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

Service-Oriented Design with Ruby and Rails

Paul Dix

Addison-Wesley
Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Library of Congress Cataloging-in-Publication Data

Dix, Paul, 1977-
Service-oriented design with Ruby and Rails / Paul Dix.
p. cm.
Includes bibliographical references and index.
ISBN 0-321-65936-8 (pbk. : alk. paper) 1. Web services.
2. Service-oriented architecture (Computer science) 3. Web sites—Design.
4. Ruby on rails (Electronic resource) I. Title.
TK5105.88813.D593 2010
006.7'8—dc22

2010021623

ISBN-13: 978-0-321-65936-1
ISBN-10: 0-321-65936-8
Text printed in the United States on recycled paper at Courier in Stoughton, Massachusetts.

First printing, August 2010

Associate Publisher
Mark Taub

Acquisitions Editor
Debra Williams Cauley

Development Editor
Michael Thurston

Managing Editor
John Fuller

Project Editor
Elizabeth Ryan

Copy Editor
Kitty Wilson

Indexer
Jack Lewis

Proofreader
Carol Lallier

Technical Reviewers
Jennifer Lindner,
Trotter Cashion

Cover Designer
Chuti Prasertsith

Compositor
LaserWords

To Pops, for encouraging my weird obsession with computers.

This page intentionally left blank

Contents

Foreword

It's an honor for me to present to you this timely new addition to the Professional Ruby Series, one that fills a crucially important gap in the ongoing evolution of all professional Rubyists and couldn't come a moment sooner! It is authored by one of the brightest minds of our international Ruby community, Paul Dix, described as "genius" and "A-list" by his peers. Paul is no stranger to the Ruby world, a fixture at our conferences and involved in some of the earliest Rails project work dating back to 2005. He's also the author of Typhoeus, a successful high-performance HTTP library that is an essential part of the service-oriented ecosystem in Ruby.

Why is this book so timely? Serious Ruby adoption in large companies and project settings inevitably necessitates service-oriented approaches to system design. Properly designed large applications, partitioned into cooperating services, can be far more agile than monolithic applications. Services make it easy to scale team size. As the code base of an application gets larger, it gets harder to introduce new developers to the project. When applications are split into services, developers can be assigned to a specific service or two. They only need to be familiar with their section of the application and the working groups can remain small and nimble.

There's also the fact that we live in the age of The Programmable Web, the boom of web applications, APIs, and innovation over the past few years that is directly attributable to the rise of interoperable web services like those described in this book. Applications that rely on web resources present unique challenges for development teams. Service-oriented traits impact various aspects of how applications should be designed and the level of attention that needs to be paid to how the application performs and behaves if those services are unavailable or otherwise limited.

My own teams at Hashrocket have run into challenges where we could have used the knowledge in this book, both in our Twitter applications as well as our large client projects, some of which we have been working on for years. In a couple of notable cases, we have looked back in regret, wishing we had taken a service-oriented approach

sooner. I assure you that this book will be on the required-reading list for all Rocketeers in the future.

Like Hashrocket, many of you buying this book already have big monolithic Rails applications in production. Like us, you might have concerns about how to migrate your existing work to a service-oriented architecture. Paul covers four different strategies for application partitioning in depth: Iteration Speed, Logical Function, Read/Write Frequency, and Join Frequency. Specific examples are used to explore the challenges and benefits of each strategy. The recurring case study is referred to often, to ensure the discussion is grounded in real, not imaginary or irrelevant situations.

Paul doesn't limit himself to theory either, which makes this a well-rounded and practical book. He gives us important facts to consider when running in a production environment, from load balancing and caching to authentication, authorization, and encryption to blocking I/O to parallelism, and how to tackle these problems in Ruby 1.8, 1.9, Rubinius, and JRuby.

Overall, I'm proud to assure you that Paul has given us a very readable and useful book. It is accurate and current, bringing in Rack, Sinatra, and key features of Rails 3, such as its new routing and ActiveModel libraries. At the same time, the book achieves a timeless feeling, via its concise descriptions of service-oriented techniques and broadly applicable sample code that I'm sure will beautifully serve application architects and library authors alike for years to come.

—Obie Fernandez
Author of *The Rails Way*
Series Editor of the Addison-Wesley Professional Ruby Series
CEO & Founder of Hashrocket

Preface

As existing Ruby on Rails deployments grow in size and adoption expands into larger application environments, new methods are required to interface with heterogeneous systems and to operate at scale. While the word *scalability* with respect to Rails has been a hotly debated topic both inside and outside the community, the meaning of the word *scale* in this text is two fold. First, the traditional definition of "handling large numbers of requests" is applicable and something that the service-oriented approach is meant to tackle. Second, *scale* refers to managing code bases and teams that continue to grow in size and complexity. This book presents a service-oriented design approach that offers a solution to deal with both of these cases.

Recent developments in the Ruby community make it an ideal environment for not only creating services but consuming them as well. This book covers technologies and best practices for creating application architectures composed of services. These could be written in Ruby and tied together through a frontend Rails application, or services could be written in any language, with Ruby acting as the glue to combine them into a greater whole. This book covers how to properly design and create services in Ruby and how to consume these and other services from within the Rails environment.

Who This Book Is For

This book is written with web application and infrastructure developers in mind. Specific examples cover technologies in the Ruby programming ecosystem. While the code in this book is aimed at a Ruby audience, the design principles are applicable to environments with multiple programming languages in use. In fact, one of the advantages of the service-oriented approach is that it enables teams to implement pieces of application logic in the programming language best suited for the task at hand. Meanwhile, programmers in any other language can take advantage of these

services through a common public interface. Ultimately, Ruby could serve simply at the application level to pull together logic from many services to render web requests through Rails or another preferred web application framework.

If you're reading this book, you should be familiar with web development concepts. Code examples mainly cover the usage of available open source Ruby libraries, such as Ruby on Rails, ActiveRecord, Sinatra, Nokogiri, and Typhoeus. If you are new to Ruby, you should be able to absorb the material as long as you have covered the language basics elsewhere and are generally familiar with web development. While the topic of service-oriented design is usually targeted at application architects, this book aims to present the material for regular web developers to take advantage of service-based approaches.

If you are interested in how Ruby can play a role in combining multiple pieces within an enterprise application stack, you will find many examples in this book to help achieve your goals. Further, if you are a Rails developer looking to expand the possibilities of your environment beyond a single monolithic application, you will see how this is not only possible but desirable. You can create systems where larger teams of developers can operate together and deploy improvements without the problem of updating the entire application at large.

The sections on API design, architecture, and data backends examine design principles and best practices for creating services that scale and are easy to interface with for internal and external customers. Sections on connecting to web services and parsing responses provide examples for those looking to write API wrappers around external services such as SimpleDB, CouchDB, or third-party services, in addition to internal services designed by the developer.

What This Book Covers

This book covers Ruby libraries for building and consuming RESTful web services. This generally refers to services that respond to HTTP requests. Further, the APIs of these services are defined by the URIs requested and the method (`GET`, `PUT`, `POST`, `DELETE`) used. While the focus is on a RESTful approach, some sections deviate from a purist style. In these cases, the goal is to provide clarity for a service API or flexibility in a proposed service.

The primary topics covered in this book are as follows:

- REST, HTTP verbs, and response codes
- API design

- Building services in Ruby
- Connecting to services
- Consuming JSON- and XML-based services
- Architecture design
- Messaging and AMQP
- Securing services

What This Book Doesn't Cover

Service-oriented architectures have been around for over a decade. During this time, many approaches have been taken. These include technologies with acronyms and buzzwords such as SOAP, WSDL, WS-*, and XML-RPC. Generally, these require greater overhead, more configuration, and the creation of complex schema files. Chapter 9, "Parsing XML for Legacy Services," provides brief coverage of consuming XML and SOAP services. However, SOAP, XML-RPC, and related technologies are beyond the scope of this book. The services you'll create in this book are lightweight and flexible, like the Ruby language itself.

This book also does not cover other methods for building complex architectures. For example, it does not cover batch processing frameworks such as MapReduce or communications backends such as Thrift. While these technologies can be used in conjunction with a web services approach, they are not the focus. However, Chapter 11, "Messaging," briefly covers messaging systems and message queues.

Additional Resources

Code examples are used heavily throughout this book. While every effort has been made to keep examples current, the open source world moves fast, so the examples may contain code that is a little out-of-date. The best place to find up-to-date source code is on GitHub, at the following address:

```
http://github.com/pauldix/service-oriented-design-with-ruby
```

In addition, you can subscribe to a mailing list to discuss the code, text, services design, and general questions on the topic of service-oriented design. You can join here:

```
http://groups.google.com/group/service-oriented-design-
  with-ruby
```

This page intentionally left blank

Acknowledgments

An unbelievable number of people contributed to this book through writing, editing, conversations about the content, or just moral support, so please forgive me if I leave anyone out. First, I need to thank Lindsey for putting up with my ridiculous schedule while writing this book and working a full-time job. Thanks to Trotter Cashion for writing Chapter 10, "Security"; to Bryan Helmkamp for writing Chapter 8, "Load Balancing and Caching"; and to Jake Howerton for writing Chapter 12, "Web Hooks and External Services." Thanks to Trotter again and Jennifer Linder for providing excellent editing work and making sure that it makes sense. Thanks to Debra, my editor at Addison-Wesley, and to Michael, my development editor at AW. Thanks to the NYC.rb crew for being smart, fun people to hang out and discuss ideas with. Thanks to the entire team at KnowMore for putting up with me while I wrote and helping me refine my thinking. Finally, thanks to my business partner, Vivek, for providing encouragement during the final editing stages.

This page intentionally left blank

About the Author

Paul Dix is co-founder and CTO at Market.io. In the past, he has worked at Google, Microsoft, McAfee, Air Force Space Command, and multiple startups, filling positions as a programmer, software tester, and network engineer. He has been a speaker at multiple conferences, including RubyConf, Goruco, and Web 2.0 Expo, on the subjects of service-oriented design, event-driven architectures, machine learning, and collaborative filtering. Paul is the author of multiple open source Ruby libraries. He has a degree in computer science from Columbia University.

This page intentionally left blank

CHAPTER 1

Implementing and Consuming Your First Service

In the grand tradition of programming books beginning with a "hello, world" example, this book starts off with a simple service. This chapter walks through the creation of a service to store user metadata and manage authentication. This includes building the web service to handle HTTP requests and the client library for interacting with the service.

What's a Service?

In this book, *service* generally refers to a system that responds to HTTP requests. Such HTTP requests are usually to write, retrieve, or modify data. Examples of public-facing HTTP services include Twitter's API (http://apiwiki.twitter.com), the Amazon S3 service (http://aws.amazon.com/s3/), the Delicious API (http://delicious.com/help/api), the Digg API (http://apidoc.digg.com), and the New York Times APIs (http://developer.nytimes.com/docs). Internally, services could exist to contain pieces of data and business logic that are used by one or more applications.

Using a broader scope of definition, *service* can refer to a system that provides functionality through a standard interface. Working at this level of abstraction are services such as relational databases (for example, MySQL), Memcached servers, message queues (for example, RabbitMQ), and other types of data stores, such as Cassandra (http://incubator.apache.org/cassandra/).

While this book touches on the broader definition of *service* in a few places, the majority of the material focuses on HTTP-based services. More specifically, this book focuses on services that are designed to roughly follow a RESTful paradigm, as described in the appendix, "RESTful Primer." Further, this book focuses on using services within an organization and infrastructure to build out applications. These services may or may not be public facing, like the previous examples.

The details of why, when, and how to use services are covered throughout the course of the book. For now the goal is to implement a simple service.

Service Requirements

A simple user management system is an example of something that can be pulled out as a service. After implementation, this service could be used across multiple applications within an organization as a single sign-on point. The goals and requirements of the service are fairly simple:

- Store user metadata, including name, email address, password, and bio.
- Support the full range of CRUD (create, update, delete) operations for user objects.
- Verify a user login by name and password.

In later versions of the user service, features could map which users work with each other, which user each user reports to, and which groups a user is a member of. For now, the basic feature set provides enough to work on.

The Ruby Tool Set

Ruby provides many tools to build both the service and client sides of services. However, this book heavily favors some specific tools due to their aesthetics or performance characteristics. The following libraries appear often throughout this book.

Sinatra

Sinatra is a lightweight framework for creating web applications. It can be described as a domain-specific language for web applications and web services. Built on top of Rack, Sinatra is perfectly suited for creating small web services like the example in this chapter. In addition to encouraging an elegant code style, Sinatra has fewer than 2,000

lines of code. With this small and readable code base, it's easy to dig through the internals to get a more specific idea of what's going on under the hood.

Sinatra was originally written by Blake Mizerany, and continued development is supported by Heroku. The official web site is http://www.sinatrarb.com, and the code repository is on GitHub, at http://github.com/sinatra/sinatra. Chapter 4, "Service and API Design," provides more in-depth coverage of Sinatra. For now, Sinatra can be installed to work through the example in this chapter using the `gem` command on the command line, like this:

```
gem install sinatra
```

ActiveRecord

ActiveRecord is the well-known object/relational mapper (ORM) that is an integral part of Ruby on Rails. It provides a simple interface for mapping Ruby objects to the MySQL, PostgreSQL, or SQLite relational databases. Since most readers are probably familiar with ActiveRecord, the choice to use it as the data library was easy. However, the focus of this book is on creating service interfaces, creating clients, and organizing service interactions. Which data store or library to use is beyond the scope of this book. Readers experienced with alternative data stores are welcome to use them in place of ActiveRecord.

ActiveRecord was originally developed by David Heinemeier Hansson as a part of Ruby on Rails. It is an implementation of the ActiveRecord design pattern by Martin Fowler (http://www.martinfowler.com/eaaCatalog/activeRecord.html). The documentation can be found at http://ar.rubyonrails.org, and the source code is part of the Rails repository on GitHub, at http://github.com/rails/rails/tree/master/activerecord/. ActiveRecord can be installed using the `gem` command on the command line, like this:

```
gem install activerecord
```

JSON

The representation of resources from HTTP services can be in any of a number of formats. HTML, XML, and JSON are the most common. JSON is quickly becoming a favorite choice because of its speed and simplicity as well as the availability of quality parsers in most languages. JSON includes built-in types such as strings, integers, floats, objects (such as Ruby hashes), and arrays. Most complex data types can be represented fairly easily and succinctly by using these basic data structures.

There is no shortage of available JSON parsers for Ruby. The most popular option is the JSON Ruby implementation found at http://flori.github.com/json/. However, Brian Marino's Ruby bindings to YAJL (yet another JSON library) look like a very solid option that can provide some performance increases. Marino's code can be found at http://github.com/brianmario/yajl-ruby, and the YAJL code can be found at http://lloyd.github.com/yajl/. For simplicity, the service example in this chapter uses the JSON Ruby implementation, which can be installed using the `gem` command on the command line, like this:

```
gem install json
```

Typhoeus

The client libraries for services must use an HTTP library to connect to the server. Typhoeus is an HTTP library specifically designed for high-speed parallel access to services. Being able to run requests in parallel becomes very important when connecting to multiple services. Typhoeus includes classes to wrap requests and response logic as well as a connection manager to run requests in parallel. It also includes raw bindings to the libcurl and libcurl-multi libraries that make up its core functionality.

Typhoeus was originally written by me, and ongoing support is provided at http://KnowMore.com. The code and documentation can be found at http://github.com/pauldix/typhoeus/, and the support mailing list is at http://groups.google.com/group/typhoeus. Typhoeus is covered in greater detail in Chapter 6. "Connecting to Services." For now, it can be install using the gem command line, like this:

```
gem install typhoeus
```

Rspec

Testing should be an integral part of any programming effort. This book uses Rspec as its preferred testing library. It provides a clean, readable domain-specific language for writing tests.

Rspec is written and maintained by the core team of Dave Astels, Steven Baker, David Chemlimsky, Aslak Hellesøy, Pat Maddox, Dan North, and Brian Takita. Coverage of Rspec is beyond the scope of this book. However, detailed documentation and examples can be found on the Rspec site, at http://rspec.info. Rspec can be installed using the `gem` command on the command line, like this:

```
gem install rspec
```

The User Service Implementation

With the list of tools to build the service and client libraries chosen, you're ready to implement the service. The server side of the system is the first part to build. Remember that this is a Sinatra application. Unlike Rails, Sinatra doesn't come with generators to start new projects, so you have to lay out the application yourself. The basic directory structure and necessary files should look something like the following:

```
/user-service
  /config.ru
  /config
    database.yml
  /db
    /migrate
  /models
  /spec
  Rakefile
```

The `user-service` directory is the top level of the service. The `config` directory contains `database.yml`, which ActiveRecord uses to make a database connection. `config.ru` is a configuration file that Rack uses to start the service. The `db` directory contains the migrate scripts for models. The models directory contains any ActiveRecord models. `Rakefile` contains a few tasks for migrating the database.

The `database.yml` file looks like a standard Rails database configuration file:

```
development:
  adapter: sqlite3
  database: db/development.sqlite3

test:
  adapter: sqlite3
  database: db/test.sqlite3
```

`Rakefile` contains the task to migrate the database after you've created the user migration:

```
require 'rubygems'
require 'active_record'
require 'yaml'
```

```
desc "Load the environment"
task :environment do
  env = ENV["SINATRA_ENV"] || "development"
  databases = YAML.load_file("config/database.yml")
  ActiveRecord::Base.establish_connection(databases[env])
end

namespace :db do
  desc "Migrate the database"
  task(:migrate => :environment) do
    ActiveRecord::Base.logger = Logger.new(STDOUT)
    ActiveRecord::Migration.verbose = true
    ActiveRecord::Migrator.migrate("db/migrate")
  end
end
```

First, the dependencies are loaded. Then the `:environment` task is created. This makes a connection to the database based on what environment is being requested. Finally, the `:db` namespace is defined with the `:migrate` task. The migrate task calls the migrate method on Migrator, pointing it to the directory the database migrations are in.

With all the basic file and directory scaffolding out of the way, you can now spec and create the service. The specs for the service define the behavior for expected interactions and a few of the possible error conditions. The specs described here are by no means complete, but they cover the primary cases.

Using `GET`

The most basic use case for the server is to return the data about a single user. The following sections outline the behavior with specs before starting the implementation.

Spec'ing `GET` User

To get the specs started, you create a file in the `/spec` directory called `service_spec.rb`. The beginning of the file and the user `GET` specs look like this:

```
require File.dirname(__FILE__) + '/../service'
require 'spec'
require 'spec/interop/test'
require 'rack/test'
```

```
set :environment, :test
Test::Unit::TestCase.send :include, Rack::Test::Methods

def app
  Sinatra::Application
end

describe "service" do
  before(:each) do
    User.delete_all
  end

  describe "GET on /api/v1/users/:id" do
    before(:each) do
      User.create(
        :name => "paul",
        :email => "paul@pauldix.net",
        :password => "strongpass",
        :bio => "rubyist")
    end

    it "should return a user by name" do
      get '/api/v1/users/paul'
      last_response.should be_ok
      attributes = JSON.parse(last_response.body)
      attributes["name"].should == "paul"
    end

    it "should return a user with an email" do
      get '/api/v1/users/paul'
      last_response.should be_ok
      attributes = JSON.parse(last_response.body)
      attributes["email"].should == "paul@pauldix.net"
    end

    it "should not return a user's password" do
      get '/api/v1/users/paul'
      last_response.should be_ok
      attributes = JSON.parse(last_response.body)
      attributes.should_not have_key("password")
    end

    it "should return a user with a bio" do
      get '/api/v1/users/paul'
      last_response.should be_ok
```

```
      attributes = JSON.parse(last_response.body)
      attributes["bio"].should == "rubyist"
    end

    it "should return a 404 for a user that doesn't exist" do
      get '/api/v1/users/foo'
      last_response.status.should == 404
    end
  end
end
```

The first 11 lines of the file set up the basic framework for running specs against a Sinatra service. The details of each are unimportant as you continue with the user specs.

There are a few things to note about the tests in this file. First, only the public interface of the service is being tested. Sinatra provides a convenient way to write tests against HTTP service entry points. These are the most important tests for the service because they represent what consumers see. Tests can be written for the models and code behind the service, but the consumers of the service really only care about its HTTP interface. Testing only at this level also makes the tests less brittle because they aren't tied to the underlying implementation.

That being said, the test still requires a user account to test against. This introduces an implementation dependency in the tests. If the service were later moved from DataMapper to some other data library, it would break the test setup. There are two possible options for dealing with setting up the test data.

First, the service could automatically load a set of fixtures when started in a test environment. Then when the tests are run, it would assume that the necessary fixture data is loaded. However, this would make things a little less readable because the setup of preconditions would be outside the test definitions.

The second option is to use the interface of the service to set up any preconditions. This means that the user `create` functionality would have to work before any of the other tests could be run. This option is a good choice when writing a service where the test data can be set up completely using only the API. Indeed, later tests will use the service interface to verify the results, but for now it's easier to work with the user model directly to create test data.

Each of the successful test cases expects the response to contain a JSON hash with the attributes of the user. With the exception of the "user not found" test, the

tests verify that the individual attributes of the user are returned. Notice that each attribute is verified in its own test. This style is common in test code despite its verbosity. When a failure occurs, the test shows exactly which attribute is missing.

The spec can be run from the command line. While in the `user-service` directory, you run the following command:

```
spec spec/service_spec.rb
```

As expected, the spec fails to run correctly before it even gets to the specs section. To get that far, the user model file and the basic service have to be created.

Creating a User Model

To create the user model, a migration file and a model file need to be created. You create a file named `001_create_users.rb` in the `/db/migrate` directory:

```
class CreateUsers < ActiveRecord::Migration
def self.up
    create_table :users do |t|
      t.string :name
      t.string :email
      t.string :password
      t.string :bio

      t.timestamps
    end
  end

  def self.down
    drop_table :users
  end
end
```

The file contains the ActiveRecord migration logic to set up the users table. The fields for the name, email address, password, and bio fields are all there as string types.

When the migration is done, you can add the user model. You create a file called `user.rb` in the `/models` directory:

```
class User < ActiveRecord::Base
  validates_uniqueness_of :name, :email
```

```
  def to_json
    super(:except => :password)
  end
end
```

The model contains only a few lines. There is a validation to ensure that the name and email address of the user are unique. The `to_json` method, which will be used in the implementation, has been overridden to exclude the password attribute. This user model stores the password as a regular string to keep the example simple. Ordinarily, a better solution would be to use Ben Johnson's Authlogic (http://github.com/binarylogic/authlogic). The primary benefit of Authlogic in this case is its built-in ability to store a salted hash of the user password. It is a big security risk to directly store user passwords, and using a popular tested library reduces the number of potential security holes in an application.

Implementing `GET` User

With the model created, the next step is to create the service and start wiring up the public API. The interface of a service is created through its HTTP entry points. These represent the implementation of the testable public interface.

In the main `user-service` directory, you create a file named `service.rb` that will contain the entire service:

```
require 'rubygems'
require 'activerecord'
require 'sinatra'
require 'models/user'

# setting up the environment
env_index = ARGV.index("-e")
env_arg = ARGV[env_index + 1] if env_index
env = env_arg || ENV["SINATRA_ENV"] || "development"
databases = YAML.load_file("config/database.yml")
ActiveRecord::Base.establish_connection(databases[env])

# HTTP entry points
# get a user by name
get '/api/v1/users/:name' do
  user = User.find_by_name(params[:name])
  if user
```