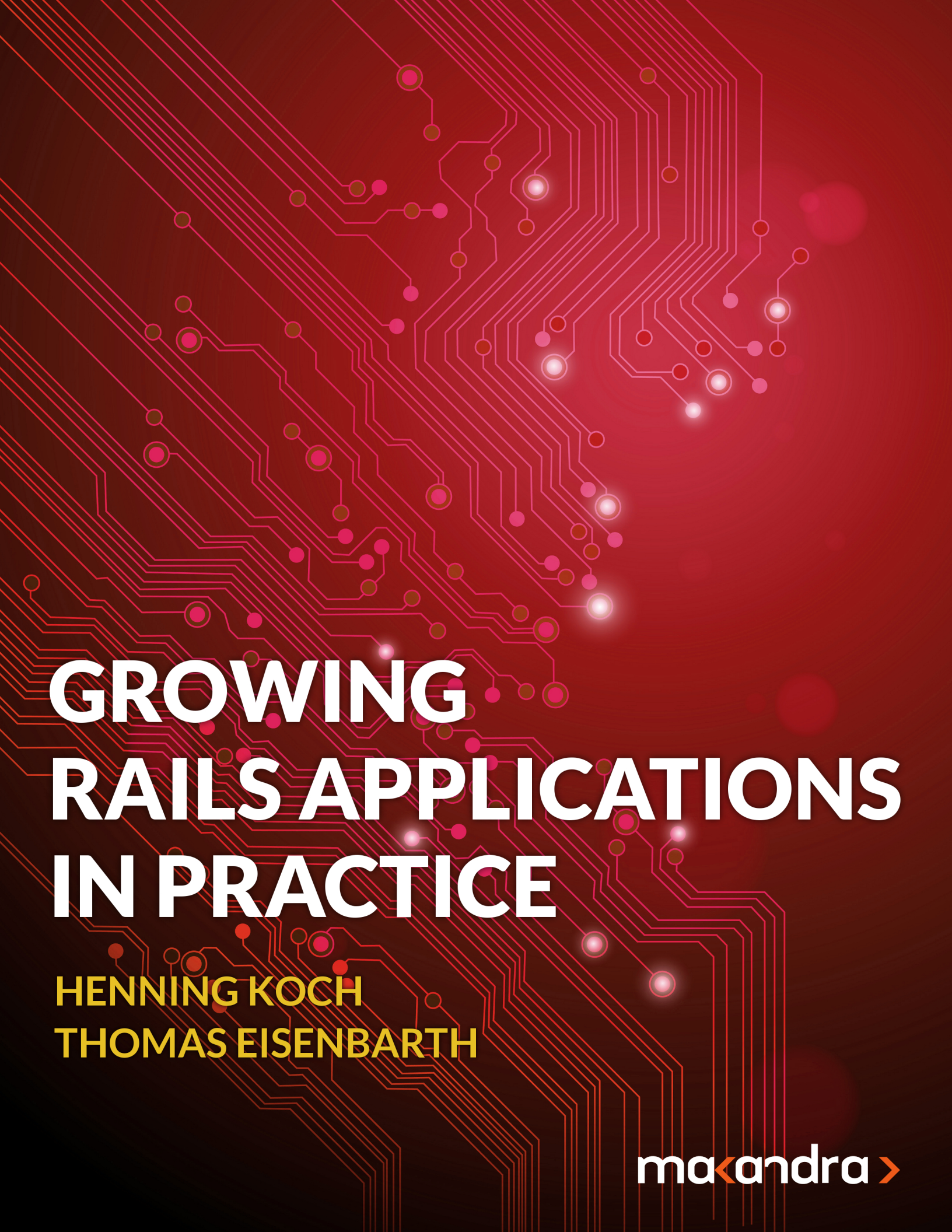
GROWING
RAILS APPLICATIONS
IN PRACTICE
HENNING KOCH
THOMAS EISENBARTH
manning

GROWING RAILS APPLICATIONS IN PRACTICE

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

By Henning Koch and Thomas Eisenbarth

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
```

```
  private

  def load_notes
    @notes ||= note_scope.to_a
  end

  def load_note
    @note ||= note_scope.find(params[:id])
  end

  def build_note
    @note ||= note_scope.build
    @note.attributes = note_params
  end

  def save_note
    if @note.save
      redirect_to @note
    end
  end

  def note_params
    note_params = params[:note]
    note_params ? note_params.permit(:title, :text, :published) : {}
  end

  def note_scope
    Note.all
  end

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. For example 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 implement authorization schemes[3] where access often depends on the current user. E.g. if we only wanted users to read and change their own notes, we could simply have `note_scope` return `Note.where(author_id: current_user.id)`. No other changes required.

- 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)

[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

- Parsing and white-listing parameters
- Loading or instantiating the model
- 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. For instance 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. For example 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