio*) is already known, no other resource is modified, and operation is on the resource as a whole (though it is possible and perfectly legal to update only part of the resource using a `Content-Range` header).

The dialog between client and server may look like the following. Here's the request:

```
PUT /authors/pavelkulchenko/bio
Content-Type: text/xml
Content-Length: NNN

<Bio xmlns="http://schema.oreilly.com/book">....</Bio>
```

Here's the response:

```
201 Created
```

The client code that posts the bio using `LWP::UserAgent` module is given in Example 11-3.

Example 11-3. A REST-style PUT request using LWP::UserAgent

```
use LWP::UserAgent;

  my $url = 'http://not-real-url.oreilly.com/authors/pavelkulchenko/bio';
  my $bio = '<Bio xmlns="http://schema.oreilly.com/book">....</Bio>';
  my $req = HTTP::Request->new(PUT => $url, HTTP::Headers->new, $bio);
  $req->content_type('text/xml');
  $req->content_length(length $bio);

  my $res = LWP::UserAgent->new->request($req);

  print $res->status_line;
```

What if the design is extended to accept plain text messages and convert them to XML? It is easy to change the content type from *text/xml* to *text/plain* and submit the text version of a bio, but can a client still use the `PUT` method? For purists, the answer is probably "no," because the `PUT` method would be used for a transformation of the representation. If transformations are allowed, it isn't clear what to do with requests that modify only part of the resource. For the practical implementation the answer is probably "yes," because requiring `POST` in this case means that the operation on the resource depends on the implementation and violates one of the REST principles: details on how to store information on the server side must be irrelevant to the client.

Implementing a Purchase Order Service

The next example is more complex. It allows customers to submit, update, or cancel purchase orders. The design uses */books/orders/* as the resource for purchase order submission. This URL works as a factory of orders, creating URLs for particular orders. Again, there are a number of choices:

- POST the purchase order to that resource and get back a URI for the order (when accepted).
- Send a GET request asking for an order number, then PUT the order there.

Two disadvantages of using the POST method

POST has a reliability problem if a client loses its contact with the server after POSTing a purchase order. The order might have been accepted, and the reply was lost, but then again the order might have been lost and should be resent. There's no way for the client to know what happened. POST can also be confusing here; a client could use POST to submit an initial order, but then PUT to update it.

Two disadvantages of the GET and PUT combination

The first disadvantage of using the GET method is that the */orders/* resource is already taken. The system may want it to do something different, for instance to return a list of orders for particular customer. Because of this, a different resource (for example, */orders/new*) is needed that will respond to GET requests and return a URL in which the order can be submitted (the only real disadvantage is that two roundtrips are required). The second disadvantage is that now the server has a potential race condition to address (if it doesn't keep the generated order numbers on the server side, it can generate the same order number for more than one customer), and the PUT request has to be smart enough to deal with that.

Choosing the implementation

Because the reliability problem seems to be the more interesting issue to address, the example will implement the GET and PUT combination.* For now, the actual XML documents for purchase orders will be skipped over (you can design these as an exercise), and the text will describe the interactions only to highlight the important aspects.

First, a client receives the information about the resource where the URL and number for a new order can be obtained (*/books/progwebsoap/orders/new* in this example). This information may be included as part of the book list (the */books/ representation*), or it may be published somewhere.

Following this, the client sends a GET request to the URL and receives back a new URL where the order should be submitted.

Here's the request:

```
GET /books/progwebsoap/orders/new
```

* Though the network may go down during the PUT request, it is safe for a client to resubmit the message when in doubt, because an order with the same number will be treated as the same order, whereas resending the order with the POST message may result in duplicate orders.

Here's the response:

```
200 OK
Location: http://www.oreilly.com/books/orders/987654321

<Order xmlns="http://schema.oreilly.com/book"
       xmlns:xlink="http://www.w3.org/1999/xlink"
       id="987654321" xlink:href="../987654321"/>
```

Finally, the client uses the new URL (from the `Location` header or the `href` attribute of the `Order` element) to submit the order using the `PUT` method. The client will also include in the request the header `If-None-Match: *`, to indicate that it wants the order to be accepted only if it doesn't already exist. The server returns a fault if someone has already submitted an order with that number.

Here's the request:

```
PUT /books/orders/987654321
Content-Type: text/xml
Content-Length: NNN
If-None-Match: *

<Order xmlns="http://schema.oreilly.com/book">....</Order>
```

The response for this request will be `201 Created` or `412 Precondition Failed`, depending on the existence of the resource.

For subsequent requests that update the resource, the client may want to include the `If-Match` header instead of `If-None-Match`, to require the presence of the resource:

```
PUT /books/orders/987654321
Content-Type: text/xml
Content-Length: NNN
If-Match: *

<Order xmlns="http://schema.oreilly.com/book">....</Order>
```

The `DELETE` request (which will mean the cancellation of the order) can be implemented in a similar fashion. Similarly, the `GET` request for this resource (*/books/orders/987654321*) may return different representations (XML, HTML, or PDF) in keeping with the role of `GET` in this kind of system.

Implementing Cover Page Service

The last example briefly describes the design that implements the cover page service. The reason for including it is that the resource in this example accepts both `POST` and `PUT` requests and shows the important difference between them. First of all, it makes sense to allow updates to a cover page only when the book is in the preproduction stage. Starting with a decision to put all related resources under a URL */preproduction/*, the server will also restrict access to it. From this, the resource */preproduction/books/* now returns the list of all books in preproduction status. Because the system is using relative URLs, there is no need to change the way XML documents are created. The service allows two different requests because the cover page can be generated in two different ways: by the technical editor, providing color, subheader, animal, color and title and using the `POST` method to submit this information; and by a graphical designer, providing the image of the cover page using the `PUT` method to submit it. A `GET` method to that resource may return PNG, GIF, PDF, or some other representation depending on request properties.

Documenting Service API

Making services available may not be enough to make them popular unless the API is documented with a widely understood format. Two specifications that look promising for documenting the APIs of REST services (besides plain text or HTML formats) are Web Services Description Language (WSDL) and Web Resource Description Language (WRDL).

Here's a quote from the WRDL specification (which can be found at *http://www.prescod.net/rest/wrdl/wrdl.html*):

> REST is defined by four interface constraints: identification of resources, manipulation of resources through representations, self-descriptive messages, and hypermedia as the engine of application state.
>
> These four things are addressed in this specification through:
>
> - the association of URIs with resourceTypes
> - the association of resourceTypes with representationTypes
> - the recognition of representationTypes through XPaths
> - the built-in support for navigation through hyperlinks

Still, there's no instant winner: the WSDL specification isn't well suited to describe REST services (e.g., even though both `QUERY STRING` and `PATH` can be parameterized (see the section "HTTP and MIME Binding in WSDL" in Chapter 9 for more information, specifically Example 9-10), it isn't possible using the current specification to specify the value of the HTTP headers to select the representation), and the WRDL specification is new and not endorsed by any standards body (which may be seen as advantage, depending on your point of view).

What Makes REST Hard

REST is simple in details and (mostly) easy to understand, but it can be hard to execute. Despite its advantages, HTTP-based, URI-centric resource modeling isn't the usual way you think about networking issues, and REST-based web services aren't very common. Applying REST style requires you to resolve problems by manipulating addressable resources instead of method calls to components. The APIs communicated to clients should be in terms of HTTP manipulations on XML documents addressed by URIs instead of familiar method calls with parameters. REST is about imposing a programming discipline of many URIs and few methods. Here are several things that have been found difficult about REST:

It's flexible
: This one of its main strengths *and* weaknesses. Its flexibility allows resources to be used in ways that can't be envisioned now, at little additional cost. Even though REST consists of a set of architectural constraints, there is still enough space for abuse. It's generally good to make choices that preclude the least number of future alternatives.

It's hard to come up with the right URI
: The significance of identity for a given URI is determined by who owns the URI. The Web relies on the author choosing a resource identifier that best fits the nature of the concept being identified.

It's hard to refactor
: As soon as a service publishes its resource, it can't take it back without breaking client applications. It can't rely on a client to follow the sequence of links before coming to the resource. One of the things that can be done is to make client applications take note of redirects.

It's hard to come up with the right fault handling model
: The currently available fault handling model, while robust and simple, may not be suitable for all applications. For those occasions where it isn't, a designed fault handling system will either duplicate or circumvent one available in the protocol.

It's hard to make and keep developers aware of the architectural style
: Currently available tools make it possible to do The Wrong Thing as easily as The Right Thing. Designing for REST requires an abstraction skill and mindset different from those developed practicing object-oriented analysis and design. Learning and practicing the REST style can allow designers and developers to do The Right Thing more often. The full value of REST will not come until all the practices are in place.

It's hard to get the full tool support when developing application in the REST style
: Any REST proponent will point out that REST has the most widely deployed and useful tool support, which is absolutely true; however, it is important to realize that these tools support the current usage patterns, which may not be exactly of the same style as that which has been described. Take this URI:

    /books/progwebsoap#isbn

    The significance of the fragment identifier is a function of the MIME type of the object, and application behavior depends on the MIME types of returned representation. Imagine that the representation is XML, and *isbn* is an XML ID of the ISBN element in that document (or maybe even written in full notation as `xpointer(id("isbn"))`). After many years, there is still very little code to properly deal with fragments like these in resource identifiers.

REST and SOAP

Lengthy and passionate discussion between REST and SOAP proponents resulted in the WebMethod feature being added to the current draft (Version 1.2) of the SOAP specification. The specification can be found at:

http://www.w3.org/TR/soap12-part2/#webmethodstatemachine

Here's a quote from the specification:

> Applications SHOULD use `GET` as the value of `webmeth:Method` in conjunction with the §6.3 SOAP Response Message Exchange Pattern to support information retrievals which are safe, and for which no parameters other than a URI are required: i.e., when performing retrievals which are idempotent, known to be free of side effects, for which no SOAP request headers are required, and for which security considerations don't conflict with the possibility that cached results would be used.

In addition to that, the SOAP 1.2 Working Group has proposed a new SOAP media type (`application/soap+xml`) that allows SOAP processors to accept `GET` requests (`POST` was the only method previously allowed) and to return a SOAP message as the resource representation in response to a `GET` request. That was allowed from the REST perspective earlier; it was just standardized from the SOAP perspective, so that a larger number of clients can access SOAP endpoints.

REST is based on resources, a limited but diverse set of operations, and a well-defined document format. Both camps agree that much of the value of provided services is in the precise description of the structure expected in requests and responses. Given the document/literal encoding and combination of the described WebMethod feature and Message Exchange Pattern defined in the SOAP 1.2 specification, SOAP and REST web services can be much closer to each other than the heated debates make them seem.

We'll leave comparing SOAP and REST interfaces to you. There are only a handful of services available through REST and SOAP/XML-RPC interfaces; Amazon (SOAP and REST), Google (SOAP and REST, though the REST interface that returns XML isn't publicly available), and Meerkat (XML-RPC and REST) are among the best known.

Whether working on world-facing web services now or in two years, the biggest challenge will be not in describing the interfaces or deciding what method to run, but in aligning business documents and processes with those of others. The technology used to move data from place to place isn't important, but the business-specific document and process modeling is (fortunately, that's the part where SOAP and REST proponents seem to agree). Given the proper format (for instance, RDF), a client application developed REST-style may be able to dynamically learn the meaning of the representation. Given links to the next state built within each response, a client application may be able to deduce dynamically which link to traverse. Dynamic learning of the representation combined with dynamic determination of link traversals would create an adaptive engine. Does this sound like the next generation of the web services?