

---

# Build a RESTful Web service

## An introduction to REST and the Restlet framework

Skill Level: Intermediate

[Andrew Glover \(aglover@stelligent.com\)](mailto:aglover@stelligent.com)

President

Stelligent Incorporated

22 Jul 2008

Representational state transfer (REST) is a style of designing loosely coupled applications that rely on named resources rather than messages. The hardest part of building a RESTful application is deciding on the resources you want to expose. Once you've done that, using the open source Restlet framework makes building RESTful Web services a snap. This tutorial guides you step-by-step through the fundamental concepts of REST and building applications with Restlets.

## Section 1. Before you start

### About this tutorial

REST is a way of thinking, not a protocol or standard. It's a style of designing loosely coupled applications — more often than not, applications oriented for the Web — that rely on named resources rather than messages. In this tutorial, you'll get to know what REST is and how to build RESTful applications with Restlets, a lightweight REST framework for Java™ applications.

### Objectives

This tutorial guides you step-by-step through the fundamental concepts of REST and

building applications with Restlets. You'll learn how to:

- Define RESTful Web services
- Implement them with the Restlet framework
- Verify them with the JUnit testing framework

When you are done with the tutorial, you will understand the benefits of designing with RESTful principles, and you'll see how the Restlet framework makes it easy.

## Prerequisites

To get the most from this tutorial, you should be familiar with Java syntax and the basic concepts of object-oriented development on the Java platform. You should also be familiar with Web applications. Familiarity with Groovy, JUnit, DbUnit, and XMLUnit is also helpful.

## System requirements

To follow along and try out the code for this tutorial, you need a working installation of either:

- [Sun's JDK 1.5.0\\_09](#) (or later)
- [IBM Developer Kit for Java technology 1.5.0 SR3](#)
- [Apache Ant 1.7 or greater](#)

There are two versions of the source code for this tutorial (see [Download](#)). One version includes all code and required dependencies (the Restlet framework, JUnit, XMLUnit, and DbUnit). Readers with a low-bandwidth connection might prefer to download the Restlet framework, JUnit, XMLUnit, and DbUnit from their respective sites (see [Resources](#)) and use the [Download](#) package version that does not include dependencies.

The recommended system configuration for this tutorial is:

- A system supporting either the Sun JDK 1.5.0\_09 (or later) or the IBM JDK 1.5.0 SR3 with at least 500MB of main memory
- At least 20MB of disk space to install the software components and examples covered

The instructions and examples in the tutorial are based on a Microsoft® Windows®

operating system. All the tools covered in the tutorial also work on Linux® and UNIX® systems.

---

## Section 2. What is REST?

REST is a style of designing loosely coupled Web applications that rely on named resources — in the form of Uniform Resource Locators (URLs), Uniform Resource Identifiers (URIs), and Uniform Resource Names (URNs), for instance — rather than messages. Ingeniously, REST piggybacks on the already validated and successful infrastructure of the Web — HTTP. That is, REST leverages aspects of the HTTP protocol such as `GET` and `POST` requests. These requests map quite nicely to standard business-application needs such as create read, update, and delete (CRUD), as shown in Table 1:

**Table 1. CRUD/HTTP mapping**

Application task	HTTP command
Create	POST
Read	GET
Update	PUT
Delete	DELETE

By associating requests, which act like verbs, with resources, which act like nouns, you end up with a logical expression of behavior — `GET` this document and `DELETE` that record, for example.

Roy Fielding, the veritable father of REST, states in his PhD dissertation that REST "emphasizes scalability of component interactions, generality of interfaces, independent deployment of components, and intermediary components to reduce interaction latency, enforce security, and encapsulate legacy systems" (see [Resources](#)). Building RESTful systems isn't difficult, and the systems are highly scalable while also being loosely coupled to the underlying data; they also leverage caching quite nicely.

Everything on the Web (pages, images, and so on) is in essence a resource. REST's reliance on named resources rather than messages facilitates loose coupling in application design, because it limits the exposure of the underlying technology. For instance, the following URL exposes a resource without implying anything about the underlying technology:

<http://thediscoblog.com/2008/03/20/unambiguously-analyzing-metrics/>

This URL represents a resource — an article called "Unambiguously analyzing metrics." A request for this resource leverages the HTTP `GET` command. Notice that the URL is noun-based. A verb-based version (which might look something like `http://thediscoblog.com/2008/03/20/getArticle?name=unambiguously-analyzing-metrics`) would violate REST principles, because it embeds a message, in the form of `getArticle`. You could also imagine posting a new resource (say, an article resource such as `http://thediscoblog.com/2008/03/22/rest-is-good-for-you/`) via HTTP's `POST` command. Although you can also imagine associated, verb-based APIs — such as `createArticle?name=rest-is-good-for-you` and `deleteArticle?name=rest-is-good-for-you` — such calls hijack the HTTP `GET` command and, for the most part, ignore the already available (and successful) HTTP infrastructure. In other words, they are not RESTful.

The beauty of REST is that resources can be anything, and how they are represented can vary too. In the preceding example, the resource is an HTML file; accordingly, the format of the response would be HTML. But the resource could have easily been an XML document, serialized object, or JSON representation. It really doesn't matter. What matters is that a resource is named and that communication with it doesn't affect its state. Not affecting state is important because stateless interactions facilitate scalability.

## Why should you care?

To quote Leonardo da Vinci, "simplicity is the ultimate sophistication." The implementation of the World Wide Web is as simple as it gets and an undeniable success. REST leverages the Web's simplicity and thus yields highly scalable, loosely coupled systems that, as it turns out, are simple to build.

As you'll see, the hardest part of building a RESTful application is deciding on the resources you want to expose. Once you've done that, using the Restlet framework makes building RESTful Web services a snap.

---

## Section 3. Off to the races: Building a RESTful API

In this section, you'll build a RESTful API for a Web service that leverages the functionality of an existing database-backed application.

### RESTful races

Imagine an online application that manages races in which contestants run various distances (such as the Chicago Marathon). The application manages races (or events) and the runners associated with them. And it reports a particular runner's time (how long it took to run the race) and rank (what place the runner finished in). The race-management company, Acme Racing, wants you to build a RESTful Web service that enables sponsors to create new races and racers for a particular race, and that can provide official results for a particular race.

Acme Racing already has a legacy fat-client application that supports similar requirements and leverages a simple database along with a domain model. Thus, the job of exposing this functionality is all that's left to do. Remember that the beauty of REST is its implicit loose coupling with an underlying application. Accordingly, your job, at the moment, isn't to worry about the data model or the technology associated with it — it's to construct a RESTful API that supports the company's requirements.

## Race URIs

Acme Races would like sponsors to be able to:

- View the details of existing races
- Create new races
- Update existing races
- Delete races

Because REST boils down to named resources, the API becomes a series of URI patterns, and the behavior associated with a resource is invoked via standard HTTP commands.

As you can see, the client's requirements map nicely to CRUD. And as you know from [Table 1](#), REST supports CRUD via the HTTP `POST`, `GET`, `PUT`, and `DELETE` requests, respectively. Accordingly, a base RESTful URI that supports these requirements could be `http://racing.acme.com/race`. Note that in this case, `race` is the resource clients would work with.

Invoking this URI with an HTTP `GET` would return a list of races. (Don't worry about the format of the response just yet.) To add a new race, you would invoke the same URI with an HTTP `POST` containing the appropriate information (for instance, an XML document containing required race information, such as name, date, and distance).

For updating and deleting existing races, you would need to act on a particular instance of a race. Accordingly, individual races can be addressed with a URI of

`http://racing.acme.com/race/race_id`. In this case, *race\_id* represents a placeholder for any race identifier (such as 1 or 600meter). Consequently, viewing an existing race instance would be an HTTP `GET` to that URI; updating or deleting a race would be a `PUT` or `DELETE` request, respectively.

Acme Racing also wishes to expose data regarding runners associated with a race. They'd like their service to support:

- Obtaining all runners for a particular race. This data should also include run times and ranks for a race that's already completed.
- Creating one or more runners for a particular race.
- Updating a runner's information (such as age) for a particular race.
- Deleting a runner for a particular race.

Acme would also like the service to let users view individual data for a particular runner in a particular race.

Just as with races, applying RESTful URIs to runners associated with a race is a logical exercise. Viewing all runners for a particular race, for example, would be implemented via a `GET` request to `http://racing.acme.com/race/race_id/runner`.

Obtaining individual data for a runner in a race would be addressed as `http://racing.acme.com/race/race_id/runner/runner_id`.

Just like *race\_id*, *runner\_id* is a placeholder for the logical implementation of IDs, which could be numbers, names, alphanumeric combinations, and so on.

Adding runners to a race would be a `POST` request to `http://racing.acme.com/race/race_id/runner`. Updating or deleting particular runners would be, respectively, `PUT` and `DELETE` requests to `http://racing.acme.com/race/race_id/runner/runner_id`.

Thus, these URIs (each supporting some or all of the four standard HTTP requests) capture Acme Racing's requirements:

- `/race`
- `/race/race_id`
- `/race/race_id/runner`
- `/race/race_id/runner/runner_id`

Remember, a particular URI can map to more than one HTTP verb (for instance, applying an HTTP `GET` to `/race` returns data; applying a `POST` with appropriate data

creates data on the server). Accordingly, some HTTP commands wouldn't be implemented. For example, `/race` wouldn't support the `DELETE` command (Acme Racing wouldn't want to delete all races); `/race/race_id` could support the `DELETE` command because removing a particular instance of a race is a business requirement.

---

## Section 4. Formatting the resource

In this section, you'll construct a series of XML documents to represent the resources that the RESTful races Web service will support.

### Race URIs

The RESTful API you built in the preceding section for Acme Racing covers the network endpoints or URIs but not the resources. As far as REST is concerned, the format of the resources doesn't matter, as I mentioned earlier. You could pass XML or binary streams back and forth, for example.

XML is arguably the *lingua franca* of machine-to-machine communication in the context of business transactions, so it makes sense to construct a series of XML documents that the RESTful service will support. The domain for racing is fairly simple, and you can use an existing data model, so the task of defining a few XML documents that represent races and runners is straightforward.

For instance, a race can be defined in XML as in Listing 1:

#### Listing 1. XML document for a race

```
<race name="McClean 1/2 Marathon" date="2008-05-12" distance="13.1" id="1">
  <uri>/races/1</uri>
  <description/>
</race>
```

Note that a `<race>` has an `id` and that Listing 1 includes a URI as part of the definition of a race. This is a key aspect of REST and indeed, the Web — resources are related and should be linked together. Accordingly, a `<race>` always contains a `<uri>` element describing its RESTful representation. The XML in Listing 1 is arguably the response of a `GET` request to `/races/1`.

To create a new race, you could omit the `id` aspect (because managing unique IDs is something the application you're building here controls). This implies you could

exclude the `<uri>` element as well. Consequently, a `POST` request would look something like Listing 2:

### Listing 2. Race-creation XML

```
<race name="Limerick 2008 Half" date="2008-05-12" distance="13.4">
  <description>erin go braugh and have a good time!</description>
</race>
```

What about runners? A runner is connected to race, right? So the `<race>` element supports holding one or more `<runner>` elements, as shown in Listing 3:

### Listing 3. Runners associated with a race

```
<race name="Limerick 200 Half" date="2008-05-12" distance="13.4" id="9">
  <uri>aces/9</uri>
  <description>erin go braugh and have a good time!</description>
  <runners>
    <runner first_name="Linda" last_name="Smith" age="25" id="21">
      <uri>/aces/9/runner/21</uri>
    </runner>
    <runner first_name="Andrew" last_name="Glover" age="22" id="20">
      <uri>/aces/9/runner/20</uri>
    </runner>
  </runners>
</race>
```

The XML document in Listing 3, for example, is what would be returned via the URI `/race/race_id/runner`. The API also supports CRUD operations performed on a single runner via the URI `/race/race_id/runner/runner_id`.

Accordingly, the XML for these CRUD actions looks like Listing 4:

### Listing 4. CRUD XML

```
<race name="Mclean 1/2 Marathon" date="2008-05-12" distance="13.1" id="1">
  <uri>/aces1</uri>
  <description />
  <runner first_name="Andrew" last_name="Glover" age="32" id="1">
    <uri>/aces/1/runner/1</uri>
    <result time="100.04" place="45" />
  </runner>
</race>
```

Note that if the race is already complete, a runner's results can be included in the XML document. Remember, using a `POST` request means creating a runner; consequently, the `<runner>` element's `id` attribute would not be present.

## Section 5. Restlets

You've defined a RESTful API that maps quite nicely to CRUDing races and runners. And you've defined the format of the communication: XML documents. In this section, you'll start to put this all together using an innovative framework that is modeled after servlets.

### The Restlet framework

Restlet applications are akin to servlet applications in that they reside in a container, but in practice they are quite different in two major ways. First, Restlets use no direct notion of HTTP or its stateful manifestations such as cookies or sessions, per se. Second, the Restlet framework is extremely lightweight. As you'll see, a fully functional RESTful application can be built with a handful of classes that extend from a few core Restlet base classes. Configuration and deployment leverage existing container models, so you simply update the customary web.xml file and deploy a standard Web archive (WAR) file.

For the most part, the bulk of a RESTful application built with the Restlet framework requires the use of two base classes: `Application` and `Resource`. Logically speaking, an `Application` instance maps URIs to `Resource` instances. `Resource` instances do the work of handling the basic CRUD commands, which are, of course, mapped to GET, POST, PUT, and DELETE.

### The race application

You create a starting point with the Restlet framework by extending from the framework's `Application` class. In this class, you define `Resources` that respond to URIs. This definition process is done with the framework's `Router` class. For example, if you have a URI such as `order/order_id`, you need to specify which object can handle these requests. This object is an instance of the framework's `Resource` type. You link objects with URIs by attaching them to a `Router` instance, as in Listing 5:

#### Listing 5. Creating Router instances and mapping URIs

```
Router router = new Router(this.getContext());
router.attach("order/{order_id}", Order.class);
```

So in this example, the URI `order/order_id` is logically mapped to an `Order` class (which, in turn, extends `Resource`).

Acme Racing has the four logical RESTful URIs that you've already defined — four patterns that work with various aspects of races and runners:

- `/race`
- `/race/race_id`
- `/race/race_id/runner`
- `/race/race_id/runner/runner_id`

The behavior of each URI (such as if it works with `POST`, `DELETE`, `GET`, and so on) isn't important at this point. The behavior of each `Resource` is the job of a `Resource` instance; however, the `Application` instance is used to map these URIs to (yet-to-be-defined) `Resources` via a `Router` instance, as shown in Listing 6:

### Listing 6. Mapping Acme Racing's URIs to Resources

```
public class RaceApplication extends Application{
    public RaceApplication(Context context) {
        super(context);
    }

    public Restlet createRoot() {
        Router router = new Router(this.getContext());
        router.attach("/race", RacesResource.class);
        router.attach("/race/{race_id}", RaceResource.class);
        router.attach("/race/{race_id}/runner", RaceRunnersResource.class);
        router.attach("/race/{race_id}/runner/{runner_id}", RaceRunnerResource.class);
        return router;
    }
}
```

The base class, `Application`, is an abstract class. Extending classes must implement the `createRoot()` method. In this method, you can create a `Router` instance and attach `Resources` to URIs as I've done in Listing 6.

As you can see, there are four different `Resource` classes. I've named them to match the desired high-level behavior of the URI. For instance, the `/race` URI is intended to work with multiple race instances; consequently the `Resource` type is named `RacesResource`. Once an `id` is included in the URI (`/race/race_id`), the implication is that a single race is being manipulated; accordingly, the `Resource` type is apply named `RaceResource`.

## Race resources

Now that you've defined the `Application` instance to handle four different URI patterns, you must implement the four `Resources`.

`Resource` types in the Restlet framework are known as *Restlets*. They are the heart of any RESTful application developed with the Restlet framework. Unlike the `Application` type, the base `Resource` class is not abstract. It's more like a template with default behavior that you can override as needed.

At a high level, `Resource` has four methods that require overriding. Not coincidentally, they map to the basic HTTP commands that are the touchstone of REST — GET, POST, PUT, and DELETE. Because the `Resource` class is nonabstract, the framework requires a paired method to be implemented for desired behavior to be invoked. For instance, if you want a particular resource to respond to DELETE requests, you would first implement the `delete()` method. Second, you also must implement the `allowDelete()` method and have this method return `true` (it defaults to `false`). By default, the corresponding PUT, POST, and DELETE allow methods return `false`, and the `allowGet()` method returns `true`. This means for read-only `Resources`, you need to override only one method (instead of two in the other three cases). You can alternatively call the `setModification(true)` in a `Resource` class and thus not have to override individual HTTP verb allow methods.

For instance, the `RacesResource` is intended to respond to GET requests with an XML document that describes the races in the system. Users can also create new races via this `Resource` type. Therefore, the `RacesResource` class overrides at least three methods from the `Resource` base class:

- `getRepresentation()`
- `allowPost()`
- `post()`

Remember, `Resources` instances, by default, are read-only. Hence the `allowGet()` method doesn't need to be overridden.

---

## Section 6. Generating XML documents

In [Formatting the resource](#), we decided to leverage XML as the data mechanism for sharing information between clients and the service. Your Restlets, therefore, must manipulate XML: build it in the case of a GET and consume it in the case of a POST, PUT, or DELETE. In this section, you'll take the pain out of generating and manipulating XML documents by leveraging the Groovy scripting language (see [Resources](#)).

## Leveraging Groovy

Working with XML is no easy chore. It can be tedious and error prone, to say the least. Fortunately, Groovy makes working with XML much easier.

You'll leverage Groovy's power to generate XML and do the tedious job of manipulating XML documents. Working with XML in Groovy couldn't be any easier. For instance, parsing an XML document is a cakewalk. Take the XML document in Listing 7:

### Listing 7. A simple XML document to parse

```
<acme-races>
  <race name="Alaska 200 below" date="Thu Jan 01" distance="3.2" id="20">
    <uri>/races/20</uri>
    <description>Enjoy the cold!</description>
  </race>
</acme-races>
```

Suppose you wanted to grab the value of the `<race>` element's `name` attribute. All you need to do is pass in an instance of the XML document to Groovy's `XMLSlurper` class, call the `parse()` method, and then navigate to your desired element or attribute, as shown in Listing 8:

### Listing 8. Parsing XML in Groovy

```
def root = new XmlSlurper().parseText(raceXML)
def name = root.race.@name.text()
```

If you want the description, it's as easy as calling `root.race.description.text()`.

Creating XML is easy too. If you wanted to create the XML snippet in Listing 7, all you have to do is create an instance of Groovy's `MarkupBuilder` class and add nodes to it, as in Listing 9:

### Listing 9. Creating XML couldn't be easier

```
def writer = new StringWriter()
def builder = new MarkupBuilder(writer)
builder."acme-races"() {
  race(name: "Alaska 200 below", date: "Thu Jan 01", distance: "3.2", id: "20") {
    uri("/races/20")
    description("Enjoy the cold!")
  }
}
println writer.toString()
```

Note how elements are added to the XML document by attaching names to the `builder` instance. I had to put quotation marks around `acme-races` because hyphens aren't allowed in string literals in Groovy; consequently, making `acme-races` a `String` solves that problem nicely.

Elements can have attributes. Attribute names and values are created by constructing a Groovy map, which links the two together (for example, `name:"Alaska 200 below"`).

---

## Section 7. The data layer

This section describes the existing domain objects that make the up data layer that your RESTful service will reuse.

### Domain objects

As you know from [Off to the races: Building a RESTful API](#), Acme Racing invested in a data layer for a previous project and wants to reuse it for the new Web service. This, of course, makes your job even easier. In a nutshell, the data layer consists of three business objects: `Race`, `Runner`, and `Result`. They are effectively managed by Spring and Hibernate; however, these frameworks are hidden from you; you simply have a JAR file that works nicely (that is, lets you easily create new races, find existing runners, and so on).

The business objects support a series of finder methods that make obtaining race and runner instances quite easy. Objects can be persisted, updated, and removed from the underlying database via `save()`, `update()`, and `remove()` methods, respectively.

For example, a `Race` object supports a series of finder methods and facilitates manipulating persisted data nicely. The `Race` object's API is straightforward, as shown in Listing 10:

#### Listing 10. Race's API

```
Collection<Race> findAll();
Race findById(long id);
Race findByName(String name);
void create(Race race);
void update(Race race);
void remove(Race race);
```

A `Race` instance has a number of properties, as shown in Listing 11:

### Listing 11. Race's properties

```
private long id;
private String name;
private Date date;
private double distance;
private Set<Runner> participants;
private Set<Result> results;
private String description;
```

All of `Race`'s properties are available via getters and setters. What's more, collections of items (such as `participants` and `results`) support adding individual items. Therefore, the `Race` object has an `addParticipant()` method, shown in Listing 12:

### Listing 12. Race's addParticipant() method

```
public void addParticipant(final Runner participant) ;
```

As you've seen, working with this domain model is quite easy.

---

## Section 8. Building and testing the service

Now that you know how you'll work with XML and already have a data layer to use, it's time to continue building your RESTful application with Restlets and do some test preparation.

### The Races service

Recall that Acme Racing would like its service to enable clients to view existing races as well as create new ones. You've already outlined the RESTful URI that'll support this behavior: `/race`.

Via the `Router` class in the `RaceApplication` class, you linked this URI to the `RacesResource` class. You already know that you must implement three methods:

- `getRepresentation()`
- `allowPost()`

- `post()`

Accordingly, create a class called `RacesResource` and ensure that it extends `org.restlet.resource.Resource`. Also, implement a three-parameter constructor, as shown in Listing 13:

### Listing 13. Three-parameter constructor in `RacesResource`

```
public class RacesResource extends Resource {
    public RacesResource(Context context, Request request, Response response) {
        super(context, request, response);
    }
}
```

Restlets must be instructed how to communicate resource representations properly. Because XML will serve as the resources format, you must direct your Restlet by adding an XML variant type. Variants, in Restlets, represent a format for Resources. The base class, `Resource`, contains a `getVariants()` method that facilitates adding various `Variant` types. Accordingly, add the line in Listing 14 to your constructor:

### Listing 14. Signifying XML as a variant

```
this.getVariants().add(new Variant(MediaType.TEXT_XML));
```

The Restlet framework supports a wide variety of media types, including images and video.

## Handling GET requests

Now it's time to implement the easiest behavior of the class: handling a GET request. Override the `getRepresentation()` method as shown in Listing 15:

### Listing 15. Overriding `getRepresentation()`

```
public Representation getRepresentation(Variant variant) {
    return null;
}
```

As you can see, this method returns a `Representation` type, of which there are multiple implementations. One implementation — aptly dubbed `StringRepresentation` — represents strings and will suffice for your needs.

As you know, you already have a legacy domain model that supports working with

the database. It also turns out that someone has already written a utility class, called `RaceReporter`, that transforms domain objects into XML documents. This class's `racestoXml()` method takes a collection of `Race` instances and returns a `String` representing an XML document looking something like Listing 16:

### Listing 16. An XML response

```
<acme-races>
  <racess>
    <race name="Leesburg 5K" date="2008-05-12" distance="3.1" id="5">
      <uri>/races/5</uri>
      <description/>
    </race>
    <race name="Leesburg 10K" date="2008-07-30" distance="6.2" id="6">
      <uri>/races/6</uri>
      <description/>
    </race>
  </racess>
</acme-races>
```

In fact, this XML document is an example of what your RESTful Web service will return when the `/race` URI is invoked with a `GET` request.

Therefore, your job is to link the retrieval of all race instances in the underlying data store; in fact, at this point, you can already write a test.

## Testing the service

Using the Restlet framework, you can construct a client instance and have it invoke your RESTful Web service. Moreover, you can leverage `XMLUnit` (see [Resources](#)) to verify that the service's output is some known XML document. Last, but not least, you can also use `DbUnit` (see [Resources](#)) to put the underlying database into a known state (so you can always get back the same XML document).

Using JUnit 4, you can create two fixtures that properly initialize `XMLUnit` and `DbUnit`, as shown in Listing 17:

### Listing 17. Setting up XMLUnit and DbUnit

```
@Before
public void setUpXMLUnit() {
    XMLUnit.setControlParser(
        "org.apache.xerces.jaxp.DocumentBuilderFactoryImpl");
    XMLUnit.setTestParser(
        "org.apache.xerces.jaxp.DocumentBuilderFactoryImpl");
    XMLUnit.setSAXParserFactory(
        "org.apache.xerces.jaxp.SAXParserFactoryImpl");
    XMLUnit.setIgnoreWhitespace(true);
}

@Before
public void setUpDbUnit() throws Exception {
```

```

Class.forName("org.hsqldb.jdbcDriver");
IDatabaseConnection conn =
    new DatabaseConnection(
        getConnection("jdbc:hsqldb:hsqldb://127.0.0.1", "sa", ""));
IDataSet data = new FlatXmlDataSet(new File("etc/database/race-db.xml"));
try {
    DatabaseOperation.CLEAN_INSERT.execute(conn, data);
} finally {
    conn.close();
}
}

```

In the `setUpDbUnit` method, an XML representation of the database is inserted into the database via the `CLEAN_INSERT` command. This XML file effectively inserts six different races. Therefore, the response to a `GET` will be an XML document with six races.

Next, you can create a test case that invokes an HTTP `GET` on the `/race` URI, obtains the response XML, and compares it to a control XML file using XMLUnit's `Diff` class, as demonstrated in Listing 18:

### Listing 18. Verifying a GET response with XMLUnit

```

@Test
public void getRaces() throws Exception {
    Client client = new Client(Protocol.HTTP);
    Response response =
        client.get("http://localhost:8080/racerrest/race/");

    Diff diff = new Diff(new FileReader(
        new File("./etc/control-xml/control-web-races.xml")),
        new StringReader(response.getEntity().getText()));
    assertTrue(diff.toString(), diff.identical());
}

```

The `control-web-races.xml` file is the expected XML response from the Web service. It contains the data shown in Listing 19:

### Listing 19. A control XML file

```

<acme-races>
  < races>
    < race name="Mclean 1/2 Marathon" date="2008-05-12" distance="13.1" id="1">
      < uri>http://localhost:8080/races/1</uri>
      < description/>
    </ race>
    < race name="Reston 5K" date="2008-09-13" distance="3.1" id="2">
      < uri>http://localhost:8080/races/2</uri>
      < description/>
    </ race>
    < race name="Herndon 10K" date="2008-10-22" distance="6.2" id="3">
      < uri>http://localhost:8080/races/3</uri>
      < description/>
    </ race>
    < race name="Leesburg 1/2 Marathon" date="2008-01-02" distance="13.1" id="4">
      < uri>http://localhost:8080/races/4</uri>
      < description/>
  </ races>
</ acme-races>

```

```
</race>
<race name="Leesburg 5K" date="2008-05-12" distance="3.1" id="5">
  <uri>http://localhost:8080/races/5</uri>
  <description/>
</race>
<race name="Leesburg 10K" date="2008-07-30" distance="6.2" id="6">
  <uri>http://localhost:8080/races/6</uri>
  <description/>
</race>
</races>
</acme-races>
```

Running this test now, of course, causes a series of failures, because you haven't implemented the RESTful service yet. Note too that the Ant build file included in the source download contains tasks for deploying a WAR file and starting and stopping Tomcat (see [Download](#)). These are prerequisites for running a successful test.

It turns out that fulfilling the GET request is a cakewalk. All that's required is to invoke the `findAll` method on the `Race` domain object, then pass in the result of that call to `RaceReporter`'s `racessToXml()` method. Accordingly, you need to update the `RacesResource` instance with a new member variable along with a new initialization in the constructor, as shown in Listing 20:

#### Listing 20. Don't forget to add the `RaceReporter`

```
public class RacesResource extends Resource {
    private RaceReporter reporter;

    public RacesResource(Context context, Request request, Response response) {
        super(context, request, response);
        this.getVariants().add(new Variant(MediaType.TEXT_XML));
        this.reporter = new RaceReporter();
    }
}
```

Now, it becomes quite easy to finish implementing the GET request. Simply add three lines to the `getRepresentation` method, as shown in Listing 21:

#### Listing 21. Finishing the GET request

```
public Representation getRepresentation(Variant variant) {
    Collection<Race> races = Race.findAll();
    String xml = this.reporter.racessToXml(races);
    return new StringRepresentation(xml);
}
```

Believe it or not, that's it!

But wait: don't you have to deploy this application to test it?

## Section 9. Deployment and verification

Before you can actually test your RESTful service that returns a list of races, you need to deploy the application. This section shows you how.

### Configuring web.xml

Luckily, deploying a Restlet application couldn't be any easier. You simply create a normal WAR file and ensure that the web.xml file is configured appropriately.

For a Restlet application to function properly in a servlet container, you must update the web.xml file to:

- Properly load your application
- Route all requests through the framework's custom servlet

Consequently, your web.xml file should look like Listing 22:

#### Listing 22. Sample web.xml file

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app id="WebApp_ID" version="2.4"
  xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
    http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">
  <display-name>RESTful racing</display-name>
  <context-param>
    <param-name>org.restlet.application</param-name>
    <param-value>RaceApplication</param-value>
  </context-param>
  <servlet>
    <servlet-name>RestletServlet</servlet-name>
    <servlet-class>com.noelios.restlet.ext.servlet.ServerServlet</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>RestletServlet</servlet-name>
    <url-pattern>*/</url-pattern>
  </servlet-mapping>
</web-app>
```

The source code for this tutorial (see [Download](#)) includes a `war` task that automatically builds a WAR file, and the build file supports deploying the WAR file to a local instance of Tomcat.

As you can see, in the first part of Listing 22, `org.restlet.application` is paired with the class name of the Restlet application, which is `RaceApplication`.

(You might need to fully qualify that name if you gave it a package name.) Note too that the last section of the document maps all requests to the `RestletServlet` type, which was previously mapped to the `com.noelios.restlet.ext.servlet.ServerServlet` class.

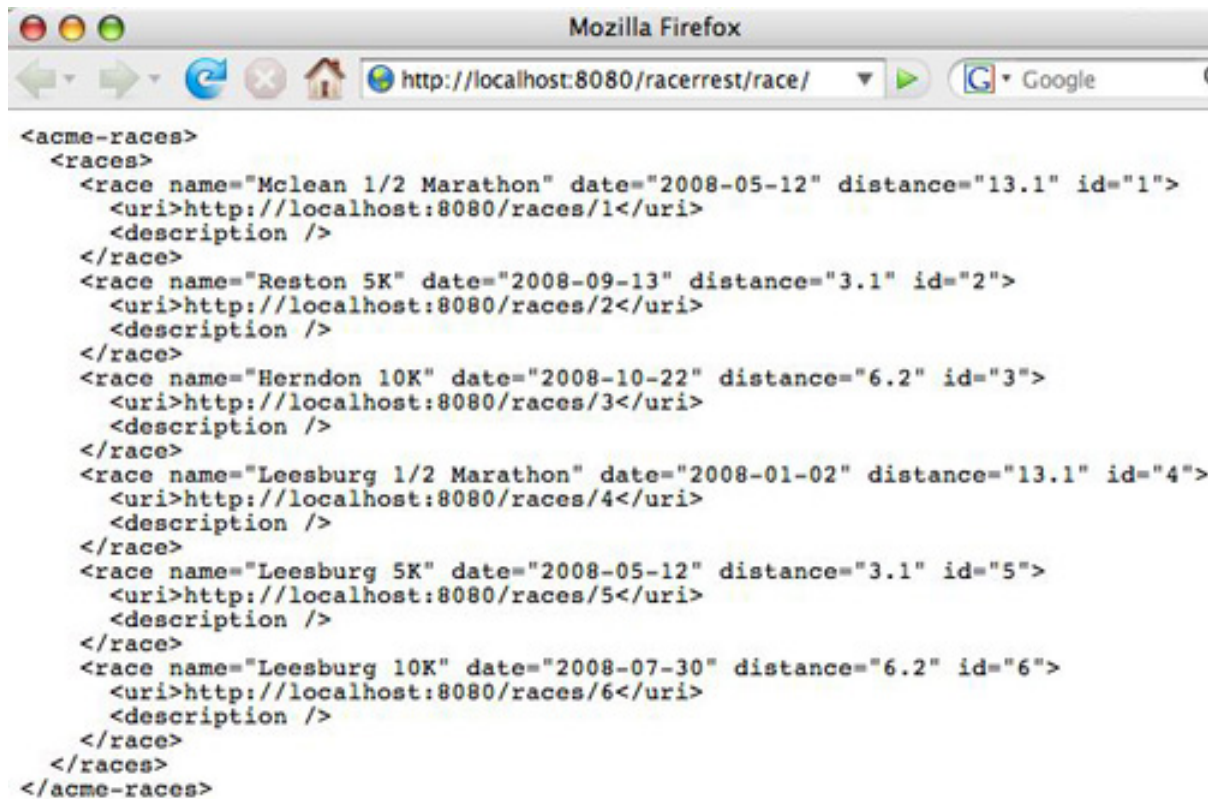
## Testing RESTfully

Testing your RESTful Web service is now a matter of rerunning the test case in [Listing 18](#).

Another look at the test starts to explain some things too. The Restlet's `Client` object supports the basic HTTP commands of `GET`, `PUT`, `POST`, and `DELETE`. And the `Client` object can take the form of different protocols — you just happen to be relying on HTTP in this case.

Your `GET` request already works (see Figure 1), so you can write another test. This time, flesh out the desired behavior of a `POST`; that is, test the creation of a new race via the `RacesResource` class.

### **Figure 1. Viewing a RESTful GET request in a browser**



```
<acme-races>
  <racess>
    <race name="Mclean 1/2 Marathon" date="2008-05-12" distance="13.1" id="1">
      <uri>http://localhost:8080/races/1</uri>
      <description />
    </race>
    <race name="Reston 5K" date="2008-09-13" distance="3.1" id="2">
      <uri>http://localhost:8080/races/2</uri>
      <description />
    </race>
    <race name="Herndon 10K" date="2008-10-22" distance="6.2" id="3">
      <uri>http://localhost:8080/races/3</uri>
      <description />
    </race>
    <race name="Leesburg 1/2 Marathon" date="2008-01-02" distance="13.1" id="4">
      <uri>http://localhost:8080/races/4</uri>
      <description />
    </race>
    <race name="Leesburg 5K" date="2008-05-12" distance="3.1" id="5">
      <uri>http://localhost:8080/races/5</uri>
      <description />
    </race>
    <race name="Leesburg 10K" date="2008-07-30" distance="6.2" id="6">
      <uri>http://localhost:8080/races/6</uri>
      <description />
    </race>
  </racess>
</acme-races>
```

Done

To test a POST, you need to fashion an XML request document with the relevant information and ensure that the service sends back a successful response. Of course, it turns out that writing this test is fairly simple. All you need to do is add some additional code to the existing JUnit class, as shown in Listing 23:

### Listing 23. The createRace test case

```
private static String raceXML = "<?xml version='1.0' encoding='UTF-8'?" +
  "<acme-races>\n" +
  " <race name='Limerick 2008 Half' date='2008-05-12' distance='13.4'>\n" +
  " <description>erin go brach</description>\n" +
  " </race>\n" +
  "</acme-races>";

@Test
public void createRace() {
  Form form = new Form();
  form.add("data", this.raceXML);
  Representation rep = form.getWebRepresentation();
  Client client = new Client(Protocol.HTTP);

  Response response =
    client.post("http://localhost:8080/racerrrest/race/", rep);
  assertTrue(response.getStatus().isSuccess());
}
```

As you can see, Listing 23 quickly fashions a `String` that represents an XML document. In this case, I'm creating a new race called the Limerick 2008 Half. Then it uses the Restlet framework's `Client` object to post this document to the server. Finally, it ensures that a success indication is returned.

Now run the test. It fails, doesn't it? That's because you haven't implemented the `POST` request code, which you'll do in the next section.

---

## Section 10. Race creation RESTfully

Creating races via the RESTful Web service is a matter of a few steps: receiving an XML document, parsing it, creating a new `Race` instance in the underlying database, and finally returning a response indicating the result of the transaction. This section covers these steps.

### Handling POST requests

To implement a create style behavior via REST, you need to handle `POST` requests logically. Accordingly, in the `RacesResource` class, you must override two methods: `allowPost()` and `post()`.

The `post()` method does all the work here. It takes a `Representation` instance, from which you can obtain posted data. Recall that the `createRace` test case in Listing 23 associated the XML document with a name: `data`. Consequently, via the Restlet framework's `Form` object, you can obtain a `String` representing the incoming XML, which you can then pass to the provided `RaceConsumer` object. This object is handy enough to accept XML documents and correspondingly manipulate the underlying database.

If the transaction works, you'll then respond accordingly with a successful response; otherwise, you'll need to respond with a failure message.

Go ahead and override `allowPost()` and `post()`, as shown in Listing 24:

#### Listing 24. Overriding POST methods

```
public boolean allowPost() {
    return true;
}

public void post(Representation representation) {}
```

Because you'll use the `RaceConsumer` object, it makes sense to add it as a member variable of the `RacesResource` class and to initialize it in the constructor. Update your object accordingly, as shown in Listing 25:

### Listing 25. Adding the `RaceConsumer` class

```
public class RacesResource extends Resource {
    private RaceReporter reporter;
    private RaceConsumer consumer;

    public RacesResource(Context context, Request request, Response response) {
        super(context, request, response);
        this.getVariants().add(new Variant(MediaType.TEXT_XML));
        this.reporter = new RaceReporter();
        this.consumer = new RaceConsumer();
    }
}
```

Next, ensure your `post()` method looks like Listing 26:

### Listing 26. Implementing `post()`

```
public void post(Representation representation) {
    Form form = new Form(representation);
    String raceXML = form.getFirstValue("data");
    Representation rep = null;
    try {
        long id = this.consumer.createRace(raceXML);
        getResponse().setStatus(Status.SUCCESS_CREATED);
        rep = new StringRepresentation(raceXML, MediaType.TEXT_XML);
        rep.setIdentifier(getRequest().getResourceRef().getIdentifier() + id);
    } catch (Throwable thr) {
        getResponse().setStatus(Status.SERVER_ERROR_INTERNAL);
        rep = new StringRepresentation("there was an error creating the race",
            MediaType.TEXT_PLAIN);
    }
    getResponse().setEntity(rep);
}
```

As you can see, a lot is going on in the `post` method; however, upon close inspection, things aren't as cerebral as they seem. First, the incoming XML is obtained via the `Form` object. The XML (in the form of a `String`) is then passed to the `createRace()` method of the `consumer` instance. If things worked (that is, the race was persisted), a response is generated that contains a successful status and then a rehashing of the incoming XML, plus the resulting URI (that is, `race/43`, where 43 is the `id` of the newly created race).

If things don't go well, the same process is essentially followed except that a failure status is returned with a failure message: no URI is returned because nothing was created.

Go ahead and rerun the `createRace` test. Assuming you've redeployed the RESTful Web application, things should work quite nicely!

---

## Section 11. A point RESTed

This tutorial managed to implement only a modest number of Acme Racing's requirements. But in the process of doing so, you've seen that working with Restlets is quite simple. The hardest part of the whole exercise was figuring out the logical RESTful API. The source code for this tutorial has all features implemented for your learning pleasure (see [Download](#)).

The poet Alexander Pope said, "There is a certain majesty in simplicity which is far above all the quaintness of wit." This couldn't be more true when it comes to REST. Remember, REST is a way of thinking — a style of designing loosely coupled applications that rely on named resources rather than messages. And by piggybacking on the already validated and successful infrastructure of the Web, REST makes these applications simple to design and implement. And REST applications scale quite well.

This tutorial covered only a handful of the Restlet framework's features, but don't let that fool you. The framework does a lot, including adding security when you need it. Restlets are a joy to code, and the code base is easy understand once you see a few Restlets coded.

Albert Einstein said, "Everything should be made as simple as possible, but not simpler." I hope you agree with me that the Restlet framework and REST itself exemplify this mantra's wisdom.

## Downloads

Description	Name	Size	Download method
Sample code with dependent libraries	j-rest.zip	19.5MB	<a href="#">HTTP</a>
Sample code without dependent libraries	j-rest2.zip	19KB	<a href="#">HTTP</a>

[Information about download methods](#)

# Resources

## Learn

- [Architectural Styles and the Design of Network-based Software Architectures](#) (Roy Thomas Fielding, University of California at Irvine, 2000): Fielding's doctoral dissertation describing REST.
- [Restlet](#): Visit the Restlet framework Web site.
- [Groovy](#): Visit Groovy's Web site.
- "[Resource-oriented vs. activity-oriented Web services](#)" (James Snell, developerWorks, October 2004): Get a quick look at the relationship between REST-style and SOAP-style Web services.
- "[Write REST services](#)" (J. Jeffrey Hanson, developerWorks, October 2007): Work through this tutorial to create REST services with Java technology and the Atom Publishing Protocol.
- "[Crossing borders: REST on Rails](#)" (Bruce Tate, developerWorks, August 2006): Read about building RESTful applications with a popular non-Java Web application development framework.
- "[Fluently Groovy](#)" (Andrew Glover, developerWorks, March 2008): Get started with Groovy. Learn about Groovy's syntax and productivity features, like native collections, built-in regular expressions, and closures. Write your first Groovy class, and test it using JUnit and pure Java code.
- "[Jump into JUnit 4](#)" (Andrew Glover, developerWorks, February 2007): This tutorial shows you how to leverage the new features in JUnit 4 enabled by annotations, including parametric tests, exception tests, and timed tests.
- "[Discover XMLUnit](#)" (Andrew Glover, developerWorks, December 2006): Developers are natural problem solvers, so it makes sense that someone has come up with an easier way to validate XML documents. This article introduces XMLUnit, a JUnit extension framework that meets all your XML validation needs.
- "[Mark it up with Groovy Builders](#)" (Andrew Glover, developerWorks, April 2005): Groovy Builders let you mimic markup languages like XML, HTML, Ant tasks, and even GUIs with frameworks like Swing. They're especially useful for rapid prototyping and, as this article shows you, they're a handy alternative to data binding frameworks when you need consumable markup in a snap!
- "[Effective Unit Testing with DbUnit](#)" (Andrew Glover, OnJava, January 2004): Writing unit tests first can be impractical when your code will depend on access to a database. Enter DbUnit, which allows you to write simple XML files to fill in for the yet-to-be populated database for testing purposes.

- Browse the [technology bookstore](#) for books on these and other technical topics.
- [developerWorks Java technology zone](#): Find hundreds of articles about every aspect of Java programming.

### Get products and technologies

- [Sun JDK 1.5 or later](#): You'll need at least version 1.5.0\_09 to follow the examples in this tutorial.

### Discuss

- [Improve your code quality](#): Andrew Glover's developerWorks discussion for developers focused on test-driven development, code quality, and reducing risk.
- Check out [developerWorks blogs](#) and get involved in the [developerWorks community](#).

## About the author

Andrew Glover

Andrew Glover is president of [Stelligent Incorporated](#), which helps companies embrace developer testing strategies and continuous integration techniques that enable teams to deliver software faster. Check out [Andy's blog](#) for a list of his publications.

## Trademarks

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.