Ruby Web Dev The Other Way

Personal best practices guide

by Ievgen Kuzminov

http://rwdtow.stdout.in

Ruby Web Dev: The Other Way

Personal best practices guide

Ievgen Kuzminov

This book is for sale at http://leanpub.com/rwdtow

This version was published on 2016-08-02

Tweet This Book!

Please help Ievgen Kuzminov by spreading the word about this book on Twitter!

The suggested hashtag for this book is #rwdtow.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#rwdtow

Thanks to my Ruby team in MobiDev, you inspire me and suffer from all my new “bright ideas”!

Contents

Intro . . . 1
- Why not “Ruby On Rails” and not “The Right Way”? . . . 1
- Damaged ecosystem . . . 1
- Manifesto . . . 2
- You can help here . . . 2

My Dream Stack . . . 3

Ground knowledge . . . 4
- Web . . . 4
- Linux . . . 4
- IDE . . . 5

Gems . . . 6

Ruby on Rails . . . 7
- Confusing environments . . . 7
- You don’t need ActiveRecord for every kind of model . . . 7
- Before filter/action . . . 8
- HTML helpers. Decorators. . . . 9
- ActiveJob and business logic . . . 11

Alternative frameworks . . . 12
- Full-featured . . . 12
  - Hanami.rb . . . 12
- Specialized . . . 12
  - Grape . . . 12
- Mini . . . 12

Architecture . . . 14
  - Trailblazer . . . 15
  - ROM.rb (Ruby Object Mapper) . . . 17
  - dry-rb . . . 18
  - Rectify . . . 18
  - Learn OOP design . . . 20

Dependency Injection and IoC containers **21**

The magical world of Metaprogramming **23**

Mixin/Module include - it is not composition **24**

Document your code **25**

Debug **26**

SQL **27**

Tests **29**
- Unit tests 29
- Test Behavior, not Configuration 31
- Integration tests 32
- Test data: Fixtures vs Factories 33
- Stub external services call 33
- Speed-up test 34
- Learn Tests Design 34

Authentication & Authorization **35**

Know your App Server! **37**

Know your CLI! **38**

Admin **39**

Templates **40**

Cache **41**
- Static content 41
- Dynamic content 41
- Nginx page microcache 41
- Application level key-value cache 41

Fulltext search **43**

Style guide and style checker **44**

Frontend **45**
- HTML / CSS 45
- JavaScript 45
  - jQuery 45
  - Debug JavaScript 45
  - Server-side rendered JS 45

Modern JavaScript 45

Performance 46

Benchmarks 46

Deployment and Server 47

VPS, Dedicated server 47

PaaS 47

CI 47

DevOps 47

Local development 48

Local tunnels. Exposing dev env in the Internet 48

Protect non-production servers with HTTP Auth 48

Web Application Security 49

Stay open-minded, stay hungry! 50

Community 51

Follow great Ruby developers 52

Books 53

Intro

This guide is born from the question: "could you write a list of all the things that a good Ruby on Rails developer should know?" I decided to expand it to the whole of Ruby web development, and related "full stack" skills, but also limit it to the web, as it is not about Ruby in general.

This guide contains sections dedicated to very important aspects of web development, explanation (if needed), and lists of tutorial links. The format and advice is inspired by "PHP The Right Way"[1].

Sometimes I will suggest tools or gems (with comparison if possible), but these are merely suggestions to get you started. It is up to you, the reader, to decide whether you would like to use them or not.

> Important notice: All suggestions in this guide are my personal opinion. They do not represent absolute truth or generally-accepted best practices. My goal is simply to make suggestions based on my experience.
>
> This guide is not a complete tutorial. Clear steps like installing Ruby (via `rbenv` or `rvm`), managing dependencies via Bundler, etc., are not described due to the wide coverage of these topics in other tutorials. If I do not mention a particular topic, that means I am not aware of any specific issues with it. If it works for you, then it's fine.

Why not "Ruby On Rails" and not "The Right Way"?

I am glad you are asking! :)

It is no secret that most Ruby web developers are introduced to Ruby via Rails. This is a double-edged sword. It lowers the barrier to entry, but it also narrows the range of knowledge. This guide contains a special Ruby on Rails section to cover Rails-specific things, and hopefully it will encourage you to look outside of Rails and the "Rails Way." I can't call the approach described here "The Right Way," as it is just another way to look at Ruby web development.

Damaged ecosystem

The main issue, that forced me to start writing, is unconscious feeling, that something is not perfect with the Rails Way. Attempts to change the behavior or add more structure had no luck. When you introduce a new part into your Rails app - it results in a tension and even active resistance (of the codebase and teammates).

[1]http://www.phptherightway.com/

The whole eco-system is deeply damaged by Rails in terms of code and mindset of developers. And yes there are no real competitors, only because of Rails is so huge in the room of Ruby, that it requires enormous forces to rise a competitor.

It results in no pluralism of ideas, no real code reusability, as most gems are made solely for Rails (and it is the real pain to customize them). Exactly because of this "it is so hard to write Ruby web app without Rails".

This guide is an attempt to describe the way to build the web app with a real understanding of each component, with knowledge that is relevant in any web language and framework. To become a better web developer and demand for a better ecosystem!

Manifesto

- prefer simple solutions
- configuration over confusion
- boilerplate over magic

You can help here

I am open to questions, suggestions, and critique based on the contents of this guide.

You can participate by opening a discussion issue on GitHub[2] or joining the Gitter chat[3].

I am not a native English speaker, so the verbiage here may be far from perfect. Please contribute with your proofreading and typographical fixes via pull requests in GitHub.

Share this guide if you consider it useful. Please use the `#RWDTOW` hashtag on social media.

[2]https://github.com/iJackUA/rwdtow/issues/new
[3]https://gitter.im/iJackUA/rwdtow

My Dream Stack

Before I start with any detailed descriptions, let us have a brief look at the “ideal” Ruby web application stack that I am trying to achieve in my work. Not everything is available yet, but “viam supervadet vadens” â€“ a journey of a thousand miles begins with a single step.

- Linux based OS with RubyMine IDE
- PostgreSQL DB and Redis cache
- Hanami.rb framework, to have a full-featured framework with as little magic as possible
- Trailblazer to organize business logic and reusable widgets, plus handle JSON API conversions
- Dependency injection in Ruby code
- Minitest and Capybara for testing, with clear OOP approaches instead of a DSL
- Vue.js + ES6/WebPack for rich JavaScript pages
- Docker for local development and production deployment

Ground knowledge

Web

It is a shame that many developers do not understand the fundamentals of web development. We are creating web apps without even knowing how the Internet (or the web) works. That is why junior developers do not understand where params come from, and the difference between HTTP request methods. Unfortunately, this lays the groundwork for reliance upon "magic."

- Mozilla: How the Web works[4]
- What really happens when you navigate to a URL[5]
- What is HTTP[6]

If you feel confident in your basic web knowledge and want to dive deeper in all aspects, read the free ebook High-Performance Browser Networking[7].

> This book provides a hands-on overview of what every web developer needs to know about the various types of networks (WiFi, 3G/4G), transport protocols (UDP, TCP, and TLS), application protocols (HTTP/1.1, HTTP/2), and APIs available in the browser (XHR, WebSocket, WebRTC, and more) to deliver the best—fast, reliable, and resilient—user experience.

- High-Performance Browser Networking by Ilya Grigorik[8]

Linux

A lot of us come to web development with Windows desktops. But even if Ruby can be installed on Windows, I recommend using a Linux machine for Ruby development – either a real desktop OS setup, or a virtual machine – because the world of web servers (VPS) is all about Linux. You must have a good understanding of Linux to do basic performance tuning and advanced web server setup for your applications in production. This requires being familiar with at least one Linux distribution – Ubuntu, for example, or one of the other top 10 distros.

[4]https://developer.mozilla.org/en-US/Learn/Getting_started_with_the_web/How_the_Web_works
[5]http://igoro.com/archive/what-really-happens-when-you-navigate-to-a-url/
[6]http://www.jmarshall.com/easy/http/
[7]https://hpbn.co/
[8]https://hpbn.co/

- Ubuntu[9]
- DistroWatch TOP[10]

IDE

You can write Ruby code in any text editor, but using a more sophisticated IDE increases productivity.

Editors like SublimeText and Atom require some additional plugin setup

The most full-featured IDE is RubyMine, but it is not free. Due to the dynamic nature of Ruby, it is hard for IDEs to do correct autocompletion most of the time. This is why RubyMine is not as powerful as similar IDEs for other languages, but it still provides quite a lot of additional integrated tools.

- RubyMine[11]
- SublimeText[12]
- Atom[13]
- NetBeans[14]

[9]http://www.ubuntu.com/
[10]http://distrowatch.com/dwres.php?resource=popularity
[11]https://www.jetbrains.com/ruby/index.html
[12]https://www.sublimetext.com/
[13]https://atom.io/
[14]https://netbeans.org/features/ruby/index.html

Gems

Before adding any utility gem to the project, try to search for alternatives, and be sure that you chose a minimal solution â€“ one that still has developer support and a community contributing to it. It is also useful to browse through curated "awesome" lists.

- The Ruby Toolbox[15]
- Curated Awesome Ruby List[16]

Be aware! Don't be gem obsessed! Most of the time, you can write the code on your own, without any gems. Your own code is much easier to customize, polish, and build into your architecture. It does only what it is supposed to do, with no ballast.

If you do decide to use a gem, look at the code inside it and try to understand what it does, and how it is implemented. A lot of gem links provided in this guide show you the most popular solution (which is usually not the simplest solution) and examples of simpler, better solutions for the topic. Investigating other people's open source code gives you a few advantages:

- you can learn how to do something from other good coders
- you don't need documentation for the gem (if the code is clear), as you can see the public interface and usage logic
- or you discover that the gem is so bad and unoptimized that you definitely need to throw it away, and write your own implementation (and you already know at least one way NOT to do it)

[15]http://www.ruby-toolbox.com/
[16]http://awesome-ruby.com/

Ruby on Rails

> Most Rails developers do not write object-oriented Ruby code. They write MVC-oriented Ruby code by putting models and controllers in the expected locations. Most will add utility modules with class-methods in lib/, but that's it. It takes 2-3 years before developers realize: "Rails is just Ruby. I can create simple objects and compose them in ways that Rails does not explicitly endorse!" – Mike Perham[17]

RoR is a good starting point for a young web developer, but it is important that our education does not end there. Learn to separate what is pure Ruby and what is Rails. Correctly recognize architectural and design issues that appear in your app, and that most of these issues are caused by misusing oversimplified approaches from Rails.

TODO: Add more points to be aware of in Rails

Confusing environments

Rails comes with the following ways to alter config according to the environment:

- Environment initializers (in the `config/environments` folder)
- YAML inheritance
- `ENV` variables
- Or you can replace config files on deploy

Often, all of these approaches are mixed without any separation or consideration. They can usually all be replaced by `ENV` vars that change the value of primitive configs and env initializers, to make bigger structural changes in initialization flow.

You don't need ActiveRecord for every kind of model

What Rails teaches you first is to model data with AR classes, and that works great while you are following "The Rails Way." Unfortunately, that teaches us to think that model structure is inseparable from the persistence layer (literally from the DB table structure).

That causes issues:

[17] https://www.mikeperham.com/2012/05/05/five-common-rails-mistakes/

- All data that is not saved to the DB flows into the app as primitive data structures (e.g. Array of Hashes).
- There is a tendency to structure data in a way that is easy to map into a relational DB, rather than a way that clearly reflects the business logic.
- Anything that is not an AR class is put into the `lib` folder.

It is an open secret: even a PORO can be a model, if it encapsulates the data and actions of an entity. Models are not required to have `find` and `save` actions (those are merely the Active Record pattern. Recognize it!). You can, and ideally should, separate models as data containers from other classes that handle persistence. It does not matter whether we get/save it via a DB, or get/send it via a REST API.

Before filter/action

The main purpose of the `before_action` hook is to authorize the call to the action – literally, to decide to allow this user to run the action or not. But often we can find "pre-population" code like this:

```ruby
before_filter :find_item, only: [:update, :show, :edit]
before_action :authenticate_user!

def show
end

private
def find_item
  @item = Item.find(params[:id])
end
```

Firstly, there is no difference between `before_filter` and `before_action`. They are aliases, and `before_filter` is deprecated in Rails 5.0.

Secondly, do you think this is DRY? No. It is misleading when you forget what `@variables` are available in what actions/views. Be explicit in such cases:

```ruby
  before_action :authenticate_user!

  def show
    @item = find_item(params[:id])
  end

  private
  def find_item(id)
    Item.find(id)
  end
```

This does not violate the DRY principle, even if you repeat the `find_item` call 10 times in different actions.

HTML helpers. Decorators.

This is the first place where young RoR developers put their view helper code. It is fine to do things like this, in the very beginning.

```ruby
module ApplicationHelper
  def page_title(title)
    title += " | " if title.present?
    title += "My Site"
  end
end
```

Unfortunately, helpers become overcomplicated very quickly with:

- a lot of semi-domain logic
- mixed responsibilities
- being included into controllers (because you also want to do some semi-domain logic in controller)
- a lot of HTML concatenation

I almost consider global view helpers as an antipattern which leads to messy code. The solution is to apply the Decorator pattern approach, with view helpers encapsulated in specialized Decorator classes, and helpers that generate a lot of HTML using partials with ERB instead of HTML string concatenation.

A Decorator could be as simple as this:

```ruby
class Coffee
  def cost
    2
  end

  def origin
    "Colombia"
  end
end

require 'delegate'

class Decorator < SimpleDelegator
  def class
    __getobj__.class
  end
end

class Sugar < Decorator
  def cost
    super + 0.2
  end
end

class Milk < Decorator
  def cost
    super + 0.4
  end
end

coffee = Coffee.new
Sugar.new(Milk.new(coffee)).cost   # 2.6
Sugar.new(Sugar.new(coffee)).cost  # 2.4
Milk.new(coffee).origin            # Colombia
Sugar.new(Milk.new(coffee)).class  # Coffee
```

Or it could be something more sophisticated, like the functionality provided in the Drapper or Disposable::Twin gems.

- Evaluating Alternative Decorator Implementations In Ruby[18]

[18]https://robots.thoughtbot.com/evaluating-alternative-decorator-implementations-in

- Gem: Draper[19]
- Gem: Disposable[20]

ActiveJob and business logic

ActiveJob is a very convenient tool, but it is very easy to write unmaintainable code with it. When you put your actual Job logic inside `process` method it is very hard to test this logic and impossible to reuse without the ActiveJob context.

The main idea that is hidden behind ActiveJob - it is an application-framework boundary, where the framework got data from to "external world" and transfer these data to you application code. It should follow the same rules as the controller does. If you follow the idea of "skinny controller", you should also apply the "skinny job" principle. It should not contain SQL queries or business logic manipulations inside but just call to model or operation object.

Look at the following similarities of `ActiveJob::process` and `Controller::[action_name]` classes:

- it is the first entry point where your code naturally "gets the wheel" and where you can do all the manipulations
- this method receives parameters from the "outside worlds"
- Browser and Job queue - are representatives of the "outside world"
- controller and job classes do parsing and unification of params for your code

Your job should like as simple as that

```ruby
class BigComplexJob < ApplicationJob
  queue_as :default

  def perform(*params)
    MyComplexLogicClass.do_a_lot_of_work(params)
  end
end
```

And you cover `MyComplexLogicClass` with simple Unit tests with the whole set of `params` conditions.

Another aspect is a job queue backend. Delayed Job seems to be a good starting point as it does not require additional infrastructure, just a database. But it quickly became a bottleneck if you have a lot of quick jobs. Redis (im-memory storage) based solutions should be used as more reliable and performant alternatives: Resque[21] or Sidekiq[22].

[19]https://github.com/drapergem/draper
[20]https://github.com/apotonick/disposable
[21]https://github.com/resque/resque
[22]http://sidekiq.org/

Alternative frameworks

The Ruby ecosystem does not end at Ruby on Rails. There are a number of alternative frameworks which excel in different areas or provide a more lightweight way to build web apps. Here are some of the most notable:

Full-featured

Hanami.rb

Hanami proposes a cleaner approach with less metaprogramming than Rails. Some interesting architectural decisions are: multi-app architecture with shared parts (that delays and simplifies the decision to break your app into smaller parts), data mapper/entity-repository approach for the persistence layer, separated actions, explicit exposition of variables for views, and more.

- What is Hanami? Where is it going?[23]

Specialized

Grape

Grape is an opinionated framework for creating REST-like APIs in Ruby. It has built-in support for common conventions, including: multiple formats, subdomain/prefix restriction, content negotiation, versioning, and much more. All these elements are described via a simple DSL.

Mini

The Mini framework focuses on providing a web request routing layer and leaves everything else to us. All Ruby web frameworks support, or are based on, Rack - a minimal interface for Ruby webservers, which standardizes how web requests are handled. Rack allows us to create a chain of “middlewares” (small specific handlers) that give us many basic things such as variable parsing, cookie and session management, etc â€“ things that we tend to consider as a “given.”

- dry-web[24]

[23]https://discuss.hanamirb.org/t/what-is-hanami-where-is-it-going/222
[24]https://github.com/dry-rb/dry-web

- Padrino[25]
- Rack[26]
- Roda[27]
- Sinatra[28]

[25]http://padrinorb.com/
[26]http://rack.github.io/
[27]http://roda.jeremyevans.net
[28]http://www.sinatrarb.com/

Architecture

MVC is not an app architecture. This may not be obvious due to the reign of MVC-based frameworks, which teach us to use routing + model + view rendering as the way to build web apps. Unfortunately, business logic has no place in this list.

The "fat model, skinny controller" approach does not solve the issue either. It just sweeps the dust under the carpet, and you will still suffer from fat models used in numerous contexts. Changing code for one usage context will break another usage context.

As a general rule in OOP code, you should break code into smaller classes with smaller responsibilities. Ideally, this code should follow SOLID principles (**S**ingle Responsibility, **O**pen-Closed, **L**iskov Substitution, **I**nterface Segregation, and **D**ependency Inversion).

In the book "Practical Object-Oriented Design in Ruby: An Agile Primer," Sandi Metz proposes that just being `SOLID` is not enough – code should also be `TRUE`.

> If you define easy to change as
>
> - Changes have no unexpected side effects
> - Small changes in requirements require correspondingly small changes in code
> - Existing code is easy to reuse
> - The easiest way to make a change is to add code that in itself is easy to change.
>
> Then the code you write should have the following qualities. Code should be
>
> - **T**ransparent - The consequences of change should be obvious in the code that is changing, and in the distant code that relies upon it.
> - **R**easonable - The cost of any change should be proportional to the benefits the change achieves.
> - **U**sable - Existing code should be usable in new and unexpected contexts.
> - **E**xemplary - The code itself should encourage those who change it to perpetuate these qualities.

TODO: Describe approaches like Form Model, Service Object, Context

- Gem: Active Interaction[29]
- SOLID Design Principles[30]

[29]https://github.com/orgsync/active_interaction
[30]https://www.practicingruby.com/articles/solid-design-principles

Trailblazer

Trailblazer is a powerful architectural framework. It can be used with any Ruby web framework, and has special adapters for Rails. It provides the missing pieces to organize business logic.

- Operations are composable entities that encapsulate an action with a context, validations, and permission checks. Almost everything you would normally write in the controller should be placed here.
- Active Record models are only used for simple finding, saving, and managing relations. They are limited to a single responsibility: data persistence operations.
- Forms are provided per operation, unbound from the single context of a Fat Model.
- Cells are small, encapsulated pieces of reusable view logic. They replace messy app helpers.
- Representers describe presentation rules for serializing and deserializing documents. These are used in a variety of places, from the internal parameter representation of Operations to the representation of data in a JSON API.

Operation example

```ruby
# CRUD action Operation
class Comment::Create < Trailblazer::Operation
  include Model
  model Comment, :create

  contract do
    property :body, validates: {presence: true}
  end

  def process(params)
    validate(params[:comment]) do
      contract.save
    end
  end
end

# Run Operation
op = Comment::Create.(comment: {body: "MVC is so 90s."})
# Get a Model from it
model = op.model
```

Cell example

```
#Cell class
class Comment::Cell < Cell::ViewModel
    property :body
    property :author

    def show
      render
    end

  private
    def author_link
      link_to author.email, author_path(author)
    end
  end

# Template
- # app/concepts/comment/views/show.haml
  %li
    = body
    By #{author_link}

# Testing
  describe Comment::Cell do
    it do
      html = concept("comment/cell", Comment.find(1)).()
      expect(html).to have_css("h1")
    end
  end
```

Representable example

```
# Class
class SongRepresenter < Representable::Decorator
  include Representable::JSON

  property :id
  property :title

  property :artist, decorator: ArtistRepresenter
end

# Serialize
```

```ruby
SongRepresenter.new(song).to_json
#=> {"id": 1, title":"Fallout", artist:{"id":2, "name":"The Police"}}

# Restore object
song = Song.new # nothing set.

SongRepresenter.new(song).
    from_json('{"id":1,title":"Fallout",artist:{"id":2,"name":"The Police"}}')

song.artist.name #=> "The Police"
```

Trailblazer (#Trbr) concepts are somewhat difficult to understand and use properly at first, but they definitely make more and more sense as you become familiar with them. It is very hard to describe the whole Trailblazer philosophy in a short text. Nick Sutterer, the author of Trbr, has quite a lot of documentation with detailed descriptions, and has written a book that covers building a Rails app with Trbr, step-by-step.

It is definitely worth a try if you want to start making your Ruby apps better.

- Trailblazer[31]
- Trailblazer Book[32]

ROM.rb (Ruby Object Mapper)

Ruby Object Mapper (ROM) is a Ruby persistence library with the goal of providing powerful object mapping capabilities without limiting the full power of your datastore.

- Isolate the application from persistence details
- Provide minimum infrastructure for mapping and persistence
- Provide shared abstractions for lower-level components
- Provide simple use of the underlying datastore when desired

All ROM components are stand-alone – they are loosely coupled, can be used independently, and follow the single responsibility principle. A single object that handles coercion, state, persistence, validation, and the all-important business logic, rapidly becomes complex. Instead, ROM provides the infrastructure that allows you to easily create small, dedicated classes for handling each concern individually, and then tie them together in a developer-friendly way.

[31]http://trailblazer.to
[32]https://leanpub.com/trailblazer

```
TODO: ROM example
```

- rom-rb[33]

dry-rb

Dry-rb Is a collection of next-generation Ruby libraries, each intended to encapsulate a common task while remaining decoupled and reusable.

TODO: Add extended dry-rb gems description

- dry-rb[34]
- list of all dry gems[35]

Rectify

The Rectify gem provides some lightweight classes that make it easier to build Rails applications in a more maintainable way. It is built on top of several other gems and adds improved APIs to make things easier.

Currently, Rectify consists of the following concepts:

- Form Objects
- Commands
- Presenters
- Query Objects

You can use these separately or together, to improve the structure of your Rails applications.

The main problem that Rectify tries to solve is where your logic should go. Commonly, business logic is either placed in the controller or the model, and the views are filled with too much logic as well. The opinion of Rectify is that these places are incorrect and that your models, in particular, are doing too much.

Rectify's opinion is that controllers should just be concerned with HTTP related things, and models should just be concerned with data relationships. The problem then becomes how and where you implement validations, queries, and other business logic.

Using Rectify, Form Objects contain validations and represent the input data of your system. Commands then take a Form Object (as well as other data) and perform a single action, which is invoked by a controller. Query objects encapsulate a single database query, and any logic it needs.

[33]http://rom-rb.org/
[34]http://dry-rb.org/
[35]http://dry-rb.org/gems/

Presenters contain the presentation logic in a way that is easily testable, and keeps your views as clean as possible.

Rectify is designed to be very lightweight and allows you to use some or all of its components. We also advise that you use these components where they make sense, not just blindly everywhere. More on that later.

Here is an example controller that shows details about a user, and also allows a user to register an account. This creates a user, sends some emails, does some special auditing, and integrates with a third party system:

```ruby
class UserController < ApplicationController
  include Rectify::ControllerHelpers

  def show
    present UserDetailsPresenter.new(:user => current_user)
  end

  def new
    @form = RegistrationForm.new
  end

  def create
    @form = RegistrationForm.from_params(params)

    RegisterAccount.call(@form) do
      on(:ok)      { redirect_to dashboard_path }
      on(:invalid) { render :new }
      on(:already_registered) { redirect_to login_path }
    end
  end
end
```

The RegistrationForm Form Object encapsulates the relevant data that is required for the action, and the RegisterAccount Command encapsulates the business logic of registering a new account. The controller is clean, and business logic now has a natural home:

```
HTTP             => Controller   (redirecting, rendering, etc)
Data Input       => Form Object  (validation, acceptable input)
Business Logic   => Command      (logic for a specific use case)
Data Persistence => Model        (relationships between models)
Data Access      => Query Object (database queries)
View Logic       => Presenter    (formatting data)
```

Learn OOP design

By learning to design small pieces – objects, in OOP – and put them together, you automatically learn how to make good app architecture overall. There is no a magical library that will suddenly make your code better. Good code comes from the combination of many tiny aspects.

- Video: SOLID Object-Oriented Design by Sandi Metz[36]
- Video: All the Little Things by Sandi Metz[37]
- Video: Nothing is Something by Sandi Metz[38]
- Video: Therapeutic Refactoring by Katrina Owen[39]
- Video: Overkill by Katrina Owen[40]
- Book: Objects on Rails by Avdi Grimm[41]
- Book: Practical Object-Oriented Design in Ruby (POODR) by Sandi Metz[42]

[36]https://www.youtube.com/watch?v=v-2yFMzxqwU
[37]https://www.youtube.com/watch?v=8bZh5LMaSmE
[38]https://www.youtube.com/watch?v=9lv2lBq6x4A
[39]https://www.youtube.com/watch?v=J4dlF0kcThQ
[40]https://www.youtube.com/watch?v=GWEEPt8VvmU
[41]http://objectsonrails.com/
[42]http://www.poodr.com/

Dependency Injection and IoC containers

This technique is widely used in many programming languages, from Java to PHP and JavaScript. In brief, it removes direct dependencies from your classes, replacing them with dependencies upon abstractions instead. Concrete instances of the abstractions are “injected” in the initializer as parameters. This is Inversion of Control â€“ objects do not instantiate their own dependencies, the dependencies are provided from the outside. Alas, it can be very messy handling all the injected dependencies manually, especially when you have a cascade of dependencies. An IoC Container is that mysterious “someone” who instantiates all objects in the app. It uses class annotations to instantiate the correct dependencies to inject. The concrete classes for each abstraction are often specified in the IoC config. Interfaces are natural abstraction identifiers, in languages that support them, but Ruby does not have interfaces. That is why dependency-injected classes require a synthetic dependency specification in Ruby.

Some time ago, DHH wrote a critique of DI usage in Ruby[43], but Piotr Solnica demonstrated reasonable examples[44] of good DI usage. Also, Sandi Metz provides good arguments and examples in her POODR book[45].

With the `dry-container` and `dry-auto_inject` libs, DI/IoC in Ruby could look something like this:

```ruby
my_container = Dry::Container.new

my_container.register(:data_store, -> { DataStore.new })
my_container.register(:user_repository, -> { container[:data_store][:users] })
my_container.register(:persist_user, -> { PersistUser.new })

# set up your auto-injection function
AutoInject = Dry::AutoInject(my_container)

# then simply include it in your class, specifying the dependencies that
# should be injected automatically from the configured container
class PersistUser
  include AutoInject[:user_repository]

  def call(user)
```

[43] http://david.heinemeierhansson.com/2012/dependency-injection-is-not-a-virtue.html
[44] http://solnic.eu/2013/12/17/the-world-needs-another-post-about-dependency-injection-in-ruby.html
[45] http://poodr.com

```
    user_repository << user
  end
end
```

- David Heinemeier Hansson: Dependency injection is not a virtue[46]
- Piotr Solnica: The World Needs Another Post About Dependency Injection in Ruby[47]
- Martin Fowler: Inversion of Control Containers and the Dependency Injection pattern[48]
- dry-container[49]
- dry-auto_inject[50]
- Effective Ruby dependency injection at scale[51]

[46]http://david.heinemeierhansson.com/2012/dependency-injection-is-not-a-virtue.html
[47]http://solnic.eu/2013/12/17/the-world-needs-another-post-about-dependency-injection-in-ruby.html
[48]http://www.martinfowler.com/articles/injection.html
[49]https://github.com/dry-rb/dry-container
[50]https://github.com/dry-rb/dry-auto_inject
[51]http://icelab.com.au/articles/effective-ruby-dependency-injection-at-scale/

The magical world of Metaprogramming

Metaprogramming is a special feature of some languages (including Ruby) used to define code dynamically, at runtime. It is code that generates other code. This is responsible for a lot of the "magic" in Rails, for example: `some_route_path` helpers, and `find_by_%attr_name%` in ActiveRecord.

It seems like an awesome feature at first, until it is misused, and unfortunately it is misused most of the time. The downsides of metaprogramming are:

- Difficult to locate method source code.
- Hidden intention in the codebase.
- IDEs can't locate these methods for auto-complete.

There is a famous saying: "if you have a problem and think that metaprogramming could help you, then congratulations! Now you have two problems." Many times, the coding challenges that you solve with metaprogramming could be solved in a simpler way, with better code quality, separation of concerns, and clearness.

Here is an example of unnecessary metaprogramming in the `rest-client`[52] gem:

```
POSSIBLE_VERBS = ['get', 'put', 'post', 'delete']

POSSIBLE_VERBS.each do |m|
  define_method(m.to_sym) do |path, *args, &b|
    r[path].public_send(m.to_sym, *args, &b)
  end
end
```

- Bala Paranj: 7 Deadly Sins of Ruby Metaprogramming[53]

[52]https://github.com/rest-client/rest-client/blob/master/bin/restclient
[53]https://www.codeschool.com/blog/2015/04/24/7-deadly-sins-of-ruby-metaprogramming/

Mixin/Module include - it is not composition

“Divide et impera” (Divide and rule) is a well known saying from ancient Rome. In programming, we also try to break our code into smaller parts, compose them, and reuse them. In Ruby, we can use `include` and `extend` to add methods from other modules. But just because your code is DRY does not mean it exhibits *composition*.

Yes, now your class/object can respond to a new set of messages, but there is a dark side: mixins create implicit (hidden) pollution of an object’s public interface. Do these methods really belong to this class/object? You should ask yourself this question every time you use a mixin (e.g. including a Concern in Rails).

Mixins make sense when they use the internal state of an Object (`@attributes` or methods of the host object). It also makes some sense to use them for static helpers that do not use the host object itself, only input parameters.

Real composition should be based on Dependency Injection and Separation of Concerns. Your methods should be grouped by an explicit receiver (e.g not `obj.a_m1`, `obj.a_m2`, `obj.b_m1`, but `obj.a.m1`, `obj.a.m2`, `obj.b.m1`). Composition in this case literally means opting not to mixin `A` and `B`, but to inject instances of `A` and `B`, and assign them to the `a` and `b` properties inside `obj`. In that way, your dependencies are not hard-coded, and you can substitute them dynamically. The main advantage of this approach is that `A` and `B` are self-contained, and can be reused.

Just think about the following use-cases, in the context of a Rails app. Do they really make sense?

- Path helpers mixed into controller/view
- Application helpers mixed into controller/view

Another issue is that mixin modules make too many assumptions about the host class interface. They expect the host class to have certain methods, or even worse, expect that the host class already includes another mixin. These expectations are completely hidden – for example, try to add the `errors` capability from ActiveModel into a PORO.

Document your code

Code documentation is more than just writing human-readable code comments. It is also about creating functional comments in specific formats (like RDoc or YARDOC) that are clear to other developers, and that can be interpreted by your IDE.

- Rails API doc Guideline[54]
- RDoc[55]
- YARDOC[56]

Remember that this documentation can help your IDE to make correct autocompletions of returned values, and object attributes/methods.

- RubyMine: Creating Documentation Comments[57]
- RubyMine: Using Annotations[58]

[54]http://edgeguides.rubyonrails.org/api_documentation_guidelines.html
[55]http://docs.seattlerb.org/rdoc/
[56]http://yardoc.org/
[57]https://www.jetbrains.com/help/ruby/8.0/creating-documentation-comments.html#create_tag
[58]https://www.jetbrains.com/help/ruby/8.0/using-annotations.html?origin=old_help

Debug

Many people are creating apps without using even standard debugging techniques. We tend to stay as "puts debuggers" because it is easy, stable and reliable. But we lose a lot of troubleshooting productivity without good debugging skills. The first step is to discover the Pry (`binding.pry`) and Byebug (`byebug`) debuggers. They allow you to set a breakpoint, stop app execution there, and print out current variables on the stack. But debugging is not limited to breakpoints. The next step is learning how to walk step-by-step inside the app â€“ stepping into functions, etc.

Great results can be achieved by combining both of these solutions with pry-byebug[59], which adds the navigation commands from Byebug to the Pry command line.

The best debugging experience would be provided via a visual debugger, like in the IntelliJ RubyMine IDE. It requires an additional Gem, and the app must be run using a special launcher, but as a result you can set breakpoints and step through code right in your IDE. You can also see all the variables on the stack at each step. Such a high density of visual debug info can dramatically increase productivity, and decrease the time needed to track down an error.

- Tenderlovemaking: I am a `puts` debuggerer[60]
- Debugging Rails Appliction[61]
- Byebug[62]
- Pry[63]
- Ruby default debugger[64]
- IntelliJ RubyMine Visual Debugger[65]

[59]https://github.com/deivid-rodriguez/pry-byebug
[60]http://tenderlovemaking.com/2016/02/05/i-am-a-puts-debuggerer.html
[61]http://guides.rubyonrails.org/debugging_rails_applications.html
[62]https://github.com/deivid-rodriguez/byebug
[63]https://github.com/pry/pry
[64]http://ruby-doc.org/stdlib-2.3.0/libdoc/debug/rdoc/DEBUGGER__.html
[65]https://www.jetbrains.com/help/ruby/2016.1/debugging.html

SQL

One of the biggest advantages of Active Record is that the developer does not need to write raw SQL queries to perform basic data manipulation in the DB. And like everything in nature, if you don't use it, you lose it. In that way, we have a bunch of Rails developers that are scared of anything more complex than `SELECT * FROM users`. That is pretty depressing.

- Learn SQL and the capabilities of specific databases!
- If you use Rails, open the console and try to understand what SQL is being generated for each AR query.
- Do not immediately reach for the "simplest" AR code. That makes a lot of unnecessary queries (N+1 queries, etc.). Think about how the DB will be accessed. Think in SQL!
- If you use Rails, use the power of AR query composition. That's fine. You do not need to write raw SQL all the time. But when you need to compose a complex query, and do not know how to do it with AR, write it in SQL first, and then gradually transform it into AR notation.
- Some queries, especially those using special abilities of the DB (like `PARTITION BY`, etc.), are better to leave in raw SQL. Think about other developers who will need to support this code. Some things are clearer in AR notation, some are clearer in pure SQL. Just consider the options.

Also, consider that different databases are best suited to different use cases. One web application can make use of multiple databases simultaneously. Think about the type of data you want to operate upon. You can even combine relational DBs (PostgreSQL, MySQL) with document oriented DBs (MongoDB). For graph-like data ("friends of my friends" relations) there is another class of DB, like Neo4J. There are even multiparadigm DBs like ArangoDB.

You should also keep an eye on the DB world. It is changing as fast as web languages and frameworks. It is up to the web developer to propose and use the best data storage solution for the situation, based on the type of data and common query patterns.

To keep productivity high while using special features of specific databases, you can use the Sequel[66] lib. It is very similar to ActiveRecord, but not coupled to Rails, and has a wide selection of plugins and extensions.

- Khan Academy. Intro to SQL: Querying and managing data[67]
- PostgreSQL Tutorials[68]

[66]http://sequel.jeremyevans.net/
[67]https://www.khanacademy.org/computing/computer-programming/sql
[68]http://www.postgresqltutorial.com/

- PostgreSQL Exercises[69]
- PostgreSQL[70]
- MongoDB[71]
- Neo4J[72]
- ArangoDB[73]
- Sequel[74]

[69]https://www.pgexercises.com/
[70]http://www.postgresql.org/
[71]https://www.mongodb.org/
[72]http://neo4j.com/
[73]https://www.arangodb.com/
[74]http://sequel.jeremyevans.net/

Tests

The Ruby community is well known for its adoption of good testing practices. For example, the PHP community is not even close the same level of understanding of the importance of testing in every stage of app development. Moreover, the average PHP developer does not write nearly as many tests as the average Ruby developer does. One of the secrets of Ruby's testing success is its great tools. Ruby testing tools provide powerful syntax and features, made possible by metaprogramming and DSLs.

Let's not pat ourselves on the back too quickly, though. As always, great power comes with great responsibility. These tools are a double-edged sword. You need to recognize the good parts and the not-so-good parts, so as not to hurt yourself.

Testing rules:

- Tests should be simple and obvious. They should not require testing, themselves.
- Each test should do only one thing.
- Tests should have clear assertions.
- Tests should be predictable and repeatable, producing the same result on each run.

Unit tests

Unit tests cover code-level logic. It instantiates an object, and makes assertions on the result of a method or function call. There are two popular unit testing solutions in Ruby: Minitest and RSpec.

RSpec is a very popular gem that provides an expressive DSL for all aspects of testing. It allows writing test in a BDD "expects" style. Its assertions tend to be human-readable, and it has lots of special extensions. However, its main advantage is also its main drawback – you need to learn a large DSL and assertion syntax. RSpec extensions make this worse, requiring you to put in quite a lot of effort to learn everything.

```ruby
RSpec.describe "Using an array as a stack" do
  def build_stack
    []
  end

  before(:example) do
    @stack = build_stack
  end

  it 'is initially empty' do
    expect(@stack).to be_empty
  end

  context "after an item has been pushed" do
    before(:example) do
      @stack.push :item
    end

    it 'allows the pushed item to be popped' do
      expect(@stack.pop).to eq(:item)
    end
  end
end
```

Minitest became a part of the Ruby standard library, and that is why it is preferable for testing gems and libs, as it does not require additional dependencies. It provides a very small assertions interface, that is easy to learn and adopt. The main advantage of Minitest is that tests are just POROs. That is why you do not need to learn anything new to structure your tests – it is just pure Ruby. Use the same techniques as in your other code. Code initialization and calls are almost identical to real usage. IMHO Minitest is the better choice for writing simple, clear and predictable tests.

```ruby
class TestMeme < Minitest::Test
  def setup
    @meme = Meme.new
  end

  def test_that_kitty_can_eat
    assert_equal "OHAI!", @meme.i_can_has_cheezburger?
  end

  def test_that_it_will_not_blend
    refute_match /^no/i, @meme.will_it_blend?
  end

  def test_that_will_be_skipped
    skip "test this later"
  end
end
```

Minitest also provides a Spec style, which has a lot of RSpec syntactic sugar, but still has a dramatically simpler code base (when you look inside Minitest gem code).

```ruby
describe Meme do
  before do
    @meme = Meme.new
  end

  describe "when asked about cheeseburgers" do
    it "must respond positively" do
      @meme.i_can_has_cheezburger?.must_equal "OHAI!"
    end
  end

  describe "when asked about blending possibilities" do
    it "won't say no" do
      @meme.will_it_blend?.wont_match /^no/i
    end
  end
end
```

Stay consistent, don't write tests in different styles with Minitest. Choose the one you like the most, and write all tests in it. But remember that using Spec style with Minitest reduces its advantages, like normal Ruby syntax and usage of normal OOP techniques.

- **Minitest**[75]
- RSpec[76]

Test Behavior, not Configuration

The shoulda-matchers[77] gem is very popular for testing aspects of ActiveRecord model configuration, like `has_one`.

[75]http://docs.seattlerb.org/minitest/
[76]http://rspec.info/
[77]https://github.com/thoughtbot/shoulda-matchers

```ruby
class Person < ActiveRecord::Base
  has_one :partner
end

class PersonTest < ActiveSupport::TestCase
  should have_one(:partner)
end
```

But why do we need to test indirect signs of expected behavior instead of testing the behavior directly?

We could better write behavior tests like this:

```ruby
class PersonTest < Minitest::Test
  def test_has_parter
    person = Person.new
    partner = Partner.new
    assert_equal partner, person.partner
  end
end
```

The test above will continue passing, regardless of any changes to the implementation of `person.partner`. This is exactly how your app expects the `Person` class to behave, so why should your tests be different, and rely upon the internal implementation?

- Test Behavior, not Configuration[78]

Integration tests

Ruby allows you to write integration tests. These emulate the real interaction between the web app and the browser. They usually contain a set of commands to click on links, visit certain pages, fill in form fields, and validate the results of these actions.

Integration tests can be written using the Capybara framework. Basically, it acts as a "fake browser," making requests to your Rack app and parsing the responses, but is not capable of running JavaScript. To fully emulate an end-user browser, you need to use the Poltergeist or Capybara Webkit add-ons. They run the same commands inside a headless Webkit browser. Of course, this incurs a speed penalty. Tests with JS run slower than tests without JS. You need to be aware of that, and activate JS per test, only in the tests that require it.

[78]http://naildrivin5.com/blog/2016/05/23/test-behavior-not-configuration.html

Understand that integration tests should not make assertions against the internal state of the web app. They should not make assertions on code variables, etc. They should only assert external output, like the response body and HTTP status codes.

Also, do not overuse integration tests. Keep in mind that in any (ANY!) case, integration tests are an order of magnitude slower than unit tests. When you write bad quality code – e.g. putting a lot of logic into Controllers – the only way to test it is to `visit` the corresponding page. You will need to execute the whole app request cycle to test only tiny things. That should encourage you to remove logic from the controller, put it into separate classes, and just call the new classes from the controller. That way, you can test 90% of the logic via unit tests (which are fast) and make only a few "smoke tests" on the controller action (to test success/failure scenarios, but not every part of the business logic).

- Capybara[79]
- Poltergeist[80] (PhantomJS)
- Capybara-WebKit[81]

Test data: Fixtures vs Factories

- Factory Girl[82]
- fabricationgem[83]
- Faker[84]

Stub external services call

Your tests should not depend on the availability of external services. Ideally, you should be able to run all tests without Internet access, but that doesn't mean that external integrations should not be tested. If your app makes a request to an external service, this request should be stubbed. Instead of making a real call, it should "pretend" and return a ready-made response (there could be a couple of different responses, to test success/failure). Gems like Webmock and VCR can help you to catch real responses and then reuse them for subsequent test runs, or to write your own response content.

Stub request with Webmock:

[79]http://jnicklas.github.io/capybara/
[80]https://github.com/teampoltergeist/poltergeist
[81]https://github.com/thoughtbot/capybara-webkit
[82]https://github.com/thoughtbot/factory_girl
[83]http://www.fabricationgem.org
[84]https://github.com/stympy/faker

```
stub_request(:post, "www.example.com")
    .with(:query => {'user' => 'Ievgen'})
    .to_return(:body => "Nice work!")
```

Catch requests and return a "fake" response with VCR:

```
  VCR.use_cassette("synopsis") do
      response = Net::HTTP.get_response(URI('http://www.iana.org/domains/reserve\
d'))
      assert_match /Example domains/, response.body
    end
```

- Stubbing external services in Rails[85]
- Webmock[86]
- VCR[87]

Speed-up test

To achieve fast tests, stub behavior, heavy computations, and external web calls, in addition to mocking dependencies. You can also run tests in parallel – another reason to keep them completely independent from one another.

- Gem: parallel_tests[88]

Learn Tests Design

To write cost-efficient and effort-efficient tests – tests that should not be rewritten from scratch after any small refactoring – you should design your tests almost as well as you design your other code.

- Only test public interface of classes
- Don't test tiny sensitive private methods (they are very likely to change often)
- Reuse repetitive test via mixins
- Avoid redundantly testing the same functionality multiple times. Code should be organized in layers that are testable separately from each other.

- Video: The Magic Tricks of Testing by Sandi Metz[89]
- Book: Practical Object-Oriented Design in Ruby (POODR) by Sandi Metz[90]

[85]https://semaphoreci.com/community/tutorials/stubbing-external-services-in-rails
[86]https://github.com/bblimke/webmock
[87]https://github.com/vcr/vcr
[88]https://github.com/grosser/parallel_tests
[89]https://www.youtube.com/watch?v=URSWYvyc42M
[90]http://www.poodr.com/

Authentication & Authorization

One of the first serious issues that are usually not covered by a framework is an authentication. In the Rails world, people tend to take the well known Devise lib, integrate it into the app with `devise :database_authenticatable, :registerable`, and forget about auth. That is, until the awkward moment when you need to customize any tiny aspect of it - anything from templates to auth logic (the latter is harder).

The truth is, you can implement auth by yourself very quickly. Just create a users table with login and password fields, encrypt the password with something like the `bcrypt-ruby` gem, store the hashed password in the DB, and write 3-5 controller actions to handle forms. Your users table will not be cluttered with lots of unknown fields, and you will have an idea of what they are for (or you can rely on these fields for other business logic).

A less obtrusive, but still more automated, option is to use Sorcery. Another option is the Tyrant gem â€“ part of the Trailblazer philosophy â€“ that tries to use just a single field of a model.

The fewer hidden pieces of data flow in your app, the easier it is to track down errors and change behavior. In such a sensitive part of the app as authentication, this is very important.

- Devise[91]
- **Sorcery**[92]
- Tyrant[93]
- rodauth[94]

Once users can be identified inside the app, the second requirement is to distinguish their abilities. One approach is to assign a marker for each user, like `role = 'admin'`, and check this model attribute all over the app. This is not flexible, however, and when your permission rules change, it requires changes to many different parts of the codebase. Permission logic can be quite complicated, so it should be encapsulated. A good example of simple authorization logic organization is Pundit.

Unfortunately, I do not know of a good example of RBAC[95] implemented in Ruby. The piece[96] gem comes close, where roles contain sets of permissions, and all of them can be assigned to a user dynamically with business rules (rules should be implemented as code that checks restrictions).

[91]https://github.com/plataformatec/devise
[92]https://github.com/NoamB/sorcery
[93]https://github.com/apotonick/tyrant
[94]http://rodauth.jeremyevans.net
[95]https://en.wikipedia.org/wiki/Role-based_access_control
[96]https://github.com/ThoughtWorksStudios/piece

- **Pundit**[97]
- CanCanCan[98]

[97]https://github.com/elabs/pundit
[98]https://github.com/CanCanCommunity/cancancan

Know your App Server!

Running a Rails app starts a WEBrick server by default. This is good enough for the development environment, but not for production. WEBrick runs code in a single thread, and can not handle requests concurrently.

It is very important to understand the different operating models of threaded and event-loop based servers, to select the most reliable and efficient server for your app.

Another aspect is how static files are handled. Do not waste CPU time by giving this task to a Ruby app. Give this work to reverse proxy servers like Nginx. They can also handle other things, like proxying (e.g. `3000` port to `80` port), multi-domain setup, limiting requests per second to prevent DDoS, "slow client" attack, etc.

TODO: Make a brief description of different models

- Rubyraptor: Description of different operation models in Rack servers[99]
- A comparison of Rack web servers for Ruby web applications[100]
- Puma[101]
- Unicorn[102]
- Passenger[103]
- Nginx[104]

[99] http://www.rubyraptor.org/how-we-made-raptor-up-to-4x-faster-than-unicorn-and-up-to-2x-faster-than-puma-torquebox/
[100] https://www.digitalocean.com/community/tutorials/a-comparison-of-rack-web-servers-for-ruby-web-applications
[101] https://github.com/puma/puma
[102] https://github.com/defunkt/unicorn
[103] https://www.phusionpassenger.com/
[104] http://nginx.org/

Know your CLI!

Everybody knows that you should use Rake tasks to run code from the CLI. But we also need to understand that Rake is more than just a part of Rails. It is an independent tool, and deserves consideration about whether it should be used or not. The main downside of Rake tasks is that they are hard to test â€“ due to their DSL nature, they require special initialization. You should be very careful when putting business logic into Rake tasks. It is just as harmful as putting business logic into controllers in a web framework.

As with any tool, Rake is not limited to `namespace`, `task` and `desc`. You should learn how to use all of the benefits and functionality hidden inside. Try to look into the advanced Rake docs.

Also, we need to understand that there are much cleaner alternatives, like Thor. Thor provides the same simple functionality for writing your own CLI tools, but does so in a much cleaner, OOP way.

- Rake[105]
- Thor[106]

[105]http://docs.seattlerb.org/rake/
[106]https://github.com/erikhuda/thor

Admin

ActiveAdmin is a very popular solution. It provides handy DSL to display CRUD interface for ActiveRecord Models. From the very beginning it helps to spare time, but the at later development stages it will require much more time to make even minor customization.

For projects, where you can't change requirements in favor of the "tool capability", prefer solutions that help to scaffold (generate) admin section files. It could look like a lot of repetitive boilerplate code, but in future, it allows customize each screen separately without a headache.

- **Administrate**[107]
- Active Admin[108]
- Rails Admin[109]

[107]https://github.com/thoughtbot/administrate
[108]http://activeadmin.info/
[109]https://github.com/sferik/rails_admin

Templates

Ruby comes with a built-in templating solution, `ERB`, that embeds Ruby into a text document. It has a number of advantages. It is very close to pure HTML, so it is easy to convert HTML into ERB, and it does not introduce any new entry barriers. It is similar to approaches in other languages (PHP, ASP), so it is transferable, common web development knowledge.

Other popular alternative templating solutions include Haml, it's successor Slim, and other Haml-like languages. Slim template code looks like this:

```
div id="footer"
  = render 'footer'
  | Copyright Â© #{year} #{author}
```

This approach has the advantage of being more clear, due to the removal of "unneeded" elements like closing tags and extra brackets. But the disadvantages are obvious: it has a larger entry barrier, it is harder to migrate HTML markup to it (especially during big redesign sprints), and it is not a transferable skill outside of Ruby.

So stick with ERB unless all of your developers and designers are familiar with Slim, or you are building a home-made, 15-min blog.

- An Introduction to ERB Templating[110]
- Slim[111]

[110]http://www.stuartellis.eu/articles/erb/
[111]http://slim-lang.com/

Cache

Caching is a multilayer technique. Each type of content requires its own caching approach.

Static content

Static files, images, and documents could be cached on the server or CDN level by filename or meta-information. Cache invalidation could be done via conditional requests or filename param modification.

Dynamic content

Dynamic content is much harder to cache as it requires clear invalidation criteria. Even harder is personalized content (that is unique per user) - be careful with it, sometimes caching and invalidating this cache could be even more resource consumption, than having no cache at all!

The most obvious solution for a personalized content is to cache on application level: whole pages, fragments or long queries. It is up to you to distinguish and measure what part of long page generation process is the slowest.

Nginx page microcache

The whole page could be cached on the web server level. Nginx server has a "microcache" feature that could be applied to the whole site or selected locations and cache even dynamic content for a short period of time (about 1 minute). For example, even if you have a highly dynamic page with ranked posts, that changes each 10 seconds (according to votes and number of comments), it seems that you need to generate it on each request to show actual data. But if you have a highload site with 1000 simultaneous visitors (that refresh a page each minute) you can do only 1 page generation instead of 1000+. Of course, it depends on the sensitivity of the data (you won't cache stock data) and you need to adjust appropriate cache lifetime while your users would not notice that data is "dramatically outdated".

Application level key-value cache

A more correct way to do caching, without tricks with total microcaching, is to have a strict cache key and to regenerate content when this key changes. Very often in-memory key-value storages like Redis and Memcached are used to store cached data (due to high speed of data access).

But even a file-cache could be productive if you cache operation that takes tens of seconds (the only thing to remember: file-system operation could be a performance bottleneck as disk read speed is very limited, so it should not be very often).

Another ting to remember - saving Ruby data structures like Hash or Array to the external cache (file or memory based) requires to serialize it into text representation (and vice versa on reading). This operation takes CPU resources and time. Usually, it is a JSON or YAML representation and not all data structures could be represented into the string automatically. Also Ruby provides a special `Marshal` class to serialize Objects.

Let's imagine the same page with the ranged list of posts, you want to cache just a plain data of ranged list and automatically rebuild cache when needed. You can make a cache key that includes data like last post date (when a new post is added, you cache key is changed and the page is regenerated). As you see here, to determine cache key you still need to perform one DB query (to get latest post date). It is a reasonable tradeoff of you spare 10-20 or more queries with this caching.

The main rule here is - **cache key computation should be as simple as possible**. Don't make to many dependencies in the cache key.

- The Benefits of Microcaching with NGINX[112]
- **Redis**[113]
- Memcached[114]
- Why Redis beats Memcached for caching[115]

[112]https://www.nginx.com/blog/benefits-of-microcaching-nginx/
[113]http://redis.io/
[114]https://memcached.org/
[115]http://www.infoworld.com/article/3063161/application-development/why-redis-beats-memcached-for-caching.html

Fulltext search

TODO

- Multi-table Full Text Search in Postgres[116]
- PostgreSQL full text seach with ts_vector[117]
- Textacular gem[118]
- Scenic gem[119]
- Sphinx[120]
- Elastic Search[121]
- Algolia[122]

[116]https://speakerdeck.com/calebthompson/multi-table-full-text-search-in-postgres
[117]http://rachbelaid.com/postgres-full-text-search-is-good-enough/
[118]https://github.com/textacular/textacular
[119]https://github.com/thoughtbot/scenic
[120]http://sphinxsearch.com/
[121]https://www.elastic.co/products/elasticsearch
[122]https://www.algolia.com/

Style guide and style checker

While Ruby does not have a lot of curly brackets `{}`, code style issues are not as painful as languages like PHP. In any case, we can write code with different paddings, spaces, and line breaks. To avoid confusion and repeated whitespace changes in your codebase - you should adopt a style guide. All team members should agree upon it.

Also, there are tools like RuboCop that can make automatic static code analysis and propose fixes or recommendations.

- RuboCop Rails styleguide[123]
- GitHub Ruby styleguide[124]
- Airbnb Ruby styleguide[125]
- RuboCop[126]
- Reek[127]

[123]https://github.com/bbatsov/rails-style-guide
[124]https://github.com/styleguide/ruby
[125]https://github.com/airbnb/ruby
[126]https://github.com/bbatsov/rubocop
[127]https://github.com/troessner/reek

Frontend

HTML / CSS

Bootstrap Foundation SemanticUI

JavaScript

TODO: Whole this section :)

jQuery

Debug JavaScript

- Breakpoints[128]

Server-side rendered JS

"replaceWith" trick

Modern JavaScript

Frameworks, Bower Modules and loaders Try to keep away from Sprockets

[128]https://developers.google.com/web/tools/chrome-devtools/debug/breakpoints/?hl=en

Performance

As a general rule, if your web page generation time in production is close to one second (or more), that is not perfect. You should invest some time in performance optimization. But before you start optimizing, you should locate the bottlenecks. Almost always, the issues are not where you think they are. Measure!

You can use special profilers that give detailed stats per function call, like PerfTools, or use continuous high-level performance monitoring with widget-like add-ons, like Rack mini profiler. Rack mini profiler will display a toolbar at the top of all pages, with a breakdown of the raw page load time.

There are also external services like New Relic that can gather code runtime performance, even on the production server. It can give general insight into the most time-consuming code and SQL queries.

TODO: Add gems that help to track N+1 queries and SQL performance

- Rack mini profiler[129]
- PerfTools.rb[130]
- Peek RBLineProf[131]
- New Relic[132]

Benchmarks

Sometimes, in the development stage, you need to measure the performance of a couple of different implementations. To get measurable timing results, the code being benchmarked should be run repeatedly, for many iterations. There are ready-made tools that can help with such measurements.

- Benchmark ips[133]
- Scientist[134]

[129]https://github.com/MiniProfiler/rack-mini-profiler
[130]https://github.com/tmm1/perftools.rb
[131]https://github.com/peek/peek-rblineprof
[132]http://newrelic.com
[133]https://github.com/evanphx/benchmark-ips
[134]https://github.com/github/scientist

Deployment and Server

TODO

- Capistrano[135]

Alternative solutions like Mina do not have any major advantages at the moment. Or a custom Deploy workflow for big projects with tools like Ansible

VPS, Dedicated server

- DigitalOcean[136] (my ref link ;) you receive $10 credit after registration)
- Amazon Web Services[137]
- Namecheap, domains registrator[138] (my ref link ;))

PaaS

- Heroku[139]

CI

- Jenkins[140]
- Travis CI[141]
- CircleCI[142]

DevOps

TODO

- Ansible
- Chef
- Puppet
- Docker

[135]http://capistranorb.com/
[136]https://m.do.co/c/20534050b97f
[137]
[138]https://www.namecheap.com/?aff=62428
[139]https://www.heroku.com/
[140]https://jenkins.io/
[141]https://travis-ci.org/
[142]https://circleci.com/

Local development

TODO

- Vagrant
- Docker

Local tunnels. Exposing dev env in the Internet

- Ngrok
- Vagrant share

Protect non-production servers with HTTP Auth

Basic Auth could be used as the main form of authentication for very simple apps, or admin backends. It is often used to prevent access to staging servers by strangers and search engines (to prevent indexing of non-production pages).

In Rails, the simplest basic auth could be added like this:

```
class ApplicationController < ActionController::Base
  http_basic_authenticate_with name: "admin", password: "hunter2"
end
```

More advanced ways can be found in the Rails docs[143].

In any other Rack based server, you can use the `Rack::Auth::Basic` middleware:

```
use Rack::Auth::Basic, "Restricted Area" do |username, password|
  [username, password] == ['admin', 'pass']
end
```

[143] http://api.rubyonrails.org/classes/ActionController/HttpAuthentication/Basic.html

Web Application Security

Frameworks are great because, not only do they simplifying our web app code, they silently provide security restrictions and best practices. But this magic security only works when you stick to the defaults. When you start to do “advanced” coding, you need to know where common web app vulnerabilities are hidden, and where you need to take care of them.

Common vulnerabilities include:

- SQL injection: unfiltered parameters concatenated into raw SQL queries. This allows hackers to run malicious SQL queries.
- CSRF: allows hackers to send requests to your app without the user’s knowledge or consent.
- XSS: allows hackers to submit malicious JavaScript to your site (e.g. in the comments) which can steal session data from other users.

There are many more potentially dangerous things. You can find more details on the OWASP website.

- OWASP Vulnerabilities list[144]

[144]https://www.owasp.org/index.php/Guide_Table_of_Contents

Stay open-minded, stay hungry!

Learn other languages to get fresh ideas and orthogonal points of view. Try to learn languages with completely different paradigms, e.g. functional languages. It changes the way you look at object-oriented languages and approaches (and no, I am not saying that you need to abandon OOP).

- Elixir[145]
- Video: Blending Functional and OO Programming in Ruby, by Piotr Solnica[146]

[145]http://elixir-lang.org
[146]https://www.youtube.com/watch?v=rMxurF4oqsc

Community

- Official Ruby site[147]
- Ruby IRC[148]
- Conferences[149]
- RubyFlow[150] - community news

[147]https://www.ruby-lang.org/en/
[148]irc://irc.freenode.net/ruby
[149]https://www.ruby-lang.org/en/community/conferences/
[150]http://www.rubyflow.com/

Follow great Ruby developers

Person	Why?	Blog	Twitter	GitHub
Aaron Patterson	active Ruby ecosystem contributor, great blogger	tenderlovemaking.com[151]	@tenderlove[152]	@tenderlove[153]
Luca Guidi	author of Hanami.rb framework	lucaguidi.com[154]	@jodosha[155]	@jodosha[156]
Nick Sutterer	author of Trailblazer	nicksda.apotomo.de[157]	@apotonick[158]	@apotonick[159]
Piotr Solnica	author of dry-rb	solnic.eu[160]	@*solnic*[161]	@solnic[162]
Sandi Metz	author of "POODR" book	sandimetz.com[163]	@sandimetz[164]	@skmetz[165]
Avdi Grimm	author of "Objects on Rails" book	devblog.avdi.org[166]	@avdi[167]	@avdi[168]
Katrina Owen	creator of http://exercism.io, great speaker	kytrinyx.com[169]	@kytrinyx[170]	@kytrinyx[171]

[151]http://tenderlovemaking.com/
[152]https://twitter.com/tenderlove
[153]https://github.com/tenderlove
[154]https://lucaguidi.com/
[155]https://twitter.com/jodosha
[156]https://github.com/jodosha
[157]http://nicksda.apotomo.de/
[158]https://twitter.com/apotonick
[159]https://github.com/apotonick
[160]http://solnic.eu/
[161]https://twitter.com/_solnic_
[162]https://github.com/solnic
[163]http://www.sandimetz.com/
[164]https://twitter.com/sandimetz
[165]https://github.com/skmetz
[166]http://devblog.avdi.org/
[167]https://twitter.com/avdi
[168]https://github.com/avdi
[169]http://kytrinyx.com/
[170]https://twitter.com/kytrinyx
[171]https://github.com/kytrinyx

Books

TODO: Add more books

- Objects on Rails[172] by Avdi Grimm.

[172]http://objectsonrails.com/