GROWING RAILS APPLICATIONS IN PRACTICE

HENNING KOCH
THOMAS EISENBARTH

makandra

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

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. For instance 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. For example 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. For instance 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`

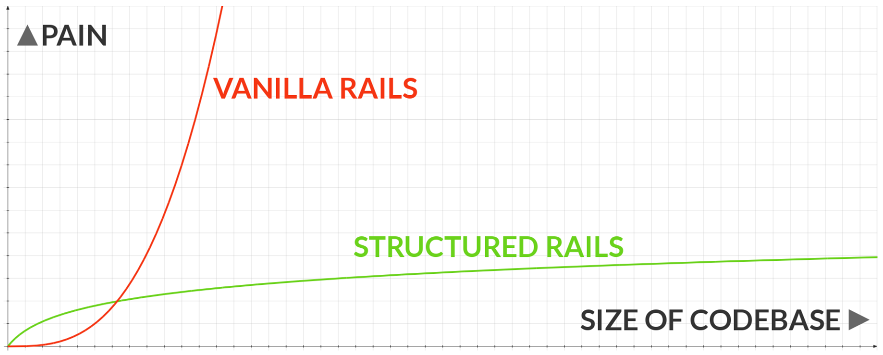
Remember when we told you that large applications are large? When you need to implement password recovery, and do not have a clear, single place to put the logic, *it will still find its way into your code*. It will spread itself across existing classes, usually making those classes harder to read and 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