
SURVIVING APIs
WITH
RAILS

REST, ROUTES, CONSTRAINTS AND NAMESPACES

LEVEL 1

A QUICK LOOK AT A WEB-BASED SERVICE.

FROM THE PERSPECTIVE OF AN END USER.

HOW A MUSIC STREAMING SERVICE WORKS

Create a Playlist from Web App

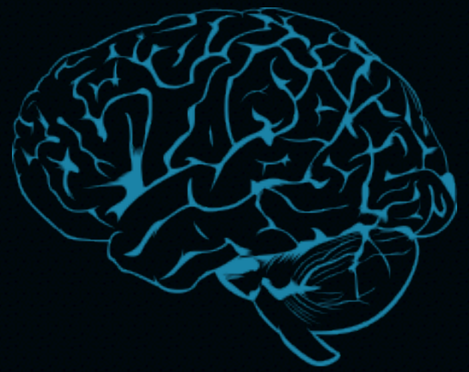
Create **Playlist A**

Access Playlist from Android app

Read **Playlist A**

Access Playlist from iPhone app

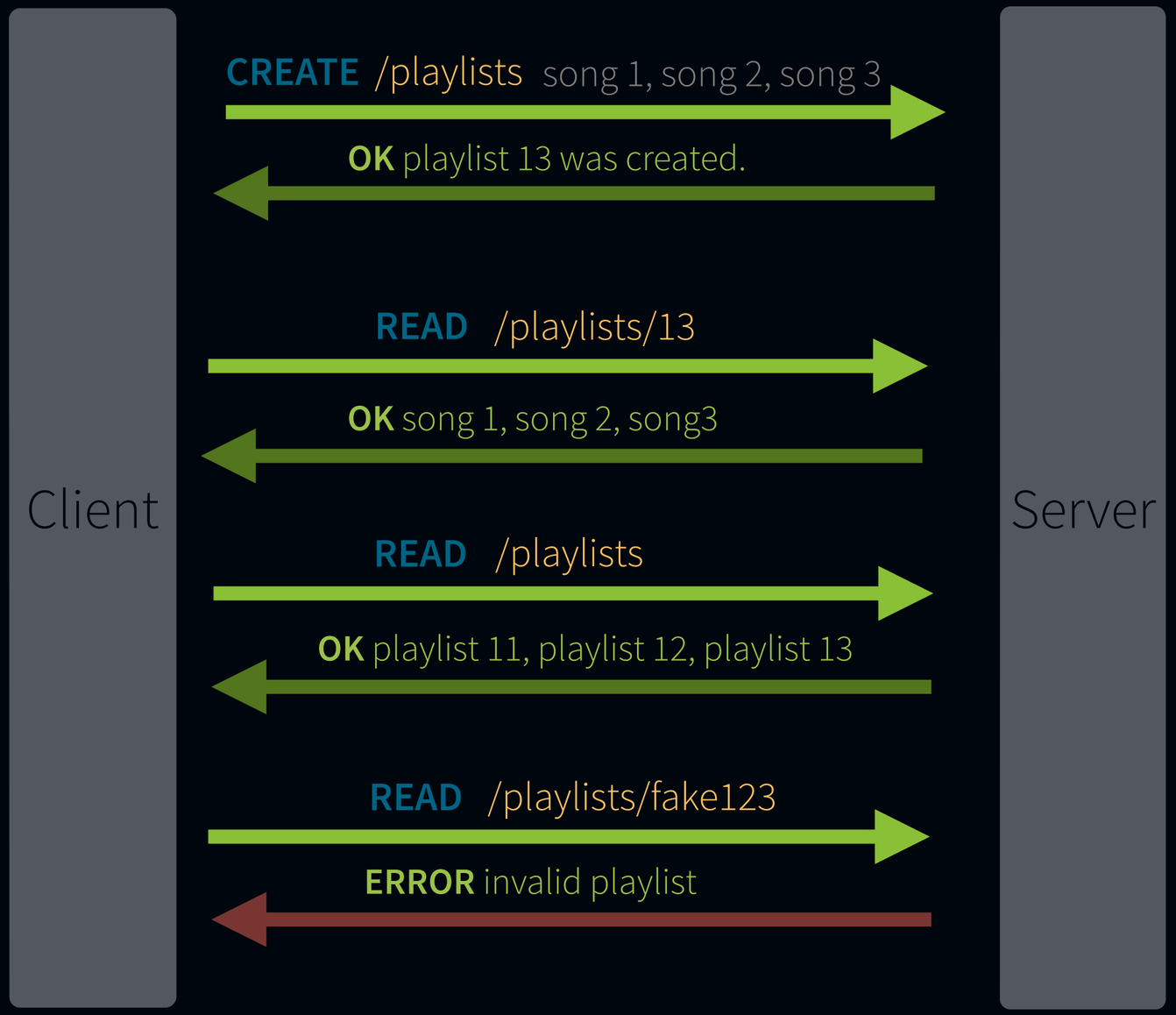
Read **Playlist A**



Web API

SURVIVING APIs WITH RAILS

HOW THE CLIENT AND SERVER COMMUNICATE



Notice the pattern:

A set of **commands** performed on **things** generates **responses.**

This is the foundation of a **REST API.**

REpresentational **S**tate **T**ransfer

By following a **strict set of operations**, REST allows building an infrastructure that can support different types of applications

WHERE OUR API STARTS

Routes are a good starting point for implementing our API.

config/routes.rb

```
resources :zombies
```

$ rake routes

lists all the routes, URIs, Controller#actions, and url helpers available on our app.

```
$ rake routes

      Prefix Verb   URI Pattern                  Controller#Action
     zombies GET    /zombies(.:format)           zombies#index
             POST   /zombies(.:format)           zombies#create
  new_zombie GET    /zombies/new(.:format)       zombies#new
 edit_zombie GET    /zombies/:id/edit(.:format)  zombies#edit
      zombie GET    /zombies/:id(.:format)       zombies#show
             PATCH  /zombies/:id(.:format)       zombies#update
             PUT    /zombies/:id(.:format)       zombies#update
             DELETE /zombies/:id(.:format)       zombies#destroy
```

append **_path** or **_url** for helper methods.

RESTRICTING ROUTES WITH EXCEPT

Restricting routes improves **memory use** and **speeds up the routing process.**

config/routes.rb

```
resources :zombies, except: :destroy
```

```
      Prefix Verb   URI Pattern                  Controller#Action
     zombies GET    /zombies(.:format)           zombies#index
             POST   /zombies(.:format)           zombies#create
  new_zombie GET    /zombies/new(.:format)       zombies#new
 edit_zombie GET    /zombies/:id/edit(.:format)  zombies#edit
      zombie GET    /zombies/:id(.:format)       zombies#show
             PATCH  /zombies/:id(.:format)       zombies#update
             PUT    /zombies/:id(.:format)       zombies#update
             DELETE /zombies/:id(.:format)       zombies#destroy
```

the **destroy** action is unreachable

RESTRICTING ROUTES WITH ONLY

config/routes.rb

```
resources :zombies, only: :index
```

the only reachable action is **index**

```
      Prefix Verb   URI Pattern                       Controller#Action
     zombies GET    /zombies(.:format)                zombies#index
             POST   /zombies(.:format)                zombies#create
  new_zombie GET    /zombies/new(.:format)            zombies#new
 edit_zombie GET    /zombies/:id/edit(.:format)       zombies#edit
      zombie GET    /zombies/:id(.:format)            zombies#show
             PATCH  /zombies/:id(.:format)            zombies#update
             PUT    /zombies/:id(.:format)            zombies#update
             DELETE /zombies/:id(.:format)            zombies#destroy
```

RESTRICTING ROUTES WITH MULTIPLE ACTIONS

Both **:only** and **:except** can also take an **array** of actions.

config/routes.rb

```
resources :zombies, only: [:index, :show]
resources :humans,  except: [:destroy, :edit, :update]
```

```
    Prefix Verb URI Pattern                Controller#Action
   zombies GET  /zombies(.:format)         zombies#index
    zombie GET  /zombies/:id(.:format)     zombies#show
    humans GET  /humans(.:format)          humans#index
           POST /humans(.:format)          humans#create
 new_human GET  /humans/new(.:format)      humans#new
     human GET  /humans/:id(.:format)      humans#show
```

SURVIVING APIs WITH RAILS

USING WITH_OPTIONS ON ROUTES

The **with_options** method is an elegant way to factor duplication out of options passed to a series of method calls.

config/routes.rb

```
resources :zombies, only: :index
resources :humans, only: :index
resources :medical_kits, only: :index
...
```

same thing

config/routes.rb

```
with_options only: :index do |list_only|
  list_only.resources :zombies
  list_only.resources :humans
  list_only.resources :medical_kits
  ...
end
```

options passed as arguments are automatically added to the resources

USING CONSTRAINTS TO ENFORCE SUBDOMAIN

Keeping our API under its own **subdomain** allows **load balancing** traffic at the DNS level.

config/routes.rb

```
resources :episodes
resources :zombies, constraints: { subdomain: 'api' }
resources :humans, constraints: { subdomain: 'api' }
```

http://cs-zombies.com/episodes

same thing

config/routes.rb

```
resources :episodes

constraints subdomain: 'api' do
  resources :zombies
  resources :humans
end
```

http://api.cs-zombies.com/zombies
http://api.cs-zombies.com/humans

USING SUBDOMAINS IN DEVELOPMENT

Network configuration for supporting subdomains in development.

/etc/hosts

```
127.0.0.1    cs-zombies-dev.com
127.0.0.1    api.cs-zombies-dev.com
```

On Windows, look for
C:\WINDOWS\system32\drivers\etc\hosts

urls available on local machine

http://cs-zombies-dev.com **:3000**
http://api.cs-zombies-dev.com **:3000**

port number of local server
is still required

Other alternatives:

- **POW** - http://pow.cx
- **LVH.me** - http://lvh.me

KEEPING WEB AND API CONTROLLERS ORGANIZED

It's not uncommon to use the same Rails code base for both web **site** and web **API.**

config/routes.rb

```
constraints subdomain: 'api' do
  resources :zombies
end

resources :pages
```

serves web site, not web API

web API and web site controllers are all under the same folder

```
      Prefix Verb   URI Pattern                Controller#Action
     zombies GET    /zombies(.:format)         zombies#index {:subdomain=>"api"}
             POST   /zombies(.:format)         zombies#create {:subdomain=>"api"}
  new_zombie GET    /zombies/new(.:format)     zombies#new {:subdomain=>"api"}
        ...
       pages GET    /pages(.:format)           pages#index
             POST   /pages(.:format)           pages#create
    new_page GET    /pages/new(.:format)       pages#new
        ...
```

VIVING APIs WITH RAILS

USING NAMESPACES TO KEEP CONTROLLERS ORGANIZED

It's a good practice to keep web controllers **separate** from API controllers.

config/routes.rb

```
constraints subdomain: 'api' do
  namespace :api do
    resources :zombies
  end
end

resources :pages
```

most useful when web site and web API share the **same** code base

web API controllers have their own namespace

```
        Prefix Verb   URI Pattern                  Controller#Action
   api_zombies GET    /api/zombies(.:format)       api/zombies#index {:subdomain=>"api"}
               POST   /api/zombies(.:format)       api/zombies#create {:subdomain=>"api"}
new_api_zombie GET    /api/zombies/new(.:format)   api/zombies#new {:subdomain=>"api"}
          ...
         pages GET    /pages(.:format)             pages#index
               POST   /pages(.:format)             pages#create
      new_page GET    /pages/new(.:format)         pages#new
          ...
```

USING NAMESPACES TO KEEP CONTROLLERS ORGANIZED

config/routes.rb

```
constraints subdomain: 'api' do
  namespace :api do
    resources :zombies
  end
end

resources :pages
```

app/controllers/api/zombies_controller.rb

```
module Api
  class ZombiesController < ApplicationController
  end
end
```

web API controllers are part of the API module

app/controllers/pages_controller.rb

```
class PagesController < ApplicationController
end
```

web site controllers remain on top-level namespace

REMOVING DUPLICATION FROM THE URL

config/routes.rb

```
constraints subdomain: 'api' do
  namespace :api do
    resources :zombies
  end
end
```

http://api.cs-zombies.com/api/zombies

Unnecessary duplication

```
        Prefix Verb   URI Pattern                       Controller#Action
    api_zombies GET    /api/zombies(.:format)          api/zombies#index {:subdomain=>"api"}
                POST   /api/zombies(.:format)          api/zombies#create {:subdomain=>"api"}
 new_api_zombie GET    /api/zombies/new(.:format)      api/zombies#new {:subdomain=>"api"}
edit_api_zombie GET    /api/zombies/:id/edit(.:format) api/zombies#edit {:subdomain=>"api"}
     api_zombie GET    /api/zombies/:id(.:format)      api/zombies#show {:subdomain=>"api"}
                PATCH  /api/zombies/:id(.:format)      api/zombies#update {:subdomain=>"api"}
                PUT    /api/zombies/:id(.:format)      api/zombies#update {:subdomain=>"api"}
                DELETE /api/zombies/:id(.:format)      api/zombies#destroy {:subdomain=>"api"}
...
```

REMOVING DUPLICATION FROM THE URL

config/routes.rb

```
constraints subdomain: 'api' do
  namespace :api, path: '/' do
    resources :zombies
  end
end
```

http://api.cs-zombies.com/zombies

namespace is preserved

```
      Prefix Verb   URI Pattern                Controller#Action
  api_zombies GET    /zombies(.:format)         api/zombies#index {:subdomain=>"api"}
              POST   /zombies(.:format)         api/zombies#create {:subdomain=>"api"}
 new_api_zombie GET    /zombies/new(.:format)     api/zombies#new {:subdomain=>"api"}
edit_api_zombie GET    /zombies/:id/edit(.:format) api/zombies#edit {:subdomain=>"api"}
    api_zombie GET    /zombies/:id(.:format)     api/zombies#show {:subdomain=>"api"}
              PATCH  /zombies/:id(.:format)     api/zombies#update {:subdomain=>"api"}
              PUT    /zombies/:id(.:format)     api/zombies#update {:subdomain=>"api"}
              DELETE /zombies/:id(.:format)     api/zombies#destroy {:subdomain=>"api"}
...
```

USING A SHORTER SYNTAX FOR CONSTRAINTS AND NAMESPACES

config/routes.rb

```
constraints subdomain: 'api' do
  namespace :api, path: '/' do
    resources :zombies
    resources :humans
  end
end
```

same thing

```
namespace :api, path: '/', constraints: { subdomain: 'api' } do
  resources :zombies
  resources :humans
end
```

API CASE CONSISTENCY

app/controllers/api/zombies_controller.rb

```ruby
module Api
  class ZombiesController < ApplicationController
  end
end
```

config/initializers/inflections.rb

acronyms are commonly all-caps

```ruby
ActiveSupport::Inflector.inflections(:en) do |inflect|
  inflect.acronym 'API'
end
```

app/controllers/api/zombies_controller.rb

```ruby
module API
  class ZombiesController < ApplicationController
  end
end
```

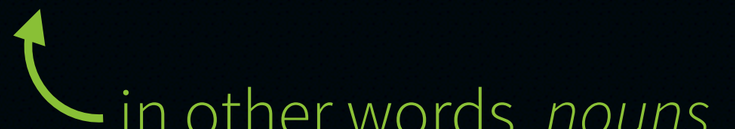
RESOURCES AND GET

LEVEL 2

IT'S ALL ABOUT THE RESOURCES

Any information that **can be named** can be a resource.

Some examples of resources:



- A music playlist
- A song
- The leader of the Zombie horde
- Survivors
- Remaining Medical Kits

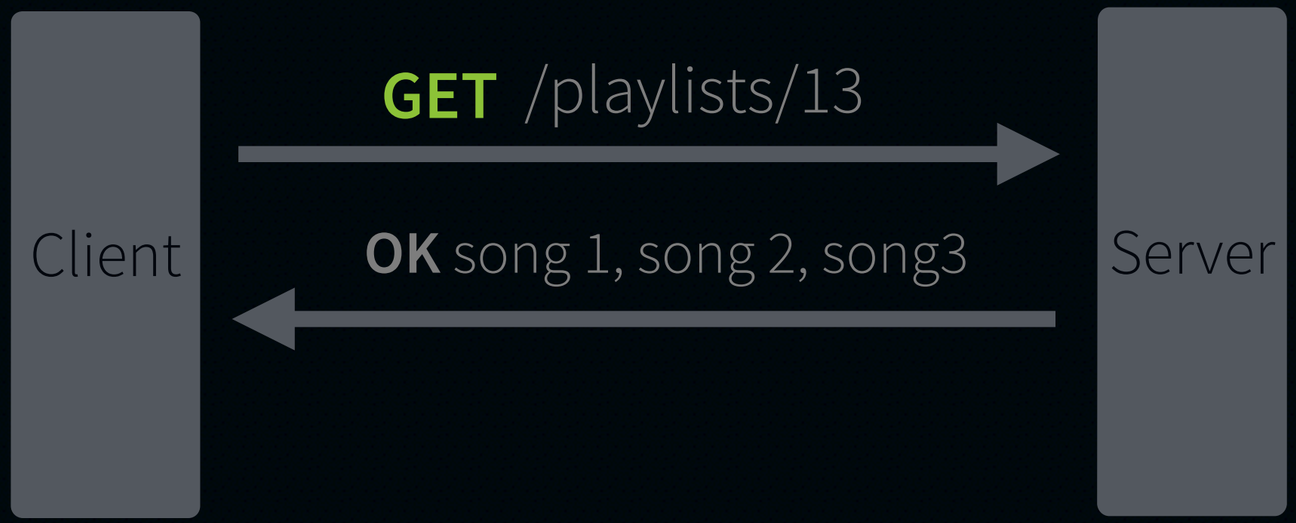
"A resource is a **conceptual mapping** to a set of entities, not the entity that corresponds to the mapping at any particular point in time."

- Steve Klabnik, **Designing Hypermedia APIs**

SURVIVING APIs WITH RAILS

UNDERSTANDING THE GET METHOD

The GET method is used to **read** information identified by a given **URI.**



Important characteristics:

- **Safe** - it should **not** take any action other than retrieval.
- **Idempotent** - sequential **GET** requests to the same URI should not generate side-effects.

SURVIVING APIs WITH RAILS

WRITING API INTEGRATION TESTS

Integration tests simulate clients interacting with our **API**

config/routes.rb

```
namespace :api, path: '/', constraints: { subdomain: 'api' } do
  resources :zombies
end
```

test/integration/listing_zombies_test.rb

```
require 'test_helper'

class ListingZombiesTest < ActionDispatch::IntegrationTest

  setup { host! 'api.example.com' }

  test 'returns list of all zombies'
end
```

required for testing
with subdomain constraint

WRITING OUR FIRST INTEGRATION TESTS

TEST FAILING

200 - Success status code means the request has succeeded

test/integration/listing_zombies_test.rb

```ruby
require 'test_helper'

class ListingZombiesTest < ActionDispatch::IntegrationTest

  setup { host! 'api.example.com' }

  test 'returns list of all zombies' do
    get '/zombies'
    assert_equal 200, response.status
    refute_empty response.body
  end
end
```

same thing

```ruby
assert response.success?
```

200 responses should include the resource in the response body

LISTING RESOURCES

app/controllers/api/zombies_controller.rb

```
module API
  class ZombiesController < ApplicationController
    def index
      zombies = Zombie.all
      render json: zombies, status: 200
    end
  end
end
```

calls the **to_json** method on **zombies**

See the **Rails 4 Patterns** course to learn about **ActiveModel Serializers**

The **to_json** method serializes all properties to JSON

```
zombies.to_json
```


```
[{"id":5,"name":"Joanna","age":null,"created_at":"2014-01-17T18:40:40.195Z",
"updated_at":"2014-01-17T18:40:40.195Z","weapon":"axe"},
{"id":6,"name":"John","age":null,"created_at":"2014-01-17T18:40:40.218Z",
"updated_at":"2014-01-17T18:40:40.218Z","weapon":"shotgun"}]
```


```
zombie.to_json
```


```
{"id":5,"name":"Joanna","age":null,"created_at":"2014-01-17T18:40:40.195Z",
"updated_at":"2014-01-17T18:40:40.195Z","weapon":"axe"}
```

PATH SEGMENTED EXPANSION

Arguments in the URI are separated using a **slash.**

```
/zombies
/zombies/:id
/zombies/:id/victims
/zombies/:id/victims/:id
```

```
/zombies?id=1
```

this routes to Zombies#index
and **NOT** to Zombies#show

MOST URIS WILL NOT DEPEND ON QUERY STRINGS

Sometimes it's ok to use query strings on URIs.

```
/zombies?weapon=axe
```
filters

```
/zombies?keyword=john
```
searches

```
/zombies?page=2&per_page=25
```
pagination

TEST LISTING RESOURCES WITH QUERY STRINGS

TEST FAILING

test/integration/listing_zombies_test.rb

```ruby
class ListingZombiesTest < ActionDispatch::IntegrationTest
  setup { host! 'api.example.com' }

  test 'returns zombies filtered by weapon' do
    john = Zombie.create!(name: 'John', weapon: 'axe')
    joanna = Zombie.create!(name: 'Joanna', weapon: 'shotgun')

    get '/zombies?weapon=axe'
    assert_equal 200, response.status

    zombies = JSON.parse(response.body, symbolize_names: true)
    names = zombies.collect { |z| z[:name] }
    assert_includes names, 'John'
    refute_includes names, 'Joanna'
  end
end
```

if creation logic gets too verbose,
use **fixtures** or **FactoryGirl.**

```
{'id' => 51, 'name' => "John"}
```

```
{:id => 51, :name => "John"}
```

LISTING RESOURCES WITH FILTER

TEST PASSING

app/controllers/api/zombies_controller.rb

```
module API
  class ZombiesController < ApplicationController

    def index
      zombies = Zombie.all
      if weapon = params[:weapon]
        zombies = zombies.where(weapon: weapon)
      end
      render json: zombies, status: 200
    end

  end
end
```

Starting in Rails 4, this returns a **chainable scope**

we can add filters dynamically

TEST RETRIEVING ONE ZOMBIE

TEST FAILING

test/integration/listing_zombies_test.rb

```
class ListingZombiesTest < ActionDispatch::IntegrationTest
  setup { host! 'api.example.com' }

  test 'returns zombie by id' do
    zombie = Zombie.create!(name: 'Joanna', weapon: 'axe')
    get "/zombies/#{zombie.id}"
    assert_equal 200, response.status

    zombie_response = JSON.parse(response.body, symbolize_names: true)
    assert_equal zombie.name, zombie_response[:name]
  end
end
```

routes to **Zombies#show**

RETURNING ONE ZOMBIE

TEST PASSING

The **:status** option accepts either numbers or symbols.

app/controllers/api/zombies_controller.rb

```
module API
  class ZombiesController < ApplicationController
    def show
      zombie = Zombie.find(params[:id])
      render json: zombie, status: 200
    end
  end
end
```

same thing

```
render json: zombie, status: :ok
```

Visit http://guides.rubyonrails.org/layouts_and_rendering.html
for a list of all numeric status codes and symbols supported by Rails.

LOOKS LIKE WE HAVE SOME DUPLICATION

test/integration/listing_zombies_test.rb

```ruby
class ListingZombiesTest < ActionDispatch::IntegrationTest
  setup { host! 'api.example.com' }

  test 'returns zombie by id' do
    zombie = Zombie.create!(name: 'Joanna', weapon: 'axe')
    get "/zombies/#{zombie.id}"
    assert_equal 200, response.status

    zombie_response = JSON.parse(response.body, symbolize_names: true)
    assert_equal zombie.name, zombie_response[:name]
  end
end
```

this method is used multiple times
across integration tests

USING OUR NEW TEST HELPER

test/integration/listing_zombies_test.rb

```ruby
class ListingZombiesTest < ActionDispatch::IntegrationTest
  setup { host! 'api.example.com' }

  test 'returns zombie by id' do
    zombie = Zombie.create!(name: 'Joanna', weapon: 'axe')
    get "/zombies/#{zombie.id}"
    assert_equal 200, response.status

    zombie_response = json(response.body)
    assert_equal zombie.name, zombie_response[:name]
  end
end
```

EXTRACTING COMMON CODE INTO A TEST HELPER

test/test_helper.rb

```
ENV["RAILS_ENV"] ||= "test"
require File.expand_path('../../config/environment', __FILE__)
require 'rails/test_help'

class ActiveSupport::TestCase
  ActiveRecord::Migration.check_pending!
  fixtures :all

  def json(body)
    JSON.parse(body, symbolize_names: true)
  end
end
```

can be reused across all tests

USING CURL TO TEST OUR API WITH REAL NETWORK REQUESTS

curl is a command line tool that issues **real HTTP requests** over the network

defaults to **GET** requests

```
$ curl http://api.cs-zombies-dev.com:3000/zombies
```

A lot of tools use curl as part of their installation process.
For example, **rvm**

RVM is the Ruby enVironment Manager (rvm).

It manages Ruby application environments and enables switching between them.

Installation

```
curl -L https://get.rvm.io | bash -s stable --autolibs=enabled [--ruby] [--rails] [--trace]
```

LOOKING AT THE RESPONSE BODY USING CURL

curl displays the response body on the command line

defaults to **GET** requests

```
$ curl http://api.cs-zombies-dev.com:3000/zombies

[{"id":5,"name":"Joanna","age":null,"created_at":"2014-01-17T18:40:40.195Z",
"updated_at":"2014-01-17T18:40:40.195Z","weapon":"axe"},
{"id":6,"name":"John","age":null,"created_at":"2014-01-17T18:40:40.218Z",
"updated_at":"2014-01-17T18:40:40.218Z","weapon":"shotgun"}]
```

Curl is is shipped with **OS X** and most **GNU/Linux** distributions.
For **Windows** installer, visit http://curl.haxx.se/download.html

USING CURL WITH OPTIONS

works with query strings too!

```
$ curl http://api.cs-zombies-dev.com:3000/zombies?weapon=axe

[{"id":7,"name":"Joanna","age":123,"created_at":"2014-01-17T18:42:47.026Z",
"updated_at":"2014-01-17T18:42:47.026Z","weapon":"axe"}]
```

use the **-I** option to only display response headers

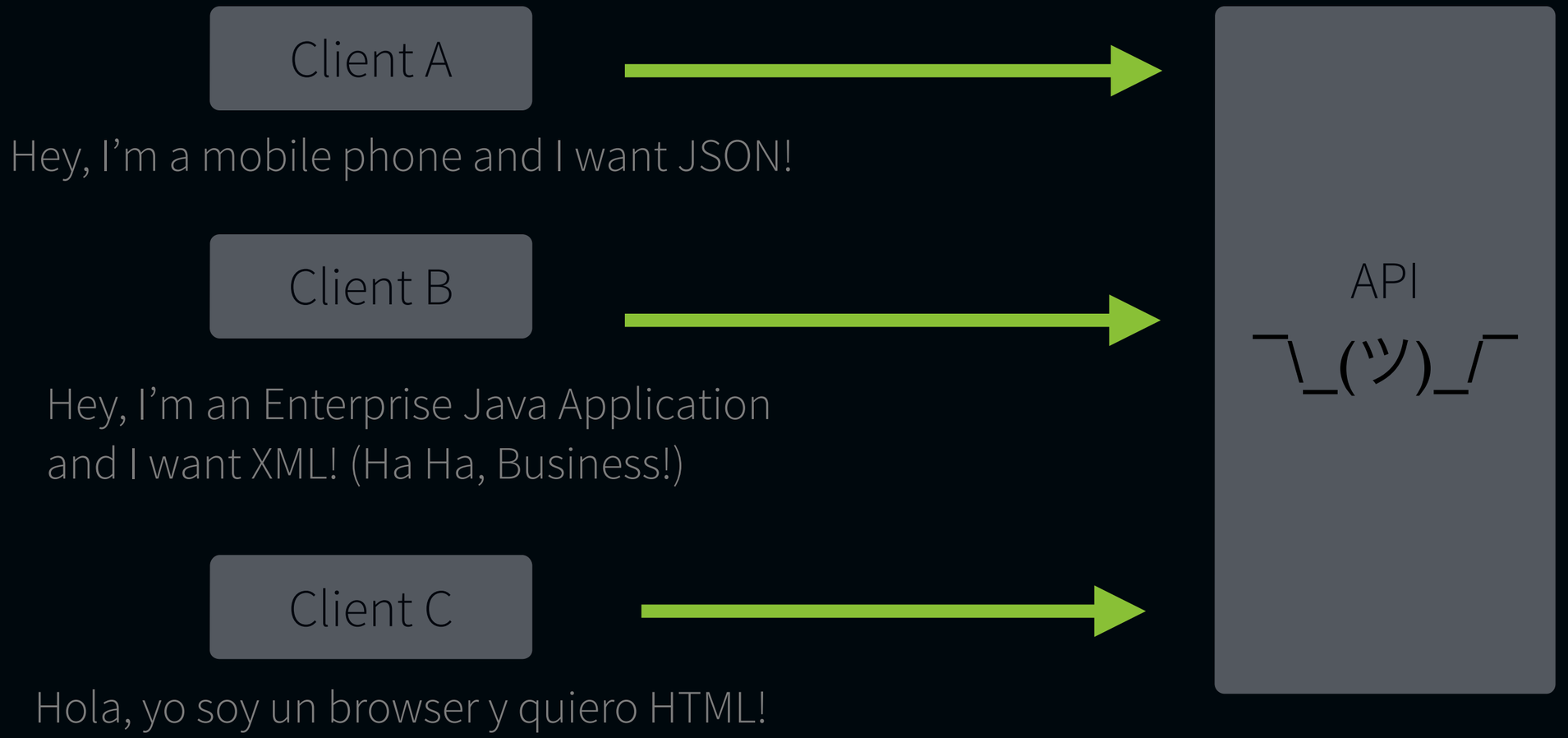
```
$ curl -I http://api.cs-zombies-dev.com:3000/zombies/7

HTTP/1.1 200 OK
X-Frame-Options: SAMEORIGIN
X-XSS-Protection: 1; mode=block
X-Content-Type-Options: nosniff
X-UA-Compatible: chrome=1
Content-Type: application/json; charset=utf-8
...
```

CONTENT NEGOTIATION

LEVEL 3

DIFFERENT CLIENTS NEED DIFFERENT FORMATS

Web APIs need to cater to different types of clients.



UNDERSTANDING CONTENT NEGOTIATION

The process in which client and server determine the best representation for a response when many are available.

SETTING THE RESPONSE FORMAT FROM THE URI

Rails allows switching formats by adding an extension to the URI

config/routes.rb

```
resources :zombies
```

http://api.cs-zombies.com/zombies.json ← returns JSON

http://api.cs-zombies.com/zombies.xml ← returns XML

```
      Prefix Verb   URI Pattern                       Controller#Action
     zombies GET    /zombies(.:format)                zombies#index
             POST   /zombies(.:format)                zombies#create
  new_zombie GET    /zombies/new(.:format)            zombies#new
 edit_zombie GET    /zombies/:id/edit(.:format)       zombies#edit
      zombie GET    /zombies/:id(.:format)            zombies#show
             ...
```

This is a nicety from Rails and it is NOT a standard.

Do NOT write API clients that rely on URI extensions.

USING THE ACCEPT HEADER TO REQUEST A MEDIA TYPE

TEST FAILING

Media types specify the scheme for resource representations.

used to be called Mime Types.

test/integration/listing_zombies_test.rb

```
class ListingZombiesTest < ActionDispatch::IntegrationTest
  test 'returns zombies in JSON' do
    get '/zombies', {}, { 'Accept' => Mime::JSON }
    assert_equal 200, response.status
    assert_equal Mime::JSON, response.content_type
  end

  test 'returns zombies in XML' do
    get '/zombies', {}, { 'Accept' => Mime::XML }
    assert_equal 200, response.status
    assert_equal Mime::XML, response.content_type
  end
end
```

USING RESPOND_TO TO SERVE JSON

The respond_to method allows controllers to serve different formats.

app/controllers/zombies_controller.rb

```
class ZombiesController < ApplicationController
  def index
    zombies = Zombie.all

    respond_to do |format|
      format.json { render json: zombies, status: 200 }
    end
  end
end
```

response body in JSON

```
[{“id":1,"name":"Jon","age":123,
"created_at":"2013-12-12T20:01:24.586Z",
“updated_at":"2013-12-12T20:01:24.586Z"},
{“id”:2,"name":"Isabella","age":93,
...
```

CHECKING RESPONSE HEADERS FOR PROPER FORMAT

The Content-Type header indicates the media type of the response

app/controllers/zombies_controller.rb

```
class ZombiesController < ApplicationController
  def index
    zombies = Zombie.all

    respond_to do |format|
      format.json { render json: zombies, status: 200 }
    end
  end
end
```

response headers

```
HTTP/1.1 200 OK
Content-Type: application/json
```

checking response headers is very useful for debugging

USING RESPOND_TO TO SERVE XML

TEST PASSING

app/controllers/zombies_controller.rb

```
class ZombiesController < ApplicationController
  def index
    zombies = Zombie.all

    respond_to do |format|
      format.json { render json: zombies, status: 200 }
      format.xml { render xml: zombies, status: 200 }
    end
  end
end
```

response body in XML

```
<?xml version="1.0" encoding="UTF-8"?>
<zombies type="array">
  <zombie>
    <id type="integer">1</id>
    <name>Jon</name>
    ...
```

USING RESPOND_TO TO SERVE XML

TEST PASSING

app/controllers/zombies_controller.rb

```
class ZombiesController < ApplicationController
  def index
    zombies = Zombie.all

    respond_to do |format|
      format.json { render json: zombies, status: 200 }
      format.xml { render xml: zombies, status: 200 }
    end
  end
end
```

response headers

```
HTTP/1.1 200 OK
Content-Type: application/xml
```

LISTING MEDIA TYPES

Rails ships with 21 different media types out of the box.

```
Mime::SET
```

← lists all supported media types

```
=> [#<Mime::Type:0x007fe2e1dc0f48 @synonyms=["application/xhtml+xml"], @symbol=:html,
@string="text/html">,#<Mime::Type:0x007fe2e1dc07c8 @synonyms=[], @symbol=:text,
@string="text/plain">, #<Mime::Type:0x007fe2e1dc0048@synonyms=["application/javascript",
"application/x-javascript"], @symbol=:js, @string="text/javascript">,#<Mime::Type:
0x007fe2e218b9e8 @synonyms=[], @symbol=:css, @string="text/css">,...
```

```
Mime::SET.collect(&:to_s)
```

← more readable output

```
=> ["text/html", "text/plain", "text/javascript", "text/css", "text/calendar", "text/csv",
"text/vcard","image/png", "image/jpeg", "image/gif", "image/bmp", "image/tiff", "video/
mpeg", "application/xml","application/rss+xml", "application/atom+xml", "application/x-
yaml", "multipart/form-data","application/x-www-form-urlencoded", "application/json",
"application/pdf", "application/zip"]
```

USING CUSTOM HEADERS TO TEST DIFFERENT RESPONSE FORMATS

send custom request headers with the -H option

```
$ curl -IH "Accept: application/json" localhost:3000/zombies

HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8
```

response in JSON format

```
$ curl -IH "Accept: application/xml" localhost:3000/zombies

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

response in XML format

TESTING WITH THE LANGUAGE SET TO ENGLISH

TEST FAILING

Use the Accept-Language request header for language negotiation

test/integration/changing_locales_test.rb

English is the default locale

```
class ChangingLocalesTest < ActionDispatch::IntegrationTest

  test 'returns list of zombies in english' do
    get '/zombies', {}, {'Accept-Language' => 'en', 'Accept' => Mime::JSON }
    assert_equal 200, response.status
    zombies = json(response.body)
    assert_equal "Watch out for #{zombies[0][:name]}!", zombies[0][:message]
  end

  ...
end
```

message in English

TESTING WITH THE LANGUAGE SET TO PORTUGUESE

TEST FAILING

test/integration/changing_locales_test.rb

```
class ChangingLocalesTest < ActionDispatch::IntegrationTest
   ...

   test 'returns list of zombies in portuguese' do
     get '/zombies', {}, {'Accept-Language' => 'pt-BR', 'Accept' => Mime::JSON }
     assert_equal 200, response.status
     zombies = json(response.body)
     assert_equal "Cuidado com #{zombies[0][:name]}!", zombies[0][:message]
   end
end
```

message in Portuguese

SETTING THE LANGUAGE FOR THE RESPONSE

Use request.headers to access request headers.

app/controllers/application_controller.rb

```ruby
class ApplicationController < ActionController::Base
  protect_from_forgery with: :exception
  before_action :set_locale

  protected
    def set_locale
      I18n.locale = request.headers['Accept-Language']
    end
end
```

returns language accepted by client

Use the I18n.locale method to set application wide locale

USING JBUILDER TO RETURN LOCALIZED JSON

Jbuilder provides a DSL for generating JSON.

app/controllers/zombies_controller.rb

```
class ZombiesController < ApplicationController
  def index
    @zombies = Zombie.all
    respond_to do |format|
      format.json { render json: @zombies, status: 200 }
    end
  end
end
```

when no block is used,
then a view is required

app/views/zombies/index.json.jbuilder

```
json.array!(@zombies) do |zombie|
  json.extract! zombie, :id, :name, :age
  json.message I18n.t('zombie_message', name: zombie.name)
end
```

looks up translation

USING LOCALE TEMPLATES

TEST PASSING

Files under config/locales are included in the translations load path.

config/locales/en.yml

```yaml
en:
  zombie_message: 'Watch out for %{name}!'
```

syntax for interpolation

JSON response in english:

```
[{"id":2,"name":"Joanna","age":251,"message":"Watch out for Joanna!"}]
```

config/locales/pt-BR.yml ← this one needs to be created

```yaml
pt-BR:
  zombie_message: 'Cuidado com %{name}!'
```

JSON response in portuguese:

```
[{"id":2,"name":"Joanna","age":251,"message":"Cuidado com Joanna!"}]
```

USING THE HTTP_ACCEPT_LANGUAGE GEM

Use the http_accept_language gem for a more robust support for locales.

Gemfile

```
gem 'http_accept_language'
```

- Sanitizes the list of preferred languages.
- Sorts list of preferred languages.
- Finds best fit if multiple languages supported.

app/controllers/application_controller.rb

```
class ApplicationController < ActionController::Base
  protect_from_forgery with: :exception
  before_action :set_locale

  protected
    def set_locale
      locales = I18n.available_locales
      I18n.locale = http_accept_language.compatible_language_from(locales)
    end
end
```

Returns [:en, :"pt-BR"]

checks header and returns the first language compatible with the available locales

POST, PUT, PATCH AND DELETE

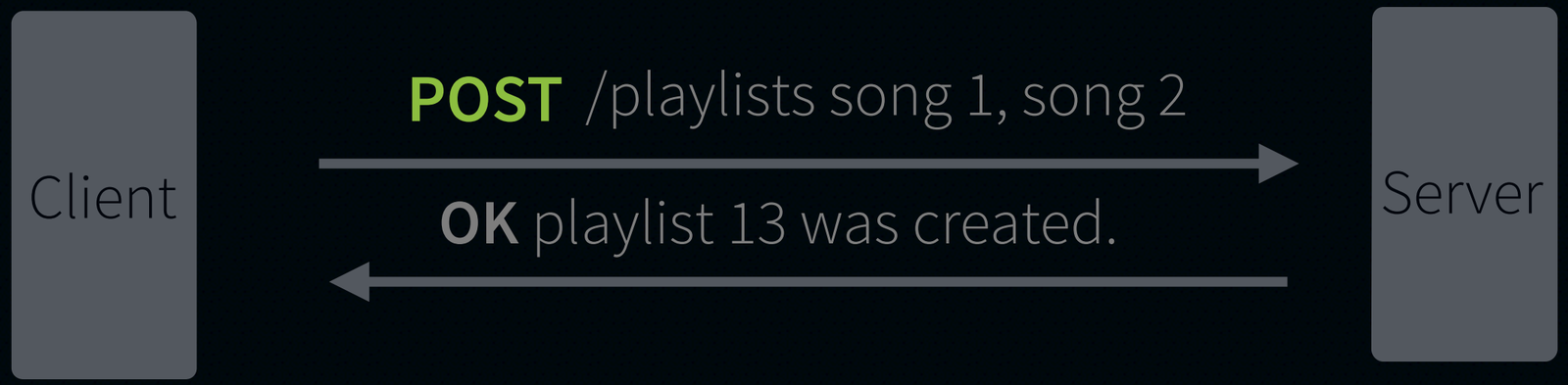
LEVEL 4

POST.

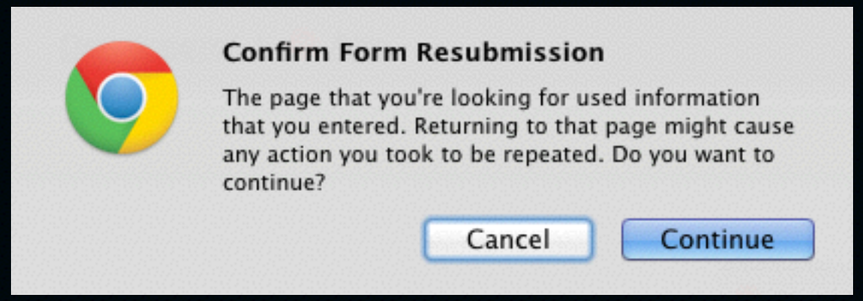
CREATING RESOURCES.

THE POST METHOD

The POST method is used to **create** new resources.



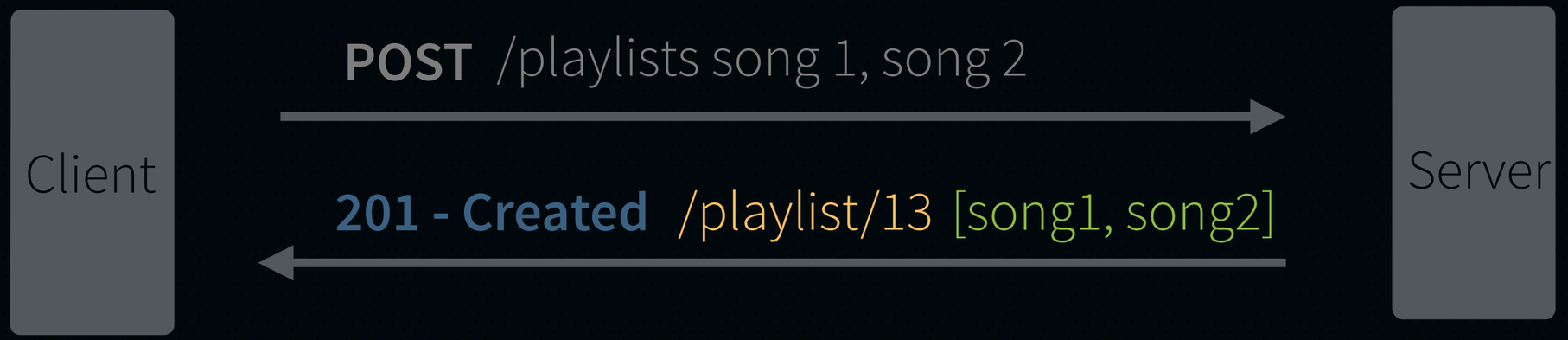
POST is neither safe or idempotent:



some browsers display a warning before re-issuing **post** requests

RESPONDING SUCCESSFULLY TO POST METHODS

A couple of things are expected from a successful POST request:



- The status code for the response should be **201 - Created**.
- The response body should contain a representation of the new resource.
- The Location header should be set with the location of the new resource.

201 - Created means the request has been fulfilled and resulted in a new resource being created

SURVIVING APIs WITH RAILS

INTEGRATION TESTING THE POST METHOD

TEST FAILING

test/integration/creating_episodes_test.rb

```
class CreatingEpisodesTest < ActionDispatch::IntegrationTest

  test 'creates episodes' do
    post '/episodes',
      { episode:
        { title: 'Bananas', description: 'Learn about bananas.' }
      }.to_json,
      { 'Accept' => Mime::JSON, 'Content-Type' => Mime::JSON.to_s }

    assert_equal 201, response.status
    assert_equal Mime::JSON, response.content_type

    episode = json(response.body)
    assert_equal episode_url(episode[:id]), response.location
  end
end
```

representation of the new episode resource in **JSON**

tells server to use JSON parsing for the content body

client can use **location** to fetch resource if needed

RESPONDING TO A SUCCESSFUL POST REQUEST

TEST PASSING

Use the **location** option on render to send back the resources' full url

app/controllers/episodes_controller.rb

```
class EpisodesController < ApplicationController

  def create
    episode = Episode.new(episode_params)
    if episode.save
      render json: episode, status: 201, location: episode
    end
  end

  private
    def episode_params
      params.require(:episode).permit(:title, :description)
    end
end
```

resolves to full resource url,
i.e. http://cs-zombies.com/episodes/3

JSON representation
of the new resource

same thing

```
status: :created
```

POSTING DATA WITH CURL

curl can help detect errors not caught by tests.

the **-X** option specifies the **method**

```
$ curl -i -X POST -d 'episode[title]=ZombieApocalypseNow' \
  http://localhost:3000/episodes

HTTP/1.1 422 Unprocessable Entity
Content-Type: text/html; charset=utf-8

...
```

use **-d** to send data on the request

422 - Unprocessable Entity means the **client** submitted request was well-formed but **semantically invalid.**

FORGERY PROTECTION IS DISABLED ON TEST

Rails checks for an **authenticity token** on POST, PUT/PATCH and DELETE.

app/controllers/application_controller.rb

```
class ApplicationController < ActionController::Base
  # Prevent CSRF attacks by raising an exception.
  # For APIs, you may want to use :null_session instead.
  protect_from_forgery with: :exception
end
```

raises error and returns a 422 response

Defaults to **disabled** on test environment.

config/environments/test.rb

the reason why CSRF error isn't raised during tests

```
# Disable request forgery protection in test environment.
config.action_controller.allow_forgery_protection = false
```

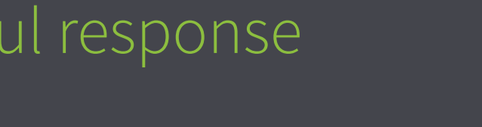
USING EMPTY SESSIONS ON API REQUESTS

API calls should be **stateless.**

app/controllers/application_controller.rb

```
class ApplicationController < ActionController::Base
  # Prevent CSRF attacks by raising an exception.
  # For APIs, you may want to use :null_session instead.
  protect_from_forgery with: :null_session
end
```

```
$ curl -i -X POST -d 'episode[title]=ZombieApocalypseNow' \
  http://localhost:3000/episodes

HTTP/1.1 201 Created
Location: http://localhost:3000/episodes/1
Content-Type: application/json; charset=utf-8
...
```



SUCCESSFUL RESPONSES WITH NO CONTENT

Some successful responses might **not** need to include a response **body**

Ajax responses can be made a lot faster with no response body.

app/controllers/episodes_controller.rb

```ruby
class EpisodesController < ApplicationController

  def create
    episode = Episode.new(episode_params)
    if episode.save
      render nothing: true, status: 204, location: episode
    end
  end

  ...
end
```

returns empty response body

204 - No Content means the server has fulfilled the request but does not need to return an entity-body

USING HEAD FOR HEADERS-ONLY RESPONSES

The **head** method creates a response consisting solely of **HTTP headers.**

app/controllers/episodes_controller.rb

```
class EpisodesController < ApplicationController

  def create
    episode = Episode.new(episode_params)
    if episode.save
      head 204, location: episode
    end
  end

  ...
end
```

```
head :no_content
```

same thing

can take arguments

TESTING UNSUCCESSFUL POST REQUESTS

TEST FAILING

When a client sends a POST request and the server is **not able** to understand.

test/integration/creating_episodes_test.rb

```
class CreatingEpisodesTest < ActionDispatch::IntegrationTest
  test 'does not create episodes with title nil' do
    post '/episodes',
      { episode:
        { title: nil, description: 'Learn about bananas.' }
      }.to_json,
      { 'Accept' => Mime::JSON, 'Content-Type' => Mime::JSON.to_s }

    assert_equal 422, response.status
    assert_equal Mime::JSON, response.content_type
  end
end
```

RESPONDING TO UNSUCCESSFUL POST REQUESTS

TEST PASSING

Use the **422** status code when creating a new resource is not successful.

app/models/episode.rb

```
class Episode < ActiveRecord::Base
  validates :title, presence: true
end
```

app/controllers/episodes_controller.rb

```
class EpisodesController < ApplicationController
  def create
    episode = Episode.new(episode_params)
    if episode.save
      render json: episode, status: :created, location: episode
    else
      render json: episode.errors, status: 422
    end
  end
  ...
end
```

helps API clients figure out the cause of the error

same thing

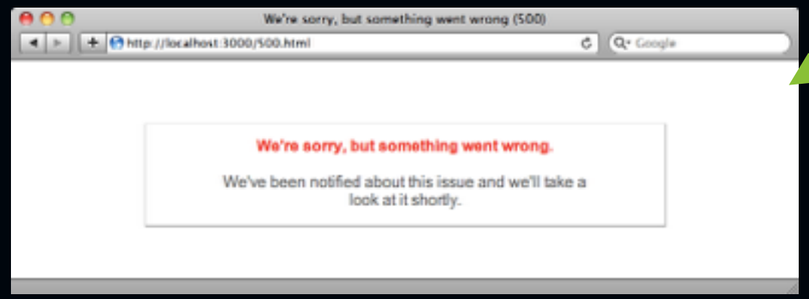
```
status: :unprocessable_entity
```

UNDERSTANDING INTERNAL SERVER ERRORS

Rails automatically handles server errors and returns a **500** response.

The best way to prevent **500 - Internal Server Errors** is to write tests.

Default Rails server error page



500 - Internal Server Error means the server encountered an unexpected condition which prevented it from fulfilling the request

PUT/PATCH.

UPDATING RESOURCES.

PUT IS FOR REPLACING RESOURCES

PUT and PATCH are used for updating existing resources.

```
      Prefix Verb   URI Pattern                       Controller#Action
    episodes GET    /episodes(.:format)               episodes#index
             POST   /episodes(.:format)               episodes#create
 new_episode GET    /episodes/new(.:format)           episodes#new
edit_episode GET    /episodes/:id/edit(.:format)      episodes#edit
     episode GET    /episodes/:id(.:format)           episodes#show
             PATCH  /episodes/:id(.:format)           episodes#update
             PUT    /episodes/:id(.:format)           episodes#update
             DELETE /episodes/:id(.:format)           episodes#destroy
```

contradicts the spec and
it's used for **partial** updates

"The PUT method requests that the enclosed entity be stored under the supplied Request-URI." - RFC 2616 / HTTP 1.1

PATCH IS FOR PARTIAL UPDATES TO EXISTING RESOURCES

Always use **PATCH** for partial updates.

```
      Prefix Verb   URI Pattern                       Controller#Action
    episodes GET    /episodes(.:format)               episodes#index
             POST   /episodes(.:format)               episodes#create
 new_episode GET    /episodes/new(.:format)           episodes#new
edit_episode GET    /episodes/:id/edit(.:format)      episodes#edit
     episode GET    /episodes/:id(.:format)           episodes#show
             PATCH  /episodes/:id(.:format)           episodes#update
             PUT    /episodes/:id(.:format)           episodes#update
             DELETE /episodes/:id(.:format)           episodes#destroy
```

Both **PUT** and **PATCH** are routed to the **update** action

"(...) the entity contains a list of differences between the original version of the resource and the desired content of the resource after the PATCH action has been applied" - RFC 2068 / HTTP 1.1

INTEGRATION TESTS WITH PATCH

TEST FAILING

Use the **patch** helper method on integration tests.

test/integration/updating_episodes_test.rb

```ruby
class UpdatingEpisodesTest < ActionDispatch::IntegrationTest
  setup { @episode = Episode.create!(title: 'First Title') }

  test 'successful update' do
    patch "/episodes/#{@episode.id}",
      { episode: { title: 'First Title Edit' } }.to_json,
      { 'Accept' => Mime::JSON, 'Content-Type' => Mime::JSON.to_s }

    assert_equal 200, response.status
    assert_equal 'First Title Edit', @episode.reload.title
  end
end
```

reloads record from the database

SUCCESSFUL UPDATES WITH PATCH

TEST PASSING

The **update** method only updates values that differ from the existing record.

app/controllers/episodes_controller.rb

```
class EpisodesController < ApplicationController

  def update
    episode = Episode.find(params[:id])
    if episode.update(episode_params)
      render json: episode, status: 200
    end
  end

  private
    def episode_params
      params.require(:episode).permit(:title, :description)
    end
end
```

successful response

UNSUCCESSFUL UPDATES WITH PATCH

TEST FAILING

A client sends a PATCH request and the server is **not able** to understand.

test/integration/updating_episodes_test.rb

```
class UpdatingEpisodesTest < ActionDispatch::IntegrationTest

  test 'unsuccessful update on short title' do
    patch "/episodes/#{@episode.id}",
      { episode: { title: 'short' } }.to_json,
      { 'Accept' => Mime::JSON, 'Content-Type' => Mime::JSON.to_s }

    assert_equal 422, response.status
  end
end
```

RESPONDING TO UNSUCCESSFUL UPDATES

TEST PASSING

app/models/episode.rb

```
class Episode < ActiveRecord::Base
  validates :title, length: { minimum: 10 }
end
```

app/controllers/episodes_controller.rb

```
class EpisodesController < ApplicationController
  def update
    episode = Episode.find(params[:id])
    if episode.update(episode_params)
      render json: episode, status: 200
    else
      render json: episode.errors, status: 422
    end
  end
  ...
end
```

client submitted invalid data

DELETE.

DISCARDING RESOURCES.

INTEGRATION TESTS WITH DELETE

TEST FAILING

The **DELETE** method indicates client is **not interested** in the given resource.

test/integration/deleting_episodes_test.rb

```
class DeletingEpisodesTest < ActionDispatch::IntegrationTest
  setup { @episode = Episode.create(title: 'I am going to be deleted') }

  test 'deletes existing episode' do
    delete "/episodes/#{@episode.id}"
    assert_equal 204, response.status
  end
end
```

response includes no content

The server can implement DELETE in a couple different ways...

RESPONDING TO DELETE BY DESTROYING RECORDS

TEST PASSING

Server **deletes** the record from the database.

app/controllers/episodes_controller.rb

```
class EpisodesController < ApplicationController

  def destroy
    episode = Episode.find(params[:id])
    episode.destroy
    head 204
  end
end
```

removes the record from the database

```
Episode.find(1)
=> ActiveRecord::RecordNotFound: Couldn't find Episode with id=1
```

SURVIVING APIs WITH RAILS

RESPONDING TO DELETE BY ARCHIVING RECORDS

Flag records as **archived** and new finder for **unarchived** records.

app/models/episode.rb

```
class Episode < ActiveRecord::Base
  def find_unarchived(id)
    find_by!(id: id, archived: false)
  end

  def archive
    self.archived = true
    self.save
  end
end
```

returns record only if **not** archived

does **not** destroy,
just flags as **archived**

RESPONDING TO DELETE METHODS BY ARCHIVING RECORD

TEST PASSING

The destroy action now uses the new methods.

app/controllers/episodes_controller.rb

```
class EpisodesController < ApplicationController

  def destroy
    episode = Episode.find_unarchived(params[:id])
    episode.archive
    head 204
  end
end
```

does **not** call destroy

does **not** remove record from the database

```
Episode.find(3)
=> #<Episode id: 3, title: "Third EP", archived: true,
created_at: "2014-02-11 12:49:12", updated_at: "2014-02-11 12:49:12">
```

RESPONDING WITH UNARCHIVED RECORDS

All other actions should also use the new methods.

app/controllers/episodes_controller.rb

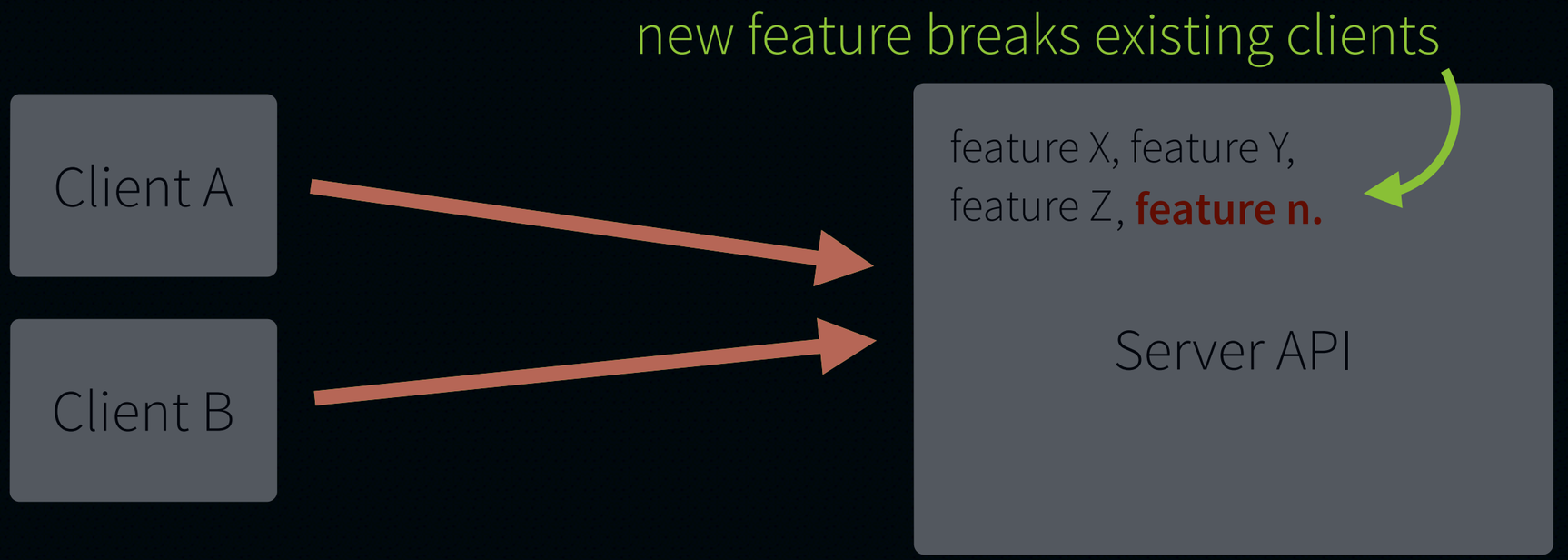
```
class EpisodesController < ApplicationController

  def index
    episodes = Episode.where(archived: false)
    render json: episodes, status: 200
  end

  def show
    episode = Episode.find_unarchived(params[:id])
    render json: episode, status: 200
  end

  ...
end
```

API VERSIONING

LEVEL 5

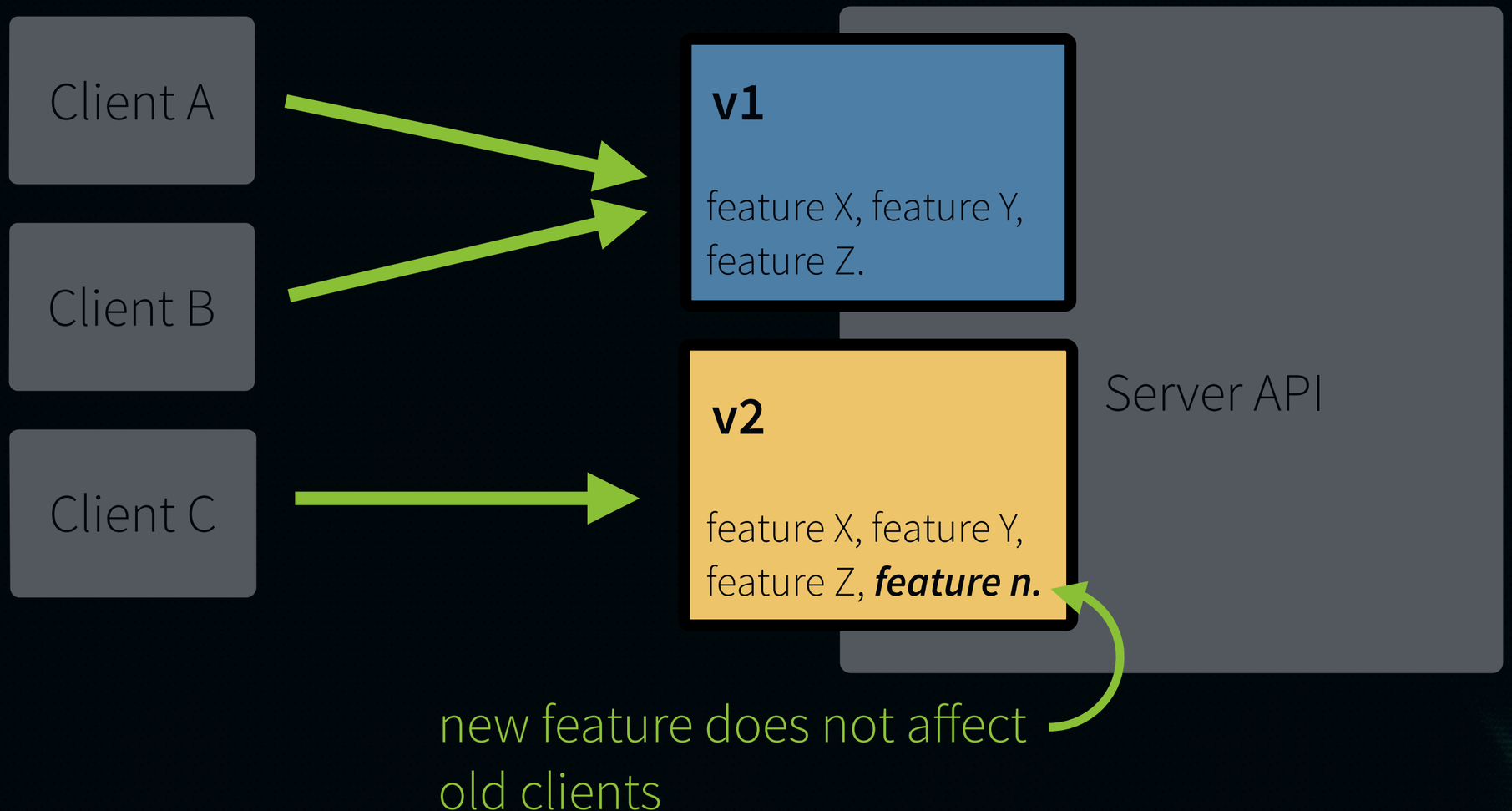
INTRODUCING CHANGES TO A LIVE API

Changes to the API cannot disrupt existing clients.



API VERSIONING

Versioning helps prevent **major changes** from **breaking** existing clients.



SURVIVING APIs
WITH
RAILS

URI VERSIONING

THE VERSION NUMBER IS PART OF THE URI.

VERSIONING USING THE URI

Namespaces help isolate controllers from different versions.

config/routes.rb

```
namespace :v1 do
  resources :zombies
end

namespace :v2 do
  resources :zombies
end
```

No need for **api** namespace.
Our app **strictly** serves an API.

http://api.cs-zombies.com/**v1**/zombies

http://api.cs-zombies.com/**v2**/zombies

```
          Prefix Verb   URI Pattern                  Controller#Action
  api_v1_zombies GET    /v1/zombies(.:format)        v1/zombies#index
                 POST   /v1/zombies(.:format)        v1/zombies#create
                 ...
  api_v2_zombies GET    /v2/zombies(.:format)        v2/zombies#index
                 POST   /v2/zombies(.:format)        v2/zombies#create
                 ...
```

TESTING ROUTES FOR URI VERSIONING

TEST FAILING

Routes test helps ensure URIs are routed to proper controller#action.

test/integration/routes_test.rb

```
class RoutesTest < ActionDispatch::IntegrationTest
  test 'routes version' do
    assert_generates '/v1/zombies', { controller: 'v1/zombies', action: 'index' }
    assert_generates '/v2/zombies', { controller: 'v2/zombies', action: 'index' }
  end
end
```

The **assert_generates** method asserts the provided options can be used to generate the expected path.

GIVING EACH VERSION ITS OWN NAMESPACE

TEST PASSING

requests for **/v1/zombies**

app/controllers/v1/zombies_controller.rb

```
module V1
  class ZombiesController < ApplicationController
  end
end
```

requests for **/v2/zombies**

app/controllers/v2/zombies_controller.rb

```
module V2
  class ZombiesController < ApplicationController
  end
end
```

REUSING CODE ACROSS DIFFERENT VERSIONS

app/controllers/v1/zombies_controller.rb

```ruby
module V1
  class ZombiesController < ApplicationController
    before_action ->{ @remote_ip = request.headers['REMOTE_ADDR'] }

    def index
      render json: "#{@remote_ip} Version One!", status: 200
    end
  end
end
```

duplication

app/controllers/v2/zombies_controller.rb

```ruby
module V2
  class ZombiesController < ApplicationController
    before_action ->{ @remote_ip = request.headers['REMOTE_ADDR'] }

    def index
      render json: "#{@remote_ip} Version Two!", status: 200
    end
  end
end
```

WRITING AN INTEGRATION TEST AS A SAFETY NET

TEST FAILING

test/integration/changing_api_versions_test.rb

```ruby
class ChangingApiVersionsTest < ActionDispatch::IntegrationTest

  setup { @ip = '123.123.12.12' }

  test '/v1 returns version 1' do
    get '/v1/zombies', {}, { 'REMOTE_ADDR' => @ip }
    assert_equal 200, response.status
    assert_equal "#{@ip} Version One!", response.body
  end

  test '/v2 returns version 2' do
    get '/v2/zombies', {}, { 'REMOTE_ADDR' => @ip }
    assert_equal 200, response.status
    assert_equal "#{@ip} Version Two!", response.body
  end
end
```

notice the version is **part of the URI**

EXTRACTING COMMON CODE TO BASE API CONTROLLER

TEST PASSING

Because our app now **strictly** serves a web API,
it's ok to use **ApplicationController** as the base class.

app/controllers/application_controller.rb

```
class ApplicationController < ActionController::Base

  before_action ->{ @remote_ip = request.headers['REMOTE_ADDR'] }

end
```

runs before all actions across controllers for all versions

CHILD CONTROLLERS INHERITING PROPERTIES FROM BASE CLASS

app/controllers/v1/zombies_controller.rb

```ruby
module V1
  class ZombiesController < ApplicationController
    def index
      render json: "#{@remote_ip} using version 1", status: 200
    end
  end
end
```

from ApplicationController

app/controllers/v2/zombies_controller.rb

```ruby
module V2
  class ZombiesController < ApplicationController
    def index
      render json: "#{@remote_ip} using version 2", status: 200
    end
  end
end
```

REUSING A VERSION SPECIFIC FEATURE

app/controllers/v2/zombies_controller.rb

```
module V2
  class ZombiesController < ApplicationController
    before_action :audit_logging_for_v2

    def audit_logging_for_v2
  end
end
```

duplication

app/controllers/v2/episodes_controller.rb

```
module V2
  class EpisodesController < ApplicationController
    before_action :audit_logging_for_v2

    def audit_logging_for_v2
  end
end
```

REUSING A VERSION SPECIFIC FEATURE

app/controllers/application_controller.rb

```
class ApplicationController < ActionController::Base
  before_action :audit_logging_for_v2

  def audit_logging_for_v2
end
```

runs for all versions

new base class for V2

app/controllers/v2/version_controller.rb

```
module V2
  class VersionController < ApplicationController
    before_action :audit_logging_for_v2

    def audit_logging_for_v2
  end
end
```

ADDING A VERSION SPECIFIC FEATURE

Using **abstract!** makes our code easier to understand.

app/controllers/v2/version_controller.rb

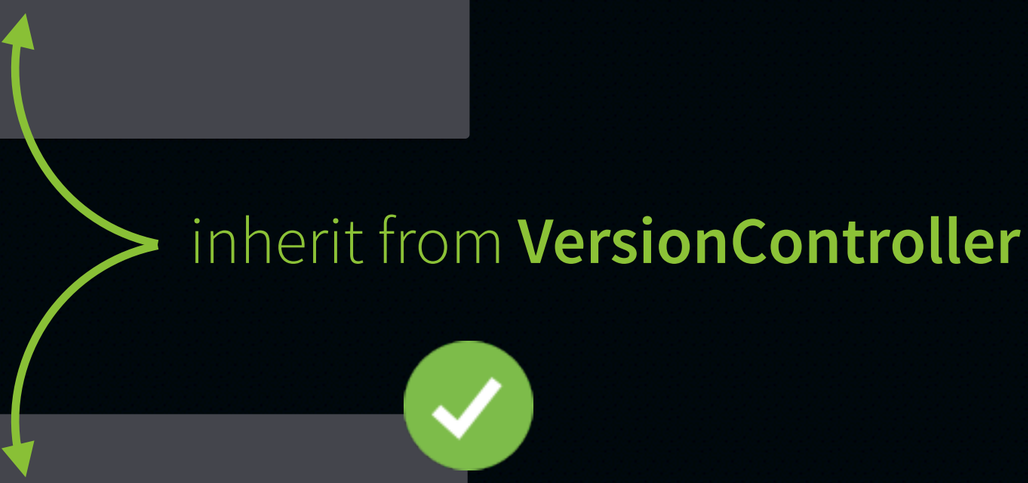
```
module V2
  class VersionController < ApplicationController
    abstract!

    before_action :audit_logging_for_v2

    def audit_logging_for_v2
  end
end
```

prevents this class from exposing its own methods as public actions

REUSING A VERSION SPECIFIC FEATURE

app/controllers/v2/zombies_controller.rb

```
module V2
  class ZombiesController < VersionController
  end
end
```



app/controllers/v2/episodes_controller.rb

```
module V2
  class EpisodesController < VersionController
  end
end
```

ACCEPT HEADER VERSIONING

THE VERSION NUMBER IS PART OF THE ACCEPT HEADER.

VERSIONING USING CUSTOM MEDIA TYPE AND THE ACCEPT HEADER

`application/json` JSON

`application/xml` XML

Custom media type

`application/vnd.apocalypse[.version]+json`

API version as part of the mime type

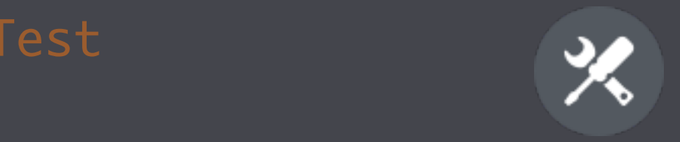
`application`	payload is **application-specific**
`vnd.apocalypse`	media type is **vendor-specific**
`[.version]`	API version
`+json`	response **format** should be JSON

INTEGRATION TESTING API VERSIONS USING THE ACCEPT HEADER

TEST FAILING

test/integration/changing_api_versions_test.rb

```
class ChangingApiVersionsTest < ActionDispatch::IntegrationTest

  setup { @ip = '123.123.12.12' }

  test 'returns version one via Accept header' do
    get '/zombies', {},
        { 'REMOTE_ADDR' => @ip, 'Accept' => 'application/vnd.apocalypse.v1+json' }

    assert_equal 200, response.status
    assert_equal "#{@ip} Version One!", response.body
    assert_equal Mime::JSON, response.content_type
  end

  ...
end
```



version one

standard format back makes it easier for different API clients to understand

INTEGRATION TESTING API VERSIONS USING THE ACCEPT HEADER

TEST FAILING

test/integration/changing_api_versions_test.rb

```
class ChangingApiVersionsTest < ActionDispatch::IntegrationTest

  setup { @ip = '123.123.12.12' }
  ...

  test 'returns version two via Accept header' do
    get '/zombies', {},
        { 'REMOTE_ADDR' => @ip, 'Accept' => 'application/vnd.apocalypse.v2+json' }

    assert_equal 200, response.status
    assert_equal "#{@ip} Version Two!", response.body
    assert_equal Mime::JSON, response.content_type
  end
end
```

version two

WRITING A ROUTE CONSTRAINT CLASS TO CHECK VERSION

lib/api_version.rb

```
class ApiVersion

  def initialize(version, default=false)
    @version, @default = version, default
  end

  def matches?(request)
    @default || check_headers(request.headers)
  end

  private
    def check_headers(headers)
      accept = headers['Accept']
      accept && accept.include?("application/vnd.apocalypse.#{@version}+json")
    end
end
```

this method is called for each request

matches specific version number

APPLYING ROUTE CONSTRAINT

TEST PASSING

config/routes.rb

```
require 'api_version'

CodeSchoolZombies::Application.routes.draw do

  scope defaults: { format: 'json' } do
    scope module: :v1, constraints: ApiVersion.new('v1') do
      resources :zombies
    end

    scope module: :v2, constraints: ApiVersion.new('v2', true) do
      resources :zombies
    end
  end
end
```

enforces response format

true indicates **default** version

default version **must** go last

TESTING ROUTES FOR THE DEFAULT API VERSION

TEST PASSING

test/integration/routes_test.rb

```
class RoutesTest < ActionDispatch::IntegrationTest
  test 'defaults to v2' do
    assert_generates '/zombies', { controller: 'v2/zombies', action: 'index' }
  end
end
```

no version on URI

defaults to API version 2

For a more robust solution for API versioning see

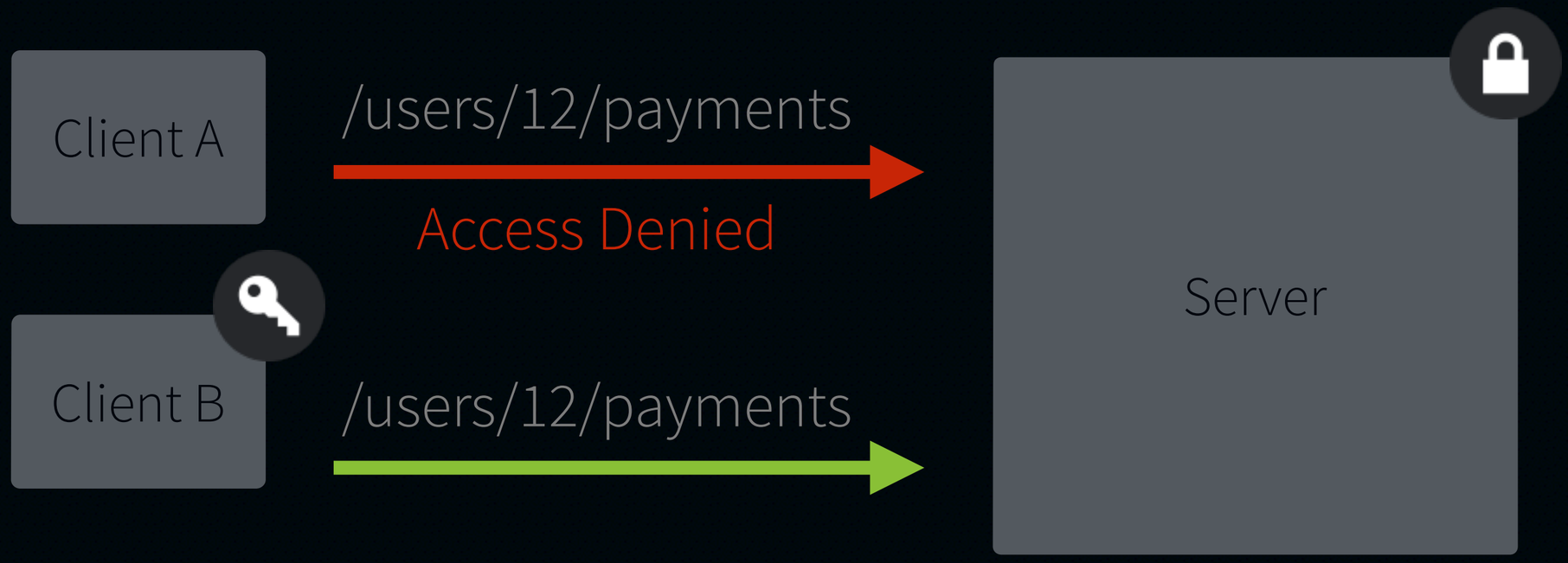
https://github.com/bploetz/versionist/

AUTHENTICATION
PART I - BASIC AUTH

LEVEL 6

API AUTHENTICATION

Authentication is how servers prevent **unauthorized** access to **protected** resources.



AUTHENTICATING USING BASIC AUTH

Credentials must be provided on HTTP requests using the **Authorization** header.

HTTP request header

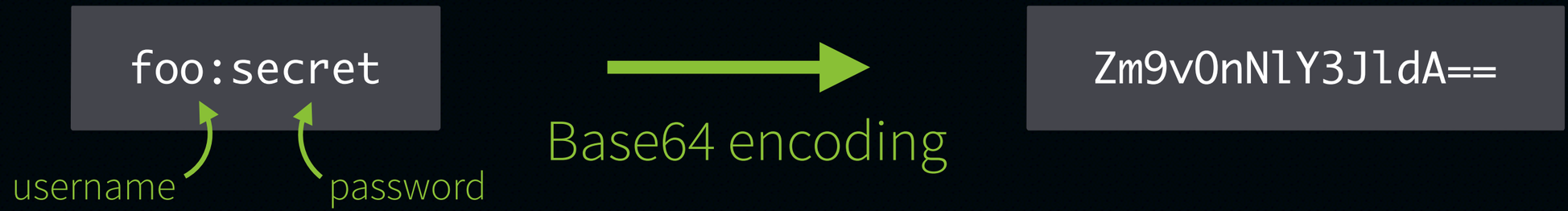
```
GET /episodes HTTP/1.1
Host: localhost:3000
Authorization: Basic dXNlcm5hbWU6cGFzc3dvcmQ=
```

Base64 encoding

Basic Auth is part of the HTTP spec under **RFC 2617.**
For more info, visit https://www.ietf.org/rfc/rfc2617

ENCODING THE CREDENTIALS

Credentials for Basic Auth are expected to be **Base64 encoded.**



Example of base64 encoding text:

ruby console (irb)

```
require 'base64'
 => true
Base64.encode64('foo:secret')
 => "Zm9vOnNlY3JldA==\n"
```

INTEGRATION TESTING WITH BASIC AUTH

TEST FAILING

test/integration/listing_episodes.rb

```ruby
class ListingEpisodesTest < ActionDispatch::IntegrationTest

  setup { @user = User.create!(username: 'foo', password: 'secret') }

  test 'valid username and password' do
    get '/episodes', {}, { 'Authorization' => 'Basic Zm9vOnNlY3JldA==' }
    assert_equal 200, response.status
  end

  test 'missing credentials' do
    get '/episodes', {}, {}
    assert_equal 401, response.status
  end
end
```

Base64 encoded

401 - Unauthorized means request requires user authentication.

IMPLEMENTING BASIC AUTH IN A CONTROLLER

TEST PASSING

Rails has **built-in** support for basic auth.

app/controllers/episodes_controller.rb

```
class EpisodesController < ApplicationController
  before_action :authenticate

  def index
    episodes = Episode.all
    render json: episodes, status: 200
  end

  protected
    def authenticate
      authenticate_or_request_with_http_basic do |username, password|
        User.authenticate(username, password)
      end
    end
end
```

reads and decodes username and password from **Authorization** header

custom authentication strategy

ADDING A TEST HELPER METHOD TO ENCODE CREDENTIALS

TEST PASSING

test/integration/listing_payments.rb

```
class ListingEpisodesTest < ActionDispatch::IntegrationTest
  setup { @user = User.create!(username: 'foo', password: 'secret') }

  test 'valid username and password' do
    get '/episodes', {}, { 'Authorization' => 'Basic Zm9vOnNlY3JldA==' }
    assert_equal 200, response.status
  end

  ...
end
```

ADDING A TEST HELPER METHOD TO ENCODE CREDENTIALS

TEST PASSING

test/integration/listing_payments.rb

```
class ListingEpisodesTest < ActionDispatch::IntegrationTest
  setup { @user = User.create!(username: 'foo', password: 'secret') }

  def encode(username, password)
    ActionController::HttpAuthentication::Basic.encode_credentials(username, password)
  end

  test 'valid username and password' do
    get '/episodes', {}, { 'Authorization' => encode(@user.username, @user.password) }
    assert_equal 200, response.status
  end

  ...
end
```

helper method returns encoded credentials

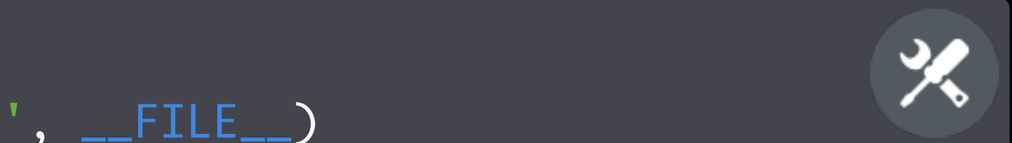
Basic Zm9vOnNlY3JldA==

EXTRACTING HELPER TO BE REUSED ACROSS ALL TESTS

TEST PASSING

test/test_helper.rb

```ruby
ENV["RAILS_ENV"] ||= "test"
require File.expand_path('../../config/environment', __FILE__)
require 'rails/test_help'

class ActiveSupport::TestCase
  ActiveRecord::Migration.check_pending!
  fixtures :all

  def encode_credentials(username, password)
    ActionController::HttpAuthentication::Basic.encode_credentials(username, password)
  end

end
```



can be reused across all tests

ADDING A TEST HELPER METHOD TO ENCODE CREDENTIALS

test/integration/listing_payments.rb

```
class ListingEpisodesTest < ActionDispatch::IntegrationTest
  setup { @user = User.create!(username: 'foo', password: 'secret') }

  test 'valid username and password' do
    get '/episodes', {},
      { 'Authorization' => encode_credentials(@user.username, @user.password) }

    assert_equal 200, response.status
  end

  ...
end
```

UNDERSTANDING UNAUTHORIZED HTTP RESPONSES

Unauthorized HTTP responses must include the **WWW-Authenticate** header.

```
HTTP/1.1 401 Unauthorized
WWW-Authenticate: Basic realm="Application"
Content-Type: text/html; charset=utf-8
```

unauthorized HTTP response

The **WWW-Authenticate** header:

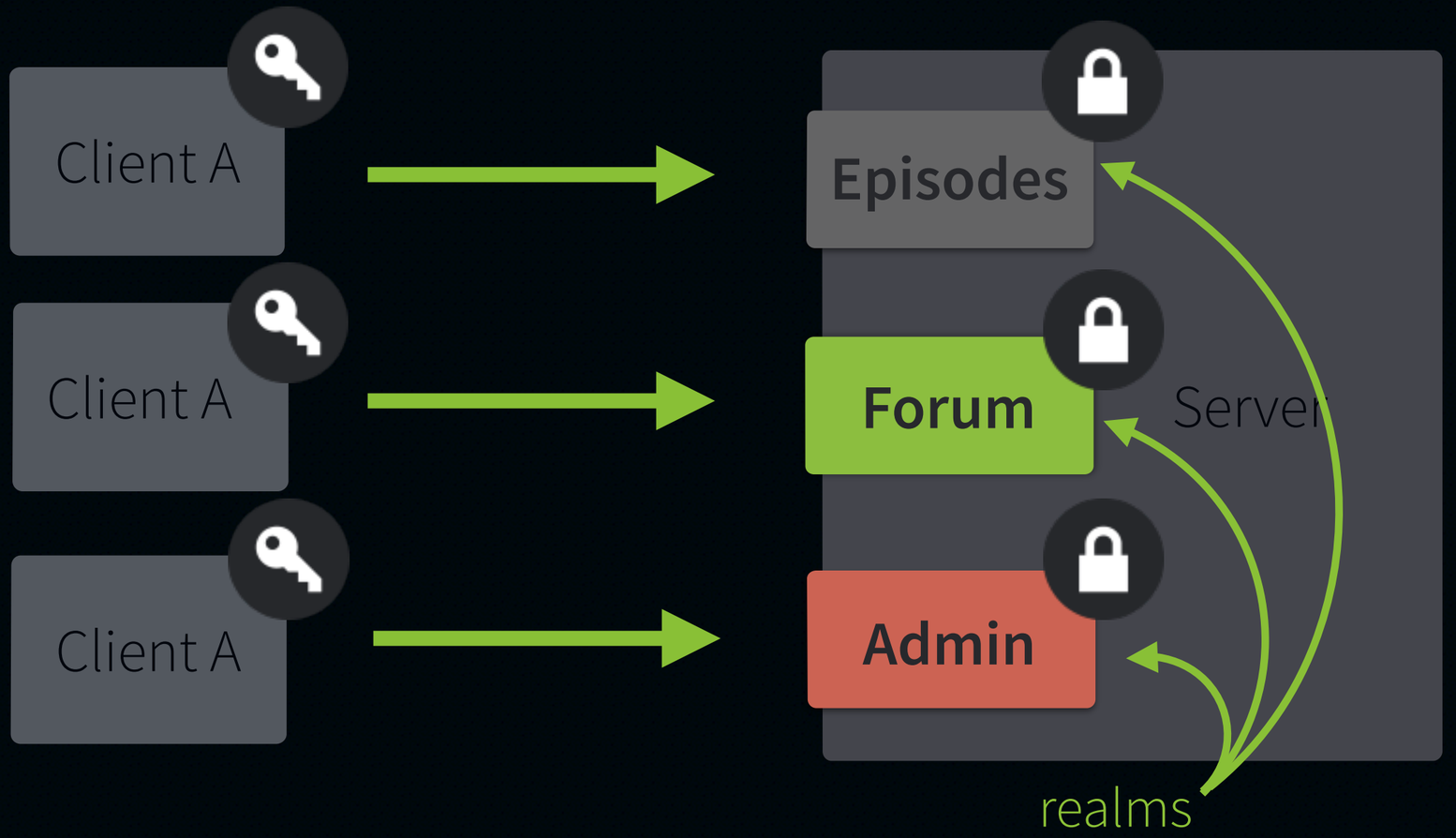
`Basic` the given resource uses HTTP **Basic** Authentication

`realm="Application"` resource is part of the **Application** realm.

USING REALMS FOR DIFFERENT PROTECTION SPACES

Realms allow resources to be partitioned into different **protection spaces.**

Each realm can implement its own authentication strategy.



SURVIVING APIs WITH RAILS

RESPONDING TO UNAUTHORIZED REQUESTS WITH CUSTOM REALM

Realms help clients know which username and password to use.

realm name as argument

```
authenticate_or_request_with_http_basic('Episodes') do |username, password|
  User.authenticate(username, password)
end
```

set on the response header
for **Unauthorized** responses

```
HTTP/1.1 401 Unauthorized
WWW-Authenticate: Basic realm="Episodes"
Content-Type: text/html; charset=utf-8
```

USING BASIC AUTH WITH CURL

Curl has built-in support for authenticated HTTP requests.

send Basic Auth credentials with the **-u** option

```
$ curl -Iu 'carlos:secret' http://localhost:3000/episodes

HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8
```

same as

send Basic Auth credentials as part of the URL

```
$ curl -I http://carlos:secret@localhost:3000/episodes

HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8
```

LIMITATIONS WITH AUTHENTICATE_OR_REQUEST_WITH_HTTP_BASIC

Unauthorized requests using **authenticate_or_request_with_http_basic** always respond with **text/html.**

client asks for **JSON**

```
$ curl -IH "Accept: application/json" \
  -u 'carlos:fakesecret' http://localhost:3000/episodes

HTTP/1.1 401 Unauthorized
Content-Type: text/html; charset=utf-8
WWW-Authenticate: Basic realm="Episodes"
```

HTTP Basic Auth

server responds with **HTML**

FIXING LIMITATIONS

Use **authenticate_with_http_basic** for more control over response.

app/controllers/episodes_controller.rb

```
class EpisodesController < ApplicationController
  before_action :authenticate
  ...
  protected
    def authenticate
      authenticate_basic_auth || render_unauthorized
    end

    def authenticate_basic_auth
      authenticate_with_http_basic do |username, password|
        User.authenticate(username, password)
      end
    end
    ...
end
```

returns a boolean and
does **not** halt the request

SETTING THE RESPONSE HEADER AND PROPER FORMAT

Manually setting the **response header** and responding with **proper format.**

app/controllers/episodes_controller.rb

```
class EpisodesController < ApplicationController
  ...
  def render_unauthorized
    self.headers['WWW-Authenticate'] = 'Basic realm="Episodes"'

    respond_to do |format|
      format.json { render json: 'Bad credentials', status: 401 }
      format.xml { render xml: 'Bad credentials', status: 401 }
    end
  end
end
```

sets proper header

responds to different formats

RESPONDING WITH PROPER CONTENT TYPE FOR BASIC AUTH

API server now responds with correct format.

client asks for **JSON**

```
$ curl -IH "Accept: application/json" \
  -u 'carlos:fakesecret' http://localhost:3000/episodes

HTTP/1.1 401 Unauthorized
Content-Type: application/json; charset=utf-8
WWW-Authenticate: Basic realm="Episodes"
```

HTTP Basic Auth

responds with **JSON**

DECODING CREDENTIALS

Base 64 encoded text can easily be decoded.

Basic Auth is **simple**, but **not secure**.

rails console

```
user, pwd = 'carlos', 'secret'
 => ["carlos", "secret"]

encoded = ActionController::HttpAuthentication::Basic.encode_credentials(user, pwd)
 => "Basic Y2FybG9zOnNlY3JldA=="

decoded = Base64.decode64(encoded.split.last).split(/:/)
 => ["carlos", "secret"]
```

network sniffers can decode credentials

AUTHENTICATION
PART II - TOKEN BASED AUTH

LEVEL 6

TOKEN BASED AUTHENTICATION

API clients use a token identifier for making authenticated HTTP requests

Benefits over Basic Auth:

- More convenience, as we can easily expire or regenerate tokens without affecting the user's account password.
- Better security if compromised, since vulnerability is limited to API access and not the user's master account.
- The ability to have multiple tokens for each user, which they can use to grant access to different API clients.
- Greater control for each token, so different access rules can be implemented.

TOKEN BASED AUTHENTICATION

Token must be provided on HTTP requests using the **Authorization** header.

HTTP request using a **token**

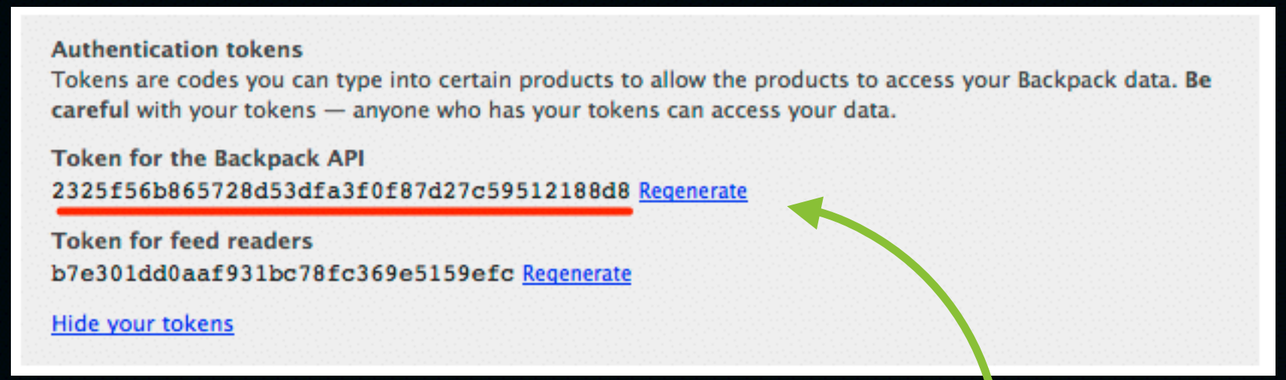
```
GET /episodes HTTP/1.1
Host: localhost:3000
Authorization: Token token=16d7d6089b8fe0c5e19bfe10bb156832
```

access token

There is currently a draft for specifying HTTP Token Access Authentication.
For more info, visit http://tools.ietf.org/html/draft-hammer-http-token-auth-01

GETTING THE API ACCESS TOKEN FOR A WEB BASED SERVICE

API tokens are usually displayed on user profile or settings page.



access token for the **Backpack** server API

using **token** from profile page

```
GET /episodes HTTP/1.1
Host: localhost:3000
Authorization: Token token=2325f56b...
```

SURVIVING APIs WITH RAILS

INTEGRATION TESTING WITH TOKEN BASED AUTHENTICATION

TEST FAILING

test/integration/listing_episodes_test.rb

```
class ListingEpisodesTest < ActionDispatch::IntegrationTest
  setup do
    @user = User.create!
  end

  test 'valid authentication with token' do
    get '/episodes', {}, { 'Authorization' => "Token token=#{@user.auth_token}"}
    assert_equal 200, response.status
    assert_equal Mime::JSON, response.content_type
  end

  test 'invalid authentication' do
    get '/episodes', {}, { 'Authorization' => @auth_header + 'fake' }
    assert_equal 401, response.status
  end
end
```

include user unique token

simulates invalid token

GENERATING USER ACCESS TOKEN

app/models/user.rb

```ruby
class User < ActiveRecord::Base
  before_create :set_auth_token

  private

    def set_auth_token
      return if auth_token.present?
      self.auth_token = generate_auth_token
    end

    def generate_auth_token
      loop do
        token = SecureRandom.hex
        break token unless self.class.exists?(auth_token: token)
      end
    end
end
```

guard clause

keeps looping until unique token is generated

AUTHENTICATING REQUESTS WITH ACCESS TOKEN

TEST PASSING

Rails has **built-in** support for token based authentication.

app/controllers/episodes_controller.rb

```
class EpisodesController < ApplicationController
  before_action :authenticate

  def index
    episodes = Episode.all
    render json: episodes, status: 200
  end

  protected
    def authenticate
      authenticate_or_request_with_http_token do |token, options|
        User.find_by(auth_token: token)
      end
    end
end
```

reads token from **Authorization** header

INTEGRATION TESTING WITH TOKEN BASED AUTHENTICATION

test/integration/listing_episodes_test.rb

```ruby
class ListingEpisodesTest < ActionDispatch::IntegrationTest
  setup do
    @user = User.create!
  end

  test 'valid authentication with token' do
    get '/episodes', {}, { 'Authorization' => "Token token=#{@user.auth_token}"}
    assert_equal 200, response.status
    assert_equal Mime::JSON, response.content_type
  end

  ...
end
```

CREATING HELPER METHOD FOR BUILDING TOKEN ACCESS HEADER

test/integration/listing_episodes_test.rb

```
class ListingEpisodesTest < ActionDispatch::IntegrationTest
  setup do
    @user = User.create!
  end

  def token_header(token)
    ActionController::HttpAuthentication::Token.encode_credentials(token)
  end

  test 'valid authentication with token' do
    get '/episodes', {}, { 'Authorization' => token_header(@user.auth_token) }
    assert_equal 200, response.status
    assert_equal Mime::JSON, response.content_type
  end
  ...
end
```

helper method returns proper header value

USING REALMS FOR TOKEN BASED AUTHENTICATION

Token based access also use realms for different protection spaces.

```
HTTP/1.1 401 Unauthorized
WWW-Authenticate: Token realm="Application"
Content-Type: text/html; charset=utf-8
```

defaults to **Application** realm

pass custom realm name as argument

```
authenticate_or_request_with_http_token('Episodes') do |token, options|
  User.find_by(auth_token: token)
end
```

added to response header
for **Unauthorized** responses

```
HTTP/1.1 401 Unauthorized
Content-Type: text/html; charset=utf-8
WWW-Authenticate: Token realm="Episodes"
```

USING CURL FOR TOKEN BASED AUTHENTICATION

Curl allows custom request headers which we can use to send the access token.

set token on **Authorization** header

```
$ curl -IH "Authorization: Token token=16d7d6089b8fe0c5e19bfe10bb156832" \
  http://localhost:3000/episodes

HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8
```

LIMITATIONS WITH AUTHENTICATE_OR_REQUEST_WITH_HTTP_TOKEN

Unauthorized requests using **authenticate_or_request_with_http_token** always respond with **text/html.**

client asks for **JSON**

```
$ curl -I -H "Authorization: Token token=fake" \
  -H "Accept: application/json" http://localhost:3000/episodes

HTTP/1.1 401 Unauthorized
Content-Type: text/html; charset=utf-8
WWW-Authenticate: Token realm="Application"
```

Token Based Auth

server responds with **HTML**

FIXING LIMITATIONS

First, we use the new **authenticate_with_http_token** method.

app/controllers/episodes_controller.rb

```
class EpisodesController < ApplicationController
  before_action :authenticate
  ...
  protected
    def authenticate
      authenticate_token || render_unauthorized
    end

    def authenticate_token
      authenticate_with_http_token do |token, options|
        User.find_by(auth_token: token)
      end
    end
    ...
end
```

returns a boolean and
does **not** halt the request

FIXING LIMITATIONS WITH AUTHENTICATE_WITH_HTTP_TOKEN

We manually set the header and respond with the proper format.

app/controllers/episodes_controller.rb

```
class EpisodesController < ApplicationController
  ...
  def render_unauthorized
    self.headers['WWW-Authenticate'] = 'Token realm="Episodes"'

    respond_to do |format|
      format.json { render json: 'Bad credentials', status: 401 }
      format.xml { render xml: 'Bad credentials', status: 401 }
    end
  end
end
```

sets proper header

can respond to different formats

RESPONDING WITH PROPER CONTENT TYPE FOR TOKEN BASED ACCESS

API server now responds with correct format.

client asks for **JSON**

```
$ curl -I -H "Authorization: Token token=fake" \
  -H "Accept: application/json" http://localhost:3000/episodes

HTTP/1.1 401 Unauthorized
Content-Type: application/json; charset=utf-8
WWW-Authenticate: Token realm="Episodes"
```

Token Based Auth

responds with **JSON**