REST WITH AMAZON'S S3

The Representational State Transfer (REST) architectural style is a popular distributed systems design pattern based on the architectural pattern of the World Wide Web—specifically, the HTTP protocol. REST is both a set of principles for building any distributed system and also a specific set of implementation choices when using HTTP.

The REST architecture is considered to be the guiding style used for the HTTP protocol. The term REST, as well as a codification of the principles it espouses, was first published in Roy T. Fielding's PH.D thesis (http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm).

At the core of the REST set of patterns is the idea of a **Resource**, the resource's **URI**, the resource's **Representation**, and a standard, codified set of operations you can have performed on the resource by a distributed system's actor:

Operation	Description	HTTP Method
Read	Access a resource without causing any side effects.	GET
Delete	Delete a resource.	DELETE
Create	Create a new resource.	PUT (and sometimes POST)
Update	Modify a resource's value.	POST (and sometimes PUT)

Notice that the access pattern for a resource is the familiar CRUD interface. HTTP's most popular verbs map very well to this pattern.

Note: There is some debate, and usage in practice varies, as to PUT and POST for Create and Update. It is most practical to consider that either verb may be used for either operation.)

A resource is an abstract concept. Usually it is a document or a piece of data. Resources have a name: a URI (Uniform Resource Indicator). For instance, the document at the root of the web server amazon.com is http://www.amazon.com/index.html. In many cases, a resource is really a collection of data items that can be viewed and processed in a variety of ways. For instance, the HTML resource returned by http://www.amazon.com/index.html may be full XHTML when a contemporary browser accesses it. But when a mobile device accesses that URI, it receives a different representation, such as a minimalist HTML for mobile devices. In HTTP, this is handled through the accept-header.

For instance, to access the resource at http://www.amazon.com/myresource.jpg, you send a GET request, using HTTP:

```
GET /myresource.jpg HTTP/1.1

Host: www.amazon.com
```

```
Date: Tue, 08 August 2008  12:00:00 +0000
```

The web server will return a representation of the resource http://www.amazon.com/myresource.jpg. Likely, there is only one representation; in this case: a jpeg image. However, there can be many representations of a single resource, and the accept-header can be used to determine which one to return.

Assuming proper authorization, that same resource could be deleted using the Delete HTTP method.

```
DELETE /myresource.jpg HTTP/1.1

Host: www.amazon.com

Date: Tue, 08 August 2008  12:00:00 +0000

Authorization: [some authorization data]
```

PUT or POST could be used, depending on the implementation, to change the image itself.

Amazon's S3 (Simple Storage Service) is a web service for storing data of any size in **buckets**. This storage schema is similar to files and folders, except that buckets may not contain other buckets, and the data contained within an S3 **object** doesn't need to be typical file-like material (though it often is.)

Amazon offers developers a REST interface as a way to access and interact with S3. It also offers a SOAP interface. The REST interface uses the standard HTTP verbs in the manner outlined in this document: GET to retrieve a document or bucket, DELETE to delete the same, etc.

Amazon has also released code toolkits in a variety of languages that wrap the REST operations in language-familiar syntax, such as Java classes. For instance, using Amazon's S3 code to create a bucket:

```
AWSAuthConnection conn = new AWSAuthConnection(

            accessKeyId, secretAccessKey,

            false, Utils.DEFAULT_HOST, Utils.INSECURE_PORT,

            CallingFormat.getPathCallingFormat());

Response createResponse = conn.createBucket("MyBucket", null, null);
```

This will result in a PUT request to Amazon's S3 server. Deleting (which corresponds to a DELETE HTTP request) the bucket is just as easy:

```
Response deleteResponse = conn.deleteBucket("MyBucket", null);
```

Note: To delete a bucket, the bucket must be empty. Also, the above code listing assumes the same AWSAuthConnection object is not null.

Creating an object inside a bucket is also a simple set of code. Essentially, you upload a file. This will result in a PUT HTTP request (POST can also be used.)

```
Byte[] file = "Hello, world".getBytes();

Response putResponse = conn.put(

            "MyBucket",

            "MyObject",

            new S3Object(file, null),

            null);
```

To retrieve this new object, a GET HTTP request can be used.

Note: GET is used both to retrieve bucket information (such as a list of all objects in a bucket), but to also retrieve the actual contents of the object.

PROBLEMS

The following problems each use a sample data set that is generated by the attached Java program "InstallS3.java". Follow the instructions in the document "Getting Started with Amazon's Web Services for Lessons Plans" for getting an Amazon Web Services account and setting up your system.

1. Java's standard API for building a REST-based web service is defined in JSR 311, also known as JAX-RS. Implement a REST-based API that is similar to Amazon's S3. Save the content in-memory.
2. Update your REST-based service from #1 to use S3 as a backing store.
3. Bonus question: JSR 311 allows for content-type-based negotiation. For instance, a client may state a preference for text/xml, but also accept text/plain. Update your implementation from #2 to support storing structured data (such as XML or CSV files) and output based on requested content-type (for instance, transform the content using XSLT or similar means.) Extra bonus: support JSON in addition to XML for bucket/folder properties.