CHAPTER 2
Identifying Resources

One of the first steps in developing a RESTful web service is designing the resource model. The resource model identifies and classifies all the resources the client uses to interact with the server. Of all the aspects of designing a RESTful web service, such as identification of resources, choice of media types and formats, and application of the uniform interface, resource identification is the most flexible part.

Because of the visible nature of HTTP (see Recipe 1.1), you can use tools like Firebug (*http://getfirebug.com*), Yahoo! YSlow (*http://developer.yahoo.com/yslow/*), or Resource Expert Droid (*http://redbot.org/*) to make reasonable assertions about whether the server is implementing HTTP correctly. But you cannot do the same with resources. *There is no right or wrong resource model.* All that matters is whether you can use HTTP's uniform interface reasonably correctly to implement your web service. This chapter goes through the following recipes to help identify resources for a number of common situations:

Recipe 2.1, "How to Identify Resources from Domain Nouns"
: Use this recipe to identify an initial set of resources from domain entities.

Recipe 2.2, "How to Choose Resource Granularity"
: Use this recipe to guide resource granularity.

Recipe 2.3, "How to Organize Resources into Collections"
: When you have several resources of the same kind, use this recipe to group those into collection resources.

Recipe 2.4, "When to Combine Resources into Composites"
: Use this recipe to combine resources into composites, based on client usage patterns.

Recipe 2.5, "How to Support Computing/Processing Functions"
: Apply this recipe to identify resources that implement processing functions.

Recipe 2.6, "When and How to Use Controllers to Operate on Resources"
: Use this recipe to design controller resources to make changes across several resources.

Designing a resource model is usually an iterative process. While developing a web service, look at backend design constraints and client needs along with other use cases, and revisit these recipes to improve resources iteratively.

2.1 How to Identify Resources from Domain Nouns

Both object-oriented design and database modeling techniques use domain entities as a basis for design. You can use the same technique to identify resources. But be warned. As you shall see later in this chapter, this recipe is simplistic and can, in some cases, provide a misleading outcome.

Problem

You want to start identifying resources from the use cases and a description of the web service.

Solution

Analyze your use cases to find domain nouns that can be operated using "create," "read," "update," or "delete" operations. Designate each noun as a resource. Use `POST`, `GET`, `PUT`, and `DELETE` methods to implement "create," "read," "update," and "delete" operations, respectively, on each resource.

Discussion

Consider a web service for managing photos. Clients can upload a new photo, replace an existing photo, view a photo, or delete a photo. In this example, "photo" is an entity in the application domain. The actions a client can perform on this entity include "create a new photo," "replace an existing photo," "view a photo," and "delete a photo."

You can apply this recipe to identify each "photo" as a resource such that clients can use HTTP's uniform interface to operate on these photos as follows:

- Method `GET` to get a representation of each photo
- Method `PUT` to update a photo
- Method `DELETE` to delete a photo
- Method `POST` to create a new photo

This recipe is what gives REST the perception that REST is suitable for CRUD-style (Create, Read, Update, Delete) applications only. If you limit yourself to identifying resources based on domain nouns alone, you are likely to find that the fixed set of methods in HTTP is quite a limitation. In most applications, CRUD operations make only part of the interface. Consider some examples:

- Find traffic directions from Seattle to San Francisco.
- Generate random numbers, or convert a given distance from miles to kilometers.

- Provide a way for a client to get a user's profile with a minimal set of properties, list of the 10 latest photos uploaded by the user, and 10 news stories that match the user's interest all in one single request.
- Approve a requisition to buy software.
- Transfer money from one bank account to another bank account.
- Merge two address books.

In all these use cases, it is easy to spot the nouns. But in each case, if you designate those nouns as resources, you will find that the corresponding actions do not map to HTTP methods such as `GET`, `POST`, `PUT`, and `DELETE`. You will need additional resources to tackle such use cases. See the rest of the recipes in this chapter to identify those additional resources.

2.2 How to Choose Resource Granularity

Bluntly mapping domain entities into resources may lead to resources that are inefficient and inconvenient to use. This recipe discusses criteria that you can use to determine appropriate granularity for resources.

Problem

You want to know the criteria for determining an appropriate granularity of resources.

Solution

Use network efficiency, size of representations, and client convenience to guide resource granularity.

Discussion

Looking at the scenarios of your application, you may find several nouns of different granularity. Take, for example, a social network where the interactions happen in the context of a "user." Each user's data may include an activity stream, list of friends, list of followers, links to share, etc. In such an application, should you model each user as a coarse-grained resource to encapsulate all this data? Or should you make the resources less coarse-grained and offer activity streams, friends, followers, etc., as resources as well? The answer depends on what a typical client for your web service does. With the former approach, user representations may be too big for clients to handle, and the latter may be more flexible. If most of the clients download the user's data onto the user's computers, store it, and then present it using some rich user interface, then offering the user resource containing all its data makes sense.

Take a much simpler case such as a user with an address. You may want to maintain a proxy HTTP cache to keep representations of all users in its memory so that clients can quickly access these representations. In this example, the representation of the user

resource that also includes the address may be too big to fit into the cache. Offering the address of each user as a separate resource makes more sense, although it can make client/server interactions chatty because of the reduced granularity.

Similarly, mapping database tables or object models in your application to resources may not produce the best results. A number of factors, such as domain modeling and allowing for efficient data access and processing, influence the design of database tables and object models. HTTP clients, on the other hand, access resources over the network using HTTP's uniform interface. Therefore, you need to design resources to suit clients' usage patterns and not design them based on what exists in a database or the object model.

So, how can you determine nouns that are candidate resources? How granular should you design them? The best way to answer these questions is to think from the client's perspective. In the first example shown previously, coarse granularity is more convenient for rich-client applications, whereas in the second example, the resources are more fine-grained to meet caching requirements. Therefore, look from the client and network point of view to determine resource granularity. The following factors may further influence resource granularity:

- Cacheablility
- Frequency of change
- Mutability

Refining resource granularity to ensure that more cacheable, less frequently changing, or immutable data is separated from less cacheable, more frequently changing, or mutable data can improve the efficiency of both clients and servers.

2.3 How to Organize Resources into Collections

Organizing resources into collections gives clients and servers an ability to refer to a group of a resources as one, to perform queries on the collection, or even to use the collection as a factory to create new resources.

Problem

You want to know how best to group together resources that share some commonality.

Solution

Identify similar resources based on any application-specific criteria. Common examples are resources that share the same database schema or the same set of attributes or properties or look similar to clients. Group similar resources into a collection resource for each similarity.

Design a representation for each collection such that it contains information about all or some of its member resources (see Recipe 3.7).

Discussion

Once you group several similar resources under a collection resource, you can refer to the group as a whole, as in the following example. You can, for instance, submit a `GET` request to fetch an entire collection instead of fetching individual resources one after the other.

Consider a social network, where all user records share the same database schema. Each user in this network has a list of friends and a list of followers. Friends and followers are other users in the same database. Users are categorized based on personal interests, such as running, cycling, swimming, hiking, etc. In this example, you can identify the following collections whose members are user resources:

- Collection of user resources
- Collection of friends of any given user
- Collection of followers of a given user
- Collections of users by the same interest

Here is an example of a users collection resource returned in response to a `GET` request to that collection:

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

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

<users xmlns:atom="http://www.w3.org/2005/Atom"> ❶
  <atom:link rel="self" href="http://www.example.org/users"/>
  <user> ❷
    <id>urn:example:user:001</id>
    <atom:link rel="self" href="http://www.example.org/user/user001"/>
    <name>John Doe</name>
    <email>john.doe@example.org</email>
  </user>
  <user>
    <id>urn:example:user:002</id>
    <atom:link rel="self" href="http://www.example.org/user/user002"/>
    <name>Jane Doe</name>
    <email>jane.doe@example.org</email>
  </user>
  ...
</users>
```

❶ A collection resource

❷ A member of the collection

Note that a collection does not necessarily imply hierarchical containment. A given resource may be part of more than one collection resource. For example, a user resource may be part of several collections such as "users," "friends," "followers," and "hikers." Here is a friends collection for a user:

```
# Request
GET /user/user001/friends HTTP/1.1
Host: www.example.org

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

<users xmlns:atom="http://www.w3.org/2005/Atom">
  <atom:link rel="self" href="http://www.example.org/user/user001/friends"/>
  <user>
    <id>urn:example:user:002</id>
    <atom:link rel="self" href="http://www.example.org/user/user002"/>
    <name>Jane Doe</name>
    <email>jane.doe@example.org</email>
  </user>
  ...
</users>
```

You can use collection resources for the following:

- To retrieve paginated views of a collection, such as browsing through the friends collection of a user, obtained 10 at a time (see Recipe 3.7).
- To search the collection for its members or to obtain a filtered view of the collection. For instance, you could search for friends who are swimmers (see Recipe 8.2).
- To create new member resources using the collection as a factory, by submitting HTTP `POST` requests to the collection resource.
- To perform the same operation on a number of resources at once (see Recipe 11.10).

2.4 When to Combine Resources into Composites

When you look at the home pages of sites like *http://www.yahoo.com* or *http://www.msn.com*, you will notice that these pages aggregate information from a number of sources, such as news, email, weather, entertainment, finance, etc. If you think of each of these sources as resources, serving each of these home pages is a result of combining those disparate resources into a single resource whose representation is an HTML page. Such web pages are composite resources; i.e., they combine information from other resources. This recipe uses the same technique to identify composite resources.

Problem

You want to know how to provide a resource whose state is made up of states from two or more resources.

Solution

Based on client usage patterns and performance and latency requirements, identify new resources that aggregate other resources to reduce the number of client/server round-trips.

Discussion

A composite resource combines information from other resources. Consider a snapshot page for each customer in an enterprise application. This page shows the customer information, such as the name, the contact information, a summary of the latest purchase orders from the customer, and any pending requests for quotes. Using the recipes discussed in this chapter so far, you can identify the following resources:

- Customer, with name, contact information, and other details
- Collection of purchase orders for each customer
- Collection of pending quotes for each customer

Given these resources, you can make the following `GET` requests and, using the responses, build a customer snapshot page:

```
# Get the customer data
GET /customer/1234 HTTP/1.1
Host: www.example.org

# Get the 10 latest purchase orders
GET /orders?customerid=1234&sortby=date_desc&limit=10 HTTP/1.1
Host: www.example.org

# Get the 10 latest pending quotes
GET /quotes?customerid=1234&sortby=date_desc&status=pending&limit=10 HTTP/1.1
Host: www.example.org
```

Although this sequence of `GET` requests may be acceptable for the server, these requests are very chatty. It may be more efficient for the client to send a single network request for all the data needed to render the page.

For the customer snapshot page, you can design a "customer snapshot" composite resource that combines all the information needed for the client to render the page. Assign a URI of the form `http://www.example.org/customer/1234/snapshot` where `1234` is an identifier that identifies a customer. Here is an example of this resource in use:

```
# Request
GET /customer/1234/snapshot HTTP/1.1
Host: www.example.org
```

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

<snapshot xmlns:atom="http://www.w3.org/2005/Atom">
  <!-- Customer info -->
  <customer>
    <id>1234</id>
    <atom:link rel="self" href="http://www.example.org/customer/1234">
    <name>...</name>
    <address>...</address>
  </customer>
  <!-- Most 10 recent orders placed by the customer -->
  <orders>
    <atom:link rel="http://www.example.org/rels/orders/recent"
          href="http://www.example.org/orders?customerid=1234&sortby=date_desc&"/>
    <order>
      <id>...</id>
      ...
    </order>
    ...
  </orders>
  <!-- Most 10 pending quotes for the customer -->
  <quotes>
    <atom:link rel="http://www.example.org/rels/quotes/recent"
          href="http://www.example.org/quotes?customerid=1234&sortby=date_desc&"/>
    ...
  </quotes>
</snapshot>
```

This response is an aggregate of representations that the client would get by submitting three `GET` requests.

Composite resources reduce the visibility of the uniform interface since their representations contain overlapping data with other resources. Therefore, before offering composite resources, consider the following:

- If requests for composites are rare in your application, composites may be a poor choice. The client may benefit from relying on a caching proxy to fetch those resources from a cache instead.
- Another factor is the network cost between the client and the server and between the server and any backend services or data stores it relies upon. If the cost of the latter is significant, then retrieving large amounts of data and combing them on the server into a composite may increase the latency for the client and reduce throughput for the server.

  In this case, you may be able to improve latency by adding a caching layer between clients and servers and avoiding composites. Conduct load tests to verify whether a composite would help.

- Finally, creating special-purpose composites for the sake of every client is not a pragmatic task. Pick the clients that are most important for your web service, and design composites to suit the needs of those clients.

2.5 How to Support Computing/Processing Functions

Processing functions are not uncommon. Websites like Babel Fish (*http://babelfish.yahoo.com/*), XE.com (*http://www.xe.com/*), and Google Maps (*http://maps.google.com*) take some inputs, process them with the help of data stored in their backend servers and some algorithms, and return results. These are all processing functions.

Problem

You want to know how to provide resource abstractions for tasks such as performing computations or validating data.

Solution

Treat the processing function as a resource, and use HTTP `GET` to fetch a representation containing the output of the processing function. Use query parameters to supply inputs to the processing function.

Discussion

One of the most common perceptions of REST's architectural constraints is that they only apply to resources that are "things" or "entities" in the application domain. Although this may be true in a number of cases, scenarios that involve processing functions challenge that perception. Here are some examples:

Distance between two places
: A client submits latitude and longitude values of both the locations to the server. The server then computes and returns the distance to the client.

Driving directions
: A client submits two locations in free form, say, "Seattle, WA" and "San Francisco, CA," and the server returns the directions as a list of driving segments and turn directions.

Validate a credit card
: The client submits credit card data such as the name of the cardholder, card number, and expiry date to the server, and the server returns data to the client indicating whether the card is valid.

All these examples share the same peculiarity. In each case, if you apply Recipe 2.1, you will find nouns on which you cannot easily apply the uniform interface. For example, if you identify each "place" as a resource, you will find that there is no HTTP equivalent of the operation "compute distance."

One way to address such use cases is to treat the processing function itself as a resource. In the first example, you can treat the distance calculator as a resource and the distance as its representation. The following request and response illustrate this resource:

```
# Request
GET /distance_calc?lats=47.610&lngs=-122.333&late=37.788&lnge=-122.406 HTTP/1.1
Host: www.example.org

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

<result xmlns:atom="http://www.w3.org/2005/Atom">
  <atom:link rel="self"
    href="http://www.example.org/distance_calc?lats=47.610&
          lngs=-122.333&late=37.788&lnge=-122.406"/>
  <distance unit="miles">808.0</distance>
</result>
```

Similarly, "direction finder," "points of interest finder," and "credit card validator" can all be resources with "directions," "points of interest," and "validation result" as representations of those resources:

```
# Request to find directions
GET /directions?from=Seattle,WA&to=San%20Francisco HTTP/1.1
Host: www.example.org

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

<directions>
  <step>
    ...
  </step>
  <step>
    ...
  </step>
</directions>

# Request to find points of interest
GET /poi?lat=47.610&lng=-122.333 HTTP/1.1
Host: www.example.org

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

<atom:feed xmlns:atom="http://www.w3.org/2005/Atom">
  <atom:title>Points of Interest</title>
  <atom:link href="http://www.example.org/poi?lat=47.610&lng=-122.333/" rel="self"/>
  <atom:updated>2009-10-01T18:30:02Z</atom:updated>
  <atom:author>
    <atom:name>All Names Made Up Inc.</atom:name>
  </atom:author>
```

```
  <atom:id>urn:uuid:60a76c80-d399-11d9-b93C-0003939e0af6</atom:id>
  <atom:entry>
    <atom:id>urn:example:poi:0012</atom:id>
    <atom:title>...</atom:title>
    <atom:updated>2009-09-13T18:30:02Z</atom:updated>
    <atom:link rel="alternate" href="http://www.example.org/poi/0012.html"/>
    <atom:content type="text">...</atom:content>
  </atom:entry>
  ...
</atom:feed>

# Request to validate a credit card (sent via HTTPS)
GET /validate?ccnum=1234567890123456 HTTP/1.1
Host: www.example.org

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

invalid
```

Because all these methods are safe and idempotent, `GET` is the most appropriate HTTP method for implementing processing functions.

URIs such as `https://www.example.org/validate` appear to denote "operations," thus undermining the uniform interface. However, URIs merely identify resources, and the syntax of URIs does not matter as far as HTTP is concerned.

2.6 When and How to Use Controllers to Operate on Resources

In the case of RESTful web services, controllers can help increase the separation of concerns between servers and clients, improve network efficiency, and let servers implement complex operations atomically.

Problem

You want to know how to tackle write operations that involve modifying more than one resource atomically, or whose mapping to `PUT` or `DELETE` is not obvious.

Solution

Designate a controller resource for each distinct operation. Let clients use the HTTP method `POST` to submit a request to trigger the operation. If the outcome of the operation is the creation of a new resource, return response code `201` (`Created`) with a `Location` header referring to the URI of the newly created resource. If the outcome is the modification of one or more existing resources, return response code `303` (`See Other`) with a `Location` with a URI that clients can use to fetch a representation of those modifications. If the server cannot provide a single URI to all the modified resources, return

response code 200 (OK) with a representation in the body that clients can use to learn about the outcome. Handle errors as described in Recipe 3.13.

Discussion

A *controller* is a resource that can atomically make changes to resources. The need for such a resource may not be apparent from your domain model, but it can help the server abstract complex business operations and provide a way for clients to trigger those operations. This in turn reduces coupling between clients and servers.

Consider merging two address books for a user. A client on the mobile phone needs a way to synchronize all the contacts with the current address book on the server. One option is to use PUT as follows:

1. Submit a GET request to the address book resource to download the complete address book from the server.
2. Load the local list of contacts, and merge them with the address book downloaded from the server.
3. Submit a PUT request to the address book resource to replace the entire address book with the merged one.

This will do the job but with some limitations. For the client's environment, downloading the entire address book and then merging it with the local list of contacts makes the client's use of the network inefficient. Moreover, some users may have very large address books on the server, and not all fields in the address book may be relevant for the client. The client may not have enough computing power for handling the merge operation. More importantly, the application logic to merge entries in the address book belongs to the server, not the client. Expecting clients to deal with this task results in the duplication of code and poor separation of concerns.

Here is another option:

1. Get each address in the address book from the server.
2. If that address matches with an entry in the local storage, merge it, and update it by submitting a PUT request.
3. If there is a new contact in the local storage that does not exist on the server, submit a POST request to the address book to add it.

This approach has the additional drawback of network chattiness, which again is not suitable for the client's constrained environment such as a mobile phone.

A more effective solution is to employ a controller resource to solve this problem. For this example, design a controller resource, and allow the client to submit the address book to the server for a merge.

```
# Request to merge an address book
POST /user/smith/address_merge HTTP/1.1
Host: www.example.org
```

```
Content-Type: text/csv;charset=UTF-8

John Doe, 1 Main Street, Seattle, WA
Jane Doe, 100 North Street, Los Angeles, CA
...

# Response
HTTP/1.1 303 See Other
Location: http://www.example.org/user/smith/address_book
Content-Type: text/html;charset=UTF-8

<html>
  <body>
    <p>See <a href="http://www.example.org/user/smith/address_book">address
    book</a> for the merged address book.</p>
  </body>
</html>
```

After merging the address books, the server redirects the client to the user's updated address book. The client can fetch a copy of the merged address book, if necessary.

Here is another example. Consider a bookstore where a store operator wants to reduce the pretax price of a book by 15 percent, and update the posttax price to reflect this discount. The server can offer the discount percentage as a resource, and clients can submit a `PUT` request to modify the current discount. The server can update the total price of the book as part of the same request.

```
# Request to update the discount value
PUT /book/1234/discount HTTP/1.1
Host: www.example.org
Content-Type: application/x-www-urlencoded

value=15

# Response
HTTP/1.1 204 No Content
```

Now consider that the client wants to offer a 30-day free access to an online version of the same book along with this 15 percent discount. The server can maintain a collection of all books that are currently being offered in the 30-day free plan, and the client can submit a `POST` request to add this particular book to that collection.

```
# Request to add the book to the list of offers
POST /30dayebookoffers HTTP/1.1
Host: www.example.org
Content-Type: application/x-www-urlencoded

id=1234&from=2009-10-10&to=2000-11-10

# Response
HTTP/1.1 201 Created
Location: http://www.example.org/30dayebookoffer/1234
Content-Length: 0
```

If your business case requires that these two changes be done atomically, you can employ a controller resource for this purpose.

```
# Request to add a discount offer and 30-day free access
POST /book/1234/discountebookoffer HTTP/1.1
Host: www.example.org
Content-Type: application/x-www-urlencoded

id=1234&discount=15&ebook_from=2009-10-10&ebook_to=2000-11-10

# Response
HTTP/1.1 303 See Other
Location: http://www.example.org/book/1234
Content-Length: 0

# Request to get the updated book
GET /book/1234 HTTP/1.1
Host: www.example.org

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

<book xmlns:atom="http://www.w3.org/2005/Atom">
  <id>urn:example:book:1234</id>
  <atom:link rel="self" href="http://www.example.org/book/1234"/>
  ...
  <discount>15</discount>
  ...
  <atom:link rel="http://www.example.org/rels/offer"
        href="http://www.example.org/30dayebookoffer/1234"/>
  ...
</book>
```

In the response, the server includes a link to let clients discover the 30-day offer. If the client is presenting a user interface to end users, it can provide a link to this offer for users to navigate to.

The key point to notice from these examples is the difficulty you might find in mapping operations in your application to the methods in the uniform interface. For example, in the discount example, the server identifies the current discount value as a resource so that clients can use `PUT` to update it. Similarly, the server identifies 30-day free electronic book offers as a collection and lets clients use `POST` to add a new book to this collection. But when it comes to combining these two tasks into a single request, a direct mapping to any HTTP method is not obvious. Controllers are most appropriate in such cases.

For use cases like the previous one, do not use the method `POST` directly on the book resource because it could lead to tunneling. Tunneling occurs whenever the client is using the same method on a single URI for different actions. Here is an example of tunneling:

```
# Request to add a discount offer
POST /book/1234 HTTP/1.1
Host: www.example.org
Content-Type: application/x-www-urlencoded

op=updateDiscount&discount=15

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

<book xmlns:atom="http://www.w3.org/2005/Atom">
  <id>urn:example:book:1234</id>
  <atom:link rel="self" href="http://www.example.org/book/1234"/>
  ...
  <discount>15</discount>
  ...
</book>

# Request to add the book for 30-day offers
POST /book/1234 HTTP/1.1
Host: www.example.org
Content-Type: application/x-www-urlencoded

op=30dayOffer&ebook_from=2009-10-10&ebook_to=2000-11-10

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

<book xmlns:atom="http://www.w3.org/2005/Atom">
  <id>urn:example:book:1234</id>
  <atom:link rel="self" href="http://www.example.org/book/1234"/>
  ...
  <atom:link rel="http://www.example.org/rels/offer"
        href="http://www.example.org/30dayebookoffer/1234"/>
</book>
```

In the requests, the parameters `op=updateDiscount` and `op=30dayOffer` signify the operation. This leads to tunneling.

Tunneling reduces protocol-level visibility (see Recipe 1.1) because the visible parts of requests such as the request URI, the HTTP method used, headers, and media types do not unambiguously describe the operation.

Avoid tunneling at all costs. Instead, use a distinct resource (such as a controller) for each operation.