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

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The author and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact:

U.S. Corporate and Government Sales
(800) 382-3419
corpsales@pearsontechgroup.com

For sales outside the United States please contact:

International Sales
international@pearson.com

Visit us on the Web: informit.com/aw

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

Copyright © 2011 Pearson Education, Inc.

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, write to:

Pearson Education, Inc
Rights and Contracts Department
501 Boylston Street, Suite 900
Boston, MA 02116
Fax: (617) 671-3447

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

Contents

Foreword xiii

Preface xv

Acknowledgments xix

About the Author xxi

1 Implementing and Consuming Your First Service 1

What's a Service? 1
Service Requirements 2
The Ruby Tool Set 2
- Sinatra 2
- ActiveRecord 3
- JSON 3
- Typhoeus 4
- Rspec 4

The User Service Implementation 5
- Using GET 6
- POSTing a User 11
- PUTing a User 13
- Deleting a User 15
- Verifying a User 16

Implementing the Client Library 18
- Finding a User 18
- Creating a User 21
- Updating a User 22
- Destroying a User 24
- Verifying a User 24

Putting It All Together 26
Conclusion 26

2 An Introduction to Service-Oriented Design 27

Use of Service-Oriented Design in the Wild 27

Service-Oriented Design Versus Service-Oriented Architecture Versus RESTful-Oriented Architecture 28

Making the Case for Service-Oriented Design 29

- Isolation 30
- Robustness 34
- Scalability 35
- Agility 36
- Interoperability 37
- Reuse 38

Conclusion 38

3 Case Study: Social Feed Reader 41

A Typical Rails Application 41

The Rails Social Feed Reader Application 45

- Features 46
- Current Setup 46

Converting to Services 54

- Segmenting into Services 54
- Breaking Up the Application into Services 54

Conclusion 58

4 Service and API Design 59

Partitioning Functionality into Separate Services 59

- Partitioning on Iteration Speed 60
- Partitioning on Logical Function 61
- Partitioning on Read/Write Frequencies 62
- Partitioning on Join Frequency 63

Versioning Services 64

- Including a Version in URIs 64
- Using Accept Headers for Versioning 65

URIs and Interface Design 66

Successful Responses 68

- HTTP Status Codes 68
- HTTP Caching 69
- Successful Response Bodies 70

Error Responses 72
HTTP Status Codes 72
Error Response Bodies 72
Handling Joins 73
Storing References 73
Joining at the Highest Level 74
Beware of Call Depth 75
API Complexity 75
Atomic APIs 76
Multi-Gets 76
Multiple Models 77
Conclusion 78

5 Implementing Services 79
The Vote Service 79
A Multi-Get Interface 81
The Vote Interface 82
API Design Guidelines 85
Models 86
Rails 88
Rails 2.3 Routes 88
Rails 3 Routes 89
The Rails Controller 90
Sinatra 95
Rack 100
Conclusion 106

6 Connecting to Services 107
Blocking I/O, Threading, and Parallelism 107
Asynchronous I/O 108
Multi-threading 108
Typhoeus 109
Making Single Requests 109
Making Simultaneous Requests 111
Multi-threaded Requests 113
JRuby 115
Logging for Performance 117

Handling Error Conditions 118
Testing and Mocking Service Calls 119
Requests in Development Environments 121
Conclusion 121

7 Developing Service Client Libraries 123
Packaging 123
    Jeweler 124
    Building and Deploying a Library 127
Parsing Logic 127
    The JSON Gem 128
    YAJL Ruby 129
    Wrapping Parsed Results 130
ActiveModel 132
    Validations 132
    Serialization 134
Connection and Request Logic 136
    Data Reads 136
    Data Writes 142
Mocks, Stubs, and Tests 143
Conclusion 146

8 Load Balancing and Caching 147
Latency and Throughput 147
Load Balancing 148
    Load Balancing Algorithms 148
    Implementing Load Balancing 152
Caching with Memcached 155
    The Memcached Client and ActiveRecord 156
    Time-Based Expiration 158
    Manual Expiration 159
    Generational Cache Keys 160
HTTP Caching 162
    Expiration-Based Caching 162
    Validation-Based Caching 163
    Implementing HTTP Caching 165
Conclusion 166

9 Parsing XML for Legacy Services 167
XML 167
REXML 170
Nokogiri 174
SOAP 177
Exploring Web Services with a WSDL File 177
Making Requests 180
Conclusion 184

10 Security 185
Authentication 185
HTTP Authentication 186
Signing Requests 187
SSL for Authentication 198
Authentication for Rails Applications 199
Authorization 201
Firewalls 201
An RBAC Authorization Service 203
Encryption 209
SSL for Encryption 210
Public/Private Key Pairs for Encryption 210
Conclusion 214

11 Messaging 215
What Is Messaging? 215
Synchronous Versus Asynchronous Messaging 216
Queues 217
Message Formats 217
RabbitMQ and AMQP 217
Queues in RabbitMQ 218
Exchanges and Bindings 218
Durability and Persistence 223
Client Libraries 224
Synchronous Reads, Asynchronous Writes 227
HTTP-Based Reads 227
Messaging-Based Writes 227
The CAP Theorem 230
Eventual Consistency 231
Designing Around Consistency 232

Data Is the API 234
- Operations on Fields 234
- Modifications to Field Operations 235

Conclusion 236

12 Web Hooks and External Services 237

Web Hooks 238
- PubSubHubbub 239
- Receiving Web Hooks 240
- Providing Web Hooks 242
- Strategies for Dealing with Failure 244

OAuth 245
- Implementing an OAuth Consumer 246
- Implementing an OAuth Provider 249

Integrating with External Services 251
- Consuming Data 251
- Pushing Data 253
- The Request Lifecycle 254
- Worker Processes 254

Ensuring Performance and Reliability 258
- Segregating Queues 259
- Metrics 259
- Throttling and Quotas 260

Conclusion 261

Appendix RESTful Primer 263

Roy Fielding's REST 263
- Constraints 264
- Architectural Elements 264
- Architectural Views 265

REST and Resources 265
- URIs and Addressability 266
- Representations 267

HTTP and the Uniform Interface 268
- HTTP Methods 268
- HTTP Headers 271
- HTTP Status Codes 274

Conclusion 275

Index 277

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
```

CHAPTER 2

An Introduction to Service-Oriented Design

Service-oriented design is about creating systems that group functionality around logical function and business practices. Services should be designed to be interoperable and reusable. The goal of service-oriented design is to split up the parts of an application or system into components that can be iterated on, improved, and fixed without having to test and verify all the other components when an individual is updated. Achieving these goals usually entails a trade-off between complexity and iteration speed. However, large and mature applications are ill-served when built in Rails monolithic style. It is necessary to segment complex or large applications into parts that can be tested and deployed separately. This chapter explores the basic goals of service-oriented design and design guidelines for splitting applications into separate services.

Use of Service-Oriented Design in the Wild

Organizations such as eBay, Amazon, LinkedIn, and other large web-based companies use layers of services to bring their applications together. While many of these environments are based in Java, the advantages that come from their approaches to architecture design can be applied to web applications and systems written in Ruby.

The architecture of Amazon most exemplifies the advantages of good service-oriented design. In May 2006 the Association for Computing Machinery (ACM) published an interview between Jim Gray and Amazon CTO Werner Vogels titled

"A Conversation with Werner Vogels."[1] In the interview Mr. Vogels states that when a user goes to the Amazon.com home page, the application calls out to more than 100 services to construct the page.

Mr. Vogels goes on to say that the move from a monolithic two-tier (database and web application) architecture to a service-oriented approach provides many advantages. These include improvements such as scalability, isolation, and developer ownership of production systems. Further, Vogels states that this has led to improved processes and increased organizational agility to develop new services and features. Amazon's service-oriented architecture has enabled the introduction of new applications and services without requiring reconfiguration of the entire system.

Amazon's approach to its internal systems has driven and informed the development of the Amazon Web Services (AWS) platforms. Every piece of the AWS architecture is exposed as a web service. Here's a breakdown of Amazon's current services:

- **S3 (Simple Storage Service)**—A service for storing files.
- **SQS (Simple Queue Service)**—A service-based messaging queue.
- **SimpleDB**—A scalable service-based database.
- **CloudFront**—A service-based content delivery network.
- **EC2 (Elastic Compute Cloud)**—A service for provisioning virtual private servers.

The AWS platform represents an example of low-level system components exposed through a services layer. A service-oriented design can take advantage of these types of lower-level components as well as services that operate a little further up the stack that provide functionality for a specific application. Higher-level services might include a user system, a comments service, a video transcoding service, and many others.

Service-Oriented Design Versus Service-Oriented Architecture Versus RESTful-Oriented Architecture

There are many approaches to designing a service-oriented application. This book takes an approach slightly different than those espoused in Java books on service-oriented architecture (SOA) or even other books that focus on REST. Within the community of professionals building service-based systems there is a bit of debate about the proper use of terminology and what qualifies as best practices.

[1] O'Hanlon, Charlene. "A Conversation with Werner Vogels." *ACM Queue.* Volume 4, Issue 4, May 2006, pp. 14–22. http://queue.acm.org/detail.cfm?id=1142065.

SOA has become a loaded term. To many people, it implies the use of tools such as SOAP, WSDL, WS-*, or XML-RPC. This implication is why the title of this book uses the word *design* as opposed to *architecture*. Even so, this book does focus on architecture. The real goal of service-oriented design is to create simple and agile services layers that can be consumed without the use of generators or strict schemas. In this regard, the style in this book is more in line with REST-based web services than with SOA.

In the book *RESTful Web Services*,[2] Leonard Richardson and Sam Ruby discuss the details for a concept they call resource-oriented architecture (ROA). ROA represents their approach for HTTP-based RESTful design and architecture. Richardson and Ruby lay out the concepts of resources, URIs, representations, and links. They also state that ROA has properties of addressability, statelessness, and connectedness, as well as a uniform interface. When it comes to the design of specific services, this book follows Richardson and Ruby's guidelines. (The appendix, "RESTful Primer," provides an overview of REST.)

The real difference between the focus of this book and that of *RESTful Web Services* lies in the interaction points—that is, how services interact with each other to create a complete working application. The focus of this book is on internal services instead of external ones. While some of an application's services can be exposed to external consumers, their real purpose is to serve other developers working within a single organization. In this regard, this book has more similarity to what one would commonly refer to as SOA. This book also includes a more specific focus on deploying and developing with Rails, Sinatra, and other Ruby libraries and tools.

Of course, SOA, ROA, and RESTful are all meant as guidelines. In the real world, it makes sense to flex a design with the needs of the application rather than adhere to the dogma of a prescribed approach such as REST or SOA. This book takes a pragmatist's view and focuses on what these things mean for a Ruby environment that uses services as part of the core infrastructure for delivering an application or a web page to a user.

Making the Case for Service-Oriented Design

Service-oriented design can appear daunting and complex. It requires more thought up front and decisions about how to separate logic and data in an application. For Rails developers, the idea of designing a complex system ahead of development may

[2] Richardson, Leonard, and Ruby, Sam. *RESTful Web Services*. Sebastopol, CA: O'Reilly, 2007.

seem like heresy. One of the biggest advantages of Rails is the ability to quickly add a few models to an application and see results immediately. The Rails development style is all about quick iterations.

However, there's more to the story. Up-front design and services give developers the ability to build apps that support greater complexity and larger team sizes. Service-oriented systems sacrifice iteration speed for stability, reuse, and robustness. The key to pairing Rails applications with services is to use Rails for its strengths and switch over to services when a more stable approach is required. A perfect example of this involves creating a new application. Most new applications have many unknowns in terms of exactly what features will be supported and how popular portions of the application will be (thus informing their need for scale). In the early stages, it is best to use the normal Rails tool set. However, as parts of an application mature, their interfaces and requirements become more concrete. These are the sections that can gain the most from services. Utilizing services is best for sections of an application that have stable, well-defined, and well-understood requirements. The following sections discuss the advantages of using services rather than using a typical monolithic application.

Isolation

Many of the benefits of a service-oriented design stem from the concept of isolation. Isolation makes a service much easier to manage and optimize. Isolated components can be tested separately from other parts of an application. Using isolated components provides an easy way of organizing larger teams. Developers can focus on isolated components. Optimally, this refers to a service running on its own systems, with self-contained business logic and a self-contained data store. The separation of a service from other areas of an application enables increased testability and code reuse. There are multiple levels of isolation, including business logic, shared system, and full isolation.

Business Logic Isolation

Services that isolate based on business logic generally have their own application code, with a shared data store and shared systems. From an organizational perspective, this can be advantageous because the business logic for parts of the system is contained in one place, without leaking into other sections of the application code base. Separation of business logic makes it easier to segment a larger group of workers into teams that can work separately. Services isolated on business logic can share data sources with other systems. Generally, this is more common within a legacy system where multiple services must interface with the same database.

Figure 2.1 shows what business logic isolation might look like for the interactions between separate components. The application servers would probably reside on the same physical server, with the database on another. To achieve true business logic

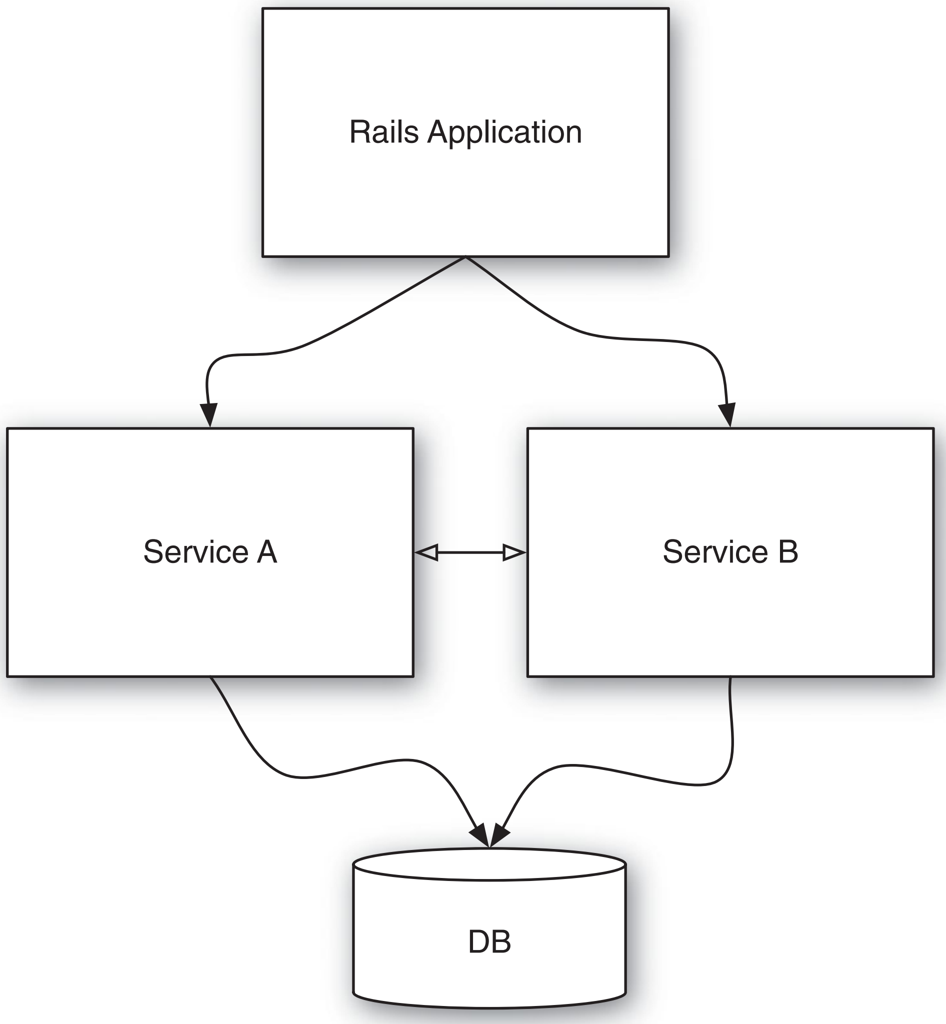

Figure 2.1 Services with separate business logic and a shared database.

isolation, the two services should have separate code bases. Further, they should not communicate with each other through the database. It's too easy to bleed business logic from the two services together through the shared database. Ideally, using services would achieve better isolation. However, for the purposes of migrating existing Rails applications to services, the shared database approach may be necessary in the early stages.

The business logic can be isolated through the use of two services, which share a database. The Rails application can still sit on top of those services. In the Rails MVC view of the world, these services occupy the Model level of the stack. The controllers and views can still be contained within the Rails application.

Shared System Isolation

Shared system isolation refers to separate services running inside their own application instances. This is like multiple Rails applications or Sinatra services running in Passenger or multiple Mongrels on the same system. Each would have its own databases, but they would be running on the same hardware. This type of system provides clean separation of business logic and data layers that is ideal. However, it changes your scaling strategy because of the shared system resources.

Figure 2.2 shows the interaction for two services that implement a shared system level of isolation. The difference between this separation and the business logic isolation just discussed is the separation of the databases. Now, each of the services communicates only with its own database and the external interface of the other service. A typical configuration would have the two databases actually residing on the same database server and the two services running on the same application server. A shared hosting environment is an example of this kind of setup. However, with shared hosting, the two services are actually two different customer applications that have nothing to do with each other.

The disadvantage with shared system isolation is that shared system resources can be tricky to manage. Further, shared system isolation adds complexity with upgrading libraries or running other system-level applications. Thus, improvements to shared system services require testing against other services when making system changes.

Full Isolation

Ideally, services should run in full isolation. With full isolation, they have their own server or virtual private server instances, completely separate code bases and repositories, and their own data stores. Over time, a system could phase from one form of

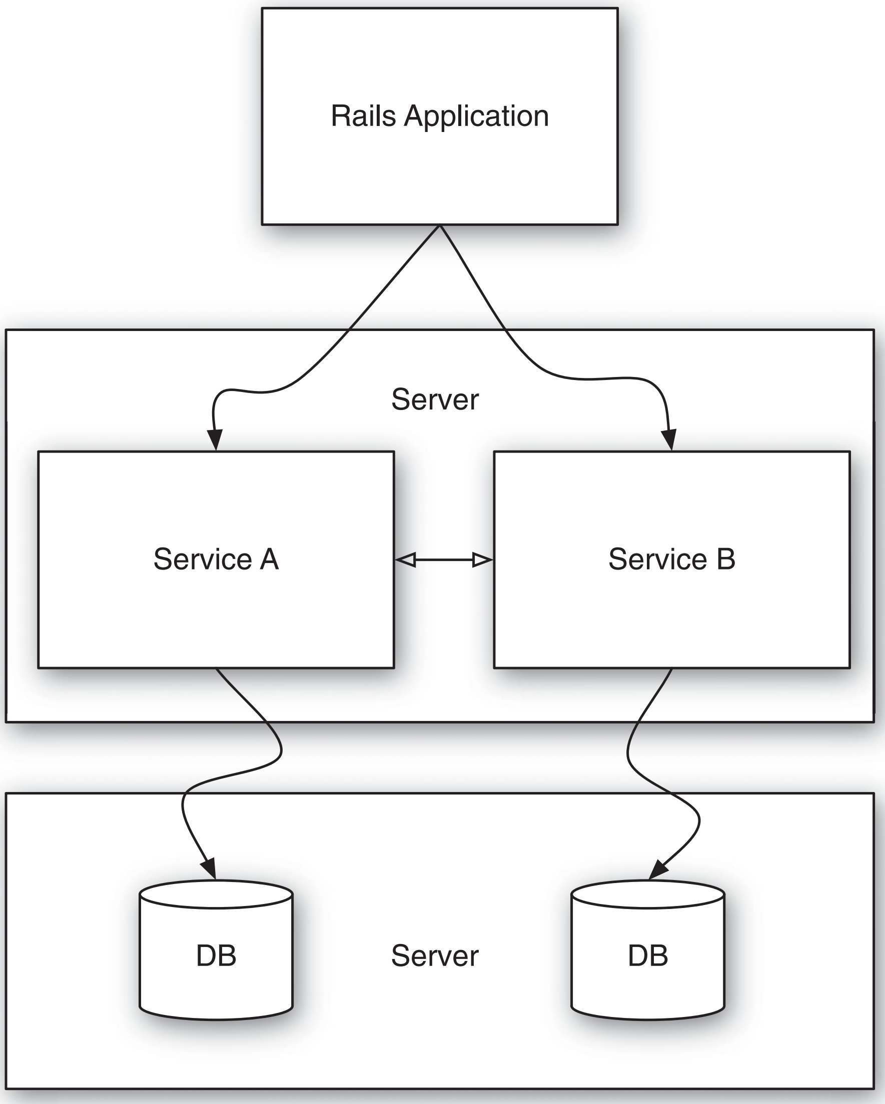

Figure 2.2 Shared systems with isolated services.

isolation to another. Migrating from a typical monolithic Rails application could start with a migration to a shared database and systems, then to isolated databases, and finally to completely isolated systems.

Testing in Isolation

Within the framework of testing, isolation provides the advantage of having a single testable interface. A public service API can be checked and agreed upon. Changes within the service need not affect how the API responds to commands. The testing of a service's interface is very much like unit or model testing within a Rails application. The only difference is that the callable methods are HTTP methods that can be called by any application. All the details of the business logic and how data is stored or optimized are hidden behind the service interface.

The big advantage isolation provides with regard to testing is in the time required to run the test suite. Rails developers working on mature projects with large code bases routinely have to wait longer than 15 minutes for their test suites to run to completion. In fact, it's not entirely uncommon for some teams to work with large code bases that take longer than 40 minutes to run their entire suite of tests. This becomes a major problem when you want to change something small. After even a small change, the full suite of tests must be run to ensure that the change didn't break anything.

With isolated application components, testing to make sure other parts of an application are functioning properly becomes a much smaller problem. For example, consider an application code base that takes 20 minutes to run the full suite of tests. Then break that application into four fairly evenly sized, separate, isolated components. Now the test time for a single change to one of these components is cut down to one-fourth of what was previously needed. If a change is made to one of the new components, it isn't necessary to run the test suite for everything. Only the tests for the single component need to be run, and in this example, that would take roughly 5 minutes rather than 20. As long as the public-facing interface for the component is well tested, changes can be deployed without concern that the other three-fourths of the application still works. Of course, if the API of a service changes, then each consumer must be tested to ensure proper operation.

Robustness

Services that are well designed provide an application with a robust architecture. That is, the architecture is able to withstand stress on the system and changes in the operating environment without loss of functionality. The underlying environment in which services run can change while a service continues to operate without the service consumers having any knowledge of these changes.

For those familiar with object-oriented design, the robustness advantages of services may sound similar to the advantages of encapsulation. Indeed, with services, the aim is to achieve encapsulation for entire sections of an application. In object-oriented design, encapsulation means that the underlying implementation can be changed without the API consumer's knowledge. For services, such changes can include code changes and more drastic changes, such as moving to a different type of database.

For example, consider a user management service. To start with, it includes only basic functionality, such as a user model, authentication, profile data, and a lightweight social network (where people can be friend each other). The initial implementation could use ActiveRecord and a MySQL database. As the load on the service picks up, it starts to outgrow the limits of the regular SQL solution. Because there is a clearly defined services interface, this can be modified without a single change to the rest of the application.

Switching underlying data stores could go something like this: After some hand wringing and careful research, you might decide to move the user data to some NoSQL data store, such as CouchDB, Redis, or MongoDB. First, you would update the implementation to use the new data store, along with migration scripts to move the data from MySQL to the new NoSQL database. Once a new server or two or three have been set up, you could migrate the code and data. When this migration is complete, the other sections of the application are still able to access the user management service without ever knowing about the complete change of the underlying data store.

Scalability

Experienced Rails developers tend to roll their eyes when people mention scalability. People outside the Rails community advise against using Rails because they say it isn't scalable. Meanwhile, Rails developers know that Rails itself isn't the problem. While it's true that scalability in general is difficult, the problem usually comes down to the database. A services approach provides more tools and ability to deal with scaling. Specifically, using services makes it easy to scale portions of an application individually. Data can be split across services, and the performance under load can be optimized for each service.

A partitioned data strategy is part of service-oriented system design. When fully isolating services, you need to make decisions about putting data in one service or another. Once data has been partitioned, changes can be made to the individual services based on their scaling needs. While one service may need to optimize for data writes, another may optimize for many reads. The advantage of a good service design is that these needs can be handled on a case-by-case basis instead of requiring optimization of a single database for all cases.

Services also make it easier to scale team size. When a programming team is larger than six people, it gets hard to coordinate changes in an application. One developer's change may step on another's. Further, as the code base of the application gets larger, it gets harder to introduce new developers to the system as a whole. When an application is split into services, developers can be assigned to a specific service or two. Thus, they need to be familiar only with their section of the application, and the working groups can remain small.

Finally, using services makes it easier to scale the absolute size of an application in terms of the code base. Larger applications have too much code for anyone to be familiar with it all at any given time. In addition, their tests take longer to run. For example, when developing Rails applications, a developer doesn't usually have to dig into the Rails code base. Rails provides a layer of abstraction that doesn't often need to be penetrated. Services can provide this same layer of abstraction for code and for actual production systems.

Agility

When thinking about complex architectures, *agility* is not the word that comes to mind. However, properly designed services can be far more agile than monolithic applications. Changes to the underlying nature of the services can be made without concern for the rest of the application. The pains of deployment can also be eased because deployments don't require the entire application to be updated with each change.

The ability to change underlying service implementations without affecting the rest of the application provides implementation agility. Switching databases or changing message queues or even changing languages can be done without worrying about the rest of the system. This kind of agility is often overlooked in the Rails community, but it becomes a huge asset to applications that mature over a period of years. Using services allows for changing or updating underlying libraries without having to dig through every part of the application code base to make sure everything is still working as expected.

In the Ruby community, a good example of an advantage offered by services is the planning for migration to Ruby 1.9. Services provide greater agility in making these kinds of updates. Services can be upgraded to 1.9 as the libraries they use are confirmed to work. Thus, services can take a phased approach to upgrading to use Ruby 1.9 and take advantage of its features.

One of the keys to maintaining agility in a service environment is proper versioning. Each service interface should be versioned when an update includes a breaking change. As long as the design includes the ability to run multiple versions of a service simultaneously, it's possible to keep somewhat agile with respect to interface changes. If an update to the service API is additive—that is, it doesn't change existing calls and only adds new functionality—the service can remain at the same version.

Interoperability

For large heterogeneous environments, interoperability is an important requirement. When working with multiple languages or interface with legacy databases, legacy systems, or external vendors, using services is a great way to connect with these systems. Web-based interfaces to these systems can provide the ability to flex with changes without breaking sections of an application. The HTTP interface also prevents being tied to a specific messaging implementation that might otherwise be used to communicate with these systems. Services ease interoperation with internal and external systems and with systems written in languages other than Ruby.

Internal interoperability refers to interfacing with systems written in different languages within an environment. Some of these systems already expose their functionality through a web API. Apache Solr, a Java-based indexing and search server, is a great example of a system that provides a service interface. By interacting with this interface, Ruby developers can take advantage of all the work done on this project without having to call into the Java code directly by using something like JRuby. The Solr interface is usually called by other services and applications within an environment.

External interoperability refers to the need to interface with external systems such as those from vendors. Many external services also provide a web-based interface for their customers. Examples include SimpleDB, SQS, SalesForce, Github, Lighthouse, Twitter, Facebook, and countless others. Writing clean, performant Ruby client libraries is key to bringing these services into an application. Writing client libraries is covered in detail in Chapter 6, "Connecting to Services," and Chapter 7, "Developing Service Client Libraries."

Environments with multiple languages in use benefit from the use of HTTP-based services. While Ruby is a great programming language, it isn't always the best tool for every job. If a section of an application would benefit from being implemented in Erlang, Scala, or even Java, HTTP services can provide the message bus for interaction between these disparate setups.

Reuse

After a service has been developed and deployed, it can be reused across the entire system. The argument for reuse is strongest in environments where multiple applications have common shared functionality, such as consultancies and large corporate environments that develop multiple internal applications for different users.

Consultancies that develop and host applications for their clients could reuse services. Currently, the most common model of code reuse across these applications is through the development of plug-ins or gems. Specific examples include user authentication, tagging systems, commenting systems, and searching. However, many of these could be developed and deployed as services. One of the possible gains to taking the service approach is the reuse of system resources across all clients. For example, a user management system could be implemented as a service (as in the example in Chapter 1, “Implementing and Consuming Your First Service”). This system could then be used across all client systems. If this is repeated for other shared functionality, new applications will have to implement and deploy only anything that is custom to their environment.

Providing public-facing APIs is another area where the services used to build a system internally can be reused. If services are created for internal use, they can be exposed later for general use. The popularity of the Twitter API shows that it can be advantageous to expose parts of an application through a RESTful HTTP interface. With the services approach, exposing application functionality to the outside world becomes as easy as simply opening up an already existing internal API to the public.

Conclusion

Hopefully, this introduction has whetted your appetite for exploring the service-oriented approach covered in this book. The extra design work and communication overhead of creating and using multiple services takes a little more effort than creating a typical Rails application. However, the benefits of a service-oriented design can far outweigh the costs associated with inter service communication and more up-front design.

Here’s a quick recap of the benefits of service-oriented design:

- **Isolation**—Robustness, scalability, and improved testing strategies all stem from the concept of isolation. Isolation gives an application architecture many of the advantages that encapsulation provides in object-oriented design.

- **Robustness**—Services are robust because their underlying implementation can change with shifting load requirements, libraries, and languages without detriment to the rest of the application.
- **Scalability**—When using services, you need to think up front about how to separate data and manage interaction. This partitioning of logic and data provides the ability to scale the size of the code base and team in addition to the number of requests to process.
- **Agility**—Upgrades to underlying system components are easier with services. Further, new versions of existing services and completely new services can be implemented outside the full architecture. This can provide much-needed agility for mature code bases where changes are typically expensive to verify with the rest of an app.
- **Interoperability**—Using HTTP-based services is a great way to expose the functionality of legacy applications or external vendors.
- **Reuse**—Service-oriented design enables reuse of components across multiple applications or clients.

Index

Numbers

1xx, HTTP status code, 275
2xx, HTTP status code, 275
3xx, HTTP status code, 275
4xx, HTTP status code, 275
5xx, HTTP status code, 275
200 -OK, HTTP status code, 68
201 -Created, HTTP status code, 69
202 -Accepted, HTTP status code, 69
304- Not Modified, HTTP status code, 69, 164–165
400 errors, 72, 118
401 Unauthorized error response, 72
404 Not Found error response, 72
405 Method Not Allowed error response, 72
406 Not Acceptable error response, 72
409 Conflict error response, 72

A

`Accept` headers
- 406 Not Acceptable error response and, 72
- defining routes for service with Rails 3, 90
- using for versioning, 65–66

access authorization screen, OAuth, 248
access key ID, AWS, 182, 192
access token, OAuth
- consuming data from external services, 251–252
- implementing, 249
- pushing data into external service, 253–254

ActiveModel library
- overview of, 132
- serialization, 134–136
- validations, 132–134

ActiveSupport library, 134–136
activity model, Rails social feed reader, 48–49
activity streams, Rails social feed reader features, 46
addressability, URIs and, 266
`after_create`
- creating subscription activities with, 51
- initiating tasks or callbacks, 238
- Memcached time-based expiration, 160

`after_save` callback, 238
`after_validation` callback, 238

agility
  achieving with REST. See REST (Representational State Transfer)
  defined, 39
  of service-oriented design, 36–37
  SOA increasing organizational, 28
AJAX widget, 254
algorithms, load balancing, 148–152
Amazon Web Services. See AWS (Amazon Web Services)
AMQP (Advanced Message Queuing Protocol)
  client library, 224–227
  defined, 217
  RabbitMQ and. See RabbitMQ and AMQP
`AMQP.start`, AMQP client, 225
Apache Commons HTTPClient library, 117
Apache web server
  SSL encryption for, 210
  URI length limit in, 82
API (application programming interfaces). See also service and API design
  data is, in messaging systems, 234–236
  design guidelines, 85–86
  designing multi-get requests, 81–82
  designing service interface, 82–85
  overview of, 79–81
architectural constraints, REST as set of, 263–264
architectural elements, REST, 265
architectural views, REST, 265
asterisk (*) wildcard operator, AMQP topic exchanges, 220–223
asynchronous communications
  messaging creating more robust, 215
  in RabbitMQ with EventMachine, 224
  synchronous vs., 216–217
  using queues for messaging in, 217
asynchronous I/O, 108–109
asynchronous writes
  field value checking and, 232
  overview of, 227–230
Atom feeds, 50, 239
atomic APIs, 75–76
`attributes` method, ActiveModel serialization, 134
`attributes=` method, ActiveModel serialization, 134–136
authentication
  defined, 185
  HTTP, 186–187
  and message signing, 187
  and message signing, public/private key pairs, 192–196
  and message signing, shared secrets with HMAC, 187–192
  and message signing, what to sign, 197
  OAuth for. See OAuth
  overview of, 185–186
  for Rails applications, 199–200
  SSL certificates for, 198–199
Authlogic, 10
authorization
  defined, 185
  firewalls, 201–203
  overview of, 201
  RBAC, 203–209
availability
  CAP theorem and, 231
  eventual consistency and, 231–232
AWS (Amazon Web Services)
  access key ID/secret authorization, 192
  development of, 28
  parsing XML with EC2 Query API, 168–174
  service-oriented design of, 27–28
  SOAP requests, 180–184
  WSDL in, 178–180

B

backend processes, and load balancing
  least-connections, 150–151
  overview of, 148

round-robin, 149–150
URI-based, 151–152
background processes
current Rails social feed reader setup, 46
with Delayed Job, 43–44
full Rails deployment with, 44–45
simple Rails deployment with, 41–43
`before_save` callback, 238
`before_validation` callback, 238
belongs_to relationships, Rails social feed reader
activity model, 48–49
comment model, 53
entry model, 51
follow model, 49–50
subscription model, 51
vote model, 52
BERT-RPC messaging standard, 216
bindings, AMQP
direct exchanges with, 219
exchanges copying messages to all queues with, 219
fanout exchanges with, 220
topic exchanges with, 220–223
using AMQP client, 225
blocking I/O, 107–109
browsers, URI length limits of, 82
Bundler, 124
Bunny client library, 226–227
business logic isolation, service-oriented design, 30–32

C

`cacert.pem`, SSL authentication, 198
cache constraint, REST design, 264
`Cache-Control` header, expiration-based caching, 162
`Cache-Control: public`, HTTP cache header, 69
cache hit ratios, 152
cache keys, generational, 160–161
caching
with Bundler, 124
defined, 147, 155
expiration-based, 158–160, 162–163
headers for service APIs, 69
highest level joins and, 74–75
with HTTP. See HTTP caching
with Memcached. See Memcached
multi-get APIs and, 77
partitioning services on read/write frequencies, 62
call depth, 75
`Callable` interface, 116
callbacks
overview of, 238
web hook. See web hooks
`canonicalized_params`, message signing, 197
CAP (consistency, availability, and partition tolerance) theorem, 230–234
capacity, 147–148
case study
conclusion, 58
converting to services, 54–58
typical Rails application, 41–45
case study, Rails social feed reader application, 45–54
activity model, 48–49
comment model, 53–54
current setup overview, 46
entry model, 51–52
features, 46
feed model, 50
follow model, 49–50
overview of, 45
subscription model, 51
user model, 47–48
vote model, 52–53
certificates, SSL. See SSL certificates
client libraries
AMQP client, 224–226
Apache Commons HTTPClient library, 117

Bunny client, 226–227
memcache-client, 156–158
parsing XML, 167
reactor, 108
working with SOAP in Ruby, 177
yajl-ruby. See yajl-ruby library
client libraries, developing for services
with ActiveModel, 132–136
with ActiveSupport, 134–136
connecting to services. See connection and request logic, client libraries
mock, stubs, and tests, 143–146
overview of, 123
packaging, 123–127
parsing logic, 127–131
summary review, 146
client library implementation
creating user, 21–22
defined, 18
destroying user, 24
finding user, 18–21
updating user, 22–23
verifying user, 24–26
client/server constraint, REST design, 264
`client.rb` file
implementing `User.create`, 22
implementing `User.destroy`, 24
implementing `User.find`, 20
implementing `User.login`, 25–26
implementing `User.update`, 23
`client_spec.rb` file
spec'ing `User.create`, 21–22
spec'ing `User.find`, 19–20
spec'ing `User.login`, 25
spec'ing `User.update`, 23
CloudFront, Amazon Web Services, 28
code base, developing Rails in single, 41
code-on-demand constraint, REST design, 264
code, scalability of service-oriented design and, 36
comment model, 53–54
comments
activity model, 49
features of, 46
partitioning on logical function, 62
reuse of services for, 38
successful response bodies for, 70–71
URIs and interface design, 67
user model, 48
components, as REST architectural elements, 265
`config.ru` file
accessing RBAC with Rack adapter, 208–209
implementing `Rack: :Throttle`, 260–261
Sinatra directory structure, 95–96
user service implementation, 5
confirmation page, OAuth consumer registration, 247
`CONNECT` method, 268
connecting to services, 107–121
blocking I/O, threading and parallelism, 107–109
client libraries, 136–143
handling error conditions, 118–119
with JRuby, 115–117
logging for performance, 117–118
with multi-threaded requests, 113–115
overview of, 107
requests in development environments, 121
summary review, 121
testing and mocking service calls, 119–120
with Typhoeus, 109–113
connection and request logic, client libraries
overview of, 136
performing data reads, 136–142
performing data writes, 142–143
connector view, REST, 265
connectors, RESTful web services design, 265

consistency
  CAP theorem and, 231
  designing around eventual, 232–234
  eventual, 231–232
constraints, architectural. See REST (Representational State Transfer)
consumers, AMQP
  AMQP client library, 224–226
  Bunny client library, 227
  function of, 218
  message acknowledgements, 224
  pushing messages to, 224
  saving persistent messages until pushed off to, 223
consumers, OAuth, 246–249
consuming data, from external services, 251–252
continuous integration servers, web hooks utilized by, 240–241
controllers
  breaking up into services, 55–58
  current setup, 46
  implementing services with Rails, 90–95
  partitioning on logical function, 62
  quick iterations and, 60
counter caches
  comment model, Rails social feed reader, 52–53
  customizing in `Rack: :Throttle`, 260–261
  vote model, Rails social feed reader using, 52–53
credentials
  encryption preventing attack on, 213
  implementing HTTP authentication, 186–187
  implementing user verification, 17
  Twitter OAuth consumer application, 247
  verification of user, 16–17, 24–25
CRUD (create, update, delete) operations
  service requirements, 2
  URIs and interface design, 67
cryptographic hashes, for signed messages, 187
CSS selectors, Nokogiri support for, 174–177
Curb library, 107
curl `commands`, implementing RBAC, 209

D

data elements, RESTful architecture, 265
data models. See models
data reads, performing
  developing client libraries, 136–142
  routing through messaging system, 227
data stores, using write exchange, 228
data view, REST, 265
data writes, performing
  in eventually consistent system, 231–234
  service client libraries and, 142–143
  through RESTful HTTP services, 227–230
database server
  shared system isolation and, 32
  simple Rails deployment, 41–42
Delayed Job, 42–44
`DELETE` method
  client library implementation, 24
  as idempotent action, 245
  as idempotent method, 268–269
  implementing HMAC signing, 189
  RESTful services design utilizing, 271
  single requests in Typhoeus, 111
  in URIs and interface design, 67
  user service implementation, 15–16
`delete` method, Sinatra, 16
denormalization, Rails social feed reader
  activity model, 48–49
  migrating model to, 87
  minimizing joins with, 64
  service-oriented design vs., 34–35

dependencies, managing with Bundler, 124
deployment, service-oriented design for, 36
desktop applications, authentication. See OAuth
destroying user, client library implementation, 24
development environments, requests in, 121
direct exchanges, RabbitMQ and AMQP, 218–220
directory structure
  Rack, 100
  service client libraries, 123–127
  Sinatra, 95–96
Distributed Ruby (DRb), 216
down votes, Rails social feed reader, 52–53
DRb (Distributed Ruby), 216
durability, RabbitMQ and AMQP, 223

E

EC2 (Elastic Compute Cloud), AWS, 28
Edge side includes (ESI), 56–58
email notifications, Rails social feed reader, 46, 61
encapsulation
  adding to underlying model over time, 88
  parsing JSON in Ruby, 131
encryption
  defined, 185
  overview of, 209–210
  public/private keys for, 210–213
  SSL for, 210
`entitystore`, Rack-Cache gateway cache, 166
`Entry` class, `ReadingList` service, 141
entry model, Rails social feed reader, 51–52
`env` variable, Rack, 106
`:environment` task, implementing user service, 6
Ernie, messaging standard, 216
error responses
  handling general data, 118–119
  handling with `PUT` method, 15
  implementing service with Rails controller code, 93–94
  service API standards for, 72–73
ESI (Edge side includes), 56–58
`ETag` header, HTTP, 69, 163–165
EventMachine, 108, 224–226
eventual consistency, 231–234
exchanges, RabbitMQ and AMQP
  direct exchanges, 218–220
  durable, 223
  fanout exchanges, 220
  overview of, 218
  topic exchanges, 220–223
`Executors` class, 116
expiration-based caching
  `Expires` header, HTTP, 69
  HTTP, 162–163
  manual, with Memcached, 159–160
  time-based, with Memcached, 158–159
exponential backoff, for callback receivers, 245
external interoperability, service-oriented design, 37
external services
  authenticating with OAuth, 245–251
  consuming data, 251–252
  dealing with failure, 244–245
  ensuring performance and reliability, 258–261
  overview of, 237
  pushing data, 253–254
  request lifecycle, 254
  summary review, 261
  worker processes, 254–258
  working with web hooks. See web hooks

F

failover
  load balancing ensuring, 148
  request lifecycle of external services and, 254
  round-robin load balancer providing, 149
FakeWeb, 120
fanout exchanges, RabbitMQ and AMQP, 220
feed-crawling system, Rails, 68
feed model, Rails, 50, 60
field operations, messaging systems, 234–236
field value checking, in eventually consistent system, 232
Fielding, Roy, 263–265, 270–271
FIFO (first-in, first-out) data structure, 217
files structure, service client libraries, 123–127
Firefox, URI length limit of, 82
firewalls, 201–203
first-in, first-out (FIFO) data structure, 217
follow model, Rails social feed reader, 49–50, 67
formats
  message, 217
  for resources, as representations, 267–268
`from_xml` method, parsing XML, 172–173, 175–176
full isolation, service-oriented design, 32–33
`Future` class, 116–117

G

`.gem` files, 124, 127
`gem install` command
  ActiveRecord, 3
  building/deploying library with RubyGems, 127
  JSON Ruby, 4
  Typhoeus, 4
`gemcutter` gems, installing Jeweler, 125
gems
  creating with gemspec file, 124
  creating with Jeweler, 124–125
  parsing JSON in Ruby with JSON, 128–129
  performing data reads, 136–142
  reusing services with, 38
gemspec file, Jeweler, 124
`Gem::Specification` documentation page, 124
`GenerateKeys`
  encryption with public/private key pairs, 211
  public/private key pairs for RSA authentication, 193–194
generational cache keys, Memcached, 160–161
`GET` method
  atomic APIs and, 76
  `HEAD` method vs., 271
  in HMAC signing, 189
  as idempotent action, 245
  as idempotent method, 268
  multi-get APIs and, 76–77, 81–82
  in Rails controller code, 94
  in `ReadingList` service, 140
  in requests in Typhoeus, 111–112
  RESTful services design utilizing, 269
  as safe method, 268
  in URIs and interface design, 67–68
  user service implementation, 6–11
`git` gems, Jeweler, 125
Github, 240–242
global interpreter lock (GIL), 109, 113
Google Gmail, 254

H

Handsoap library, 177
HAProxy load balancer, 153–155
hardware load balancers, 152–155
Hash-based Message Authentication Code (HMAC) signing, 180–184, 187–192
has_many relationships, Rail social feed reader
  entry model, 51
  feed model, 50
  user model, 47
has_many: through relationship, user model, 47–48
`HEAD` method
  as idempotent method, 245, 268
  rare use of, 268
  RESTful services design utilizing, 271
  as safe method, 268
headers, HTTP, 271–274
highest level, joining at, 74–75
HMAC (Hash-based Message Authentication Code) signing, 180–184, 187–192
`HMAC::SHA256` signer, 189
`HmacSignature`, 189–191
`HOOK: INCOMING TRAFFIC`, firewalls, 202
`host` object, data reads, 137–138
Hpricot, 174
`.html` file, sepcifying representations, 267
HTTP authentication, 186–187
HTTP-based services, 1–2
HTTP caching
  expiration-based, 162–163
  HTTP-based data reads, 227
  implementing with Rack-Cache, 165–166
  implementing with Squid and Varnish, 166
  overview of, 162
  validation-based, 163–165
HTTP headers, RESTful services design, 271–274
HTTP libraries
  defined, 4
  as message transport protocol in SOA, 215
  service API standards for responses, 68–73
  service interface design methods, 82–85
  Typhoeus as preferred, 21
HTTP methods, RESTful services design, 268–271
HTTP requests
  in development environments, 121
  logging for performance, 117–118
  multiple-threaded, 113–115
  in parallel with JRuby, 115–117
  services responding to, 1
  Typhoeus simultaneous, 111–113
  Typhoeus single, 109–111
HTTP status codes
  RESTful services design, 274–275
  service API standards for errors, 72, 93–94, 117–118
  service API standards for successful responses, 68–69, 94
hybrid system, designing application as, 60–61
`hydra` class
  defined, 137
  logging for performance, 117–118
  making single requests, 109–110
  performing data reads, 137–139
  pushing data into external service, 253–254

I

idempotent methods
  defined, 245
  overview of, 268–269

`info` routing key, AMQP direct exchanges, 220
integration servers, web hooks utilized by, 240–242
Integrity integration server, 240
interface design, and URIs, 66–68
internal interoperability, service-oriented design, 37
Internet Explorer, URI length limit of, 82
interoperability
  achieving with REST. See REST (Representational State Transfer)
  service-oriented design and, 37, 39
invalid credentials, testing for, 17
invalidation, cache, 158–161
IP addresses, and firewalls, 201–203
`iprange` filter, firewalls, 202
`iptables`, firewalls, 201–202
`iptables-restore` command, firewalls, 203
`iptables-save` command, firewalls, 203
isolation
  achieving with REST. See REST (Representational State Transfer)
  business logic, 30–32
  full, 32–33
  service-oriented design and, 38
  shared system, 32
  testing in, 34
iteration speed, partitioning services on, 60–61

J

Jeweler, 123–126
`jeweler-help` command, 125
`jgem` command, 128–129
joins
  call depth issues, 75
  at highest level, 74–75
  overview of, 73
  partitioning services by minimizing, 63–64
  performing data reads, 137
  storing references, 73–74
  URIs and interface design, 67
JRuby
  connecting to services, 115–117
  installing JSON gem on, 128
  Nokogiri compatibility with, 174
  running requests in parallel with, 115–117
  using kernel-level threads, 109
JSON
  ActiveModel serialization module for, 134–136
  implementing service with Sinatra, 96–100
  overview of, 3–4
  parsing in request body, 94
  parsing in Ruby. See parsing JSON in Ruby
  parsing with yajl-ruby library, 94
  serialization standard, 216
  spec'ing `POST` user, 12
`.json` file, sepcifying representations, 267
`JSON.parse` call, 129

K

key/value stores, Memcached, 156

L

languages
  interoperability of service-oriented design, 37
  specifying for representation of resources, 267–268
`Last-Modified` header, HTTP caching, 69, 163–165

latency
  caching increasing, 155
  least-connections load balancer improving, 150
  load balancing maintaining, 148
  overview of, 147–148
layered system constraint, REST design, 264
least-connections load balancers, 150–151
least-recently used (LRU) eviction algorithm, 160–161
legacy services. See parsing XML, for legacy services
libcurl, 4, 109
libcurl-multi libraries, 4, 109
libraries. See client libraries
libxml library, installing Nokogiri, 174
LibXml-Ruby, 174
libxslt library, installing Nokogiri, 174
`LimitRequestLine` directive, Apache, 82
load balancing
  defined, 147
  implementing, 152–155
  least-connections, 150–151
  overview of, 148
  round-robin, 149–150
  URI-based, 151–152
lock service, enforcing uniqueness, 233
logging
  handling error conditions, 118
  for performance, 117–118
  using AMQP client, 225
logical function, partitioning on, 61–62
lower-level services, partitioning, 61–62
LRU (least-recently used) eviction algorithm, 160–161

M

man-in-the-middle attacks, preventing, 187, 209–210
manual expiration, Memcached, 159–160
mapping, Ruby objects with ActiveRecord, 3
`Marshal` module, memcache-client library, 157–158
memcache-client library, 156–158
Memcached, 155–161
  deploying Rails with background processes, 44–45
  generational cache keys, 160–161
  manual expiration, 159–160
  Memcached client and ActiveRecord, 156–158
  overview of, 155–156
  time-based expiration, 158–159
memory, caching objects in. See Memcached
message signing
  authentication using, 187
  public/private key pairs with RSA, 192–196
  shared secrets with HMAC, 187–192
  what to sign, 197
messaging, 215–236
  benefits of signing, 185
  CAP theorem and, 230–234
  creating systems tolerant of usage spikes, 143
  data as API, 234–236
  formats, 217
  moving outside request life cycle with, 243–244
  queues, 217
  RabbitMQ and AMQP. See RabbitMQ and AMQP
  summary review, 236
  synchronous vs. asynchronous, 216–217, 227–230
  understanding, 215–216
`metastore` option, Rack-Cache, 166
methods, HTTP, 268–271
metrics, performance of services, 259–260
migration. See also Rails social feed reader application
  agility of service-oriented design for, 36
  creating user model with, 9–10

MIME types, indicating service versions, 65–66
mocking
  service calls, 119–120
  service client libraries, 143–146
model file, 9–10
models
  breaking up into into services, 55–58
  creating user model, 47–48
  current setup, 46
  implementing services with, 86–88
  quick iterations and, 60
mod_ssl, SSL for encryption, 210
Mongrel, HAProxy with, 154–156
monitoring
  latency of services, 147
  partitioning on logical function, 61
multi-get APIs, 75–77
multi-get requests, 81–82
multi-threaded requests, connecting to services, 113–115
multiple-model APIs, 77–78
multithreaded I/O, 107–109

N

named scopes, creating models, 86–87
naming conventions
  creating gems with Jeweler, 125–126
  specifying representation of requested resources, 267
  URIs and, 266
native (kernel-level) threads
  achieving parallelism, 108–109
  thread pools for, 115
NeverBlock library, 108
next page functionality
  creating pagination, 94
  implementing services, 91–92, 97–98, 103–104
  performing data reads, 140
Nginx web server
  enabling SSL for encryption, 210
  URI length limit of, 82
Nokogiri, 167
notifications, receiving, 228–230
notify exchange, messaging-based writes, 228–230

O

OAuth
  consuming data from external services, 251–252
  implementing consumer, 246–249
  implementing provider, 249–251
  overview of, 245
object/relational mapper (ORM), 3
objects, JSON, 127–128
`on_complete` handler
  logging for performance, 117–118
  `ReadingList` service, 140–141
  simultaneous requests in Typhoeus, 112–113
  single requests in Typhoeus, 110
online references
  ActiveModel built-in validators, 134
  ActiveRecord, 3
  Amazon EC2 Query API documentation, 168–170
  Amazon's Product Advertising API, 178
  AMQP, 218
  AMQP client, 224
  Apache Commons HTTPClient library, 117
  Bundler, 124
  Bunny client library, 226–227
  creating gem with gemspec file, 124
  Delayed Job, 43
  ESI (Edge side includes), 57
  EventMachine, 224
  `Gem::Specification` documentation page, 124

HTTP request headers, 273
Jeweler, 124–127
JSON, 4, 94
libcurl installation, Typhoeus, 109
libraries for working with SOAP, 177
messaging standards for Ruby, 216
mocking service calls, 120
performing data reads, 137
presenter pattern, 94
PubSubHubbub, 239
`Rack: :Throttle`, 260
reactor design pattern implementations, 108
receiving web hooks, 240
REST concept, 263
REXML, 170
Rspec, 4
ruby-hmac gem, 187
Ruby on Rails guide to security, 200
Sinatra, 3
Typhoeus, 4
vendor tree, 66
worker processes for servolux library, 254
WSDL file, 178
XPath, 170
yajl-ruby project, 129
`openssl`, public/private key pairs, 193
optimistic locking, for uniqueness, 233
optimistic systems, for transactions, 233
`OPTIONS` method
rare use of, 268
RESTful services design utilizing, 271
ORM (object/relational mapper), 3

P

packaging library
organizing structure of files and directories, 123–126
with RubyGems, 127
parallel request method, atomic APIs, 76
parallelism
with asynchronous I/O, 108
for data read requests, 136
with JRuby, 115–117
with multiple-threaded requests, 113–115
with multithreaded I/O, 107–109
overview of, 107–108
with Typhoeus, 109
`params` hash
implementing `GET` user, 11
pushing data into external service, 253–254
parsing JSON in Ruby
with JSON gem, 128–129
overview of, 127–128
wrapping parsed results, 130–131
yajl-ruby library, 94, 129–130
parsing XML, for legacy services
summary review, 184
using Nokogiri, 174–177
using REXML, 170–174
writing classes using SOAP calls, 177–184
XML, 167–170
partition tolerance, 231–232
partitioning into separate services
on iteration speed, 60–61
on join frequency, 63–64
on logical function, 61–62
on read/write frequencies, 62–63
scalability of service-oriented design, 35
transactions using, 234
passwords
HTTP authentication, 186–187
user authentication, 185
user verification, 16–18
validating user model, 10
patterns, AMQP topic exchanges messages based on, 220–223
performance
external services and, 254, 258–261
logging for, 117–118
multiple-threaded requests for, 113–115
persistency slowing, 223

RabbitMQ and AMQP, 223
Squid and Varnish benefits, 166
threaded I/O and, 109
typical Rails issues, 45
persistency, RabbitMQ and AMQP, 223
Phusion Passenger load balancer, 153
pings, PubSubHubbub, 239
plugins
RBAC, 203
reuse of services, 38
ports, restricting with firewalls, 202–203
POST body
encryption with public/private key pairs, 210–211
message signing, 197
POST method
designing service interface, 85
HMAC signing, 189
implementing service with multi-get interface, 81–82
pushing data into external service, 253–254
RESTful services design utilizing, 269
single requests in Typhoeus, 111
URIs and interface design, 67–68
user service implementation, 11–13
post-receive hooks, web hooks, 240–242
pound sign (#) wildcard operator, AMQP topic exchanges, 220–223
presenter pattern, 94
previous page functionality
creating pagination, 94
implementing services, 91–92, 97–98, 103–104
performing data reads, 140
process view, REST, 265
producer, AMQP, 219
profiles, Rails social feed reader, 46
The Programmable Web, 237
provider, implementing OAuth, 249–251
public-facing APIs, reusing services with, 38
public/private key pairs
for encryption, 210–212
generating with RSA, 192–196
PubSubHubbub, 239
pushing data, into external services, 224, 253–254
PUT method
designing service interface, 85
HMAC signing and, 189
as idempotent action, 245
as idempotent method, 268–269
implementing `User.update`, 23
RESTful services design utilizing, 269–271
single requests in Typhoeus, 111
URIs and interface design, 67
user service implementation, 13–15

Q

query strings
authentication with, 195
encryption with, 210–213
signing, 197
signing requests using HMAC, 187, 190–191
querying user authentication, RBAC service, 204–206, 209
queues
acknowledgement in AMQP-based, 245
basic metrics in, 259
messaging, 217
RabbitMQ and AMQP, 218, 223, 225
running multiple-threaded requests in, 114–115
segregating by service, 259
using worker processes, 254–258
quotas, monitoring external services, 260

R

RabbitMQ and AMQP
acknowledgement, 245
client libraries, 224–227

durability and persistence, 223–224
exchanges and bindings, 218–223
moving work outside request life cycle, 243–244
overview of, 217–218
queues in RabbitMQ, 218
Rack
HTTP authentication in, 186–187
implementing services with, 100–106
Rails and Sinatra built on top of, 79, 95
throttling in, 260
Rack adapter
implementing encryption, 212–213
implementing HMAC signing, 190–192
implementing RBAC, 206–209
Rack-Cache gateway cache, HTTP caching, 165–166
`Rack: :Throttle`, 260–261
`Rack::SignatureValidator`, 191–192
`rackup` file, Rack, 100–106
Rails applications
authentication for, 199–200
implementing service with, 79
implementing service with Sinatra vs., 100
Rails 2.3 routes, 88–89
Rails 3 routes, 89–90
Rails controller code, 90–95
typical, 41–45
Rails social feed reader application
activity model, 48–49
breaking up into services, 54–58
comment model, 53–54
current set up, 46
entry model, 51–52
features, 46
feed model, 50
follow model, 49–50
overview of, 45
segmenting into services, 54
service and API design. See service and API design
subscription model, 51
summary review, 58
user model, 47–48
vote model, 52–53
ratings
ActiveModel serialization, 135–136
ActiveModel validation, 132–134
data reads, 137
data writes, 142–143
simultaneous requests in Typhoeus, 113
RBAC authorization service
overview of, 203–206
Rack adapter, 206–209
reactor design pattern, 108
read/write frequencies, partitioning services on, 62–63
`read_attribute_for_validation` method, ActiveModel, 133
reading data
logic for, 136–142
from queue in RabbitMQ, 218
`ReadingList` service, 137–142
recording failures, 245
references, storing joined data, 73–74
registration, OAuth, 246
relationships, Rails social feed reader
comment model, 54
current setup, 46
entry model, 52
feed model, 50
subscription model, 51
user model, 47–48
vote model, 53
reliability, external services, 258–261
replay attacks, preventing, 189
replicated data, minimizing joins with, 64
representation headers, RESTful services design, 273
representations, REST and, 267–268
`request` class, 109–110, 116
request headers, RESTful services design, 272–273

request lifecycle, external services, 254, 258
request objects, ReadingList service, 142
request signing
  authentication and, 187
  public/private key pairs with RSA, 192–196
  shared secrets with HMAC, 187–192
  what to sign, 197
request token, OAuth, 248–249
Request wrapper, Rack, 106
requests, SOAP, 180–184
Resource class, RBAC, 204
resource-oriented architecture (ROA), 29
resources
  RBAC roles for, 203–206
  REST and, 265–268
response bodies, 70–73
response codes, customizing in Rack::Throttle, 262
response headers, RESTful services design, 274
response object, Typhoeus, 109–110
response time, 147–148
REST (Representational State Transfer)
  service versions, 65
  sessions as resources in, 17
  SOA vs. service-oriented design vs., 28–29
  this book focusing on services for, 2
  using HTTP APIs for messaging, 215
REST (Representational State Transfer), understanding
  architectural elements, 264–265
  architectural views, 265
  constraints, 264
  HTTP headers, 271–274
  HTTP methods, 268–271
  HTTP status codes, 274–275
  overview of, 263–264
  and resources, 265–268
  uniform interface of, 268
RESTful Web Services (Richardson and Ruby), 29
retry logic
  designing callback receivers, 245
  handling error conditions, 119
reuse, 38–39
Revactor library, 108
REXML, 167, 170–174
ROA (resource-oriented architecture), 29
robust architecture, service-oriented design, 34–35
robustness, 39
RoleBased class, RBAC, 204
roles, RBAC, 203–206
round-robin load balancers, 149–150
routers, exchanges vs., 218
routes, defining for services
  Rails 2.3, 88–89
  Rails 3, 89–90
  Sinatra vs. Rails, 100
routes.rb, 88–90
routing keys, RabbitMQ and AMQP
  defined, 218
  in direct exchanges, 218–220
  in fanout exchanges, 220
  modifications to field operations, 235–236
  in topic exchanges, 220–223
RSA algorithm, public/private key pairs with, 192–196
Rspec, testing library, 4
RSS feeds, 50, 239
rssCloud, 239
Ruby bindings to YAJL, 4
ruby-hmac gem, implementing HMAC, 187
Ruby tool set
  ActiveRecord, 3
  JSON, 3–4
  Rspec, 4
  Sinatra, 2–3
  Typhoeus, 4
RubyGems
  building/deploying library, 127
  installing memcache-client library, 156–157

structuring library files/directories, 123–126
run call, Typhoeus, 110

S

safe methods, 268
save method, data writes, 142–143
Savon library, 177–178
Savon:Client object, SOAP requests, 183
Savon:Result object, SOAP requests, 182
scalability
  achieving with REST. See REST (Representational State Transfer)
  defined, 39
  load balancing allowing, 148
  Rails applications issues, 45
  round-robin load balancer providing, 149
  in service-oriented design, 35–36
searching, reusing services in, 38
security
  authentication. See authentication
  authorization. See authorization
  encryption, 209–213
  overview of, 185
  password. See passwords
  for SOAP requests, 180
segregating queues by service, 259
serialization
  ActiveModel, 134–136
  as goal of message formats, 216
  JSON. See JSON
  XML as format for, 167
service and API design, 59–78
  API complexity and, 75–78
  handling error responses, 72–73
  handling joins, 73–75
  partitioning on iteration speed, 60–61
  partitioning on join frequency, 63–64
  partitioning on logical function, 61–62
  partitioning on read/write frequencies, 62–63
  successful responses, 68–71
  summary review, 78
  URIs and interface design, 66–68
  versioning, 64–66
Service class, 96, 100
service library, 55–58
service-oriented architecture (SOA)
  comparing with other design strategies, 28–29
  HTTP in, 215
service-oriented design
  agility, 36–37
  case study. See case study
  interoperability, 37
  isolation, 30–34
  making case for, 29–30
  reuse, 38
  robustness, 34–35
  scalability, 35–36
  SOA vs. RESTFUL architecture vs., 28–29
  summary review, 38–39
  understanding, 27
  uses of, 27–28
service.rb file
  implementing DELETE user, 15–16
  implementing GET user, 10–11
  implementing POST user, 12–13
  implementing PUT user, 14–15
  implementing service with Sinatra, 96–100
  implementing user verification, 17–18
  spec'ing User.destroy, 24
  spec'ing User.find, 18–20
service.ru file, 100
services
  breaking up Rails application into, 54–58
  requirements, 2
  Ruby tool set, 2–4
  segmenting Rails application into, 54
  understanding, 1–2

services, implementing
  API design guidelines, 85–86
  creating data models, 86–88
  creating full Rails application, 88–95
  multi-get interface, 81–82
  summary review, 106
  using raw Rack interface, 100–106
  using Sinatra interface, 95–100
  vote interface, 82–85
  vote service. See vote service
`service_spec.rb` file
  spec'ing `DELETE` user, 15
  spec'ing `GET` user, 6–9
  spec'ing `POST` user, 12
  spec'ing `PUT` user, 13
  spec'ing user verification, 16
servolux library, worker processes, 254–258
shared key value store, partitioning, 61–62
shared messaging system, partitioning, 61–62
shared secret key, SOAP requests, 182
shared secrets, HMAC signing, 187–192
shared system isolation, service-oriented design, 32–33
`SIGHUP`, worker processes, 257–258
signature class, creating HMAC, 188–189
`signature_is_valid`? HMAC signing, 190
`SIGTERM`, worker processes, 257–258
Simple Object Access Protocol. See SOAP (Simple Object Access Protocol)
Simple Queue Service (SQS), Amazon Web Services, 28, 62
Simple Storage Service (S3), Amazon Web Services, 28, 62
SimpleDB, AWS, 28
simultaneous HTTP requests, Typhoeus, 111–113
Sinatra
  defined, 79, 95
  handling RBAC service users/roles, 205–206
  implementing services with, 95–100
  overview of, 2–3
single HTTP requests, Typhoeus, 109–111
single-table inheritance (STI), 49, 52
SOA (service-oriented architecture)
  comparing with other design strategies, 28–29
  HTTP in, 215
SOAP (Simple Object Access Protocol)
  making requests, 180–184
  as messaging format, 215–216
  overview of, 177
  with WSDL file, 177–180
soap4r library, 177
social feed reader case study. See Rails social feed reader application
software load balancers, 152–155
specs, user service implementation
  `DELETE` user, 15
  `GET` user, 6–9
  `POST` user, 12
  `PUT` user, 13–14
  `User.destroy`, 24
  `User.login`, 25
  verification, 16–17
SQS (Simple Queue Service), Amazon Web Services, 28, 62
Squid gateway cache, 166
SR/AW (synchronous reads, asynchronous writes)
  field value checking and, 232
  HTTP-based reads, 227
  messaging-based writes, 227–230
  understanding, 227
  and value uniqueness, 233
SSL certificates
  authentication with, 198–200
  encryption with, 210
  using basic HTTP authentication with, 187
stateless constraint, REST design, 264
status codes. See HTTP status codes
status update, Twitter, 253–254
STI (single-table inheritance), 49, 52

storage service
  feed entry, 60–61
  requirements, 2
strings, Memcached storing types other than, 157–158
stubbing, service client libraries, 143–146
`subscribe` blocks, consumers for Bunny client library, 227
subscriptions
  activity model, 48–49
  queues in RabbitMQ, 218, 225
  Rails social feed reader, 49, 51
  synchronous reads, asynchronous writes and, 229
  user model, 47
successful responses, service API standards, 68–71
synchronous communications, 216–217
synchronous reads, asynchronous writes. See SR/AW (synchronous reads, asynchronous writes)

T

tagging systems, reusing services, 38
TCP requests, firewalls, 202
team size
  Rails applications issues, 45
  scalability of service-oriented design, 36
testing
  `GET` user, 8–9
  in isolation, 34
  Rails applications, 41
  Rspec as preferred library for, 4
  service calls, 119–120
  service client libraries, 143–146
Thin, HAProxy with, 154
thread pools, 114–117
threading
  achieving parallelism with, 108–109
  running multiple-threaded requests, 113–115
throttling, monitoring external services, 260
throughput, 147–148
time-based expiration-based caching, Memcached, 158–159, 160
time sensitivity, request lifecycle of external services, 254
time value
  HMAC signing in Rack, 191
  preventing replay attacks, 189
`timestamp_and_signature` method, HMAC signing, 182
tokens
  OAuth provider, 249–251
  OAuth requests, 248–249
topic exchange, RabbitMQ and AMQP, 220–223, 228
topics, PubSubHubbub, 239
`TRACE` method, 268
transactions, in eventually consistent system, 233–234
tweet, pushing data into Twitter, 253–254
`TweetSender` module, 256–257
Twitter
  consuming data, 251–252
  open access of, 251
  pushing data into, 253–254
  request lifecycle, 254
  using OAuth authentication, 245–251
  worker processes, 254–258
Typhoeus
  defined, 109
  HTTP authentication, 186
  as HTTP library, 4
  logging for performance, 117–118
  making simultaneous requests, 111–113
  making single requests, 109–111
  mocking service calls, 119–120
  using re-try logic, 118–119

U

Unicorn load balancer, 153–155
uniform interface constraint
  HTTP headers, 271–274
  HTTP methods, 268–271
  HTTP status codes, 274–275
  in REST design, 264
uniqueness, eventually consistent systems, 232–233
up votes, Rails social feed reader, 52–53
updates, user
  agility of service-oriented design for, 36
  client library implementation, 23
  `PUT` method for, 13–15
updating service versions, 64–65
URI-based load balancers, 151–152
URIs (uniform resource identifiers)
  defining for Sinatra, 100
  `GET` limitation on length of, 82
  handling joins, 73–74
  interface design and, 66–68
  `ReadingList` service, 140
  REST and, 266
  service design implementation, 80–81
  service interface design, 82–85
  service version indicated in, 64–65
  specifying representation of requested resources, 267
  spec'ing user verification, 17
  successful response bodies and, 70
URLs
  messaging, 215
  RBAC service implementation, 205
user authentication
  HTTP, 186–187
  messaging signing for, 185
  overview of, 185
  for Rails, 199–200
  request signing for, 187
  reuse of services for, 38
`User` class, RBAC, 204
user-level threads, 108–109, 113
user login verification, 2
user model
  creating, 9–10
  Rails social feed reader, 47–48
user name
  HTTP authentication, 186–187
  implementing user verification, 17–18
  as simplest form of user authentication, 185
  spec'ing user verification, 16–17
user profiles, 46
`user-service` directory, 5, 10–11
user service implementation
  for authentication, 199–200
  deleting user, 15–16
  overview of, 5–6
  `POST`ing user, 11–13
  `PUT`ing user, 13–15
  using `GET`, 6–11
  verifying user, 16–18
`User.create`, 21–22
`User.destroy`, 24
`User.find`, 18–21
`User.login`, 25–26
`user.notify` exchange, 229–230
users
  client library implementation. See client library implementation
  creating/authorizing roles for, 203–206
`/users/:name/authorization`, 205
`User.update`, 23
`user.write` exchange, 229–230
UUIDs, multi-get, 82

V

valid credentials, testing for, 17
`validates_acceptance_of`, ActiveModel, 133

validates_confirmation_of, ActiveModel, 133
validates_exclusion_of, ActiveModel, 134
validates_format_of, ActiveModel, 134
validates_inclusion_of, ActiveModel, 134
validates_length_of, ActiveModel, 134
validates_numericality_of, ActiveModel, 134
validates_presence_of, ActiveModel, 134
validates_uniqueness_of, 134
validates_with, ActiveModel, 134
validation. See also authentication
  ActiveModel, 132–134
  creating model, 86–87
  creating user model, 10
  HTTP caching based on, 163–164
  not occuring in client library, 22
  with Rails controller code, 93
  verifying user credentials, 24–26
Validations module, ActiveModel, 132–134
value uniqueness, eventually consistent systems, 232–233
Varnish gateway cache, 166
vendor tree, defining application MIME types, 66
vendors, interoperability of service-oriented design, 37
verification, user
  client library implementation, 25–26
  user service implementation, 16–18
versioning services
  agility and, 37
  overview of, 64–65
version.rb file, Jeweler, 126
views
  breaking up into services, 55–58
  current setup, 46
  partitioning on logical function, 62
  quick iterations and, 60
  REST architectural, 265
Vogels, Werner, 27–28
vote model, Rails social feed reader, 52–53
vote service
  API design guidelines, 85–86
  creating data models, 86–88
  multi-get interface, 81–82
  overview of, 79–81
  with Rack, 100–106
  with Rails, 88–95
  with Sinatra, 95–100
  summary review, 106
  vote interface, 82–85
votes
  activity model, 49
  features, 46
  partitioning services on read/write frequencies, 63
  vote model, Rails social feed reader, 52–53
VotesService class, 105–106

W

web hooks
  overview of, 237
  providing, 242–244
  PubSubHubbub, 239
  receiving, 240–242
  strategies for dealing with failure, 244–245
  summary review, 261
  understanding, 238–239
web server
  full Rails deployment with, 43–45
  simple Rails deployment with, 41–43
Web Services Description Language. See WSDL (Web Services Description Language)
WebhookCallback model, 243–244
wildcards, for AMQP topic exchanges, 220–223

worker processes, external services, 254–258
WorkerPool module, 255–258
wrapping parsed results, JSON, 130–131
write exchange
  and field value checking, 232
  with minimal writes, 236
  overview of, 228–230
writing
  classes that make SOAP calls. See SOAP (Simple Object Access Protocol)
  classes to parse service responses. See parsing XML, for legacy services
  messaging-based writes, 228–230
  partitioning services on read/write frequencies, 62–63
  performing data writes, 142–143
WSDL (Web Services Description Language)
  constructing requests without, 183–184
  exploring web services, 177–180
  making requests, 180–182
WSSE (WS-Security), SOAP requests, 180

X

X-Auth-Sig header
  HMAC signing, 189
  using public/private key pairs, 194
X-Auth-User header
  HMAC signing, 190
  using public/private key pairs, 194
XML
  implementing web services. See SOAP (Simple Object Access Protocol)
  parsing. See parsing XML, for legacy services
.xml file, sepcifying representations, 267
XPath
  defined, 170
  REXML full support for, 170

Y

yajl-ruby library, 94, 129–130