CHAPTER 4
Designing URIs

URIs are identifiers of resources that work across the Web. A URI consists of a scheme (such as `http` and `https`), a host (such as `www.example.org`), a port number followed by a path with one or more segments (such as `/users/1234`), and a query string. In this chapter, our focus is on designing URIs for RESTful web services:

Recipe 4.1, "How to Design URIs"
: Use this recipe to learn some commonly practiced URI design conventions.

Recipe 4.2, "How to Use URIs As Opaque Identifiers"
: Use this recipe to learn some dos and don'ts to keep URIs as opaque identifiers.

Recipe 4.3, "How to Let Clients Treat URIs As Opaque Identifiers"
: Treating URIs as opaque identifiers helps decouple clients from servers. This recipe shows techniques that the server can employ to help clients treat URIs as opaque.

Recipe 4.4, "How to Keep URIs Cool"
: Since URIs are a key part of the interface between clients and servers, it is important to keep them "cool," i.e., stable and permanent. Use this recipe to learn some practices to help keep URIs cool.

4.1 How to Design URIs

URIs are opaque resource identifiers. In most cases, clients need not be concerned with how a server designs its URIs. However, following common conventions when designing URIs has several advantages:

- URIs that support convention are usually easy to debug and manage.
- Servers can centralize code to extract data from request URIs.
- You can avoid spending valuable design and implementation time inventing new conventions and rules for processing URIs.
- Partitioning the server's URIs across domains, subdomains, and paths gives you operational flexibility for load distribution, monitoring, routing, and security.

Problem

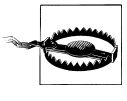
You want to know the best practices to design URIs for resources.

Solution

- Use domains and subdomains to logically group or partition resources for localization, distribution, or to enforce various monitoring or security policies.
- Use the forward-slash separator (`/`) in the path portion of the URI to indicate a hierarchical relationship between resources.
- Use the comma (`,`) and semicolon (`;`) to indicate nonhierarchical elements in the path portion of the URI.
- Use the hyphen (`-`) and underscore (`_`) characters to improve the readability of names in long path segments.
- Use the ampersand (`&`) to separate parameters in the query portion of the URI.
- Avoid including file extensions (such as *.php*, *.aspx*, and *.jsp*) in URIs.

Discussion

URI design is just one aspect of implementing RESTful applications. Here are some conventions to consider when designing URIs.



As important as URI design is to the success of your web service, it is just as important to keep the time spent in URI design to a minimum. Focus on consistency of URIs instead.

Domains and subdomains

A logical partition of URIs into domains and subdomains provides several operational benefits for server administration. Make sure to use logical names for subdomains while partitioning URIs. For example, the server could offer localized representations via different subdomains, as in the following:

```
http://en.example.org/book/1234
http://da.example.org/book/1234
http://fr.example.org/book/1234
```

Another example is, partition based on the class of clients.

```
http://www.example.org/book/1234
http://api.example.org/book/1234
```

In this example, the server offers two subdomains, one for browsers and the other for custom clients. Such partitioning may let the server allocate different hardware or apply different routing, monitoring, or security policies for HTML and non-HTML representations.

Forward-slash separator

By convention, the forward slash (/) character is used to convey hierarchical relationships. This is not a hard and fast rule, but most users assume this when they scan URIs. In fact, the forward slash is the only character mentioned in RFC 3986 as typically indicating a hierarchical relationship. For example, all the following URIs convey a hierarchical association between path segments:

```
http://www.example.org/messages/msg123
http://www.example.org/customer/orders/order1
http://www.example.org/earth/north-america/canada/manitoba
```

Some web services may use a trailing forward slash for collection resources. Use such conventions with care since some development frameworks may incorrectly remove such slashes or add trailing slashes during URI normalization.

Underscore and hyphen

If you want to make your URIs easy for humans to scan and interpret, use the underscore (_) or hyphen (-) character:

```
http://www.example.org/blog/this-is-my-first-post
http://www.example.org/my_photos/our_summer_vacation/first_day/setting_up_camp/
```

There is no reason to favor one over the other. For the sake of consistency, pick one and use it consistently.

Ampersand

Use the ampersand character (`&`) to separate parameters in the query portion of the URI:

```
http://www.example.org/print?draftmode&landscape
http://www.example.org/search?word=Antarctica&limit=30
```

In the first URI shown, the parameters are `draftmode` and `landscape`. The second URI has the parameters `word=Antarctica` and `limit=30`.

Comma and semicolon

Use the comma (`,`) and semi-colon (`;`) characters to indicate nonhierarchical portions of the URI. The semicolon convention is used to identify matrix parameters:

```
http://www.example.org/co-ordinates;w=39.001409,z=-84.578201
http://www.example.org/axis;x=0,y=9
```

These characters are valid in the path and query portions of URIs, but not all code libraries recognize the comma and semicolon as separators and may require custom coding to extract these parameters.

Full stop, or period

Apart from its use in domain names, the full stop (`.`), or period, is used to separate the document and file extension portions of the URI:

```
http://www.example.org/my-photos/flowers.png
http://www.example.org/index.html
http://www.example.org/api/recent-messages.xml
http://www.example.org/blog/this.is.my.next.post.html
```

The last example in the previous list is valid but might introduce confusion. Since some code libraries use the period to signal the start of the file extension portion of the URI path, URIs with multiple periods can return unexpected results or might cause a parsing error.

Except for legacy reasons, there is no reason to use this character in URIs. Clients should use the media type of the representation to learn how to process the representation. "Sniffing" the media type from extensions can lead to security vulnerabilities. For instance, various versions of Internet Explorer are prone to security vulnerabilities because of its implementation of media type sniffing (*http://msdn.microsoft.com/en-us/library/ms775148(VS.85).aspx*).

Implementation-specific file extensions

Consider the following URIs:

```
http://www.example.org/report-summary.xml
http://www.example.org/report-summary.jsp
http://www.example.org/report-summary.aspx
```

In all three cases, the data is the same and the representation format may be the same, but the file extension indicates the technology used to generate the resource representation. These URIs will need to change if the technology used needs to change.

Spaces and capital letters

Spaces are valid URI characters, and according to RFC 3986, the space character should be percent-encoded to `%20`. However, the `application/x-www-form-urlencoded` media type (used by HTML `form` elements) encodes the space character as the plus sign (`+`). Consider the following HTML:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html lang="en">
  <head>
    <title>Search</title>
  </head>
  <body>
    <form method="GET" action="http://www.example.org/search"
      enc-type="application/x-www-form-urlencoded">
      <label for="phrase">Enter a search phrase</label>
      <input type="text" name="phrase" value=""/>
      <input type="submit" value="Search"/>
```

```
    </form>
  </body>
</html>
```

When a user submits the search phrase "Hadron Supercollider," the resulting URI (using `application/x-www-form-urlencoded` rules) would be as follows:

```
http://www.example.org/search?phrase=Hadron+Supercollider
```

Code that is not aware of how the URI was generated will interpret the URI using RFC 3986 and treat the value of the search phrase as "Hadron+Supercollider."

This inconsistency can cause encoding errors for web services that are not prepared to accept URIs encoded using the `application/x-www-form-urlencoded` media type. This is not just a problem with common web browsers. Some code libraries also apply these rules inconsistently.

Capital letters in URIs may also cause problems. RFC 3986 defines URIs as case sensitive except for the scheme and host parts. For example, although `http://www.example.org/my-folder/doc.txt` and `HTTP://WWW.EXAMPLE.ORG/my-folder/doc.txt` are the same, but `http://www.example.org/My-Folder/doc.txt` isn't. However, Windows-based web servers treat these URIs as the same when the resource is served from the filesystem. This case insensitivity does not apply to characters in the query portion. For these reasons, avoid using uppercase characters in URIs.

4.2 How to Use URIs As Opaque Identifiers

Treating URIs as opaque identifiers is, in most cases, trivial. It only requires you to make sure that each resource has a distinct URI. However, some practices illustrated in this recipe can lead to overloading URIs. In such cases, URIs may become generic gateways for unspecified information and actions. This can result in improperly cached responses, possibly even the leakage of secure data that should not be shared without appropriate authentication.

Problem

You want to know how to avoid situations that prevent URIs from being used as unique identifiers.

Solution

Use only the URI to determine which resource processes a request.

Do not tunnel repeated state changes over `POST` using the same URI or use custom headers to overload URIs. Use custom headers for informational purposes only.

Discussion

Designating URIs as unique resource identifiers is a straightforward exercise except when you overload some HTTP methods or use something other than the URI to determine how to process a request.

Here is an example that uses a custom HTTP header to determine what to return:

```
# Request
GET /news HTTP/1.1
Host: www.example.org
X-Filter: science;sports;weather

# Response
HTTP/1.1 200 OK
Content-Type: application/xml;charset=UTF-8

... message body ...
```

In this example, the URI `http://www.example.org/news` is overloaded by the contents of the `X-Filter` header. If another client makes a similar request but with a different value in this custom header (e.g., `politics;economy;healthcare`), the server will return the representation of a different resource.

Such practices are easy to avoid. In this example, the server should offer different URIs for different news filters.

Another common practice that uses URIs as gateways and not as unique identifiers is tunneling repeated state changes using `POST`. This is the default practice in several web frameworks including ASP.NET, JavaServer Pages, and some Ajax toolkits:

```
# Request
POST /ajax-endpoint HTTP/1.1
Host: www.example.org

<request>
  <filter>science</filter>
  <filter>sports</filter>
  <filter>weather</filter>
</request>

# Response
HTTP/1.1 200 OK
Content-Type: application/xml;charset=UTF-8

... message body ...

# Request
POST /ajax-endpoint HTTP/1.1
Host: www.example.org

<request>
  <filter>politics</filter>
  <filter>economy</filter>
```

```
  <filter>healthcare</filter>
</request>

# Response
HTTP/1.1 200 OK
Content-Type: application/xml;charset=UTF-8

... message body ...
```

Such practices are usually a result of treating HTTP as a transport protocol. As long as you avoid such practices, treating URIs as unique identifiers should be relatively easy.

4.3 How to Let Clients Treat URIs As Opaque Identifiers

No matter how you design your URIs, it is important that web services make it possible for clients to treat them as opaque identifiers to the extent possible. Clients should be able to use server-provided URIs to make additional requests without having to understand how the server's URIs are structured.

Problem

You want to know how to ensure clients treat URIs as opaque.

Solution

Whenever possible, provide URIs at runtime using links in the body of representations (see Recipes 5.1 and 5.2) or headers (see Recipe 5.3).

When it is not reasonable to provide a complete set of possible URIs, consider using URI templates (see Recipe 5.7), or establish out-of-band rules to let clients construct URIs programmatically.

Discussion

Neither the architectural constraints of REST nor HTTP require that clients treat URIs as opaque. But doing so reduces coupling between servers and clients. A server expecting clients to construct URIs from bits of information returned in representations or offline knowledge (e.g., documentation or reverse-engineering) indicates tight coupling. This coupling can break existing clients when the web service makes changes to the way it creates new URIs.

In most cases, the process of creating URIs belongs to the server, not the client. For example, consider a photo-sharing web service, returning a list of photos uploaded recently to the server.

```
<?xml version="1.0" encoding="utf-8" ?>
<photos>
  <photo>
    <id>nj1-1234</id>
    <user-id>987</user-id>
```

```
    <server-id>east-nj1</server-id>
  </photo>
  <photo>
    <id>nj4-1235</id>
    <user-id>988</user-id>
    <server-id>east-nj4</server-id>
  </photo>
  ...
</photos>
```

Since no URIs are provided in this representation, anyone implementing a client for this web service must read documentation and write client code to programmatically create URIs to each photo.

```
http://east-nj1.photos.example.org/987/nj1-1234
http://east-nj4.photos.example.org/988/nj4-1235
```

These URIs contain implementation-level data such as server names, photo IDs, and user IDs. If the server makes architectural changes that result in changes in URIs for all new photos, clients will have to make changes in the way they create URIs.

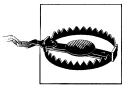
When your web service requires clients to create URIs based on the implementation details of your web service, those details will become part of your web service's public interface. Avoid or minimize leaking such implementation details to clients.

To decouple the client from these implementation details, the server can provide links in the representation.

```
<?xml version="1.0" encoding="utf-8" ?>
<photos xmlns:atom="http://www.w3.org/2005/Atom">
  <photo>
    <atom:link href="http://east-nj1.photos.example.org/987/nj1-1234"
          rel="alternate"
          title="Sunset view from our backyard"/>
    <atom:link href="http://east-nj1.photos.example.org/987"
          rel="http://www.example.org/rels/owner"/>
    <id>nj1-1234</id>
  </photo>
  <photo>
    <atom:link href="http://east-nj4.photos.example.org/988/nj4-1235"
          rel="alternate"/>
    <atom:link href="http://east-nj1.photos.example.org/988"
          rel="http://www.example.org/rels/owner"/>
    <id>nj4-1235</id>
  </photo>
  ...
</photos>
```

This representation uses links to encode implementation details into URIs directly. Each photo in this representation has a link with a URI to fetch the image file and another link to fetch the owner resource of each photo. To realize which link points to

which, clients do not have to know how to manufacture URIs. They just need to understand the meaning of the values of the `rel` attribute.



Note that requiring clients to treat URIs as opaque may require you to tradeoff against performance. Usually URIs are longer in length than database identifiers, and hence transporting URIs over the network increases the message size. This may matter when the representation needs to convey a large number of URIs.

In cases where it is impractical for web services to supply the client with a list of all the possible URIs in the representation (e.g., supporting ad hoc searching), use "semi-opaque" URI templates (see Recipe 5.7). You will also need to loosen/ignore opacity if you want to protect against request tampering by using digitally signed URIs (see Recipe 12.5) or to encrypt parts of the URI to shield sensitive information. For this purpose, clients and servers will need to exchange details of how to sign URIs out of band.

4.4 How to Keep URIs Cool

URIs should be designed to last a long time. Clients may store URIs in databases and configuration files, or may even hard-code them in code. In fact, the Web works under the assumption that URIs are permanent. This design principle is referred to with the axiom "Cool URIs don't change" (*http://www.w3.org/Provider/Style/URI*). When a server decides to change its URIs, clients will fail to function. Cool URIs are those that never change.

The effect of URI changes may seem insignificant when your web service is operating in a private and controlled network. However, URIs make up a vital part of the interface between clients and servers, and changes to URIs are bound to be disruptive. This recipe shows you how to keep URIs permanent.

Problem

You want to know how to support the axiom "Cool URIs don't change."

Solution

Design URIs based on stable concepts, identifiers, and information. Use rewrite rules on the server to shield clients from implementation-level changes. In cases where URIs must change (e.g., when merging two applications, major redesign, etc.), honor old URIs and issue redirects to clients with the new URI using `301` (`Moved Permanently`) responses or, in rare cases, by issuing a `410` (`Gone`) for URIs that are no longer valid.

URIs cannot be permanent if the concepts or identifiers used for URIs cannot be permanent for business, technical, or security reasons. See Recipe 5.6 for ways to deal with such cases.

Discussion

The permanence of URIs depends on stability and the permanence of concepts and identifiers used to create URIs. For example, the URI `http://www.example.org/2009/11/my_trip_report` for a document titled "My Trip Report" is stable as long as the server treats the title as unchangeable once the document has been published. Usually, unique identifiers used to store data of resources help design stable URIs. Such identifiers rarely change.

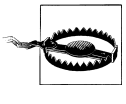
Even when the concepts/identifiers used to create URIs change, it may be possible to hide such changes by employing rewrite rules supported by web servers such as Apache mod_rewrite (*http://httpd.apache.org/docs/2.0/mod/mod_rewrite.html*) and Internet Information Services (IIS) server's URLRewrite (*http://www.iis.net/extensions/URLRewrite*). You can use these web server extensions to hide URI changes that may be caused by merging server applications, changing paths, etc.

If you are not able hide URI changes, respond to all requests to the old URI with a `301` (`Moved Permanently`) and the new URI in the `Location` header:

```
# Request
GET /users/1 HTTP/1.1
Host: www.example.org
Accept: application/json

# Response
HTTP/1.1 301 Found
Location: http://www.example2.org/users/1
```

When a client receives the `301` (`Moved Permanently`) response, it should remove any copies of the old URI from the client's local storage and replace them with the new URI. This will reduce the number of redirects the client needs to follow.



Do not disable support for redirects in client applications. Instead, consider a sensible limit on the number of redirects a client can follow. Also verify that the `Location` URI maps to a trusted domain or IP address. Disabling redirects altogether will break the client when the server decides to change URIs.

Once you set up redirection, monitor request traffic on the server for the old URIs. Maintain redirection services for old URIs until you are confident the majority of clients have updated their stored links to point to the new URI. When you cannot monitor the old URIs, establish and communicate an appropriate end-of-life policy for old URIs.

Once the traffic has fallen off or the preset time interval has passed, convert the `301` (`Moved Permanently`) responses to `410` (`Gone`) or `404` (`Not Found`). Also include a message body to indicate where the new (or related) resources may be found.

```
# Request
GET /users/ HTTP/1.1
Host: www.example.org
```

```
Accept: application/xml;charset=UTF-8

# Response
HTTP/1.1 410 Gone
Content-Type: application/xml;charset=UTF-8;
Expires: Sat, 01 Jan 2011 00:00:00 GMT

<error xmlns:atom="http://www.w3.org/2005/Atom">
  <atom:link rel="help" href="http://www.example2.org"/>
  <message xml:lang="en-US">This resource no longer exists.
  Related information may be found at http://www.example2.org</message>
</error>
```

Note that the previous example shows the `410` (`Gone`) response is marked with an `Expires` header value far into the future. For more on caching responses, see Chapter 9.