Sharing the Message

Section 6

In this Section, we are going to take a look at...

- REST and RESTful APIs
- Using Rails 5 to create API project
- Authenticating our API's consumers
- Testing our API
- Consuming our API from client app

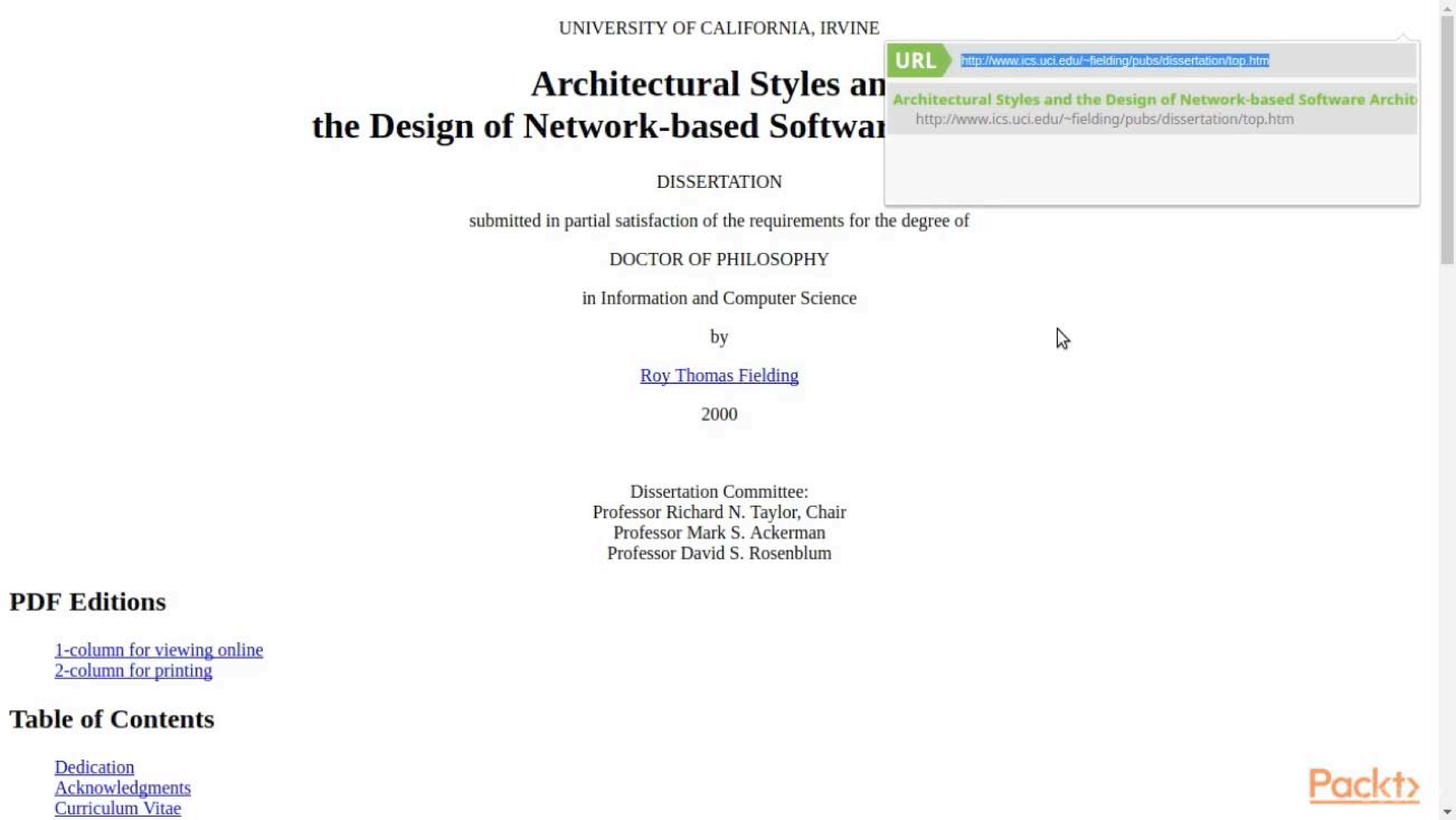
RESTful APIs

In this Video, we are going to take a look at...

- The REST philosophy and the REST constraints
- The 6 essential knowledge areas required for building APIs
- The reasons why we encapsulate domain functionality in APIs

What Is REST?

- An architectural style for distributed hypermedia systems
- Roy Fielding, 2000
  - http://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf



UNIVERSITY OF CALIFORNIA, IRVINE

URL http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm

Architectural Styles and the Design of Network-based Software Archit

http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm

Architectural Styles an
the Design of Network-based Softwar

DISSERTATION

submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in Information and Computer Science

by

Roy Thomas Fielding

2000

Dissertation Committee:
Professor Richard N. Taylor, Chair
Professor Mark S. Ackerman
Professor David S. Rosenblum

PDF Editions

- 1-column for viewing online
- 2-column for printing

Table of Contents

[Top] [Prev] [Next]

CHAPTER 5

Representational State Transfer (REST)

This chapter introduces and elaborates the Representational State Transfer (REST) architectural style for distributed hypermedia systems, describing the software engineering principles guiding REST and the interaction constraints chosen to retain those principles, while contrasting them to the constraints of other architectural styles. REST is a hybrid style derived from several of the network-based architectural styles described in Chapter 3 and combined with additional constraints that define a uniform connector interface. The software architecture framework of Chapter 1 is used to define the architectural elements of REST and examine sample process, connector, and data views of prototypical architectures.

5.1 Deriving REST

The design rationale behind the Web architecture can be described by an architectural style consisting of the set of constraints applied to elements within the architecture. By examining the impact of each constraint as it is added to the evolving style, we can identify the properties induced by the Web's constraints. Additional constraints can then be applied to form a new architectural style that better reflects the desired properties of a modern Web architecture. This section provides a general overview of REST by walking through the process of deriving it as an architectural style. Later sections will describe in more detail the specific constraints that compose the REST style.

5.1.1 Starting with the Null Style

There are two common perspectives on the process of architectural design, whether it be for buildings or for software. The first is that a designer starts with nothing--a blank slate, whiteboard, or drawing board--and builds-up an architecture from familiar components until it satisfies the needs of the intended system. The second is that a designer starts with the system needs as a whole, without constraints, and then incrementally identifies and applies constraints to elements of the system in order to differentiate the design space and allow the forces that influence system behavior to flow naturally, in harmony with the system. Where the first emphasizes creativity and unbounded vision, the second emphasizes restraint and understanding of the system context. REST has been developed using the latter process. Figures 5-1 through 5-8 depict this graphically in terms of how the applied constraints would differentiate the process view of an architecture as the incremental set of constraints is applied.

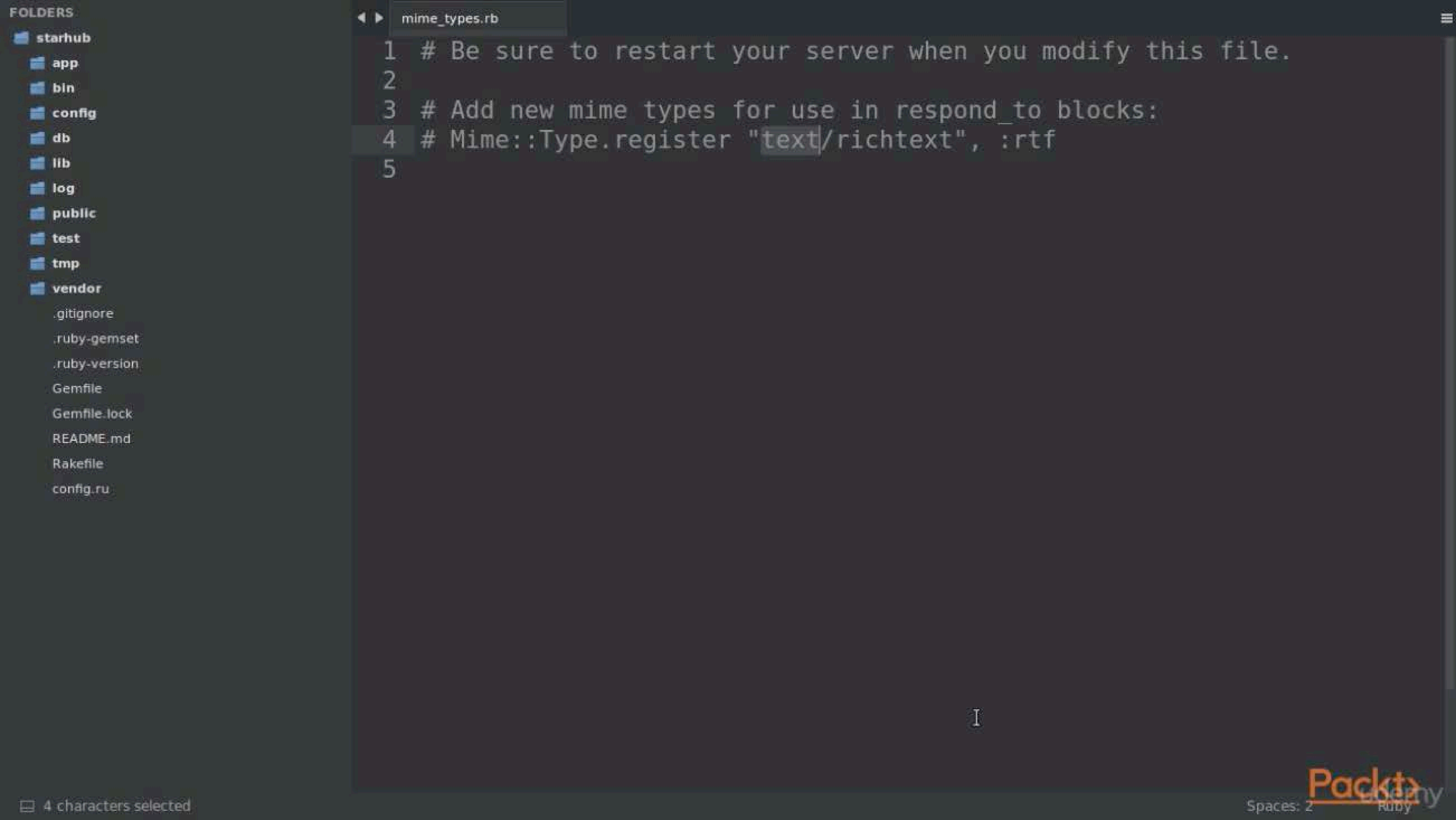
The Null style (Figure 5-1) is simply an empty set of constraints. From an architectural perspective, the null style describes a system in which there are no distinguished boundaries between components. It is the starting point for our description of REST.

The RESTful Constraints

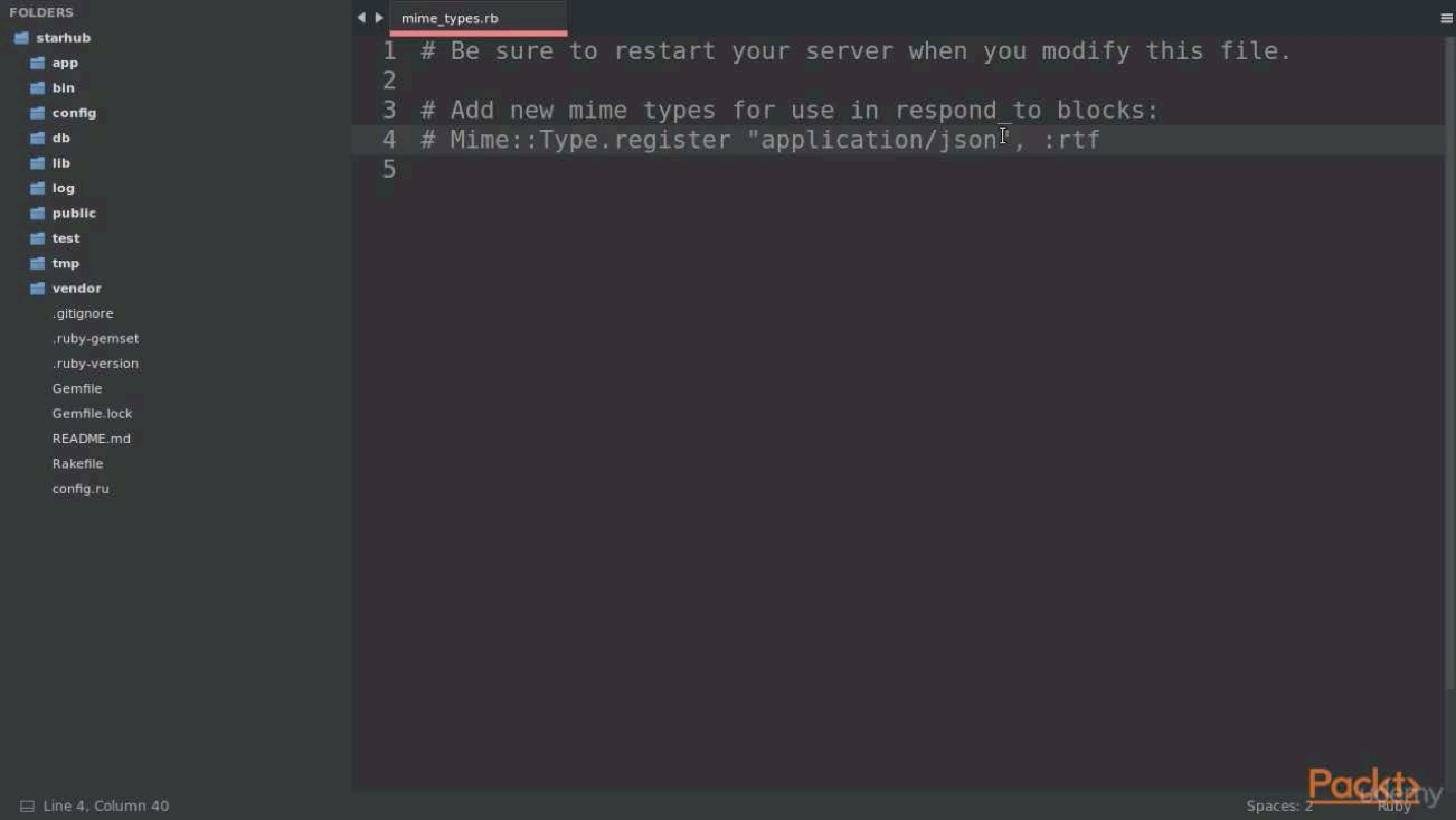
- Client server
- Stateless
- Cache declaration
- Uniform Interface
  - Identification of resources
  - Resources through representations
  - Self-descriptive messages
  - Hypermedia As The Engine Of Application State (HATEOAS)
  - Layered system
- Code-On-Demand

Areas to Consider

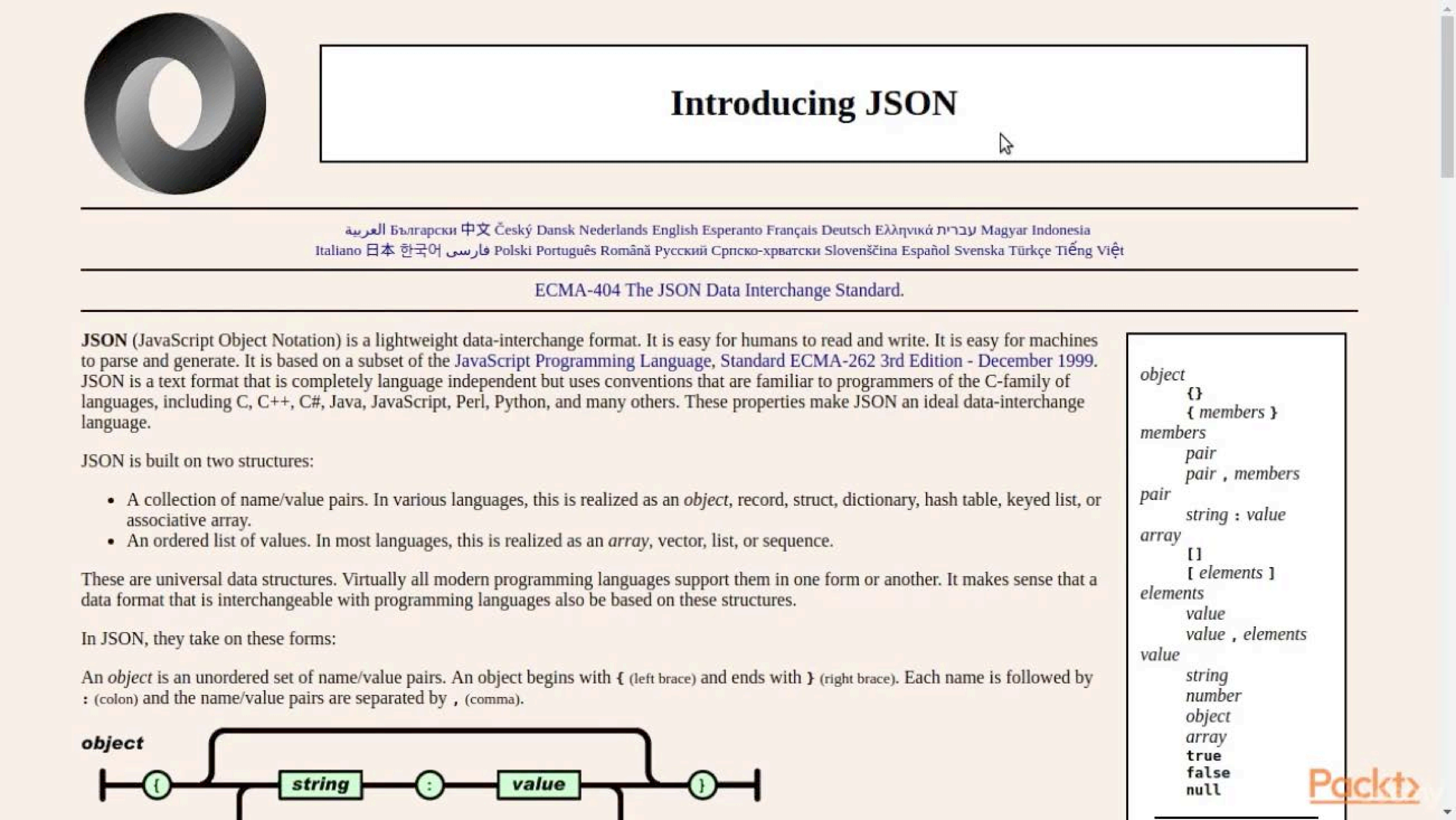
- Media types
- Versioning
- Caching
- Authentication
- Security



```ruby
# Be sure to restart your server when you modify this file.

# Add new mime types for use in respond_to blocks:
# Mime::Type.register "text/richtext", :rtf

```



```ruby
# Be sure to restart your server when you modify this file.

# Add new mime types for use in respond_to blocks:
# Mime::Type.register "application/json", :rtf

```



Introducing JSON

العربية Български 中文 Český Dansk Nederlands English Esperanto Français Deutsch Ελληνικά עברית Magyar Indonesia
Italiano 日本 한국어 فارسی Polski Português Română Русский Српско-хрватски Slovenščina Español Svenska Türkçe Tiếng Việt

ECMA-404 The JSON Data Interchange Standard.

JSON (JavaScript Object Notation) is a lightweight data-interchange format. It is easy for humans to read and write. It is easy for machines to parse and generate. It is based on a subset of the JavaScript Programming Language, Standard ECMA-262 3rd Edition - December 1999. JSON is a text format that is completely language independent but uses conventions that are familiar to programmers of the C-family of languages, including C, C++, C#, Java, JavaScript, Perl, Python, and many others. These properties make JSON an ideal data-interchange language.

JSON is built on two structures:

- A collection of name/value pairs. In various languages, this is realized as an *object*, record, struct, dictionary, hash table, keyed list, or associative array.
- An ordered list of values. In most languages, this is realized as an *array*, vector, list, or sequence.

These are universal data structures. Virtually all modern programming languages support them in one form or another. It makes sense that a data format that is interchangeable with programming languages also be based on these structures.

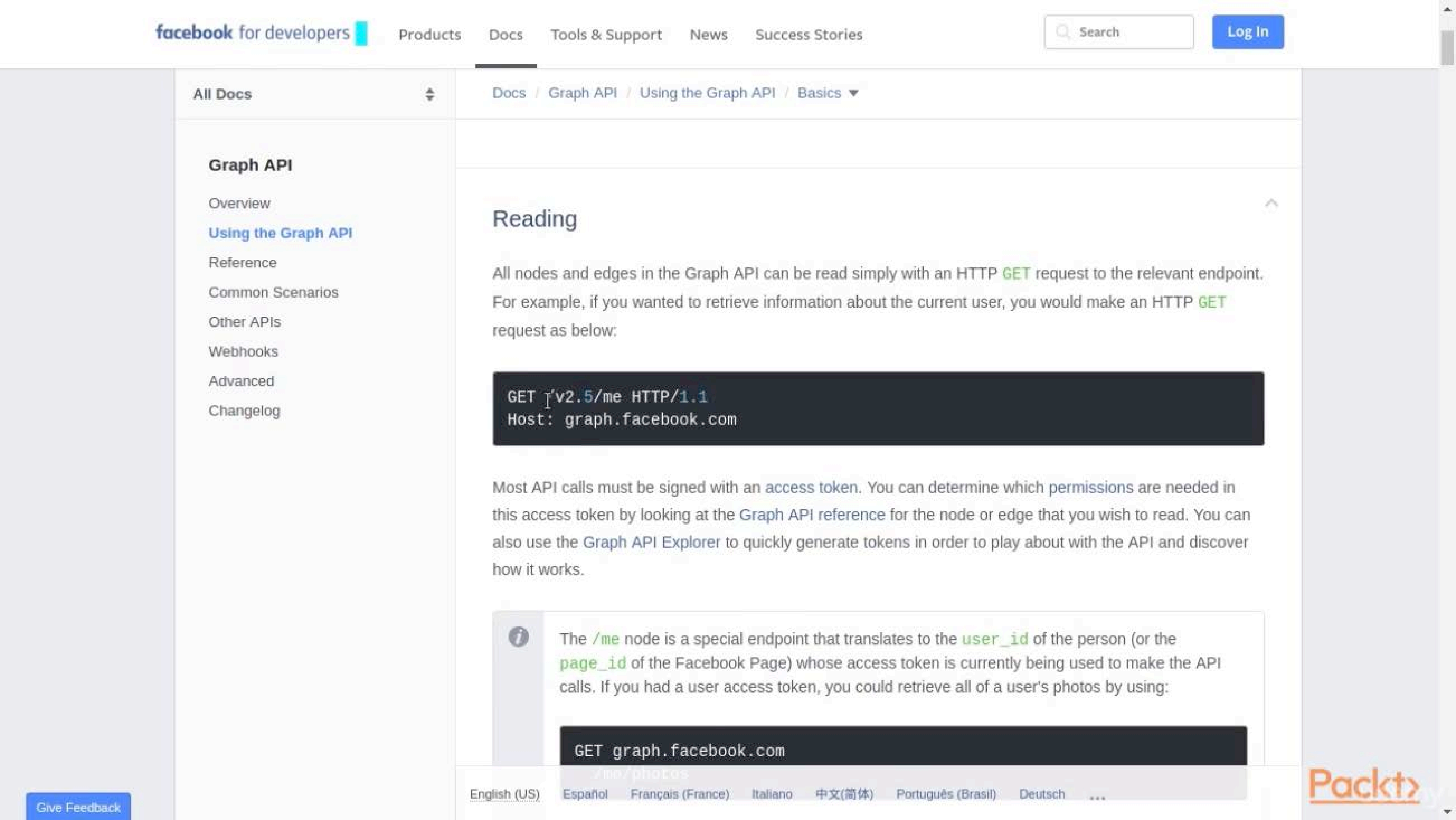
In JSON, they take on these forms:

An *object* is an unordered set of name/value pairs. An object begins with **{** (left brace) and ends with **}** (right brace). Each name is followed by **:** (colon) and the name/value pairs are separated by **,** (comma).

```
object
    {}
    { members }
members
    pair
    pair , members
pair
    string : value
array
    []
    [ elements ]
elements
    value
    value , elements
value
    string
    number
    object
    array
    true
    false
    null
```

Areas to Consider

- Media types
- Versioning
- Caching
- Authentication
- Security



Reading

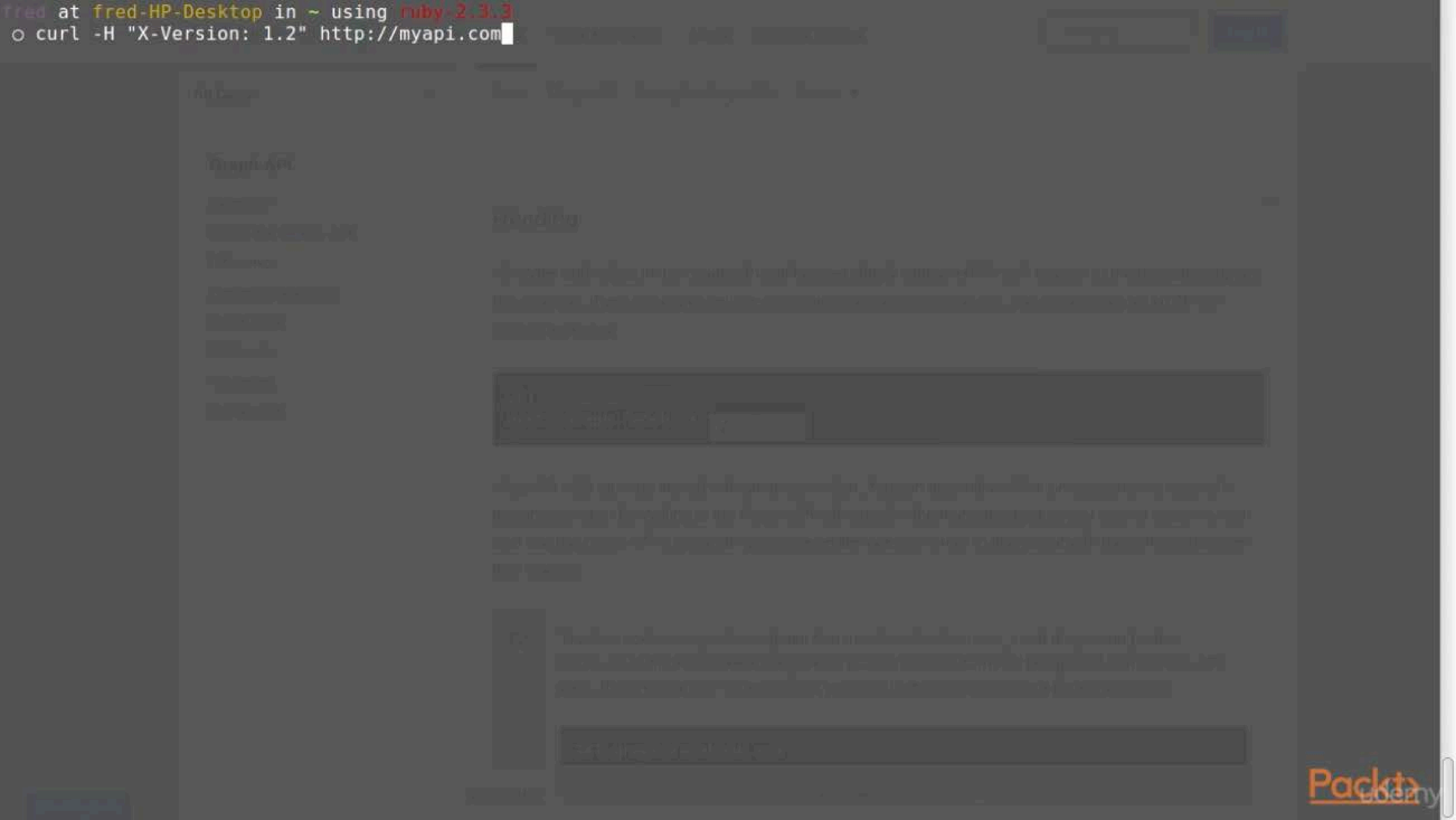
All nodes and edges in the Graph API can be read simply with an HTTP `GET` request to the relevant endpoint. For example, if you wanted to retrieve information about the current user, you would make an HTTP `GET` request as below:

```
GET /v2.5/me HTTP/1.1
Host: graph.facebook.com
```

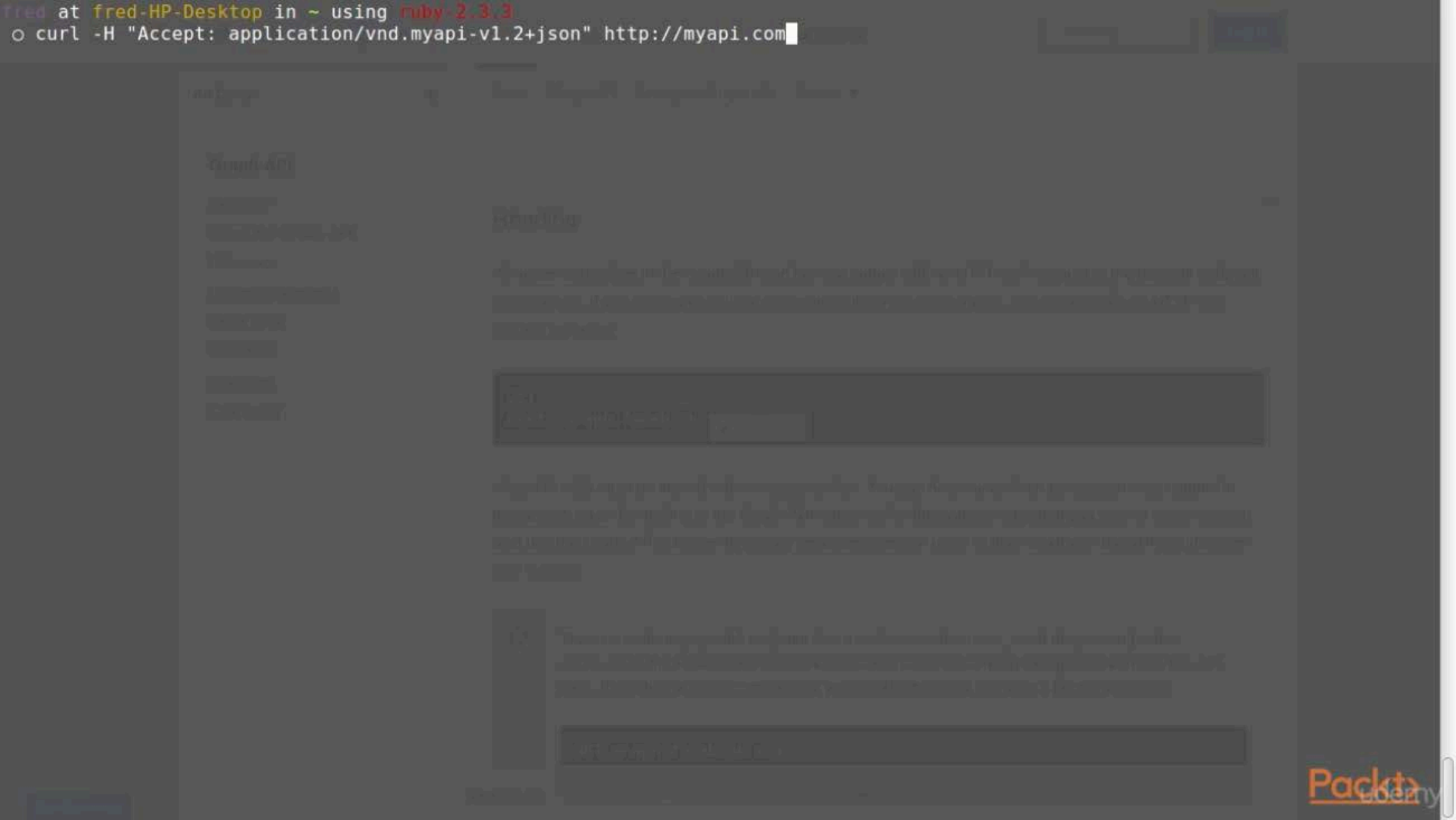
Most API calls must be signed with an access token. You can determine which permissions are needed in this access token by looking at the Graph API reference for the node or edge that you wish to read. You can also use the Graph API Explorer to quickly generate tokens in order to play about with the API and discover how it works.

The `/me` node is a special endpoint that translates to the `user_id` of the person (or the `page_id` of the Facebook Page) whose access token is currently being used to make the API calls. If you had a user access token, you could retrieve all of a user's photos by using:

```
GET graph.facebook.com
  /me/photos
```



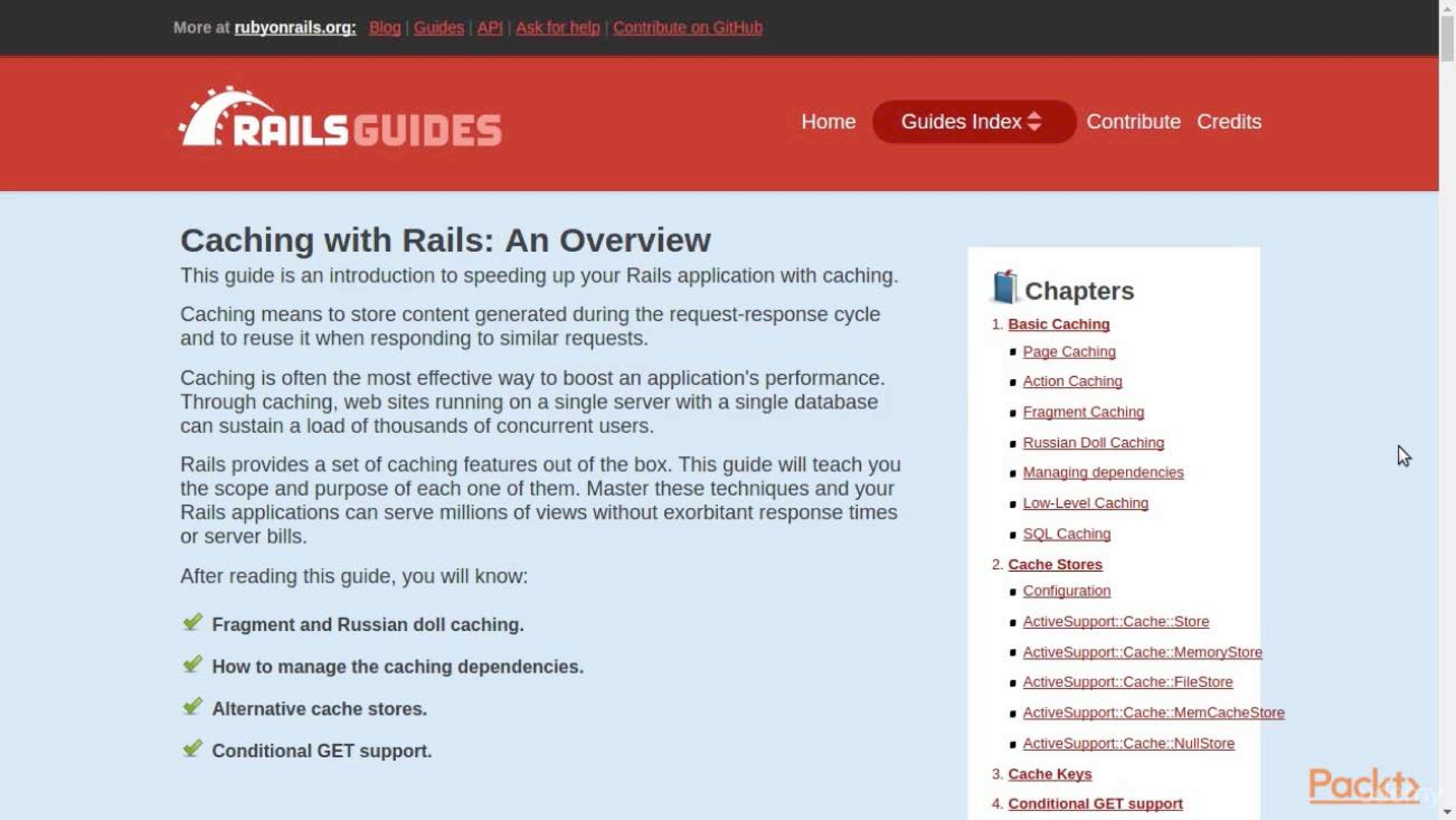
```
fred at fred-HP-Desktop in ~ using ruby-2.3.3
○ curl -H "X-Version: 1.2" http://myapi.com
```



```
fred at fred-HP-Desktop in ~ using ruby-2.3.3
○ curl -H "Accept: application/vnd.myapi-v1.2+json" http://myapi.com
```

Areas to Consider

- Media types
- Versioning
- Caching
- Authentication
- Security



More at rubyonrails.org: Blog | Guides | API | Ask for help | Contribute on GitHub

Caching with Rails: An Overview

This guide is an introduction to speeding up your Rails application with caching.

Caching means to store content generated during the request-response cycle and to reuse it when responding to similar requests.

Caching is often the most effective way to boost an application's performance. Through caching, web sites running on a single server with a single database can sustain a load of thousands of concurrent users.

Rails provides a set of caching features out of the box. This guide will teach you the scope and purpose of each one of them. Master these techniques and your Rails applications can serve millions of views without exorbitant response times or server bills.

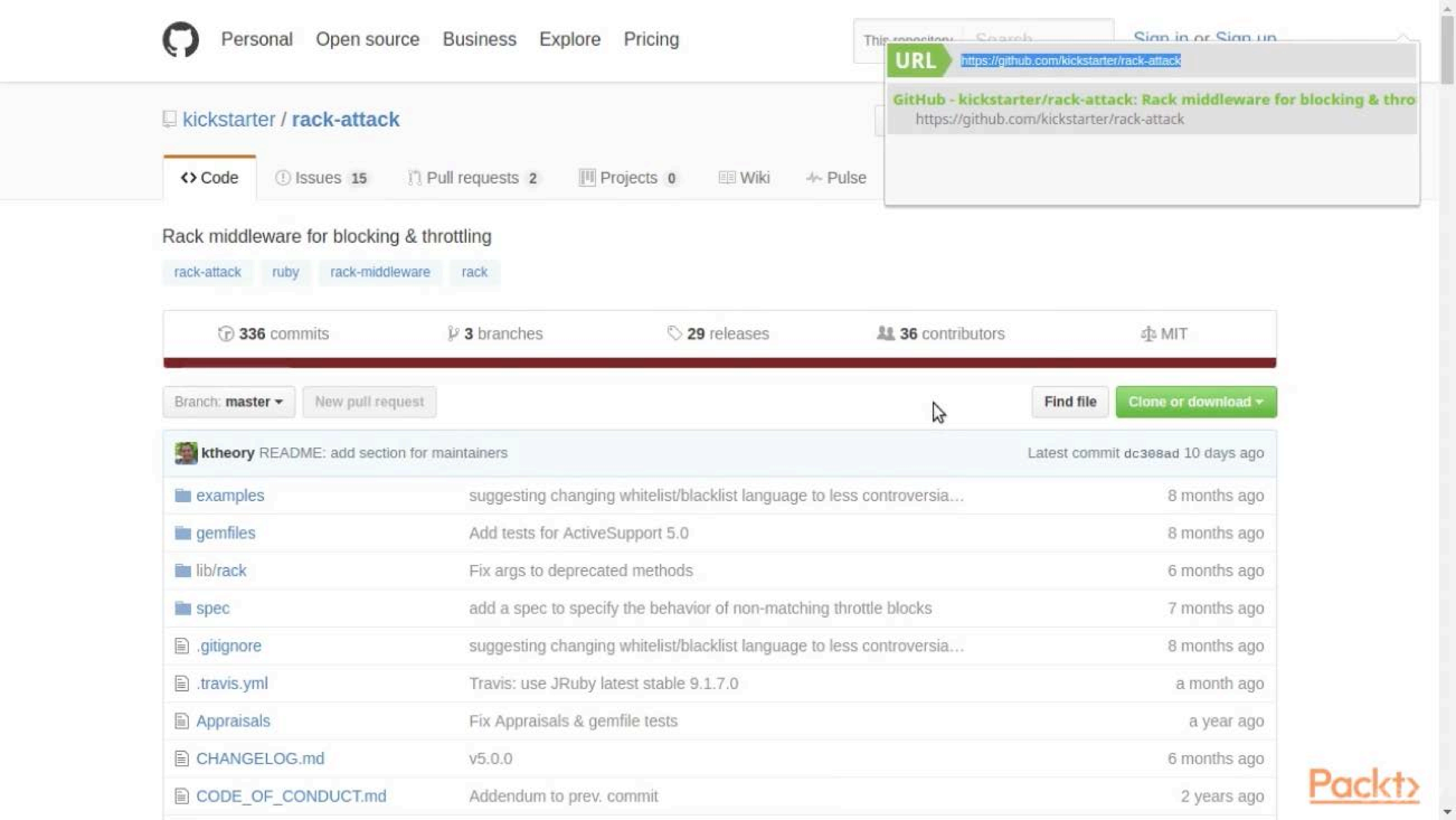
After reading this guide, you will know:

- ✓ **Fragment and Russian doll caching.**
- ✓ **How to manage the caching dependencies.**
- ✓ **Alternative cache stores.**
- ✓ **Conditional GET support.**

Chapters

Areas to Consider

- Media types
- Versioning
- Caching
- Authentication
- Security



URL https://github.com/kickstarter/rack-attack

GitHub - kickstarter/rack-attack: Rack middleware for blocking & thro
https://github.com/kickstarter/rack-attack

kickstarter / rack-attack

Rack middleware for blocking & throttling

ktheory README: add section for maintainers — Latest commit dc308ad 10 days ago

examples	suggesting changing whitelist/blacklist language to less controversia…	8 months ago
gemfiles	Add tests for ActiveSupport 5.0	8 months ago
lib/rack	Fix args to deprecated methods	6 months ago
spec	add a spec to specify the behavior of non-matching throttle blocks	7 months ago
.gitignore	suggesting changing whitelist/blacklist language to less controversia…	8 months ago
.travis.yml	Travis: use JRuby latest stable 9.1.7.0	a month ago
Appraisals	Fix Appraisals & gemfile tests	a year ago
CHANGELOG.md	v5.0.0	6 months ago
CODE_OF_CONDUCT.md	Addendum to prev. commit	2 years ago

Areas to Consider

- Media types
- Versioning
- Caching
- Authentication
- Security

Why Split Functionality into APIs?

- Can be re-used
- Easier to test
- Easier to maintain
- Flexible deployment and scaling

Creating an API Project with Rails

Next Video