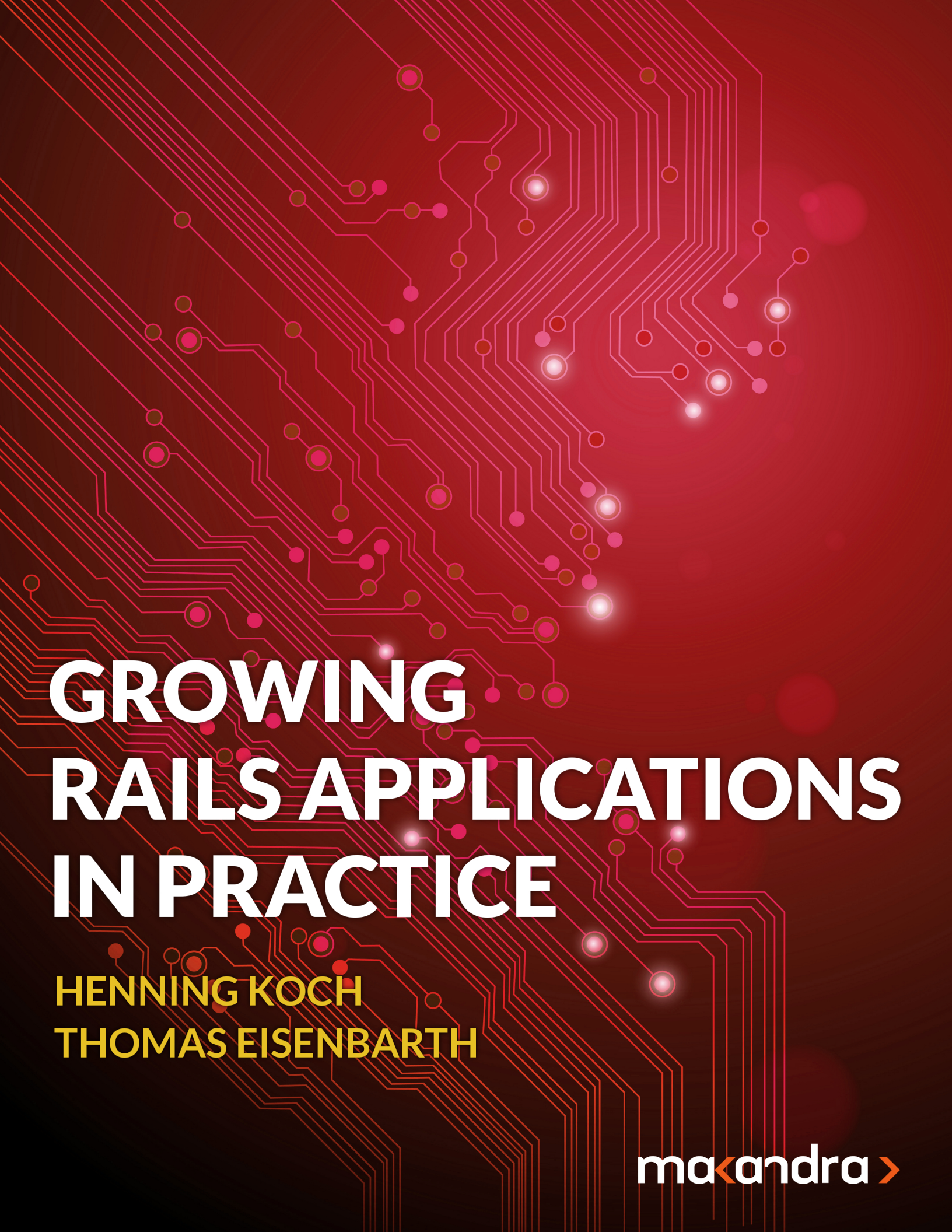

GROWING
RAILS APPLICATIONS
IN PRACTICE
HENNING KOCH
THOMAS EISENBARTH
manning

Growing Rails Applications in Practice

Structure large Ruby on Rails apps with the tools you already know and love

Henning Koch and Thomas Eisenbarth

This book is for sale at http://leanpub.com/growing-rails

This version was published on 2014-07-02

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Contents

1 Introduction

In this book we demonstrate low-ceremony techniques to scale large, monolithic Rails applications. Instead of introducing new patterns or service-oriented architecture, we will show how to use discipline, consistency and code organization to make your code grow more gently. Instead of using new gems we will use the tools built into Rails to accomplish our goal. The tools you already know and love.

How we got here

When you started working with Rails some years ago, it all seemed so easy. You saw the blog-in-ten-minutes video. You reproduced the result. ActiveRecord felt *great* and everything had its place.

Fast forward two years. Your blog is now a full-blown CMS with a hundred models and controllers. Your team has grown to four developers. Every change to the application is a pain. Your code feels like a house of cards.

You turn to the internet for assistance, and find it filled with silver bullets. You should move away from fat controllers, it says. But do avoid fat models. And use DCI. Or CQRS. Or SOA. As you cycle through patterns, your application is becoming a patchwork of different coding techniques. New team members are having a hard time catching up. And you are beginning to question if all those new techniques actually help, or if you are just adding layers of indirection.

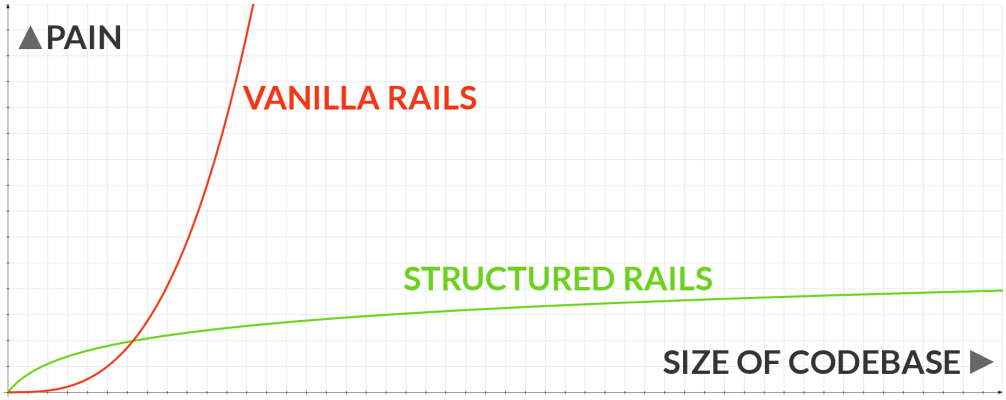
You start missing the early days, when everything had seemed so easy and every new piece of code had its place. You actually *liked* ActiveRecord before it drowned you in a sea of callbacks. If only there was a way to do things "the Rails way" without it falling apart as your application grows.

The myth of the infinitely scalable architecture

We would like to show you one path to write Rails apps that are a joy to understand and change, even as your team and codebase grows. This book describes a complete toolbox that has served us well for all requirements that we have encountered.

Before we do that we need to let you in on an inconvenient secret: *Large applications are large*. The optimal implementation of a large application will always be more complex than the optimal representation of a smaller app. We cannot make this go away. What we *can* do is to organize a codebase in a way that "scales logarithmically". Twice as many models should not mean twice as many problems.

A business management consultant would use a chart like this:



Vanilla Rails vs. Structured Rails

In order to achieve the green line in the chart above, you do not necessarily need to change the way your application is built. You do not necessarily need to introduce revolutionary architectures to your code. You can probably make it with the tools built into Rails, if you use them in a smarter way.

Compare this to sorting algorithms. When a sorting function is too slow, your first thought is very likely not "we should install a Hadoop cluster". Instead you simply look for an algorithm that scales better and go the Hadoop way when you become Amazon.

In a similar fashion this book is not about revolutionary design patterns or magic gems that make all your problems go away. Instead, we will show how to use discipline, consistency and organization to make your application grow more gently.

How we structured this book

We divided the book into three parts:

Part I: New rules for Rails

In this part we unlearn bad Rails habits and introduce design conventions for controllers and user-facing models. By being consistent in our design decisions we can make it easier to navigate and understand our application even as its codebase grows.

The first part contains the following chapters:

- Beautiful controllers
- Relearning ActiveRecord
- User interactions without a database

Part II: Creating a system for growth

As we implement more and more requirements, all that code has to go somewhere. If all we do is add more lines to existing classes and methods, we end up with an unmaintainable mess.

In this part we show how to organize code in a way that encourages the creation of new classes which in turn enhances comprehensibility and maintainability. We also lean to contain the side effects that code can have on unrelated parts of our application.

The second part discusses the following topics:

- Dealing with fat models
- A home for interaction-specific code
- Extracting service objects
- Organizing large codebases with namespaces
- Taming stylesheets

Part III: Building applications to last

We show how to think about future maintenance when making decisions today. We make a case for adopting new technologies and patterns with care, and for taking full responsibility for those techniques and technologies that you *do* choose to adopt.

The last part contains the following chapters:

- On following fashions
- Surviving the upgrade pace of Rails
- Owning your stack
- The value of tests

New rules for Rails

2 Beautiful controllers

Let's talk about controllers. Nobody loves their controllers.

When a developer learns about MVC for the first time, she will quickly understand the purpose of models and views. But working with controllers remains awkward:

- It is hard to decide whether a new bit of functionality should go into your controller or into your model. Should the model send a notification e-mail or is that the controller's responsibility? Where to put support code that handles the differences between model and user interface?
- Implementing custom mappings between a model and a screen requires too much controller code. Examples for this are actions that operate on multiple models, or a form that has additional fields not contained in the model. It is too cumbersome to support basic interactions like a "form roundtrip", where an invalid form is displayed again with the offending fields highlighted in red.
- Any kind of code put into a controller might as well sit behind a glass wall. You can see it, but it is hard to test and experiment with it. Running controller code requires a complex environment (request, params, sessions, etc.) which Rails must conjure for you. This makes controllers an inaccessible habitat for any kind of code.
- Lacking clear guidelines for designing controllers, no two controllers are alike. This makes working on existing UI a chore, since you have to understand how data flows through each individual controller.

We cannot make controllers go away. However, by following a few simple guidelines we can reduce the importance of controllers in our application and move controller code to a better place. Because the less business logic is buried inside controllers, the better.

The case for consistent controller design

Ruby on Rails has few conventions for designing a controller class. There are no rules how to instantiate model classes or how we deal with input errors. The result is a `controllers` folder where every class works a little differently.

This approach does not scale. When every controller follows a different design, a developer needs to learn a new micro API for every UI interaction she needs to change.

A better approach is to use a *standard controller design* for every single user interaction.

This reduces the mental overhead required to navigate through a large Rails application and understand what is happening. If you knew the layout of a controller class before you even open the file, you could focus on models and views. That is our goal.

Having a default design approach also speeds up development of new controllers by removing implementation decisions. You always decide to use CRUD and quickly move on to the parts of your application you really care about: Your models and views.

This point becomes more important as your team grows, or once your application becomes too large to fit into your head entirely.

Normalizing user interactions

How to come up with a default controller design when your application has many different kinds of user interactions? The pattern we use is to reduce every user interaction to a Rails CRUD resource[1]. We employ this mapping even if the *user* interface is not necessarily a typical CRUD interface at first glance.

Even interactions that do not look like plain old CRUD resources can be modeled as such. A screen to *cancel a subscription* can be thought of as *destroying a subscription* or *creating a new cancellation*. A screen to upload multiple images at once can be seen as *creating an image batch* (even if there is no `ImageBatch` model).

By normalizing every user interaction to a CRUD interaction, we can design a beautiful controller layout and reuse it again and again with little changes.

A better controller implementation

When we take over maintenance for existing Rails projects, we often find unloved controllers, where awkward glue code has been paved over and over again, negotiating between request and model in the most unholy of protocols.

It does not have to be that way. We believe that controllers deserve better:

- Controllers should receive the same amount of programming discipline as any other type of class. They should be short, DRY[2] and easy to read.
- Controllers should provide the *minimum amount of glue code* to negotiate between request and model.
- Unless there are good reasons against it, controllers should be built against a standard, proven implementation blueprint.

[1] http://guides.rubyonrails.org/routing.html#resource-routing-the-rails-default
[2] http://en.wikipedia.org/wiki/Don't_repeat_yourself

What is a controller implementation that is so good you want to use it over and over again? Of course Rails has always included a `scaffold` script that generates controller code. Unfortunately that generated controller is unnecessarily verbose and not DRY at all.

Instead we use the following standard implementation for a controller that CRUDs an ActiveRecord (or ActiveModel) class:

```
class NotesController < ApplicationController

  def index
    load_notes
  end

  def show
    load_note
  end

  def new
    build_note
  end

  def create
    build_note
    save_note or render 'new'
  end

  def edit
    load_note
    build_note
  end

  def update
    load_note
    build_note
    save_note or render 'edit'
  end

  def destroy
    load_note
    @note.destroy
    redirect_to notes_path
  end

```

```
37    private
38
39    def load_notes
40      @notes ||= note_scope.to_a
41    end
42
43    def load_note
44      @note ||= note_scope.find(params[:id])
45    end
46
47    def build_note
48      @note ||= note_scope.build
49      @note.attributes = note_params
50    end
51
52    def save_note
53      if @note.save
54        redirect_to @note
55      end
56    end
57
58    def note_params
59      note_params = params[:note]
60      note_params ? note_params.permit(:title, :text, :published) : {}
61    end
62
63    def note_scope
64      Note.scoped
65    end
66
67  end
```

Note a couple of things about the code above:

- The controller actions are delegating most of their work to helper methods like `load_note` or `build_note`. This allows us to not repeat ourselves and is a great way to adapt the behavior of multiple controller actions by changing a single helper method. E.g. if you want to place some restriction on how objects are created, you probably want to apply the same restriction on how objects are updated. It also facilitates the DRY implementation of custom controller actions (like a `search` action, not visible in the example).
- There is a private method `note_scope` which is used by all member actions (`show`, `edit`, `update`, and `destroy`) to load a `Note` with a given ID. It is also used by `index` to load the list of all notes.

Note how at no point does an action talk to the `Note` model directly. By having `note_scope` guard access to the `Note` model, we have a central place to control which records this controller can show, list or change. This is a great technique to e.g. implement authorization schemes[3] where access often depends on the current user.

- There is a private method `note_params` that returns the attributes that can be set through the `update` and `create` actions. Note how that method uses strong parameters[4] to whitelist the attributes that the user is allowed to change. This way we do not accidentally allow changing sensitive attributes, such as foreign keys or admin flags. Strong parameters are available in Rails 4+. If you are on Rails 3 you can use the strong_parameters gem[5] instead. And if you are on Rails 2 LTS[6] you can use `Hash#slice`[7] to a similar effect. In any case we recommend such an approach in lieu of the `attr_accessible` pattern that used to be the default in older Rails versions. The reason is that authorization does not belong into the model, and it is *really* annoying to have attribute whitelisting get in your way when there is not even a remote user to protect from (e.g. the console, scripts, or background jobs).
- Every controller action reads or changes a *single model*. Even if an update involves multiple models, the job of finding and changing the involved records should be pushed to an orchestrating model. You can do so with nested forms[8] or form models (which we will learn about in a later chapter). By moving glue code from the controller into the model it becomes easier to test and reuse.
- Although the controller from the code example maps directly to an ActiveRecord model called `Note`, this is by no means a requirement. For instance, you might want to use a custom model for forms that are complicated or do not persist to a database. This book will equip you with various techniques for providing a mapping between a user interface and your core domain models.

Why have controllers at all?

We advocate a very simple and consistent controller design that pushes a lot of code into the model. One might ask: Why have controllers at all? Can't we conjure some Ruby magic that automatically maps requests onto our model?

Well, no. There are still several responsibilities left that controllers should handle:

- Security (authentication, authorization)
- Parsing and white-listing parameters
- Loading or instantiating the model

[3] Also see our talk: Solving bizarre authorization requirements with Rails.
[4] http://api.rubyonrails.org/classes/ActionController/StrongParameters.html
[5] https://github.com/rails/strong_parameters
[6] https://railslts.com
[7] http://apidock.com/rails/Hash/slice
[8] http://api.rubyonrails.org/classes/ActiveRecord/NestedAttributes/ClassMethods.html

- Deciding which view to render

That code needs to go *somewhere* and the controller is the right place for it.

However, a controller **never does the heavy lifting**. Controllers should contain the minimum amount of glue to translate between the request, your model and the response.

We will learn techniques to extract glue code into classes in "User interactions without a database" and "A home for interaction-specific code".

But before we do that, we need to talk about ActiveRecord.

A note on controller abstractions

There are gems like Inherited Resources[a] or Resource Controller[b] that generate a uniform controller implementation for you. E.g. the following code would give you a fully implemented `UsersController` with the seven RESTful default actions[c] with a single line of code:

```
class UsersController < ResourceController::Base
end
```

When Ruby loads the `UsersController` class, Resource Controller would dynamically generate a default implementation[d] for your controller actions.

Following the idea of convention over configuration, one would reconfigure the default implementation only if needed:

```
class UsersController < ResourceController::Base

  create.after do
    Mailer.welcome(@user).deliver
  end

end
```

We used to like this idea a *lot*. However, having used it in several large projects, we now prefer to write out controllers manually again. Here are some reasons why we no longer like to use `resource_controller` and friends:

Configuration can be very awkward

Many things that have a natural place in a hand-written controller are awkward to write in a controller abstraction. E.g. using a different model for `new/create` and `edit/update` is almost impossible to implement using the configuration options of Resource Controller.

Too much magic

We rotate on projects a lot. Often new developers on projects using gems such as InheritedResources have a hard time understanding what is happening: Which methods are generated automatically? How do I disable them if not all of them are necessary? Which configuration options do exist? At the end of the day, we saw colleagues spending more time reading documentation than writing code. It's simply too much magic, too much implicit behavior.

Controller abstractions can be useful if you know their pros and cons. We are using them only in very simple CRUD apps with *extremely* uniform user interfaces.

---

[a] https://github.com/josevalim/inherited_resources
[b] https://github.com/makandra/resource_controller
[c] http://guides.rubyonrails.org/routing.html#resource-routing-the-rails-default
[d] https://makandracards.com/makandra/637-default-implementation-of-resource_controller-actions

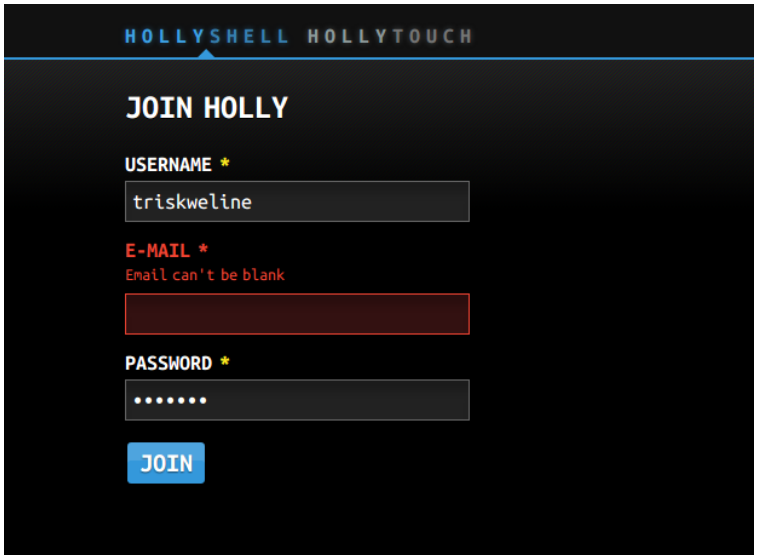
3 Relearning ActiveRecord

Developers burnt by large Rails applications often blame their pain on ActiveRecord. We feel that a lof of this criticism is misplaced. ActiveRecord can be a highly effective way to implement user-facing models, meaning models that back an interaction with a human user.

An example for a user-facing model is a sign up form that takes a username and password, informs the user if some data is invalid, and only creates a user once when all the information is valid.

Since humans are not perfect, a user-facing model needs to revolve around dealing with broken input data. It should make it easy to implement things like the following:

- Validation of data entered by a user (e.g. a username must not be taken, a password must have a minimum number of characters)
- Form roundtrips: When any input field contains invalid data, the form is displayed again with the invalid fields being highlighted. During a roundtrip all input fields retain their values. Only when everything is valid an action is performed (like creating a record).
- Lifecycle callbacks: We want to run code at different points during the human-model-interaction, e.g. do something during the data validation or send an e-mail after successful submission.



A form roundtrip after a validation error

For this type of model, ActiveRecord can be a great choice. ActiveRecord is focused around error handling and input validation. It allows you to collect and process possibly broken user input with a minimum amount of code in controllers and views.

Unfortunately, ActiveRecord also comes with plenty of opportunities to shoot yourself in the foot. When you do not model your classes in the way ActiveRecord wants you to, you will find yourself fighting against the library instead of leveraging its power.

Understanding the ActiveRecord lifecycle

The insidious part about ActiveRecord is that it pretends to get out of your way and let you write a simple Ruby class with methods of your choice. The reality however is that ActiveRecord requires your models to be written in a certain style in order to be effective.

Say you have an `Invite` model. Invites are created without an associated `User`, and can be accepted later. When an invite is accepted, the current user is assigned and a `Membership` is created. So we provide an `accept!` method that does all of that:

```
class Invite < ActiveRecord::Base

  belongs_to :user

  def accept!(user)
    self.user = user
    self.accepted = true
    Membership.create!(:user => user)
    save!
  end

end
```

The problem with the code above is that there are a dozen ways to circumvent the `accept!` method by setting the `user` or `accepted` attribute with one of ActiveRecord's many auto-generated methods. For instance, you cannot prevent other code from simply changing the `accepted` flag without going through `accept!`:

```
invite = Invite.find(...)
invite.accepted = true
invite.save!
```

That is unfortunate, since now we have an `Invite` record in a strange state: It is accepted, but no user was assigned and the membership record is missing. There are many other ways to accidentally create such an invite, e.g.:

- `invite.update_attributes!(accepted: true)`

- `invite[:accepted] = true; invite.save`
- `Invite.new(accepted: true).save`
- `Invite.create!(accepted: true)`

In Objects on Rails[1] Avdi Grim called this the "infinite protocol" of ActiveRecord.

So what does this mean for our `Invite` class? We do not want other code to violate data integrity by accidentally calling the wrong method. After all one of the most basic OOP principles is that it should be very hard to misuse a class.

Make it hard to misuse your model.

One approach we have seen is to get rid of all the "bad" methods that ActiveRecord generates for you[2], leaving only a whitelist of "good" methods that you can control. But we prefer another way. We *love* the powerful API that ActiveRecord exposes. We like to be able to use setters, `create!` or `update_attributes!` as we see fit.

So instead of fighting the nature of ActiveRecord, consider an alternative implementation of the `Invite` class:

```
1  class Invite < ActiveRecord::Base
2
3    belongs_to :user
4
5    validates_presence_of :user_id, :if => :accepted?
6
7    after_save :create_membership_on_accept
8
9    private
10
11   def create_membership_on_accept
12     if accepted? && accepted_changed?
13       Membership.create!(:user => user)
14     end
15   end
16
17 end
```

[1] http://objectsonrails.com/

[2] https://github.com/objects-on-rails/fig-leaf

The implementation above works by expressing its API in *validations and callbacks*. This way clients of the model are free to use any ActiveRecord method to manipulate its state. It can ensure data integrity because ActiveRecord guarantees that callbacks will be called[3].

Never rely on other code to use the custom model methods that you provide. To enforce an API in ActiveRecord, you must express it in validations and callbacks.

Given this statement, you might wonder if there is even a place for model methods that are not callbacks (e.g. the `accept!` method from a previous code example). And there is! Feel free to pepper your model with convenience methods that facilitate the reading and changing model records. Just do not rely on other code using those methods. Always enforce the integrity of your data with validations and callbacks.

Aren't callbacks evil?

If you have been working with ActiveRecord for a while, you have probably been bitten by models with too many callbacks.

If you are now shaking your fist at our suggestion to use *more* instead of *less* callbacks, we ask for your patience. We will discuss and solve this issue in a later chapter (Dealing with Fat Models).

The true API of ActiveRecord models

Once you start expressing your model's behavior in validations and callbacks, you realize that the API of your models actually looks like this:

- You instantiate a record using *any API that suits your needs*.
- You manipulate the record using *any API that suits your needs*.
- Manipulating a record does **not** automatically commit changes to the database. Instead the record is put into a "dirty" state. You can inspect dirty records for errors and preview the changes that will be committed.
- Once a record passes validations, all changes can be committed to the database *in a single transaction*.

Once you fully embrace the restriction of only using callbacks to express yourself, you can reap many benefits:

[3] There are a few ActiveRecord methods like `update_attribute` (singular) that can update a record while skipping callbacks, but they are not used during a regular `save` call.

- Your views become significantly simpler by leveraging ActiveRecord's error tracking. E.g. you no longer need to write and read controller variables to indicate an error state. Invalid form fields are highlighted automatically by the Rails form helpers.
- Developers no longer need to understand a custom API for each model they encounter, making it much easier to understand how a model behaves or how a valid record can be created.
- It is no longer possible to accidentally misuse a model and end up with records in an unexpected state.
- You can use a standard, lean controller design (see chapter Beautiful controllers).
- You will find that there are many useful libraries that work with the standard ActiveRecord API. E.g. the state_machine[4] gem will prevent invalid state transitions by hooking into ActiveRecord's validation mechanism. Or the paper_trail[5] gem can hook into ActiveRecord's lifecycle events to track changes of your model's data.

Note that this is not exclusive to `ActiveRecord` models. We will now take a look on how it works for non-persisted classes.

[4] https://github.com/pluginaweek/state_machine
[5] https://github.com/airblade/paper_trail

4 User interactions without a database

In the previous chapter we have shown that ActiveRecord's focus on error handling makes it an effective tool to implement interactions with a human user.

However, models do not necessarily have to be backed by a database table. There are many UI interactions that do not result in a database change at all. Here are some examples:

- A sign in form
- A search form
- A payment form like Stripe's where the user enters a credit card, but the credit card data is stored on another server (using an API call)

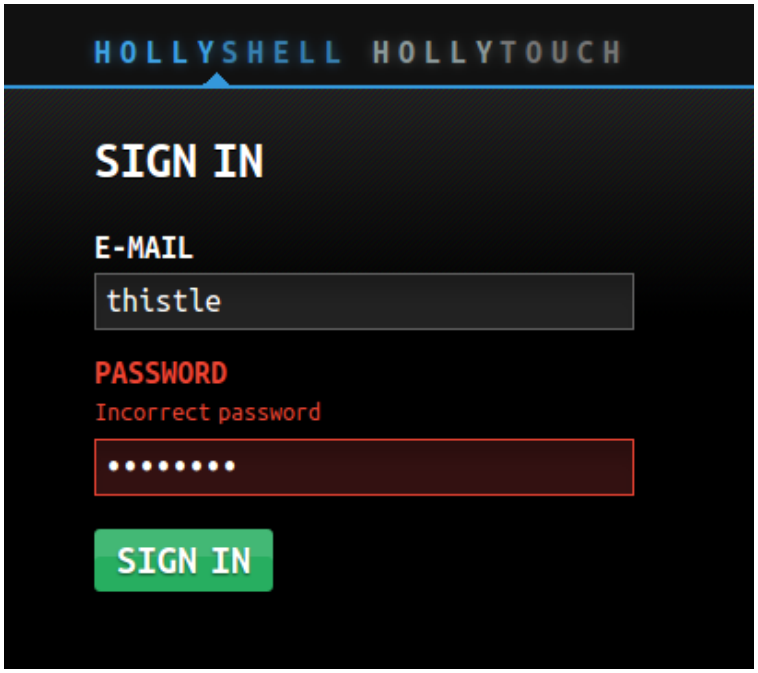
Yet it easy to see why you would want an ActiveRecord-like model for such interactions. After all ActiveRecord gives you so so many convenient features:

- Validations and form roundtrips, where invalid fields are highlighted and show their error message.
- Attribute setters that cast strings to integers, dates, etc. (everything is a string in the params).
- Language translations for model and attribute names.
- Transactional-style form submissions, where an action is only triggered once all validations pass.

In this chapter we will show you how to configure Active**Model** to give you all the convenience of ActiveRecord without a database table, and without many lines of controller code.

Writing a better sign in form

Everyone has written a sign in form. For all purposes, a sign in form behaves like a form implemented with `form_for` and an ActiveRecord model. Input is validated and invalid information is highlighted in the form:



Sign in form after a validation error

The key difference to ActiveRecord is that when the form is submitted, no table row is inserted. Instead, the result of this form submission is the user being signed in (usually by setting a session cookie).

To implement this form, we start by creating a new model class that will house all the logic for this interaction. We call the class `SignIn` and give it accessors for the user's `email` and `password`. The class also validates if a user with the given e-mail address exists, and if the password is correct:

app/models/sign_in.rb

```
1  class SignIn < PlainModel
2
3    attr_accessor :email
4    attr_accessor :password
5
6    validate :validate_user_exists
7    validate :validate_password_correct
8
9    def user
10     User.find_by_email(email) if email.present?
11   end
12
13   private
14
15   def validate_user_exists
```

```
    if user.blank?
      errors.add(:user_id, 'User not found')
    end
  end

  def validate_password_correct
    if user && !user.has_password?(password)
      errors.add(:password, 'Incorrect password')
    end
  end

end
```

"Wait a minute!" we hear you say. "What is this `PlainModel` class that we inherit from?" `PlainModel` is little more than a 20-line wrapper around ActiveModel, which is baked into Rails. If you are not familiar with ActiveModel, it is basically a low-level construction set for classes like `ActiveRecord::Base`. You can find the full source code for `PlainModel` below. For the sake of this example, simply assume that `PlainModel` is a base class that makes plain old Ruby classes feel a lot like ActiveRecord.

Below is the controller that uses the `SignIn` class. It has two actions: The `new` action shows the sign in form while the `create` action validates the user input and sets a session cookie if the credentials are correct.

app/controllers/sessions_controller.rb

```
class SessionsController < ApplicationController

  def new
    build_sign_in
  end

  def create
    build_sign_in
    if @sign_in.save
      # remember user in cookie here
    else
      render 'new'
    end
  end

  private

```

```
18    def build_sign_in
19      @sign_in = SignIn.new(sign_in_params)
20    end
21
22    def sign_in_params
23      sign_in_params = params[:sign_in]
24      sign_in_params.permit(:email, :password) if sign_in_params
25    end
26
27  end
```

Note how the controller differs in no way from a controller that works with a database-backed ActiveRecord model. The `@sign_in` instance has the same lifecycle as an ActiveRecord object:

- Attributes are set from the params
- The object is asked to validate its attributes
- If the object can be "saved", some action is performed

In fact, the `SessionsController` above is almost a line-by-line copy of the default controller implementation we introduced earlier. We like this.

Finally we have the view for the login form:

app/views/sessions/new.html.erb

```
1  <h1>Sign in</h1>
2
3  <%= form_for(@sign_in, url: sessions_path) do |form| %>
4
5    <%= form.label :email %>
6    <%= form.text_field :email %>
7
8    <%= form.label :password %>
9    <%= form.password_field :password %>
10
11   <%= form.submit %>
12
13 <% end %>
```

Note how the view also works entirely like a form for an ActiveRecord model. We can use `form_for` and form helpers like `text_field`. Invalid fields are automatically highlighted (Rails will wrap them in a `<div class="field_with_errors">`) and retain their dirty value until the form submission was successful.

Building PlainModel

In our example above we had the `SignIn` class inherit from `PlainModel`, our custom wrapper around Rails' ActiveModel. We initially used a `PlainModel` that looked like this:

```
1  class PlainModel
2
3    include ActiveModel::Model
4    include ActiveSupport::Callbacks
5    include ActiveModel::Validations::Callbacks
6
7    define_callbacks :save
8
9    def save
10     if valid?
11       run_callbacks :save do
12         true
13       end
14     else
15       false
16     end
17   end
18
19 end
```

Over time we backported more and more features from ActiveRecord into `PlainModel`. For instance, we added coercion (attributes that automatically cast their values to integers, dates, etc.) and something like `belongs_to` (where setting a foreign key like `user_id` sets an associated record like `user` and vice versa).

In the end we wrapped it all in a gem called ActiveType[1]. You can browse through the documentation on GitHub[2].

Using ActiveType our `SignIn` model would look like this:

[1] http://rubygems.org/gems/active_type
[2] https://github.com/makandra/active_type

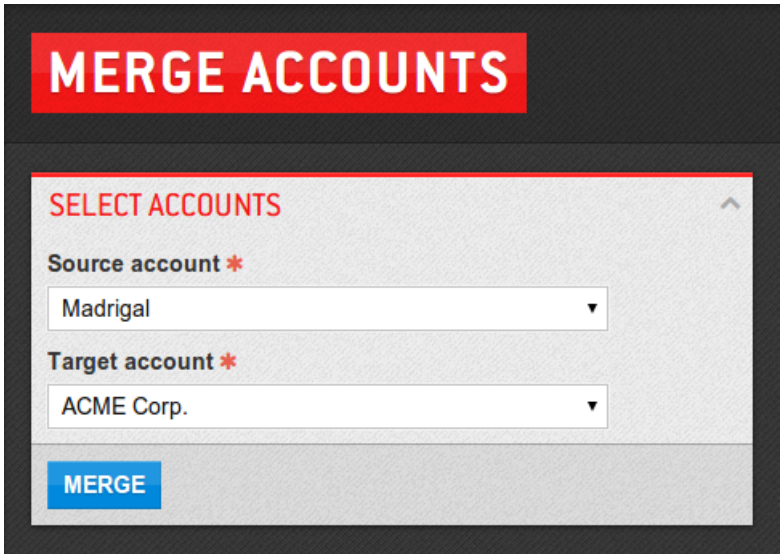
```
class SignIn < ActiveType::Object

  attribute :email, :string
  attribute :password, :string

  validate :validate_user_exists
  validate :validate_password_correct

  def user
    User.find_by_email(email) if email.present?
  end

  private

  def validate_user_exists
    if user.blank?
      errors.add(:user_id, 'User not found')
    end
  end

  def validate_password_correct
    if user && !user.has_password?(password)
      errors.add(:password, 'Incorrect password')
    end
  end

end
```

The only difference to the previous example is that we now inherit from `ActiveType::Object`, and we define attributes using the `attribute` macro (and set a type like `:string`, `:integer` or `:date`) instead of using Ruby's `attr_accessor`. We will learn more about ActiveType in a later chapter, when we talk about extracting interaction-specific code.

Refactoring controllers from hell

ActiveModel-based model classes are a fantastic way to refactor controllers from hell. When we say "controllers from hell" we usually mean controllers that are long, emulate validations, and contain a lot of logic.

We will now show you an example for such a controller, and how we can improve it using ActiveType. Note that this is actual production code. It supports a UI interaction that lets the user select two accounts for merging:



User selects two accounts for merging

Once the user has selected two valid accounts, the merge is performed. The merge involves the following actions:

1. All credits from the first account are transferred to the second account.
2. All subscriptions from the first account are re-assigned to the second account.
3. The first account is deleted.
4. The owner of the first account is notified by e-mail that her account no longer exists.

Before refactoring, our controller from hell looked like this:

```
1  class AccountsController < ApplicationController
2
3    def merge_form
4      @source_missing = false
5      @target_missing = false
6    end
7
8    def do_merge
9      source_id = params[:source_id]
10     target_id = params[:target_id]
11     if source_id.blank?
12       @source_missing = true
13     end
14     if target_id.blank?
15       @target_missing = true
```

```
    end

    if source_id.present? && target_id.present?
      source = Account.find(source_id)
      target = Account.find(target_id)
      Account.transaction do
        target.update_attributes!(:credits => target.credits + source.credits)
        source.subscriptions.each do |subscription|
          subscription.update_attributes!(:account => target)
        end
        Mailer.account_merge_notification(source, target).deliver
        source.destroy
      end
      redirect_to target
    else
      render 'merge_form'
    end

  end

end
```

Below you can see the view from hell that goes with that controller. Since the author did not use an ActiveRecord-like object she could not use `form_for` and had to do everything manually. Note how much pain the view goes through in order to support a "form roundtrip" and display errors near invalid fields:

```
<h1>Merge two accounts</h1>

<%= form_tag do_merge_accounts_path do %>

  <%= label_tag :source_id, 'Source account' %>

  <% if @source_missing %>
    <div class="error">
      You must select a source account!
    </div>
  <% end %>

  <% selectable_accounts = Account.all.collect { |a| [a.name, a.id] } %>
  <%= select_tag :source_id, options_for_select(selectable_accounts) %>

```

```
16    <%= label_tag :target_id, 'Target account' %>
17
18    <% if @target_missing %>
19      <div class="error">
20        You must select a target account!
21      </div>
22    <% end %>
23
24    <%= select_tag :target_id, options_for_select(selectable_accounts) %>
25
26    <%= submit_tag %>
27
28  <% end %>
```

By moving code from the controller into a class inheriting from ActiveType we can reduce the view to the following:

```
1   <h1>Merge two accounts</h1>
2
3   <%= form_for @merge do |form| %>
4
5     <%= form.error_messages_on :source_id %>
6     <%= form.label :source_id %>
7     <%= form.collection_select :source_id, Account.all, :id, :name %>
8
9     <%= form.error_messages_on :target_id %>
10    <%= form.label :target_id %>
11    <%= form.collection_select :target_id, Account.all, :id, :name %>
12
13    <%= form.submit %>
14
15  <% end %>
```

Note how short and and smooth the view has become! Since the `@merge` object supports the ActiveModel API (we will see its implementation in a second) we can now use `form_for` and its many convenient helpers. In case of a validation error, invalid fields are highlighted automatically and all input fields retain their dirty values.

Here is the refactored controller that renders and processes this new form:

```
class AccountMergesController < ApplicationController

  def new
    build_merge
  end

  def create
    build_merge
    if @merge.save
      redirect_to @merge.target
    else
      render 'new'
    end
  end

  private

  def build_merge
    @merge ||= AccountMerge.new(params[:merge])
  end

end
```

Note how the controller has almost become a line-by-line copy of the default controller implementation we introduced earlier.

All the crufty code to emulate validations and perform the actual merge has moved into a much better home. Meet our shiny new `AccountMerge` class:

```
class AccountMerge < ActiveType::Object

  attribute :source_id, :integer
  attribute :target_id, :integer

  validates_presence_of :source_id, :target_id

  belongs_to :source, class_name: 'Account'
  belongs_to :target, class_name: 'Account'

  after_save :transfer_credits
  after_save :transfer_subscriptions
  after_save :destroy_source
  after_save :send_notification
```

```
15
16    private
17
18    def transfer_credits
19      target.update_attributes!(:credits => target.credits + source.credits)
20    end
21
22    def transfer_subscriptions
23      source.subscriptions.each do |subscription|
24        subscription.update_attributes!(:account => target)
25      end
26    end
27
28    def destroy_source
29      source.destroy
30    end
31
32    def send_notification
33      Mailer.account_merge_notification(source, target).deliver
34    end
35
36  end
```

Note how the merge has moved into private methods that are called after successful validation:

```
1  after_save :transfer_credits
2  after_save :transfer_subscriptions
3  after_save :destroy_source
4  after_save :send_notification
```

You might wonder if all we did was move code around, and add another file. And you are right! We did not remove any logic. Note how the `AccountMerge` class describes its validations and merge process much clearer. Compare it with the controller from hell we started with, where everything was just a long blob of spaghetti code.

Also, by moving the code from the controller into the model, it now lives in a place where we can unit-test it much easier than in a controller. Testing controller code requires a complex environment (request, params, sessions, etc.) whereas testing a model method simply involves calling the method and observing its behavior.

In addition, by being able to leverage `form_for` and its many helpers, the view has become *much* shorter and easier to read.

And finally the controller now uses the same control flow as *any other controller* in your application:

1. Instantiate an object
2. Assign attributes from the params
3. Try to `save` the object
4. Render a view or redirect

This consistency makes it easy for another developer to understand and navigate your code. Since she already knows how your controller works, she can concentrate on the `AccountMerge` model.

Creating a system for growth

5 Dealing with fat models

In our chapter "Beautiful controllers" we moved code from the controller into our models. We have seen numerous advantages by doing so: Controllers have become simpler, testing logic has become easier. But after some time, your models have started to exhibit problems of their own:

- You are afraid to save a record because it might trigger undesired callbacks (like sending an e-mail).
- Too many validations and callbacks make it harder to create sample data for a unit test.
- Different UI screens require different support code from your model. E.g. a welcome e-mail should be sent when a `User` is created from public registration form, but not when an administrator creates a `User` in her backend interface.

All of these problems of so-called "fat models" can be mitigated. But before we look at solutions, let's understand why models grow fat in the first place.

Why models grow fat

The main reason why models increase in size is that they must serve more and more purposes over time. E.g. a `User` model in a mature application needs to support a lot of use cases:

- A new user signs up through the registration form
- An existing user signs in with her email and password
- A user wants to reset her lost password
- A user logs in via Facebook (OAuth)
- A users edits her profile
- An admin edits a user from the backend interface
- Other models need to work with `User` records
- Background tasks need to batch-process users every night
- A developer retrieves and changes `User` records on the Rails console

Each of these many use cases leaves scar tissue in your model which affects *all* use cases. You end up with a model that is very hard to use without undesired side effects getting in your way.

Lets look at a typical `User` model one year after you started working on your application, and which use cases have left scars in the model code:

Use case	Scar tissue in `User` model
New user registration form	Validation for password strength policy Accessor and validation for password repetition Accessor and validation for acceptance of terms Accessor and callback to create newsletter subscription Callback to send activation e-mail Callback to set default attributes Protection for sensitive attributes (e.g. admin flag) Code to encrypt user passwords
Login form	Method to look up a user by either e-mail or screen name Method to compare given password with encrypted password
Password recovery	Code to generate a recovery token Code to validate a given recovery token Callback to send recovery link
Facebook login	Code to authenticate a user from OAuth Code to create a user from OAuth Disable password requirement when authenticated via OAuth
Users edits her profile	Validation for password strength policy Accessor and callback to enable password change Callback to resend e-mail activation Validations for social media handles Protection for sensitive attributes (e.g. admin flag)
Admin edits a user	Methods and callbacks for authorization (access control) Attribute to disable a user account Attribute to set an admin flag
Other models that works with users	Default attribute values Validations to enforce data integrity Associations Callbacks to clean up associations when destroyed Scopes to retrieve commonly used lists of users
Background tasks processes users	Scopes to retrieve records in need of processing Methods to perform required task

That's a lot of code to dig through! And even if readability wasn't an issue, a model like this is **a pain to use**:

- You are suddenly afraid to update a record because who knows what callbacks might trigger. E.g. a background job that synchronizes data accidentally sends a thousand e-mails because some `after_save` callback informs the user that her profile was updated (*"it made sense for the user profile"*).

- Other code that wants to create a `User` finds itself unable to save the record because some access control callback forbids it (*"it made sense for the admin area"*).
- Different kind of `User` forms require different kind of validations, and validations from that other form are always in the way. You begin riddling the model with configuration flags to enable or disable this or that behavior.
- Unit tests become impossible because every interaction with `User` has countless side effects that need to be muted through verbose stubbing.

The case of the missing classes

Fat models are often the symptoms of undiscovered classes trying to get out. When we take a close look at the huge table above we can discover new concepts that never made it into their own classes:

- `PasswordRecovery`
- `AdminUserForm`
- `RegistrationForm`
- `ProfileForm`
- `FacebookConnect`

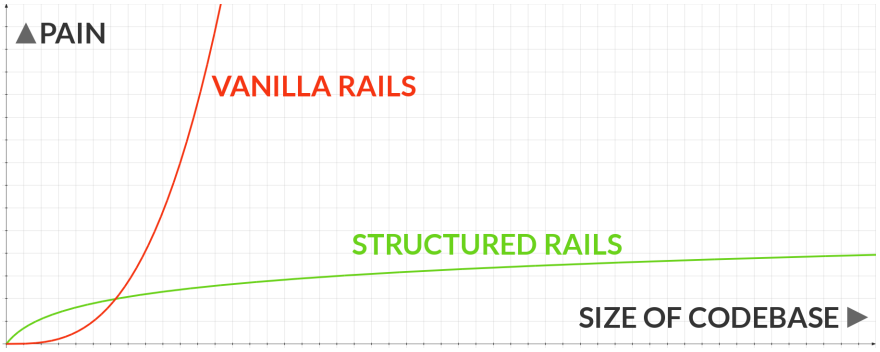
Remember when we told you that large applications are large? When you need to implement password recovery but don't create a place where to put code related to password recovery, *that code doesn't go away*. It will spread itself across existing classes, usually making those classes harder to read and use.

Compare this to your apartment at home and what you do with letters you need to deal with later. Maybe there's a stack of these letters sitting next to your keys or on your desk or dining table, probably all of the above. Because there's no designated place for incoming letters, they are spread all over the apartment. It's hard to find the letter you're looking for. They clutter up your desk. And they're in the way for dinner, too!

Of course there's a simple solution to our letter problem. We can make a box, label it "Inbox" and put it on a shelf above our desk. With all of our letters sitting in a designated place they are no longer in the way for dinner or for working on our desk.

Code never goes away. You need to actively channel it into a place of your choice or it will infest an existing class.

Note that our apartment still contains the same number of letters as it did before. Neither can we make those letters go away. But instead of accepting an increase in clutter we have provided the organizational structure that can carry more items. Remember our chart from the first chapter?



Vanilla Rails vs. Structured Rails

Getting into a habit of organizing

Organizing your letters in an inbox is not hard. But realizing that all those letters lying around are actually yelling *"Make an inbox!"* takes some practice.

In similar fashion, when you are looking for a place to add new code, don't immediately look for an existing ActiveRecord class. Instead look for *new* classes to contain that new logic.

In the following chapters we will show you:

- How to get into a habit of identifying undiscovered concepts in your code
- How to keep a slim core model by channelling interaction-specific support code into their own classes
- How to identify code that does not need to live inside an ActiveRecord model and extract it into service classes
- How to do all of this with the convenience that you are used to from ActiveRecord

6 A home for interaction-specific code

Most mature Rails applications suffer from large models that must serve too many purposes. In our previous chapter we looked at a typical `User` model, and why it grew fat.

The first step to deal with a fat model should be to reduce its giant class to a slim *core model*. All the code you chop away goes into multiple interaction-specific *form models*.

Your core model should only contain the absolute minimum to exist:

- A minimum set of validations to enforce data integrity
- Definitions for associations (`belongs_to`, `has_many`)
- Universally useful convenience methods to find or manipulate records

Your core model should **not** contain logic that is specific to individual screens and forms, such as:

- Validations that only happen on a particular form (e.g. only the sign up form requires confirmation of the user password)
- Virtual attributes to support forms that do not map 1:1 to your database tables (e.g. tags are entered into a single text field separated by comma, the model splits that string into individual tags before validation)
- Callbacks that should only fire for a particular screen or use case (e.g. only the sign up form sends a welcome e-mail)
- Code to support authorization (access control) [1]
- Helper methods to facilitate rendering of complex views

All this interaction-specific logic is better placed in *form models*, which only exist to facilitate a single UI interaction (or a group of closely related UI interactions).

If you consistently extract interaction-specific logic into form models, your core models will remain slim and mostly free of callbacks.

A modest approach to form models

The idea of form models is nothing new. You will find that GitHub is full of gems to support "presenters", "exhibits" or "form models". Unfortunately you will find many of those solutions to be lacking in practice. Here are some problems we encountered:

[1] Also see our talk: Solving bizarre authorization requirements with Rails.

- You can not use them with Rails' form helpers (`form_for`)
- You can not use ActiveRecord macros like `validates_presence_of`
- You can not use nested forms
- They only work for individual objects, but not for a collection of objects (e.g. `index` actions).
- Creating new records and editing existing records requires two different presenter classes (although the UI for both is almost identical)
- They require you to copy & paste validations and other logic from your core model
- They use delegation (which can have confusing semantics of `self`)
- They add a lot of files and mental overhead

Having been disappointed by various gems again and again we wondered: Is it possible to have all the convenience of ActiveRecord and still contain screen-specific logic in separate classes?

We found out that yes, we can, and we do not even need a fancy gem to do it. In fact plain vanilla *inheritance* is all we need in order to extract screen-specific code into their own form models.

Let's look at an example. Consider the following `User` model which has grown a little too fat for its own good:

```
1  class User < ActiveRecord::Base
2
3    validates :email, presence: true, uniqueness: true
4    validates :password, presence: true, confirmation: true
5    validates :terms, acceptance: true
6
7    after_create :send_welcome_email
8
9    private
10
11   def send_welcome_email
12     Mailer.welcome(user).deliver
13   end
14
15 end
```

Clearly a lot of this code pertains to the *sign up form* and should be removed from the `User` core class. You also start having trouble with this obese model, since the validations are sometimes impractical (the admin form does not require acceptance of terms) and you do not always want to send welcome e-mails whenever creating a new record, e.g. when using the console.

So let's reduce this obese model to the minimum amount of logic that is universally useful and necessary:

```
class User < ActiveRecord::Base

  validates :email, presence: true, uniqueness: true

end
```

All the parts that pertain to the sign up form are moved into a new class `User::AsSignUp` which simply inherits from `User`:

```
class User::AsSignUp < User

  validates :password, presence: true, confirmation: true

  validates :terms, acceptance: true

  after_create :send_welcome_email

  private

  def send_welcome_email
    Mailer.welcome(user).deliver
  end

end
```

Note that we did not need to repeat the validation of the `email` attribute within our form model. Since the form model inherits from the core model, it automatically has all the behavior from the core model and can now build on top of that.

The controller that processes the public sign up form simply uses the new form model `User::AsSignUp` instead of the original `User` class

```
class UsersController < ActionController

  def new
    build_user
  end

  def create
    build_user
    if @user.save
      redirect_to dashboard_path
```

```
    else
      render 'new'
    end
  end

  private

  def build_user
    @user ||= User::AsSignUp.new(user_params)
  end

  def user_params
    user_params = params[:user]
    user_params.permit(
      :email,
      :password,
      :password_confirmation,
      :terms
    )
  end

end
```

Note that the name of the class (`User::AsSignUp`) is simply a naming convention we like to use. There is nothing that forces you to prefix form models with "`As`" or even use a namespace. However, we really like to organize form models in the directory structure like this:

File	Class definition
`user.rb`	`class User < ActiveRecord::Base`
`user/as_sign_up.rb`	`class User::AsSignUp < User`
`user/as_profile.rb`	`class User::AsProfile < User`
`user/as_admin_form.rb`	`class User::AsAdminForm < User`
`user/as_facebook_login.rb`	`class User::AsFacebookLogin < User`

See our section on namespacing for more about code organization.

More convenience for form models

As we kept using form models more and more, we discovered that we could make working with them more convenient by adding a few minor tweaks. Eventually we packaged these tweaks into ActiveType that we learned about earlier. Even though you do not need to use ActiveType to take

advantage of the form model approach, here are a few examples how ActiveType can make your life easier:

- Rails helpers like `url_for`, `form_for`, and `link_to` use the class name to guess which route to link to. This guess is usually wrong for class names like `User::AsSignUp`. ActiveType contains a tweak so that `link_to(@user)` calls the route `user_path(@user)` even if `@user` is an instance of a form model like `User::AsSignUp`.
- When using single table inheritance[2] you probably do not want to store a class name like `User::AsSignUp` in your `type` column. This is also managed by ActiveType, so `User` will be stored as the `type` of a `User::AsSignUp`.
- In form models you often want virtual attributes with coercion (attributes that automatically cast their values to integers, dates, etc.). This is extremely useful when building forms that do not map exactly to database columns. ActiveType comes with some simple syntax to declare typed virtual attributes within the class definition of your form models.

Switching to ActiveType is simple. Instead of inheriting directly like this …

```
1 class User::AsSignUp < User
```

… we inherit like this:

```
1 class User::AsSignUp < ActiveType::Record[User]
```

Everything else will work the same as before, but you get the added convenience tweaks that we mentioned above.

Note that aside from making form models more convenient, ActiveType is also awesome to make any plain Ruby class implement the ActiveModel API. We discussed this in a previous chapter.

In the next chapter we will discuss how to reduce your core model's weight even more.

Can't I just split my model into modules?

A different technique to deal with fat models is to split them into multiple modules, "concerns", or "traits"[a].

While this technique can be useful to share behavior across multiple models, you must understand that it is *only* a form of file organization. Since all those modules are loaded into your model at

[2] http://api.rubyonrails.org/classes/ActiveRecord/Base.html#class-ActiveRecord::Base-label-Single+table+inheritance

runtime, it does not actually reduce the amount of code in your model. In different words: Callbacks and methods in your model will not go away by dividing it into multiple modules.

[a] http://signalvnoise.com/posts/3372-put-chubby-models-on-a-diet-with-concerns

7 Extracting service objects

It is important to realize that writing ActiveModel classes involves a trade-off. ActiveModel shines when used for user-facing forms, where form roundtrips and validation errors are a core part of the user interaction. In these cases, you get a lot of convenience with a few lines of code.

But not every class in `app/models` is required to use ActiveRecord or ActiveModel. When you don't *need* the features of ActiveModel (such as validations, callbacks, dirty tracking, etc.), you should not object your class to the constraints that the ActiveModel API forces upon you. The class no longer needs to be a "bag of attributes with some callbacks".

When looking through a fat model, you will usually find a lot of code that does not need to be tied to ActiveRecord. This code is often only called by other models, never talks to the user at all, or only supports a very simple interaction like registering an event or looking up a value.

Such code should be extracted into plain Ruby classes that do one job and do them well. A fancy name for that sort of class is "service object", although it's a simple Ruby class. A great way to slim down fat models is to walk through the code and look for a set of methods that can be extracted into their own service object.

Example

Consider a model called `Note`. The model has a `search` method that greps all available notes by performing a naive MySQL `LIKE` query[1]. So when we call `Note.search('growing rails')` the resulting SQL query would like this:

```
1  SELECT * FROM notes WHERE
2    title LIKE "%growing%" OR body LIKE "%growing%" OR
3    title LIKE "%rails%" OR body LIKE "%rails%";
```

Here is the implementation of `Note` that does just that:

[1] http://dev.mysql.com/doc/refman/5.7/en/string-comparison-functions.html

```ruby
class Note < ActiveRecord::Base

  validates_presence_of :title, :body

  def self.search(query)
    parts = []
    bindings = []
    split_query(query).each do |word|
      escaped_word = escape_for_like_query(word)
      parts << 'body LIKE ? OR title LIKE ?'
      bindings << escaped_word
      bindings << escaped_word
    end
    where(parts.join(' OR '), *bindings)
  end

  private

  def self.split_query(query)
    query.split(/\s+/)
  end

  def self.escape_for_like_query(phrase)
    phrase.gsub("%", "\\%").gsub("_", "\\_")
  end

end
```

When you think about it, the methods `search`, `split_query`, and `escape_for_like_query` do not need to live inside the model class. A trivial refactoring is to take these closely related methods and move them into a new class that is all about search:

```ruby
class Note::Search

  def initialize(query)
    @query = query
  end

  def matches
    parts = []
    bindings = []
    split_query.each do |word|
```

```
11        escaped_word = escape_for_like_query(word)
12        parts << 'body LIKE ? OR title LIKE ?'
13        bindings << escaped_word
14        bindings << escaped_word
15      end
16      Note.where(parts.join(' OR '), *bindings)
17    end
18
19    private
20
21    def split_query
22      @query.split(/\s+/)
23    end
24
25    def escape_for_like_query(phrase)
26      phrase.gsub("%", "\\%").gsub("_", "\\_")
27    end
28
29  end
```

`Note::Search` does only one thing and does it well: It takes a query and returns a set of matching notes. It's a simple class with a single responsibility.

The `Note` class itself is now reduced to this:

```
1  class Note < ActiveRecord::Base
2
3    validates_presence_of :title, :body
4
5  end
```

Not only have we freed Note from some code, we also created a place where new search-related requirements can live in the future. If new requirements appear, such as stemming or moving the search engine from MySQL to Lucene, `Note::Search` will be a good home for the code required to do that.

Finding the seams along which you can cut your code into independent concepts like "Note" or "Search" or "Excel export" is one of the key strategies to keep a growing application maintainable in the long run.

Aggressively look for opportunities to extract service objects. Often times it is not hard to identify code that can be extracted without much effort. Here are more examples to get your imagination started:

Fat model style	Service object style
`Note.dump_to_excel(path)`	`Note::ExcelExport.save_to(path)`
`User.authenticate(username, password)`	`Login.authenticate(username, password)`
`Project#changes`	`Project::ChangeReport.new(project).changes`
`Invoice#to_pdf`	`Invoice::PdfRenderer.render(invoice)`

Did we just move code around?

When looking at our `Note` class after the refactoring, you might notice that we did not actually remove a single line of code. In fact we even gained a line or two by an additional class declaration. However, what used to be a monolithic blob of intertwined logic is now separated into multiple, loosely coupled components.

By stripping down our core models to basic data integrity, associations and universally useful convenience methods, our classes become easier to use:

- The side effect of any interactions are limited. E.g. you are no longer afraid to save a record because it might trigger a cascade of e-mails or other updates.
- Your core model becomes easier to test, since with less validations and callbacks it is much easier to create sample records.
- It becomes easier to navigate through your growing amount of code. Instead of having one huge class that mixes many different responsibilities, we have many small classes dedicated to a single purpose (`Note::Search`, `Note::Export`, etc.).

Just like the letter box on our desk helps our appartment to remain tidy while carrying more items, channeling logic into interaction models and service objects creates the organizational structure that enables your application to carry more logic.

8 Organizing large codebases with namespaces

As a Rails application grows, so does its `app/models` folder. We've seen applications grow to hundreds of models. With an `app/models` directory that big, it becomes increasingly hard to navigate. Also it becomes near-impossible to understand what the application is about by looking at the `models` folder, where the most important models of your core domain sit next to some support class of low significance.

A good way to not drown in a sea of `.rb` files is to aggressively namespace models into sub-folders. This doesn't actually reduce the number of files of course, but makes it much easier to browse through your model and highlights the important parts.

Namespacing a model is easy. Let's say we have an `Invoice` class and each invoice can have multiple invoice items:

```
class Invoice < ActiveRecord::Base
  has_many :items
end

class Item < ActiveRecord::Base
  belongs_to :invoice
end
```

Clearly `Invoice` is a composition of `Items` and an `Item` cannot live without a containing `Invoice`. Other classes will probably interact with `Invoice` and not with `Item`. So let's get `Item` out of the way by nesting it into the `Invoice` namespace. This involves renaming the class to `Invoice::Item` and moving the source file to `app/models/invoice/item.rb`:

app/models/invoice/item.rb

```
class Invoice::Item < ActiveRecord::Base
  belongs_to :invoice
end
```

What might seem like a trivial refactoring has great effects a few weeks down the road. It is a nasty habit of Rails teams to avoid creating many classes, as if adding another file was an expensive thing to do. And in fact making a huge `models` folder even larger is something that does not feel right.

But since the `models/invoice` folder already existed, your team felt encouraged to create other invoice-related models and place them into this new namespace:

File	Class
app/models/invoice.rb	Invoice
app/models/invoice/item.rb	Invoice::Item
app/models/invoice/reminder.rb	Invoice::Reminder
app/models/invoice/export.rb	Invoice::Export

Note how the namespacing strategy encourages the use of Service Objects in lieu of fat models that contain more functionality than they should.

Real-world example

In order to visualize the effect that heavy namespacing has on a real-world-project, we refactored one of our oldest applications, which was created in a time when we didn't use namespacing.

Here is the `models` folder before refactoring:

app/models before refactoring

```
activity.rb
amortization_cost.rb
api_exchange.rb
api_schema.rb
budget_calculator.rb
budget_rate_budget.rb
budget.rb
budget_template_group.rb
budget_template.rb
business_plan_item.rb
business_plan.rb
company.rb
contact.rb
event.rb
fixed_cost.rb
friction_report.rb
internal_working_cost.rb
invoice_approval_mailer.rb
invoice_approval.rb
invoice_item.rb
invoice.rb
invoice_settings.rb
invoice_template.rb
invoice_template_period.rb
listed_activity_coworkers_summary.rb
```

```
note.rb
person.rb
planner_view.rb
profit_report_settings.rb
project_filter.rb
project_link.rb
project_profit_report.rb
project_rate.rb
project.rb
project_summary.rb
project_team_member.rb
project_type.rb
rate_group.rb
rate.rb
revenue_report.rb
review_project.rb
review.rb
staff_cost.rb
stopwatch.rb
task.rb
team_member.rb
third_party_cost.rb
third_party_cost_report.rb
topix.rb
user.rb
variable_cost.rb
various_earning.rb
workload_report.rb
```

Looking at the huge list of files, could you tell what the application is about? Probably not (it's a project management and invoicing tool).

Let's look at the refactored version:

app/models after refactoring

```
/activity
/api
/contact
/invoice
/planner
/report
/project
activity.rb
contact.rb
planner.rb
invoice.rb
project.rb
user.rb
```

Note how the `app/models` folder now gives you an overview of the core domain at one glance. Every single file is still there, but neatly organized into a clear directory structure. If we asked a new developer to change the way invoices work, she would probably find her way through the code more easily.

Use the same structure everywhere

In a typical Rails application there are many places that are (most of the time) structured like the `models` folder. For instance, you often see helper modules or unit tests named after your models.

When you start using namespaces, make sure that namespacing is also adopted in all the other places that are organized by model. This way you get the benefit of better organization and discoverability in all parts of your application.

Let's say we have a namespaced model `Project::Report`. We should now namespace helpers, controllers and views in the same fashion:

File	Class
`app/models/project/report.rb`	`Project::Report`
`app/helpers/project/report_helper.rb`	`Project::ReportHelper`
`app/controllers/projects/reports_controller.rb`	`Projects::ReportsController`
`app/views/projects/reports/show.html.erb`	View template

Note how we put the controller into a `Projects` (plural) namespace. While this might feel strange at first, it allows for natural nesting of folders in in `app/views`:

```
app/
  views/
    projects/
      index.html.erb
      show.html.erb
      reports/
        show.html.erb
```

If we put the controller into a `Project` (singular) namespace, Rails would expect view templates in a structure like this:

```
app/
  views/
    project/
      reports/
        show.html.erb
    projects/
      index.html.erb
      show.html.erb
```

Note how two folders `project` (singular) and `projects` (plural) sit right next to each other. This doesn't feel right. We feel that the file organization of our views is more important than keeping controller namespace names in singular form.

Organizing test files

When we have tests we nest the test cases and support code like we nest our models. For instance, when you use RSpec and Cucumber, your test files should be organized like this:

File	Description
`spec/models/project/report_spec.rb`	Model test
`spec/controllers/projects/reports_controller_spec.rb`	Controller test
`features/project/reports.feature`	Cucumber untegration test
`features/step_definitions/project/report_steps.rb`	Step definitions

Other ways of organizing files

Of course models/controllers/tests don't always map 1:1:1, but often they do. We think it is at the very least a good default with little ambiguity. When you look for a file in a project structured like this, you always know where to look first.

If another way to split up your files feels better, just go ahead and do it. Do not feel forced to be overly consistent, but always have a good default.

9 Taming stylesheets

You might be surprised to read about CSS in a book about maintainable Rails applications.

However, we found that there is hardly any part of your application that can spiral out of control as quickly and horribly as a `stylesheets` folder.

Does the following sound familiar?

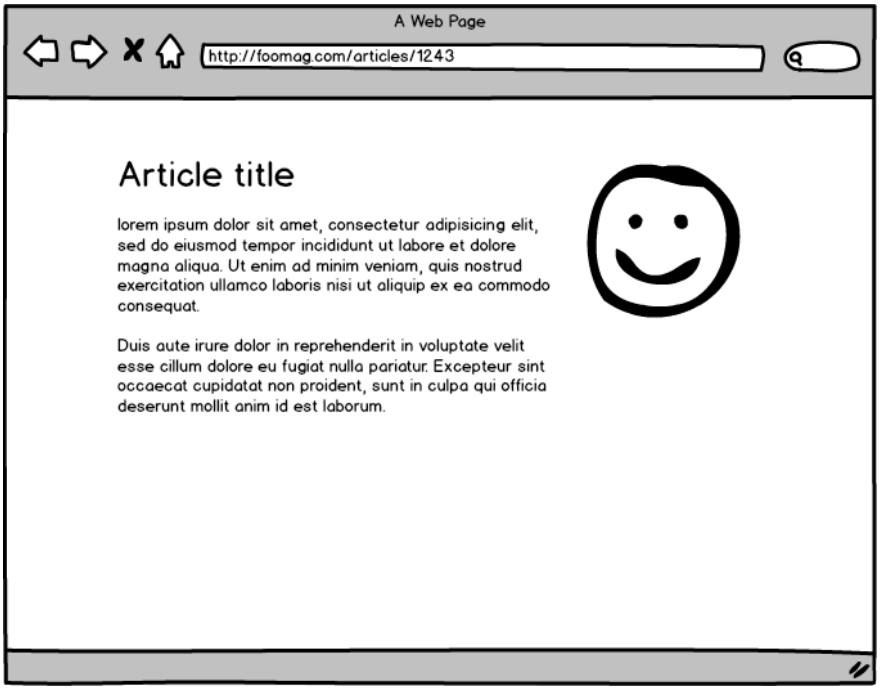
- You are afraid to change a style because it might break screens you are not aware of.
- An innocent tag like `<label>` or `<p>` immediately inherits a million styles and you spend a lot of time overwriting inherited styles that get into your way before you can get started with your own work.
- You often see your own styles being overshadowed by other styles that are more specific because of rules you do not care about. You often need to go nuclear with `!important`.

In this chapter we want to examine why CSS grows unmaintainable, and what we can do to bring back order and sanity.

How CSS grows out of control

We will look at a short example how the most innocent of stylesheets can succumb to the dark side.

Let's say we are making a blog. We begin by styling an article page that shows the article's title, text, and the avatar of the author:



Article show view

We can describe this layout with a couple of HTML tags:

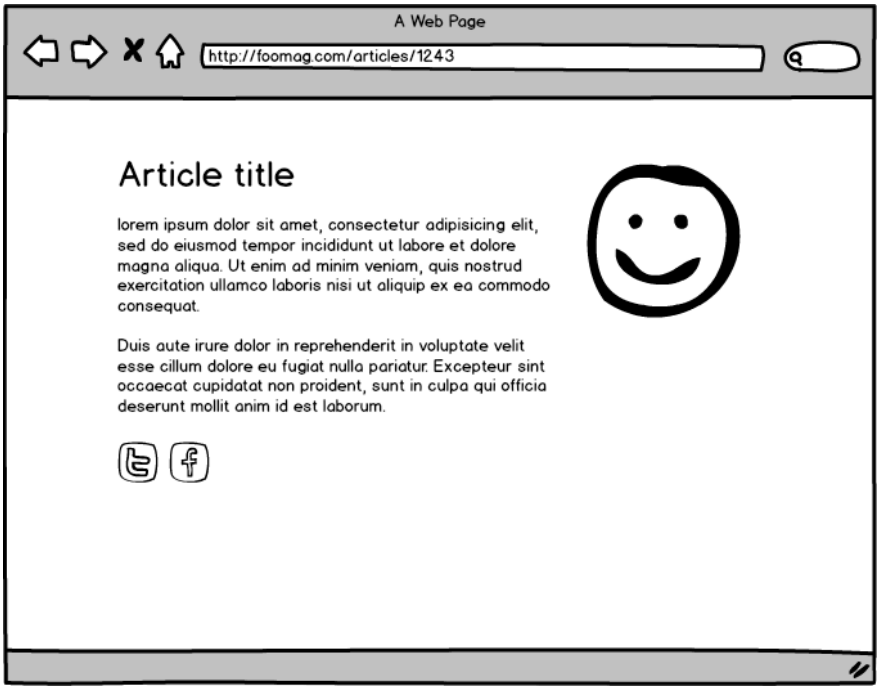
```
<div class="article">
  <img src="avatar.png" />
  <h1>Article title</h1>
  <p>Lorem ipsum...</p>
</div>
```

The CSS that goes along with it is easy enough:

```
.article img {
  width: 200px;
  float: right;
}
```

We test the stylesheet in a browser. Everything looks great and we deploy the first version of our blog.

A few hours later an e-mail arrives. It's a feature request from our product manager: At the end of each article there should be links to share the article via Twitter and Facebook. The e-mail even included a mock-up of the desired look:

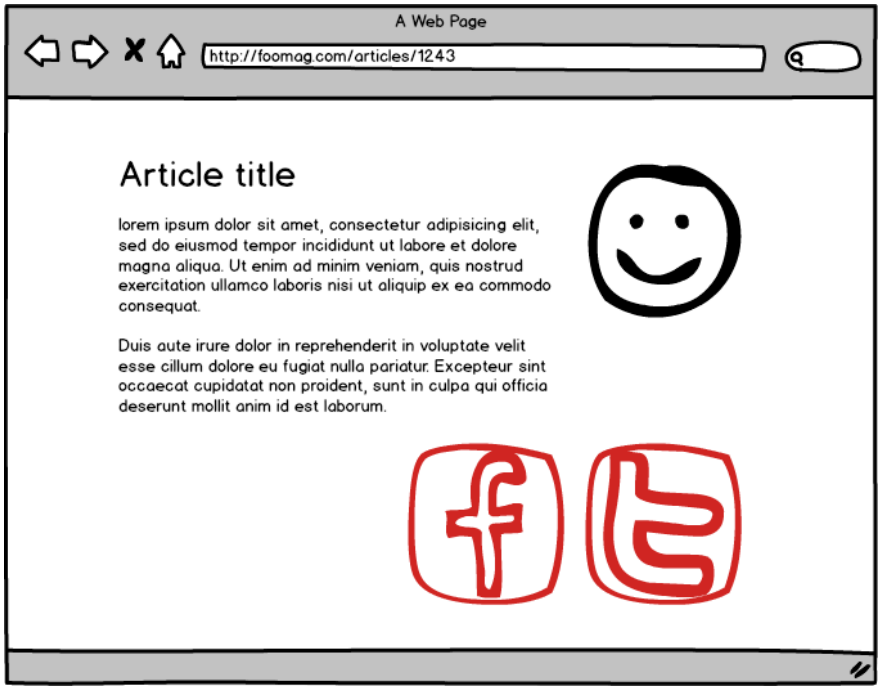


Article with social links

No problem! We quickly add a `<div class="social">` container to our HTML and nest links to our social profiles inside it:

```
<div class="article">
  <img src="avatar.png" />
  <h1>Article title</h1>
  <p>Lorem ipsum...</p>
  <div class="social">
    <a href=".."><img src="twitter.png" /></a>
    <a href=".."><img src="facebook.png" /></a>
  </div>
</div>
```

When we test these changes in our browser, we realize that the images are *huge* and that they are hanging on the right side of the screen. And oddly, the Twitter and Facebook links seem to have switched places:



Broken image sizes

We go back to our HTML and quickly see what went wrong. When we implemented the author avatar, we applied a `size` and `float` style to all the `<img>` tags inside an `.article`. And the icon images for Twitter and Facebook now inherit those styles.

But we won't give up that easily. After all we can simply reset the inherited styles for `<img>` within a `.social` container:

```
.social img {
  width: auto;
  float: none;
}
```

We test our changes in the browser and all is well. We deploy the changes and inform our product manager.

New feature request! The homepage should have a list of recent articles. However, articles on the homepage should *not* show author avatars because it distracts from the huge banner image on the home page. No problem! We will just give the `<body>` tag on the homepage a class `homepage` like this:

```
<html>
  <body class="homepage">
    ...
  </body>
</html>
```

Now we can hide the author avatars on the homepage like this:

```
body.homepage .article img {
  display: none;
}
```

We check the browser for changes. The avatars are gone, so we go ahead and deploy.

It only takes a few minutes for the first bug report to come in. Articles on the homepage are missing their social media links. Embarrassed we quickly deploy a fix that restores them:

```
.article .social img {
  display: inline !important;
}
```

While that fixes the site for now, the CSS we have created is a mess of awkward style dependencies:

1. There is still the original style `.article img`
2. Sometimes the attributes of that style are overshadowed by `.social img`
3. In some contexts the `<img>` element gets hidden by `body.homepage .article img`
4. The element's visibility is restored by `.social img`, by using `!important`.

This cascade of rules overshadowing one another is the "C" in CSS. Its effect is nothing like inheritance in object-oriented programming languages. In CSS every attribute of every selector can be overshadowed by any number of attributes from any other selector. The sum of all side effects of all selectors determines how the site renders in the end. The problem grows with the number of styles in your stylesheet.

Maintainable CSS is hard.

No sane programmer would allow her Ruby code to work like this. As programmers we strive for simple APIs with clear input and output. We avoid side effects and tight coupling whenever we can. We should do the same for our stylesheets.

Luckily, there is a better way.

An API for your stylesheets

Let's imagine we wrote a wish list what a better CSS structure should do for us. Our list would probably look something like this:

- We want to restrict how styles can influence each other
- We want to encourage the reuse of styles
- We want to limit code duplication
- We want to easily see which styles are already available
- We want clear rules where to find existing styles
- We want clear rules how to add new styles
- We want to be able to refactor styles without the fear of removing an intended side effect

A systematic approach to CSS that delivers all of the above is *BEM* (short for **B**lock, **E**lement, **M**odifier). BEM is not a library you can download. It is a set of rules how to structure your stylesheets.

BEM is to your oldschool stylesheets what a cleanly cut domain model is to 10000 lines of PHP spaghetti code. It divides responsibilities and provides a clear API for your styles. Most importantly, it draws clear lines in the sand to dictate where side effects may flow.

At first, BEM will feel limiting. You might even ask yourself why you should restrict your freedom to write styles in any way you want. But just like spaghetti vs. OOP, the benefits will quickly become clear with use. After a week with BEM the old school way of writing CSS will feel dirty and unprofessional to you.

Blocks

In BEM you structure every style in the same, uniform way. Instead of wildly adding styles where it makes sense at the time, BEM stylesheets consist of a simple, flat list of "blocks".

Here are some examples for blocks:

- A navigation bar
- A blog article
- A row of buttons
- Columns that divide the available horizontal space

Think of blocks as the *classes* of your stylesheet API. Blocks are implemented as simple CSS selectors:

```
.navigation {
  ...
}

.article {
  ...
}

.buttons {
  ...
}
```

Elements

Components are often more complex than a single style rule. That's why BEM has the notion of "elements".

Here are some examples for elements:

- The block "navigation bar" comprises multiple "section" elements
- The block "blog article" comprises an element "title" and an element "text"
- The block "columns" contains multiple "column" elements.

Think of elements as the *methods* of your stylesheet classes (blocks). Elements are also implemented as simple CSS selectors, prefixed with the block name:

```
<div class="article">

  <div class="article__title">
    Awesome article
  </div>

  <div class="article__text">
    Lorem ipsum dolor ...
  </div>

</div>
```

In the accompanying stylesheet we put elements alongside blocks:

```
.article {
  ...
}

.article__title {
  ...
}

.article__text {
  ...
}
```

Note that elements can only exist as the child of one specific block, made explicit by the prefixed block name. An element can never exist without a matching block to contain it.

Also note that the list of selectors is again *flat*. We avoid nesting selectors to evade selector specificity[1]. This keeps the overshadowing rules very simple - styles can be overwritten by styles further down in the file. It can also improve browser rendering performance.[2]

You might sniff your nose at the double underscores that separate the block name from the element name. There are other conventions, such as double dashes (`article--title`). In any case you should pick a separator that you can distinguish from a regular word boundary (e.g. the single underscore in `navigation_bar`). Whatever syntax you pick, you will get used to it within a day.

Modifiers

We do not like to repeat code. So what if we need a block that behaves 90% like another block?

- A smaller version of an article
- A differently colored button for a form's primary action
- Highlighting the current navigation section

Think of modifiers as the *optional parameters* of your stylesheet classes (blocks) and methods (elements). Modifiers are implemented as conjunctive selectors on an existing block or modifier. Modifiers then use the cascade to overshadow existing styles *from the same block*:

[1] https://developer.mozilla.org/en-US/docs/Web/CSS/Specificity

[2] Browsers need to match every DOM node against every CSS selector. Short rules can be matched faster than long chains of nested selectors.

```
.button {
  background-color: #666;
  ...
}

.button.is_primary {
  background-color: #35d;
}
```

Note that modifiers are not prefixed with the name of their block. Instead we choose names that indicate "modifierness" such as `is_small` or `has_children`. Since modifiers only attach to existing blocks or elements, we do not need to worry about name conflicts due to the missing prefix.

Block modifiers can also modify elements as long as they are within the same block:

```
.article.is_summary .article__title {
  font-size: 12px;
  ...
}
```

The BEM prime directive

The main reason CSS becomes unmaintainable is the mess of side effects between styles that influence each other. BEM contains these side effects by requiring blocks to be 100% independent from other blocks. In other words: A block is an independent building block that *always renders the same* regardless of context.

The BEM prime directive: A block must never influence the style of other blocks.

Block independence has a massive effect on the maintainability of our CSS:

1. We minimize effects of the cascade. We rarely inherit unwanted styles anymore.
2. We can freely modify blocks without the fear of breaking styles that rely on implicit side effects. For the first time we can refactor CSS *without* needing to go through every single screen looking for breakage.
3. We promote the re-use of styles. Since blocks don't require a particular context to render as intended, it becomes very easy to re-use a block on a new screen.

Let's look how a **violation** of the BEM prime directive looks like. For this let's assume we have an `.article` block to style a blog article. We also have a `.sidebar` block for secondary information that is displayed alongside our main content.

Sometimes we want to display the summary of a related article in the sidebar. Article summaries should be displayed in a smaller font. The quickest way to write that in CSS is this:

```
.article {
  ...
}

.sidebar .article {
  font-size: 12px;
}
```

That violates the BEM prime directive. By modifying the `.article` style in the context of `.sidebar` we created a dependency between those two selectors. Our `.article` is no longer an independent block, since changes in `.sidebar` can now break `.article`.

Thoughtlessly coupling blocks together will throw us back into the style dependency hell that we just emerged from.

Resolving violations

If you need an `.article` to behave differently within a `.sidebar`, you have two options. You could make `.article` an element of `.sidebar`:

```
.sidebar {
  ...
}

.sidebar__article {
  font-size: 12px;
}
```

Interactions between blocks and their elements are allowed.

But in this case this severely limits the reusability of the article block. A better way here is to give `.article` a well-named modifier like `.is_summary`:

```
.article {
  ...
}

.article.is_summary {
  font-size: 12px;
}

.sidebar {
  ...
}
```

This way `.article` can implement the desired effect (being smaller) without knowing about `.sidebar`. Since there is no coupling between the two selectors, they can both be used and refactored independently from each other.

Escape from the CSS cascade hell by isolating blocks.

Full BEM layout example

To show the application BEM with a larger, practical example, we will implement the following site layout:

Example site layout

Below is the HTML required to implement this layout in BEM-style. Note that we used `<div>` tags for everything, even though there are more meaningful tags in HTML5. We did this to keep the example focused on structure and class name choices.

```
<div class="layout">

  <div class="layout__head">

    <div class="logo">...</div>

    <div class="navigation">
      <a href="..." class="navigation_section is_current">Home</div>
      <a href="..." class="navigation_section">Archives</div>
      <a href="..." class="navigation_section">About</div>
    </div>

  </div>

  <div class="layout__main">

    <div class="title">Featured article</div>

    <div class="article">
```

```
      <div class="article__title">Giraffe escapes from zoo</div>
      <div class="article__text">
        ...
      </div>
    </div>

  </div>

  <div class="layout__side">

    <div class="title">Related articles</div>

    <div class="article is_summary">
      <div class="article__title">Lion escapes from circus</div>
      <div class="article__text">
        ...
      </div>
    </div>

    <div class="article is_summary">
      <div class="article__title">Penguins love ice cream</div>
      <div class="article__text">
        ...
      </div>
    </div>

  </div>

</div>
```

Here is our BEM stylesheet structure:

```
.layout {
  ...
}

.layout__head {
  ...
}

.layout__main {
  ...
```

```
}

.layout__side {
  ...
}

.logo {
  ...
}

.navigation {
  ...
}

.navigation__section {
  ...
}

.navigation__section.is_current {
  ...
}

.title {
  ...
}

.article {
  ...
}

.article__title {
  ...
}

.article__text {
  ...
}

.article.is_summary .article__title {
  ...
}

```

```
.article.is_summary .article__text {
  ...
}
```

Note some things:

- No block ever influences another block. E. g. even though articles in the sidebar are smaller than in the main content area, an `.article` doesn't simply render smaller within `.layout__side`. Instead we choose an explicit modifier `article.is_small` to contain the style variant within the `.article` block.
- Class names are unambiguous. Although there are two kinds of "titles" in the example, they have two distinct names: `.title` indicates a section heading, `.article__title` indicates the title of an article. Not reusing names is important to prevent the CSS cascade and keep blocks independent from each other.
- The page layout is simply another block `.layout`. The individual parts of the layout, such as page head, main content area and sidebar, make up the elements of the `.layout` block.
- Sometimes it is ambiguous whether a new element should become its own block or if it should be added as an element of an existing block. In the example above it could be argued that `.logo` should be `.layout__logo`, since the site logo is part of the layout. Often both solutions are fine. Do be wary of "god" blocks that grow too large.

Organizing stylesheets

We recommend to use one file for each BEM block. This enforces the independence of blocks. If you do not even see another block selector in the same file, you are even less tempted to violate the BEM prime directive. Foreign block selectors should stick out like a sore thumb.

Your `stylesheets` folder will look like this:

app/assets/stylesheets

```
normalize.css
blocks/
  button.css
  columns.css
  layout.css
  map.css
  pagination.css
```

You can easily concat these many files using the Rails asset pipeline[3]:

[3] http://guides.rubyonrails.org/asset_pipeline.html

app/assets/stylesheets/application.css

```
/*
 *= require ./normalize
 *= require_tree ./blocks
 */
```

BEM anti-patterns

Using BEM is no guarantee for a maintainable library of reusable styles. We have seen *horrible* BEM stylesheets, featuring anti-patterns like the following:

- Badly named elements (discouraging reuse)
- Giant "god" blocks that are many pages long (making reuse impossible)
- Blocks that describe individual screens (instead of attempting to extract reusable components)
- Violations of the BEM prime directive (making the whole exercise pointless)

It is easy to get side-tracked by all the restrictions that BEM places on you, especially when you have years of bad CSS habits to unlearn. Struggling to implement your designs at all you might resort to one of the anti-patterns listed above.

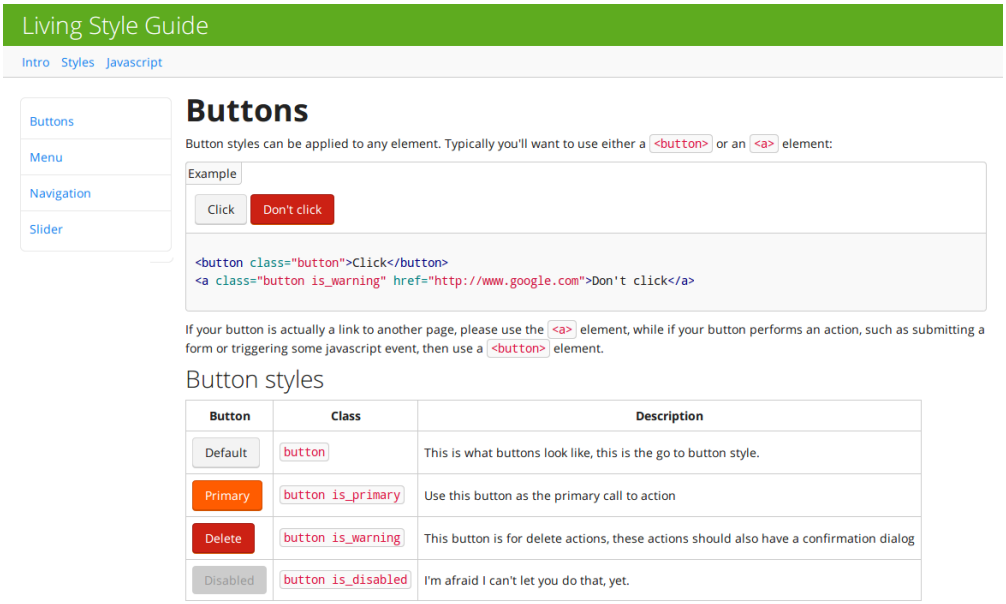
It's important to never forget the point of all of this. What we are doing here is creating a *public API for your stylesheets*. Put yourself in the position of your colleague developer who starts on your project next week. What would you like to see?

Your stylesheets should be like a delicious menu of aptly named and reusable styles from which you can compose new and beautiful screens. *A BEM block should scream at you what it does.*

Write the stylesheet library you would want use.

Living style guides

A great way to make your team care for your library of BEM blocks is to use a *living style guide*. The idea is that you annotate your CSS with documentation and usage examples. There are tools that use those annotations to generate a style guide document like this:



A style guide generated from CSS comments

Developers can now browse this document to see which BEM blocks are available for reuse or extension. This greatly promotes the care of your style library. A team with a living style guide is less likely to employ BEM anti-patterns.

The annotations can live in CSS comments above your styles like this:

```
/*doc
---
title: Buttons
name: button
category: Styles
---

Button styles can be applied to any element. Typically you'll want to
use either a `<button>` or an `<a>` element:

```html_example
<button class="button">Click</button>
<a class="button is_warning" href="http://www.google.com">Don't click</a>
```

*/

.button {
  ...
```

```
}

.button.is_warning {
  ...
}
```

In order to generate the style guide for the example above we used the hologram gem[4] from the people at Trulia[5]. There are many other gems that do similar things. What we like about `hologram` is that the documentation lives in the BEM blocks themselves (rather than in a a separate file).

Pragmatic BEM

BEM is a coding discipline. In order to enjoy the advantages of independent styles and better organisation you need to stick to the rules. Once you start breaking the BEM prime directive the benefits will go away and you are left with a folder of strangely named styles.

However, there are instances where it makes sense to *slightly* deviate from the BEM dogma.

An example for this are *compound HTML structures*. The block/element relationship can be found in vanilla HTML. E. g. a `<table>` tag comprises multiple `<tr>` and `<td>` elements. If we made a table block like `table.grid`, BEM dogma would require us to add classes to every `<tr>` and `<td>` it contains. The resulting HTML structure would like this:

```
<table class="grid">
  <tr class="grid__row">
    <td class="grid__cell">...</td>
    <td class="grid__cell">...</td>
  </tr>
</table>
```

We believe this is too much code for little benefit. In practice we assume an implicit block/element relationship between `<table>` and the contained `<tr>` and `<td>`. We would only write this:

[4] http://trulia.github.io/hologram/
[5] http://www.trulia.com/

```
<table class="grid">
  <tr>
    <td>...</td>
    <td>...</td>
  </tr>
</table>
```

When to be pragmatic

We made some rules for ourselves when to consider a pragmatic interpretation of BEM:

- We never violate the BEM prime directive. Once we assign a BEM block name, we *never* reference it from another block.
- The effect of our pragmatism must be contained within small boundaries. In the example above, styling `<td>` tags within a `table.grid` only affects table cells within that `table.grid` block, so we feel the damage is contained. However, if our design or content would often feature nested tables of different block types, we *would* give `<tr>`s and `<td>`s element names.

Here are some examples where we give ourselves a little freedom to violate the BEM dogma:

- We don't give classes to the children of compound HTML elements (`<table>`, `<dl>`, `<ul>`, ...)
- We allow CSS to insert unnamed elements with `:before` and `:after` statements instead of adding explicit elements into the HTML.
- We allow pseudo-selectors like `:hover` or `:first-child` (instead of using an implicit modifier).
- We allow media queries (instead of using an implicit modifier set by Javascript).
- When we have long-form text content (e. g. the text of a blog post) with a limited set of tags (like `<p>`, `<b>` and `<a>`) we sometimes style plain HTML tags within a container like `.text_content`. This is often the case when we let users enter formatted text in a WYSIWYG or Markdown editor, and we want to display that text on the site. It would simply be impractical to go through the user-provided HTML and replace every `<a>` tag with a `<a class="post__hyperlink">`.
- Sometimes other libraries generate HTML for us, e.g. for a lightbox or another canned UI widget. We cannot always control how these libraries use CSS class names, and generated HTML rarely fits into whatever BEM structure we decided on for our own HTML. In that case we simply style the generated tags within a container.

Flirting with a pragmatic interpretation of BEM's structural rules is a dangerous game. It can save a lot of time, but make a few badly chosen exceptions and you will be back in the hell of style dependencies that you came from.

So tread carefully. And if you are ever unsure, just be strict about it.

Building applications to last

10 On following fashions

The ecosystem of Ruby on Rails is not immune to fashion. From one day to another social media is buzzing with praise for a new architectural pattern, or for a new gem that promises to completely change the way you think about Ruby.

It is often hard to judge which of these fashions are just short-lived trends, and which are here to stay.

Before/after code comparisons

A good way to judge a new technique is to rewrite a small part of your application in the proposed style, then compare the code before and after the rewrite. As you see both versions side-by-side you should ask yourself:

- Does the code *feel* easier to read and change after the refactoring? Can you find words to back up your impression?
- How did the amount of code, the number of files, the coupling between classes change? If there is an increase of artifacts or complexity, is it outweighed by the benefits you gain?
- Did you run into new issues that the old code did not have?

If you have colleagues, ask them for an opinion.

If the new technique requires a new library, consider the cost of upgrading (also see the next chapter: "Surviving the upgrade pace of Rails").

Once you understand a new pattern and the trade-off it involves, adapt what works for you and discard what does not.

Understanding trade-offs

There are no silver bullets. New patterns will always be incremental improvements, or simply exchange one trade-off for another. One technique might have prettier syntax, but makes your classes harder to test. Another pattern might make your code a lot shorter, but at the price of too much magic behavior going on behind the scenes.

Get into a habit of scanning new patterns for trade-offs. See our note on controller abstractions for a detailed breakdown of one technology and the trade-offs it involves.

The value of consistency

Give yourself a break and trust that the default MVC style of Rails will carry you a long way if you are well-organized and consistent in your design decisions.

If you follow every fashion as soon as it arises, your code base will be a patchwork of different styles, making it very hard to understand or change. When adapting new techniques, consider the cost of migrating your complete application or burdening your team with the mental overhead of having to understand multiple code styles.

If you appreciate a consistent code style in your application, it might be worthwhile not to adapt a superior technique until you find the time for a full refactoring.

11 Surviving the upgrade pace of Rails

In contrast to languages such as Java, the Rails ecosystem has little tradition of keeping APIs backward compatible. Hence, new versions of Rails and Ruby gems often break existing code, making an upgrade very time intensive.

Not upgrading is not an option either. Once you get too far behind the latest version of Rails you will not receive any further security updates[1]. Given the severity[2] of some of the vulnerabilities that have been disclosed in the past, you probably do not want to expose an unpatched Rails application to the open Internet.

There are products like Rails LTS[3][4] that offer security patches for old versions of Rails. However, if you want to take advantage of the latest features, you will find yourself locked in an eternal rat race of upgrading your application dependencies every few months.

This chapter wants to give some advice for dealing with this sitation.

Gems increase the cost of upgrades

When adding a new gem dependency, consider the cost of upgrading that gem through the lifespan of your application. Will the gem's author still be interested in maintaining the library two years down the road? When push comes to shove, would *you* be willing to replace that gem or take over maintenance if there is no version compatible with a new version of Rails?

Be aware of different upgrade costs between libraries that provide low-level abstractions and those that offer highly coupled mini frameworks. E.g. a library that supplies geographical calculations might not even have a dependency on Rails and is unlikely to ever break during an upgrade. Whereas a gem that dynamically generates an admin backend for your application will almost certainly break after a new Rails release due to its many hooks into the internals of Rails.

Upgrades are when you pay for monkey patches

Monkey patches (or "freedom patches") describe the practice of opening up an existing class from a gem dependency and overriding some method with custom behavior. While this can be a way to quickly move on when encountering a fatal library bug, monkey patches are usually the first thing that break when upgrading Rails.

---

[1] http://weblog.rubyonrails.org/2013/2/24/maintenance-policy-for-ruby-on-rails/

[2] http://blog.codeclimate.com/blog/2013/01/10/rails-remote-code-execution-vulnerability-explained/

[3] https://railslts.com

[4] Full disclosure: Rails LTS is a product of our company makandra.

Be careful when monkey patching the internals of a class that does not belong to you. You will pay for it later.

If you find a bug in a gem that you like, consider forking the gem, committing your fix with a test and creating a pull request to the original author. This way your monkey patch can soon be replaced with the next official version of the gem. It also makes for good karma.

Don’t live on the bleeding edge

There is no need to upgrade to a new version of Rails on the day it becomes available. Wait until a major Rails release has matured into a couple of patch levels before upgrading, i.e. don’t upgrade to Rails 5.0.0 right away, but wait until Rails 5.0.3 is published.

This also gives authors of your other gem dependencies a chance to update their library to integrate with the new version of Rails.

12 Owning your stack

When you browse through the Ruby Toolbox[1] you will see that there is a gem provided for nearly every imaginable requirement. And indeed the idea of building an application by plugging together pre-assembled components sounds fantastic. However, adding a component to your project has implications that you should be aware of.

When you add a gem or technology to your stack, you must understand that **you now own that component**. That means:

- You are responsible for its behavior under load.
- You are responsible for future security updates.
- You are responsible for upgrading it when you upgrade other parts of your stack. For instance, new versions of Rails are notorious for breaking existing gems.
- You are responsible for maintaining it if the maintainer loses interest, or for swapping it out for an alternative that is still being maintained.
- You also own any of its dependencies, and the dependencies of those dependencies. Adding a single gem to your application often adds a dozen other gems to your stack.

You must own your stack.

In order to decide whether adding a new library is worth the cost of ownership, here are some questions to ask yourself:

- Take a glance at the source code by browsing through the GitHub repository. Does the code look well-groomed or is it a mess of long classes and methods?
- Does it have tests? You should see a `test` or `spec` directory in the GitHub repository. Make sure that it contains actual tests, not just auto-generated stubs.
- Is the library under active development? Check the list of recent commits and if the maintainers are responding to GitHub issues.
- Are you ready to maintain or replace this library if the maintainers lose interest?
- Is the functionality provided by the library significant enough to accept the cost of integration and future maintenance? Take a minute to sketch out the effort required to build the functionality yourself. E.g. instead of adding a library for autocomplete suggestions you could also add a new controller action and some lines of jQuery. But rolling your own XML parser would mean implementing a complex format with all of its intricacies.

[1] https://www.ruby-toolbox.com/

By making informed decisions about when to accept a library into your stack you can speed up development without sabotaging future maintenance work.

Accepting storage services into your stack

Every other week a new database solution is trending on Hacker News, promising to take the pain out of managing and scaling your data. When you accept a new storage service into your stack, you accept the same responsibilities as when you add a new library.

Here are some additional questions you should ask before integrating new storage technology:

- How does it behave under load?
- Can you distribute it? With which guarantees?
- How does it deal with hardware failure?
- Does it have mature Ruby bindings?
- Can you backup data without locking the world?
- Does it offer fast and convenient access to your types of data?
- Are there other people using it in production?

Maxing out your current toolbox

An alternative to adding new technology to your project is to see if one of your current tools can already handle the job. You can always make things more complicated later.

Some examples:

Requirement	New technology	Technology you already have
Key-value store	Redis	Another MySQL table
Full text search	Lucene / Solr / Sunspot	MySQL `LIKE` queries or `FULLTEXT` indexes
Background tasks	Sidekiq queue	Cron job [2]

Note that we do not recommend to flat out refuse new technology. Our point is that you should consider going with a simple solution until your load or data volume justifies the switch.

To be able to easily swap in a new solution later, remember our advice about service objects and hide service access behind a simple API. Say you are implementing a full text search and decide to start with simple MySQL `LIKE` queries instead of integrating an indexer like Solr. Instead of scattering your code with MySQL queries, hide the implementation details behind a service object like this:

[2] Whenever is a tiny gem that automatically installs Cron jobs when you deploy with Capistrano.

```
Article::Search.find('query goes here')
```

This way, when your data outgrows MySQL and you need a more sophisticated replacement, you can simply swap out the implementation of `Article::Search.find` without needing to touch other code.

13 The value of tests

Learning to test your applications effectively might be the most important skill you can learn in your entire career. If we could offer you only one advice for the future, tests would be it.

Recognized benefits you can gain from an effective test suite include the following:

- The frequency of bugs is reduced to a point where you don't think about bugs at all anymore. With that comes a lot of peace of mind. No more lying awake at night, wondering if yesterday's code release is silently eating up customer data.
- The ability to release often. Since you don't need to manually test every part of your application after a change you can release whenever your test suite is green. New features can be shipped to customers and generate business value in no time.
- The freedom to refactor without fear. Since tests will tell you if a refactoring inadvertently breaks peripheral functionality, you can move code around with broke strokes instead of carefully tip-toeing through your source.
- The ability to work on one part of the application without knowing the rest. This becomes critical once your application or team grows to a point where no single person can memorize the whole code base anymore. Having a safety net for regressions is also great for new team members, or when you jump back and forth between multiple projects.

Tests also come with some drawbacks, such as the following:

- Tests take time to write.
- Tests take time to run.
- Tests increase the amount of code you need to maintain during changes.
- Writing tests is a completely separate skill set you need to learn.

Many books have been written about software testing, and you will probably not find two developers who would completely agree about what to test, or how to test it.

What we want to do in this chapter is to give you a high-level overview of what has worked for us.

Choosing test types effectively

There are many types of tests. Unit tests, integration tests, performance tests, the list goes on and on. By choosing the right kind of test to observe a particular behavior you can dramatically increase the effectiveness of your test suite.

We have found the most value in *unit tests* and *full-stack integration tests*. We use unit tests to describe the fine-grained behavior of classes and methods. We use full-stack integration to test the frontend from a user's point of view, with a simulated web browser that magically clicks through the user interface, fills out forms, presses buttons and so on.

Here is an example for a full-stack integration test written in Cucumber[1]. It tests a search form for an address book with company contacts:

```
Given there is a company with the name "Acme"
And there is a company with the name "Madrigal"
When I go to the search form
And I fill in "Query" with "Acme"
And I press "Search"
Then I should see a search result for "Acme"
But I should not see a search result for "Madrigal"
```

In addition to the integration test above, we usually have multiple *unit tests* like the following:

```
describe Company::Search do

  describe '.find' do

    it 'should find companies by a keyword' do
      match = Company.make(:name => 'Acme')
      no_match = Company.make(:name => 'Madrigal')
      Company::Search.find('Acme').should == [match]
    end

    it 'should be case-insensitive' do
      match = Company.make(:name => 'Acme')
      no_match = Company.make(:name => 'Madrigal')
      Company::Search.find('aCmE').should == [match]
    end

    it 'should find companies by a phrase in double-quotes' do
      match = Company.make(:name => 'Acme Corporation')
      no_match = Company.make(:name => 'Acme Toys Corporation')
      Company::Search.find('"Acme Corporation"').should == [match]
    end

  end
```

[1] http://cukes.info/

Note how the unit test is covering a lot more edge cases (like case insensitivity) than the integration test. We could have tested each of these edge cases with integration tests. However, this would have required an awful lot of overhead only to run a simple query and observe the results.

On the other hand, we wouldn't want to leave out the integration test. If we only had the unit test, a lot of code in the controller and view would have no test coverage at all.

Here are some practical guidelines for choosing the right type of test:

1. Try to cover as much ground as possible using unit tests, which are much faster and easier to set up than full-stack integration tests. By moving as much code as possible from controllers and views into form models and service objects, large chunks of behavior will live in a place where it is easy to benefit from a short and fast unit test.
2. Every UI screen should be "kissed" by at least one full-stack integration test. It is sufficient for the test to quickly run one or two paths through the screen, without necessarily covering every edge case. For instance, test that a form shows validation errors, but do not test every single combination of errors.
3. Prefer unit tests to test methods with many code paths or edge cases. An example for this is the `Company::Search.find` method in the example above.

How many tests are too many?

How much to test is a question that every test-driven developer struggles with. How many tests are too many? How few tests are too few? Do I need to write a test for every theoretically possible edge case?

The answer comes with experience, but here is a good rule of thumb. We call it the "Dark House Rule":

The Dark House Rule

Imagine your application as a dark house. Tests are the light you can use to illuminate that house. Integration tests are ambient light that radiates broad beams through a room, but is hard to focus. Unit tests are small, concentrated spotlights that you can shine into a corner.

A test suite does not need to trench the whole house into the glaring lights of an operating table (which would be 110% test coverage). But the house should be lit well enough for you to feel comfortable and to give you the confidence that no mold or monsters grow in a dark corner.

In the end, it always comes down to what you care about the most. If you are an online merchant you probably want the checkout process to be tested in great detail. The same merchant might not care so much about the admin backend that only two people ever see.

When to repeat yourself in tests - and when not to

DRY ("Don't Repeat Yourself"[2]) is one of the first techniques a developer needs to master. After a while DRY becomes so primal to a programmer's thinking that she cringes whenever she sees that principle violated.

When the same programmer learns test driven development, she will be surprised at the level of apparent duplication going on. Seeing the same piece of logic touched by separate unit tests, controller tests and integration tests will seem horribly wrong. But a reasonable level of repetition in your tests is actually a good thing.

First, different kinds of tests do not actually test the same thing. They look at your code through different lenses, some narrowly focused, some gazing at the broader picture. Combining those different points of views allows a much richer description of your code. It also makes your test suite more brittle, often resulting in multiple tests failures after a single code change. This trade-off has a sweet spot.

Sometimes an additional type of test does not add value beyond what other types of tests already cover. As argued above, we often find the combination of unit and integration tests to be sufficient to describe a given feature. Supplementing controller and view tests for the same feature would add little coverage for the amount of code they impose on the project. In that case these tests can and should be omitted. It takes some experience to know when to cut corners this way. When in doubt, write the test.

There is another reason why some repetition in your tests is OK: Your tests are not supposed to be too clever. Clever things should happen in the code your tests are cross-checking. When tests gain complexity through nested method calls and metaprogramming magic, they themselves become subject to defects.

Some people go as far as to not use even basic helpers, like routes, in their tests. They'd rather type out an URL as a string than call the router. We find such purism to be impractical. Your tests should be able to call code that is tested elsewhere. For example, when a complex route requires further description, this can be covered by a dedicated test. This way your other tests don't need to repeat details outside their focus.

Better design guided by tests

It is difficult to design a great class without having some example code that uses that class. E.g. you write a method that works with input streams, but once you start working with that method you discover you would much rather have it accept file paths.

We're big fans of exploring a future class design by writing some caller code *before* actually implementing the class. Unit tests are great for that. Simply "use" the class in your unit tests before

[2] http://en.wikipedia.org/wiki/Don't_repeat_yourself

it even exists. Then implement the class once its usage feels right. You might have heard about this as "test-driven design". By practicing test-driven design you can significantly improve the quality of your classes and methods.

Getting started with tests in legacy applications

You don't need a greenfield Rails project to be able to benefit from tests. Even if you have an existing code base you can easily add tests on top of any legacy code.

Recalling to the Dark House Rule, your untested application is currently trenched in darkness. You should start by installing broad ambient lighting. You do this by writing a full-stack integration test that walks along the "happy path" that is most central to your application. If you're an online merchant, this happy path could look like this:

- User searches for an article
- User inspects the details of an article that she likes
- User adds the article to her shopping cart
- User enters her credit card
- User reviews and places her order
- An admin receives an order confirmation via e-mail

Even if all you have is this one test you are already *so* much better off than when you had to manually click through the order process whenever you make a change to the code. Whatever you do from here on, you can always rely on that test to know that the checkout process works flawlessly.

You don't need full test coverage to benefit from tests.

Now you have a great foundation to expand upon. Whenever you make a change, add another test to describe the new behavior. For instance, when you add a feature to remove an item from the shopping cart, add a test that checks that the item is actually gone.

Overcoming resistance to testing in your team

The best test suite won't help you unless everyone in your team is convinced by the usefulness of having tests. As a developer driving the adoption of TDD in your team, you should make it as easy is possible for your colleagues to benefit from your test suite.

One way to do so is to ensure that there is *one* command that runs *all* the tests to see if the code still works as expected. There should be no additional setup necessary to run your test suite. E.g.

if your tests require sample data in the database, the test cases should insert the required records automatically.

Another principle you should put in place is that no code should be committed with failing or pending tests. Once your test suite is anything but 100% green, it degrades quickly. It can no longer give you feedback if a change has introduced a regression, if it was already failing before the change. It also lowers the barrier for your team to accept more and more failing test cases, since no-one yelled about the previous one.

14 Closing thoughts

This has been version 1.0 of *Growing Rails Applications in Practice*. Did you find it useful? Please let us know by sending your feedback to info@growingrails.com.

In case you need help with your application, note that you can hire us for Rails development and consulting. Simply send a message to info@makandra.com and we'll be in touch.

Finally, if you enjoyed the book, please help us spread the word on Twitter and other social networks. The best way to tell others about the book is to link to our website:

http://growingrails.com[1]

Updates of the book

We plan to update the contents as we learn about better ways of doing things. As a reader you'll always be able to download new versions from your online bookstore at no additional cost.

[1] http://growingrails.com

Copyright notices

Cover image