Addison-Wesley Professional Ruby Series

SERVICE-ORIENTED DESIGN WITH RUBY AND RAILS

Foreword by **Obie Fernandez,** *Series Editor*

PAUL DIX

with TROTTER CASHION ▪ BRYAN HELMKAMP ▪ JAKE HOWERTON

Service-Oriented Design with Ruby and Rails

This puts the responsibility for constructing URIs to joined data inside each service. It's a slight departure from REST because the service is not storing the actual URI. However, it enables easy updates for resources that put version numbers in the URI. When the user service updates its version, the comment service has to update its URI generation logic. Of course, the update to the user system should allow the upgrade of the comment service to happen over time rather than having to update both systems in lock step.

Still another argument can be made for passing raw IDs in the responses. With some services, it makes sense to have the client contain business logic that wraps the data instead of just passing the raw data to the service. A pure RESTful approach assumes that no business logic will be written around the responses. Take basic data validation as an example. When attempting to create a user, you may want to ensure that the email address is in the proper form, the user name isn't blank, and the password isn't blank. With a pure RESTful approach, these validations would occur only in the service. However, if a service client wraps the data with a little bit of business logic, it can perform these kinds of data validations before sending to the service. So if you're already writing clients with business logic, it isn't that much of a stretch to have a client also be responsible for URI construction.

Joining at the Highest Level

The *highest level* for a join is the place in the service call stack where joins should occur. For example, with the social feed reader, consider a service that returns the five most recent reading items for a user. The application server will probably make calls to the user service for the basic user data, the subscription service for the list of reading items in a specific user's list, the feed entry service for the actual entries, and the ratings service for ratings on the list of entries. There are many places where the joins of all this data could possibly occur.

The subscriptions and ratings services both store references to feed entries and users. One method would be to perform the joins within each service. However, doing so would result in redundant calls to multiple services. The ratings and reading lists would both request specific entries and the user. To avoid these redundant calls, the joins on data should occur at the level of the application server instead of within the service.

There are two additional reasons that joins should be performed at the application server. The first is that it makes data caching easier. Services should have to cache data only from their own data stores. The application server can cache data from any

service. Splitting up caching responsibility makes the overall caching strategy simpler and leads to more cache hits. The second reason is that some service calls may not need all the join data. Keeping the join logic in the application server makes it easier to perform joins on only the data needed.

Beware of Call Depth

Call depth refers to the number of nested joins that must be performed to service a request. For example, the request to bring back the reading list of a user would have to call out to the user, subscription, feed storage, and ratings services. The call chain takes the following steps:

1. It makes simultaneous requests to the user and subscription services.
2. It uses the results of the subscription service to make simultaneous requests to the feed storage and ratings services.

This example has a call depth of two. There are calls that have to wait for a response from the subscription service before they can start. This means that if the subscription service takes 50 milliseconds to respond and the entry and ratings services complete their calls in 50 milliseconds, a total time of 100 milliseconds is needed to pull together all the data needed to service the request.

As the call depth increases, the services involved must respond faster in order to keep the total request time within a reasonable limit. It's a good idea to think about how many sequential service calls have to be made when designing the overall service architecture. If there is a request with a call depth of seven, the design should be restructured so that the data is stored in closer proximity.

API Complexity

Service APIs can be designed to be as specific or as general as needed. As an API gets more specific, it generally needs to include more entry points and calls, raising the overall complexity. In the beginning, it's a good idea to design for the general case. As users of a service request more specific functionality, the service can be built up. APIs that take into account the most general cases with calls that contain the simplest business logic are said to be *atomic*. More complicated APIs may enable the retrieval of multiple IDs in a single call (multi-gets) or the retrieval or updating of multiple models within a single service call.

Atomic APIs

The initial version of an API should be atomic. That is, it should expose only basic functions that can be used to compose more advanced functionality. For example, an API that has a single `GET` that returns the data for different models would not be considered atomic. The atomic version of the API would return only a single type of model in each `GET` request and force the client to make additional requests for other related models. However, as an API matures, it makes sense to include convenience methods that include more data and functionality.

However, simple atomic APIs can be taken too far. For example, the interface for retrieving the list of comments for a single user in the social feed reader is non-atomic. It returns the actual comments in the response. A more atomic version of the API would return only the comment IDs and the URIs to get each comment. The expectation is that the client will get the comments it needs. As long as these `GET`s can be done in parallel, it doesn't matter that you have to do many to get a full list of comments.

In theory, the parallel request model works. However, in practice it's more convenient to just return the list of comments in the actual request. It makes the most sense for clients because in almost all cases, they want the full comments instead of just the IDs.

Atomicity is about returning only the data that a client needs and not stuffing the response with other objects. This is similar to performing joins at the highest level. As you develop an application and services, it may make sense to have special-case requests that return more data. In the beginning, it pays to make only the simplest API and expand as you find you need to.

Multi-Gets

You might want to provide a multi-get API when you commonly have to request multiple single resources at one time. You provide a list of IDs and get a response with those objects. For example, the social feed reader application should have multi-get functionality to return a list of feed entry objects, given a list of IDs. Here is an example of a request/response:

```
REQUEST GET /api/v1/feed_entries?ids=1,2
RESPONSE:
[{
  "id" : 1,
  "feed_id" : 45,
```

```
    "title" : "an entry",
    "content" : "this is an entry",
    "url" : "http://pauldix.net/1"},
  {"id" : 2,
    "feed_id" : 78,
    "title" : "another",
    "content" : "another entry",
    "url" : "http://pauldix.net/2"}
]
```

For service clients, the advantage of using this approach is that the clients are able to get all the entries with a single request. However, there are potential disadvantages. First, how do you deal with error conditions? What if one of the requested IDs was not present in the system? One solution is to return a hash with the requested IDs as the keys and have the values look the same as they would if they were responses from a single resource request.

The second possible disadvantage is that it may complicate caching logic for the service. Take, for example, a service that runs on multiple servers. To ensure a high cache hit ratio, you have a load balancer in front of these servers that routes requests based on the URL. This ensures that a request for a single resource will always go to the same server. With multi-get requests, the resources being requested may not be cached on the server the request gets routed to. At that point, you either have to make a request to the server that has the cached item or store another copy. The workaround for this is to have the servers within a service communicate to all of the service's caching servers.

Multiple Models

A multiple-model API call is a single call that returns the data for multiple models in the response. An API that represents a basic social network with a user model and a follows model is a good example that shows where this would be useful. The follows model contains information about which user is the follower, which user is being followed, when the follow occurred, and so on. To retrieve a user's full information, the atomic API would require two requests: one to get the user data and one to get all the follows data for that user. A multiple-model API could reduce that to a single request that includes data from both the user and follows models.

Multiple-model APIs are very similar to multi-gets in that they reduce the number of requests that a client needs to make to get data. Multiple-model APIs are usually best

left for later-stage development. You should take the step of including multiple models within the API only when you're optimizing calls that are made very frequently or when there is a pressing need. Otherwise, your API can quickly become very complex.

Conclusion

When designing an overall services architecture, you need to make some decisions up front about the services and their APIs. First, the functionality of the application must be partitioned into services. Second, standards should be developed that dictate how to make requests to service APIs and what those responses should look like. Third, the design should account for how many joins have to occur across services. Finally, the APIs should provide a consistent level of functionality.

APIs should take into account the partitioning of the application, consistency in their design, and versioning. Strategies for partitioning application into services include partitioning based on iteration speed, on logical function, on read/write frequency, and on join frequency. Developing standards in the URI design and service responses makes it easier to develop client libraries to access all services. The service APIs should aim for consistency to make URI and response parsing easier to extract into shared libraries for use by everyone. Services should be versioned and be able run multiple versions in parallel. This gives service clients time to upgrade rather than having to upgrade both client and server in lock step.

CHAPTER 5
Implementing Services

This chapter explores a few of the options in Ruby for implementing services. Specifically, in this chapter, you will implement a service from the social feed reader application with Rails, Sinatra, and as a basic Rack application. Implementing the same functionality in all these frameworks will help reveal the differences in these choices and the possible strengths and weaknesses of each.

The Rails framework provides an option that most readers are familiar with for complete web applications. Rails supplies a lot of innate functionality, but at the cost of speed and extra code and complexity that isn't necessary in smaller services. Rails is a good place to start, but you should examine other frameworks to see other ways of organizing logic for handling requests. Sinatra is part of the newest generation of Ruby frameworks that diverge from Rails in design. Rack is the basic web server interface that both Rails and Sinatra are built on.

The Vote Service

The service this chapter focuses on is the vote system from the social feed reader application. Basically, the vote service should provide a way to let users give a "thumbs up" or "thumbs down" to a feed entry. It should also provide different ways to retrieve previous vote data. In addition, the service should provide the following specific features:

- **Vote an entry "up" or "down"**—The service should provide an API call to rate an entry. This call should require a user ID, an entry ID, and whether the vote is up