Exploding Rails

Ryan Bigg

Exploding Rails

Ryan Bigg

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

This version was published on 2018-06-19

Contents

Introduction

Thank you for reading Exploding Rails.

If you find any misteaks while reading this book, please email them to me@ryanbigg.com.

If you have a problem to do with your code, then please put the code on GitHub and link me to it in the email so I can clone it and attempt to reproduce the problem myself. If you don't understand something, then it's more likely that I'm the idiot and rushed it when I wrote it. Let me know!

This far into the book and there's already one error. You should expect that it is not alone. It has friends and their ways are devious. They are coming after your perception of reality. **Beware**.

In similar fashion to my Toy Robot Walkthrough[1], this book started out as a blog post that I was going to put up at ryanbigg.com[2]. I knew that this post was going to be very long the very first day I started writing it, because it was 5,000 words by the end of the day and I still had a lot to say on the topic of organising a Rails application in a "better" way. It's now about 15,000 words. So I think it deserves to be a little book on Leanpub, rather than just a blog on my website.

The idea for this blog-post-turned-book came out of the Culture Amp[3] Junior Engineering Program that I am running. One of the junior developers that I mentor – Seb Pearce – came up with the idea of building an expense tracker

[1] https://leanpub.com/toyrobot
[2] http://ryanbigg.com
[3] https://cultureamp.com

application. So I thought, OK, this is going to be a run-of-the-mill Rails application with a `Transaction` model and so on. Then he told me that he wanted to build the application in Ruby, *without* Rails.

Well then.

So we went about doing that, sticking to plain Ruby code as much as possible. Eventually, we introduced some gems like `rom-rb`, `rom-sql`, `dry-validation` and `dry-transaction`. I had only really toyed around with these gems previously, and had never built anything "serious" with them, and so when pairing with Seb it gave us both an opportunity to explore using these gems in a semi-"serious" application.

And it was *bloody brilliant*.

The code for validating a transaction's attributes lived over *here*. The code for talking with the database lived over *here*. The "models" of the application were slimmed right down to classes that knew only what attributes and business logic `Transaction` objects should have.

It reminded me a lot of how I felt when I saw Rails for the first time: everything had a neat orderly *place*. There were clean lines of responsibility between each part of the application. The model was no longer a one-stop-shop for validation, persistence and business logic. The model stopped feeling like that moving box labelled "miscellaneous" that you dump all your trinkets into during a move. The model felt like it represented the structure of data for a particular *thing* in the application, and that was beautiful. The model only had a single responsibility.

(I should also mention that Phoenix[4] applications are structured somewhat similarly, but since this is a Ruby book that is all I will be saying about Elixir, for now.)

[4] https://phoenixframework.org

Seb and I have just recently gotten to the point of integrating his plain Ruby code with a Ruby web framework, with the intention of providing an API that a React[5] frontend can read from. After a little bit of deliberation over what non-Rails framework he could use, Seb settled on Sinatra. It's a fine choice that works well for Seb's application.

Meanwhile, I wanted to play around with `rom-rb` and `dry-rb` in a Rails application of my own making. My first goal was that I wanted to build a Rails application *without* Active Record. The second goal was that if the logic in the controller got too messy, I would have to abstract that out to a "service object" with `dry-transaction`. You'll see how I've accomplished both of those goals in this guide.

I found out that there was a gem called `rom-rails` and I started experimenting. The notes that I took eventually coalesced into this guide.

This book is best suited for *experienced* Rails developers. I'll assume a lot of knowledge around things like the normal structure of Rails applications, feature testing, what at least the concept of a "service object" is, and so on.

If you're new to Ruby, you will probably understand some of the terms I'm using here, but to get the most out of this guide you should have some Rails experience under your belt.

Ideally, you should've encountered places where using vanilla Rails has hurt you, your friends and possibly even your family members. But that's not necessary because there's a few examples peppered throughout this guide that will give you a good inkling. Rails is good, but it is not without its flaws.

[5] https://reactjs.org

This book is not supposed to be "The Bible of How to Integrate ROM into your Rails application". It's merely a short guide covering how one eager developer built the beginnings of a little Rails application using the `rom-rb` and `dry-rb` gems.

Acknowledgements

I'd like to thank Tiya Belayneh, Andy Holland, Tim Riley, Chris Flipse, and Piotr Solnica for their feedback on early editions of this book. It has been invaluable to have such dedicated readers reviewing this book. I would also like to thank Rob Jacoby and Seb Pearce (again) for inspiring me to write the first chapter of the book.

Thank you to Francois Beausoleil, Sean Liu, Bruno Bonamin, Brian Buchalter, Rocio Diaz-Meco, Tristan Penman, Rob Howard, and Michael Kohl for reporting errata in this book.

There's probably more errata to report, so if you find any, please email me@ryanbigg.com with what you find and you too can go on this list of outstanding people.

Preface

When Rails came out, it was revolutionary. There was an order to everything.

Code for your business logic or code that talks to the database *obviously* belongs in the model.

Code that presents data in either HTML or JSON formats *obviously* belongs in the view.

Any special (or complex) view logic goes into helpers.

The thing that ties all of this together is *obviously* the controller.

It was (and still is) neat and orderly. Getting started with a Rails application is incredibly easy thanks to everything having a pre-assigned home.

The Rails Way™ enforces these conventions and suggests that this is the One True Way™ to organise a Rails application. This Rails Way™ suggests that, despite there being over a decade since Rails was crafted, that there still is no better way to organise an application than the MVC pattern that Rails originally came with.

While I agree that this way is still extremely simple and great for *getting started* within a Rails application, I do not agree that this is the best way to organise a Rails application in 2018 with long-term maintenance in mind.

As a friend of mine, Bo Jeanes[6] put it neatly once:

> Code is written for the first time only once.
>
> Then there is anywhere between 0 and infinite days of having to change that code, understand that code, move that code, delete that code, document that code, etc. Rails makes it easy to write that code and to do some of those things early on, but often harder to do all the those things on an ongoing basis.
>
> We benefit by being patient in that first period and maybe trading off some of that efficiency for a clarity and momentum for the *life* of the project.

A decade of Ruby development has produced some great alternatives to Rails' MVC directory structure that are definitely worthwhile to consider.

In this guide, I want to show an *alternative* viewpoint on how a Rails application should be organised. It is not to say that my suggestion is The Rails Way™ 2.0 and everyone would be silly to stick with The Original Rails Way™ and Rails 6 should ultimately adopt this directory structure or face a certain doom. It's merely an *alternative* viewpoint that I think will lead to applications that are easier to work with over the longer-term. Think of it as a suggestion more than doomsaying, please.

To expand on this quote from Bo, here's my take on where Rails falls down.

Where Rails falls down

The Original Rails Way™ falls down in at least two major areas in my opinion: models and controllers.

[6] https://twitter.com/bjeanes

Due to the lack of choice in a Rails application, most of a Rails application's code usually ends up in the model. Where does business logic go? The model. Where do validations go? The model. Where do you define queries for working with the database? The model. How do you persist a model instance back to the database? The model takes care of that. The end result after even just a few months can be unsightly. All this logic is muddled up as the model is the only One True Place to put this sort of code. Rails provides no other sensible choices; it *must* go in the model.

The controllers aren't any better. The controller's actions talk to the model, asking the model to create, read, update or delete records in a database. And then this controller code might do more: send emails, enqueue background jobs, make requests to external services. There is no pre-determined, widely agreed-upon location for this logic; the controller is the place. A controller action can often have request logic, business logic, external API calls and response logic all tied up in the one method.

Testing all these intertwining parts individually is hard work. To make sure that it all works together, you often have to write many feature and/or request tests to test the different ways that the controller action is called and utilized. The logic of the controller's actions – those calls out to the model – get intimately accquainted with the logic for handling the request and response for that action. The lines between the incoming request, the business logic and the outgoing response become blurred. The controller's responsibilities are complex because there is no other sensible place for this code to go.

If the controller action is even mildly complex, then one solution is to just add more private methods to the controller, abstracting out the complexities to these methods. But then you have a jumble of private methods at the bottom of the controller, with no clear indications of what methods are used in what actions. It can be quite a mess. Or you'll just go ahead and chuck a method into

`ApplicationController` because more than one controller uses that method.

In this guide, I want to show an alternative viewpoint on how you can structure code that talks to a database, as well as an alternative viewpoint on code that would *usually* belong in a controller. This code leads to a clearer separation of concerns between the many different responsibilities of a Rails application. Code doesn't just have to go into controllers, models, views and helpers. We can break the code's responsibilities better than that. Code doesn't have to live in a very small handful of locations.

The Rails Way™ doesn't have to be the *only* way. In recent years, there's been a groundswell of support behind projects like rom-rb[7], and the dry-rb[8] suite of gems. These are the alternatives I was speaking of earlier.

These projects offer ways to make your application's code leaner, cleaner and simpler to understand. In this guide, I'll demonstrate how you can use them to clean up the code that you might write in a Rails application. Your models will be lean and clean. Your controllers will be lean and clean. And the things we use to replace this complexity will also be lean and clean.

But first, I want to go more into depth about the problems with models that use the Active Record gem.

The problems with Active Record Models

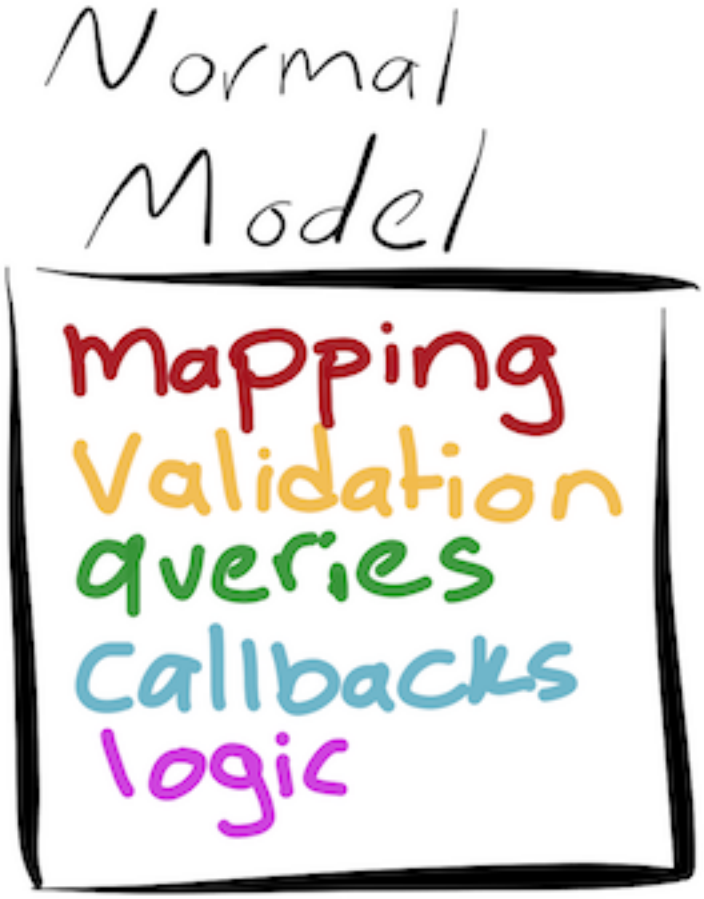
An Active Record model is responsible for *at least* the following things:

- Mapping database rows to Ruby objects
- Containing validation rules for those objects
- Managing the CRUD operations of those objects in the database (through inheritance from `ActiveRecord::Base`)

[7] http://rom-rb.org/
[8] http://dry-rb.org/

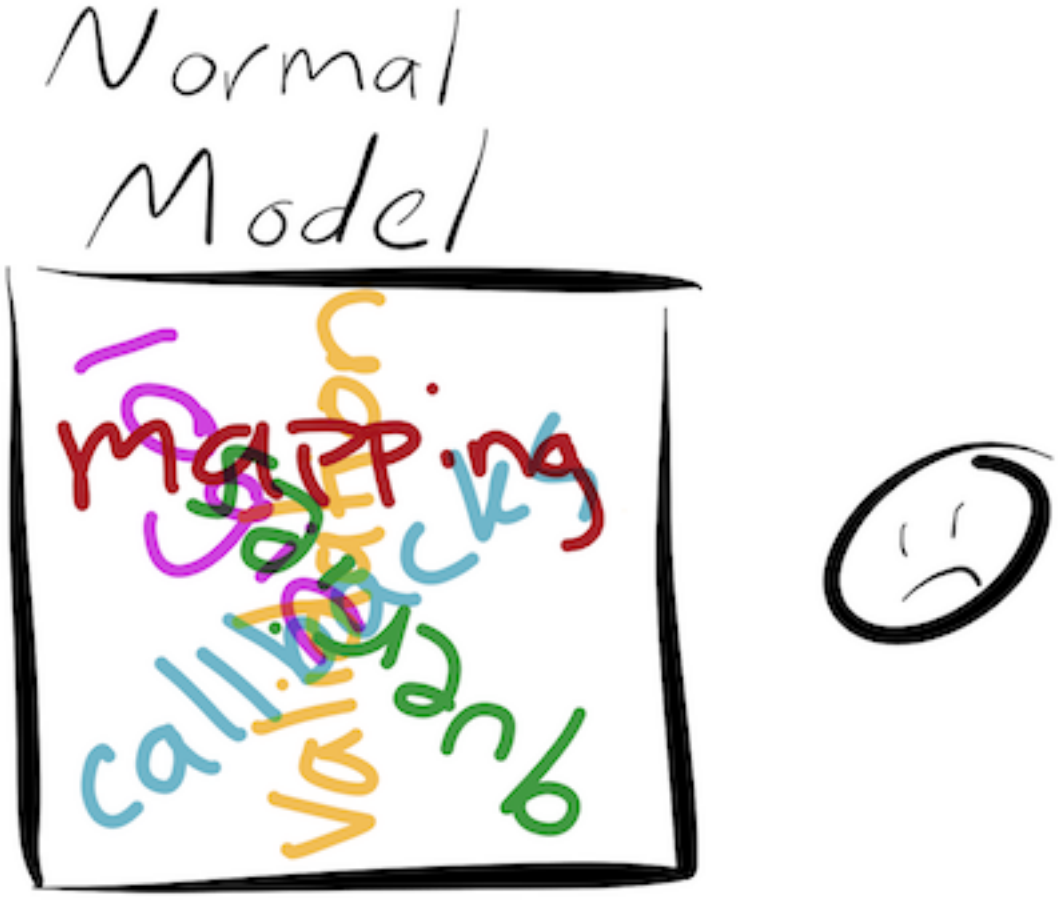
- Providing a place to put code to run before those CRUD operations (callbacks)
- Containing complicated database queries
- Containing business logic for your application

If you were to colour each responsibility of your model, it might look something like this:



Normal model

Or really, it might look like this:



Normal model

In traditional Rails models, all of this gets muddled together in the model, making it very hard to disentangle code that talks to the database and code that is working with plain-Ruby objects.

For instance, if you saw this code:

```ruby
class Project < ApplicationRecord
  has_many :tickets

  def contributors
    tickets.map(&:user).uniq
  end
end
```

You might know *instinctively* that this code is going to make a database call to the `tickets` association for the `Project` instance, and then for each of these `Ticket` objects it's going to call its `user` method, which will load a `User` record from the database.

Someone unfamiliar with Rails – like, say, a junior Ruby developer with very little prior Rails exposure – might think this is bog-standard Ruby code because that's *exactly* what it looks like. There's something called `tickets`, and you're calling a `map` method on it, so they might guess that `tickets` is an array. Then `uniq` further indicates that. But `tickets` is an association method, and so a database query is made to load all the associated tickets.

This kind of code is very, very easy to write in a Rails application because Rails applications are intentionally designed to be easy. "Look at all the things I'm *not* doing"[9] and "provide sharp knives"[10] and all that.

However, this code executes one query to load all the `tickets`, and then one query *per ticket* to fetch its users. If we called this method in the console, then the query output might look like this:

[9] http://youtu.be/Gzj723LkRJY
[10] http://rubyonrails.org/doctrine/#provide-sharp-knives

```
Project Load (0.2ms)  SELECT  "projects".* FROM "projects" ORDER BY "projects"."id" \
ASC LIMIT ?  [["LIMIT", 1]]
Ticket Load (0.1ms)  SELECT "tickets".* FROM "tickets" WHERE "tickets"."project_id" \
= ?  [["project_id", 1]]
User Load (0.1ms)  SELECT  "users".* FROM "users" WHERE "users"."id" = ? LIMIT ?  [[\
"id", 1], ["LIMIT", 1]]
User Load (0.1ms)  SELECT  "users".* FROM "users" WHERE "users"."id" = ? LIMIT ?  [[\
"id", 2], ["LIMIT", 1]]
User Load (0.1ms)  SELECT  "users".* FROM "users" WHERE "users"."id" = ? LIMIT ?  [[\
"id", 3], ["LIMIT", 1]]
User Load (0.1ms)  SELECT  "users".* FROM "users" WHERE "users"."id" = ? LIMIT ?  [[\
"id", 1], ["LIMIT", 1]]
User Load (0.1ms)  SELECT  "users".* FROM "users" WHERE "users"."id" = ? LIMIT ?  [[\
"id", 2], ["LIMIT", 1]]
User Load (0.1ms)  SELECT  "users".* FROM "users" WHERE "users"."id" = ? LIMIT ?  [[\
"id", 3], ["LIMIT", 1]]
```

This is a classic N+1 query, which Rails does not stop you from doing. It's a classic Active Record footgun. And this is all because Active Record makes it *much* too easy to call out to the database from within the model. This code for `Project#contributors` combines business logic intent ("find me all the contributors to this project") with database querying and it's *the* major problem with Active Record's design. And mongoid[11]'s too, since it also follows the Active Record pattern.

Database queries are cheap to make because Active Record makes it so darn easy. When looking at the performance of a large, in-production Rails application, the number one thing I come across is slow database queries caused by methods just like this. Programmers writing innocent looking Ruby code that triggers not-so-innocent database activity is something that I've had to fix too many times within a Rails application.

[11] https://rubygems.org/gems/mongoid

Active Record makes it way too easy to make calls to the database. Once these database calls are ingrained in the model like this and things start depending on those calls being made, it becomes hard to refactor this code to reduce those queries. Even tracking down where queries are being made can be difficult due to the natural implicitness that *some* method calls produce database queries.

Thankfully, there are tools like Skylight[12] and New Relic[13] that point directly at the "smoking guns" of performance hits in a Rails application. Tools like these are invaluable. It would be nice to not need them so much in the first place, however.

The intention here with the `contributors` method is very innocent: get all the users who have contributed to the project by iterating through all the tickets and finding their users. If we had a `Project` instance (with thousands of tickets[14]), running that contributors method would cause thousands of database queries to be executed against our database.

Of course, there is a way to make this all into two queries through Rails:

```
class Project < ApplicationRecord
  def contributors
    tickets.includes(:user).map(&:user).uniq
  end
end
```

This will load all the tickets *and* their users in two separate queries, rather than one for tickets and then one for each ticket's user, thanks to the *power of eager loading*. (Which you can read more about in the Active Record Querying guide[15].)

The queries look like this:

[12] https://skylight.io
[13] https://newrelic.com
[14] https://github.com/rails/rails
[15] http://guides.rubyonrails.org/active_record_querying.html#eager-loading-associations

```
Ticket Load (0.4ms)  SELECT "tickets".* FROM "tickets" WHERE "tickets"."project_id" \
= ?  [["project_id", 1]]
User Load (0.4ms)  SELECT "users".* FROM "users" WHERE "users"."id" IN (1, 5)
```

Active Record loads all the ticket objects that it needs to, and then it issues a query to find all the users that match the `user_id` values from all the tickets.

You can of course not load all the tickets at the start either, you could load only the 100 most recent tickets:

```
class Project < ApplicationRecord
  def contributors
    tickets.recent.includes(:user).map(&:user).uniq
  end
end

class Ticket < ApplicationRecord
  scope :recent, -> { limit(100) }
end
```

But I think this is still too much of a mish-mash of database querying and business logic. Where is the clear line between database querying and business logic in this method? It's hard to tell. This is because Active Record *allows* us to do this sort of super-easy querying; intertwining Active Record's tentacles with our business logic.

It should be possible to work with the business logic of your application without these database calls being made; and without the database at all. Being able to reach into the database from your business logic *should* be hard work. Your business logic should have everything it needs to work by that stage. A class containing only business logic and being passed some data should not need to know also about how that data is validated, any "callbacks" or how that data is persisted too. If a class knows about all of those things, it has too many responsibilities.

The Single Responsibility Principle says that a class or a module should only be responsible for one aspect of the application's behaviour. It should only have one reason to change. An Active Record model of any meaningful size has many different reasons to change. Maybe there's a validation that needs tweaking, or an association to be added. How about a scope, a class method or a plain old regular method, like the contributors one? All more reasons why changes could happen to the class.

An Active Record model flies in the face of the Single Responsibility Principle. I would go as far as to say this: Active Record leads you to writing code that is hard to maintain from the very first time you set foot in a Rails application. Just look at any sizeable Rails application. The models are usually the messiest part and I really believe Active Record – both the design pattern and the gem that implements that pattern – is the cause.

Having a well-defined boundary between different pieces of code makes it easier to work with each piece. Active Record does not encourage this. Validations and persistence should be their own separate responsibilities and separated into different classes, as should business logic. There should be a class that only has the responsibility to talk to the database. Clear lines between the responsibilities here makes it so much easier to work with this code.

It becomes easier then to say: this class works with only validations and this other class talks to the database. There's no muddying of the waters between the responsibilties of the classes. Each class has perhaps not *one* reason to change, but at least *fewer* reasons to change than Active Record classes.

In contrast with Active Record, the ROM (Ruby Object Mapper) library draws these crystal clear lines in a simple manner. The design patterns that ROM encourages are leaps and bounds better than Active Record. This guide will serve as an example of that.

We're going to use the rom-rb suite of gems – and some from the dry-rb

suite too – to interact with our database and to write better code for working with objects from a database. We'll keep our validation and persistence logic completely separate to our business logic. You will be amazed at the cleanliness once we're done.

Chapter 1 starts out with an empty directory. We'll fill out this directory with a few plain Ruby files, showing how it is possible to get started with ROM without using Rails at all. We'll also spend this time getting familiar with some of the concepts of ROM.

Chapter 2 will have us installing ROM into a new Rails application and configuring the first model: a model called `Project`. This chapter will give you a pretty good idea of how separated the code is within a project that uses ROM.

Chapter 3 will look at how we can connect the parts that we build in Chapter 1 to a Rails controller. You'll be surprised at how not-different this looks to a regular controller.

Chapter 4 will continue the work from Chapter 2, adding an `index` and a `show` action to that controller. We'll be using some more ROM methods in this chapter too.

Chapter 5 covers validations within this application, proving that the model isn't the only place validations can live. We'll also look at how we can present the validation messages back to the user once a validation fails.

Chapter 6 talks about service objects, discussing the pitfalls of common service object design and presenting a better alternative in what the `dry-transaction` gem provides.

Chapter 7 introduces the second and third models of the application: a `Ticket` model and a `User` model. It's in this chapter that we'll look at how we can make

our `Project#contributors` method still use the beautiful code we've seen before, but without making calls out to the database.

In the book's "epilogue", there is some short homework for you (yes, that means you) to do. You should do this homework to practice working with a ROM-powered application, just so you can experience how easy it is to use. Doing it yourself is much better than following the bouncing ball of a technical guide like this.

The final part of the book's "epilogue" discusses a radically different architecture for a Rails application, where Rails is the dumb host to an application's business logic.

So without further ado, let's get started using this ROM thing. You really will be amazed at the cleanliness of the code.

Getting started with ROM

> ### Need to see the code?
>
> The code for all chapters of this book is on GitHub here: https://github.com/radar/exploding-rails-examples.
>
> Chapter 1's code is in the `projects` directory.
>
> The remaining chapters are in the `ticketee` directory.

Even though this book has spent a large portion of time talking about Rails, the first chapter is only going to mention Rails in passing. What we'll do here is demonstrate that it is possible to use the ROM library in a completely isolated fashion, separate to a Rails application. We'll also spend some time getting acquainted with the concepts of ROM.

In particular, we'll find out about these ROM concepts:

- Relations: Classes that know about database queries
- Repositories: The "neat" layer between relations and your application
- Migrations: Alter your database, creating or dropping tables
- Models: Provide a class that is used to represent data from our database in Ruby

You'll know the last two from Rails. ROM's version of migrations aren't too different from Rails. The models are much lighter though, with the querying logic living in the relations and repositories instead of in the model. By separating

these concerns out into different classes it will make it easier to reason about each concern individually, as we'll see as we go through this chapter.

We'll spend this chapter building a small ROM-powered Ruby application that can create records in a PostgreSQL database table. It's not until the next chapter that we'll integrate ROM and Rails together.

Introducing ROM

ROM[16] (Ruby Object Mapper) is a suite of gems designed to make it easy to work with databases, but not as easy as Active Record. That is an intentional design decision. ROM favors explicitness over implicitness, and in my opinion that leads to better code design.

Rather than having a single class that encapsulates all of the logic of working with databases – like how a typical model in a Rails application behaves – ROM chooses to split this logic over several separate classes:

- Repositories - Manages the interaction between your application and relations. Also handles CRUD actions.
- Relations - A place to put database query logic
- Models - A class that is used to represent data from our database in Plain Ol' Ruby.

Validations in these applications are often split off into their own classes, called "schemas", with the help of the dry-validation[17] gem. Which we'll see used a little later on in this book.

By breaking things apart like this, it makes it easier to test and use each part individually. The business logic in your Plain Ol' Ruby Class isn't interspersed

[16] http://rom-rb.org/
[17] https://github.com/dry-rb/dry-validation

with Active Record methods that trigger database calls. It's not possible to make database queries from this object. Instead, it works as an individual unit in your application. It's the Relations classes that will make those database queries instead.

We could keep talking about the theory or we could actually write some code.

Let there be ROM

We'll now setup ROM within a brand-new Ruby application.

A PostgreSQL server is required

This section will require you to have a PostgreSQL server that you can connect to. If you don't have one now, you can install it:

- Mac: https://postgresapp.com/
- Ubuntu: https://www.postgresql.org/download/linux/ubuntu
- Windows: https://www.postgresql.org/download/windows

We're going to start with nothing and end up with something. That nothing is going to be an empty directory, which we'll call `projects`:

```
mkdir projects
```

Inside this directory, we can create a `Gemfile` which will be used to specify the gems this application uses:

Gemfile

```
source 'https://rubygems.org'

gem 'rom', '4.2.1'
gem 'rom-sql', '2.5.0'
gem 'pg'
gem 'rake'
```

In this file, we've added in four gems:

- `rom`: The core functionality of ROM
- `rom-sql`: The SQL functionality of ROM
- `pg`: A gem that can talk to PostgreSQL databases
- `rake` A gem for running Ruby tasks

The functionality of talking to a SQL database has been split out from the `rom` gem, as `rom` can work with more than just SQL databases – it can also work with document databases, like MongoDB. In case, we're going to be interacting with a PostgreSQL database, and so we want to use `rom-sql`.

The `rake` gem has been added here too, as we will need it later on when we run Rake tasks within this project.

The Sequel gem

One important thing to note here is that the `rom-sql` gem is built on top of another excellent gem, called `sequel`.

The `sequel` gem is pretty extensive, so I won't cover it here. What I will say is that you should check out the Sequel documentation, just to see what it's capable of.

You can find that documentation here: https://sequel.jeremyevans.net/.

Let's install these gems with this command now:

```
bundle install
```

Creating the ROM container

Installed gems don't do much without code to match, and so we need to write some code now to use these gems. Let's create a new file to do this:

lib/projects.rb

```
require 'rom'
require 'rom-sql'

module Projects
  DB = ROM.container(:sql, 'postgres://localhost/projects_dev')
end
```

In this code, we're using all three of the gems that we've just installed. The `ROM.container` line comes from the `rom-core` gem, which is a dependency of `rom`.

This line configures a *container* that encapsulates all the configuration for how our application talks to a database. All it knows so far is that we're talking to a SQL database, and that we can find that database at the URL of `postgres://localhost/projects_dev`.

The `postgres://` part in here tells Sequel, that underlying gem of `rom-sql`, to connect to a PostgreSQL database. The Sequel gem will use the `pg` gem to initiate and manage that connection.

Building the database

This database that we're connecting to doesn't exist yet, but we can create it very easily by using the `createdb` command line tool:

```
createdb projects_dev
```

A database without tables doesn't really do very much. It's good practice to create tables within a database by using *migrations*; Ruby files that allow us to build up our database tables (or "schema") gradually.

ROM has the concept of migrations too, but we need to do a bit of setup first before we can create our first migration. That setup will involve creating a `Rakefile` within our project, that will then provide us with some tasks to create and run these migrations.

Let's create that `Rakefile` now:

Rakefile

```
require 'rom/sql/rake_task'

require_relative 'lib/projects'

namespace :db do
  task :setup do
    ROM::SQL::RakeSupport.env = Projects::DB
  end
end
```

This `Rakefile` first requires `rom/sql/rake_task`, which adds some Rake tasks from ROM to our application. At this point we can run `rake -T` to see a list of these tasks:

```
rake db:clean
  # Perform migration down (erase all data)
rake db:create_migration[name,version]
  # Create a migration (parameters: NAME, VERSION)
rake db:migrate[version]
  # Migrate the database (options [version_number])]
rake db:reset
  # Perform migration reset (full erase and migration up)
```

These four tasks come from that `rom/sql/rake_task` require.

Inside the `Rakefile`, we define a new task called `db:setup`, which these Rake tasks will call before acting. This code points ROM to our `Projects::DB` ROM container object, and when we run these Rake tasks, they will run on our configured database.

The first one of these Rake tasks that we'll run is the one to create a migration. We'll use this migration to create the first table within our database, a table called `projects`. Let's create that migration with this command now:

```
rake db:create_migration[create_projects]
```

As the output shows, this has created a new migration file:

```
<= migration file created db/migrate/[timestamp]_create_projects.rb
```

This migration file doesn't contain very much:

db/migrate/[timestamp]_create_projects.rb

```
ROM::SQL.migration do
  change do
  end
end
```

Unlike in Rails, it's not possible to get `rom-sql` to fill out a migration for us. It's going to be up to us to fill it out. Migrations with `rom-sql` are based on Sequel's migrations, which you can learn about here[18]. In this migration, we're going to create a `projects` table, which will just have two fields, an `id` and a `name` field. Let's change this migration now to make it create this table:

db/migrate/[timestamp]_create_projects.rb

```
ROM::SQL.migration do
  change do
    create_table :projects do
      primary_key :id
      column :name, :text
    end
  end
end
```

The syntax here is not very different from the Rails syntax. The big difference is that in a `rom-sql` / Sequel migration, we must use the `primary_key` method to tell it that we want a primary key to exist, and what the name of that primary key is. In a Rails migration, the primary key is assumed by default to be wanted and called "id".

The `column :name, :text` line tells Sequel that we want to create a `name` column and it is of the type `:text`.

We can run this migration with this command:

[18] https://github.com/jeremyevans/sequel/blob/master/doc/migration.rdoc

```
rake db:migrate
```

This command will not output much information:

```
<= db:migrate executed
```

By default, ROM is not configured to log anything. So how are we supposed to know if this migration was actually successful and did create our table? Well, we could look at the database itself and see if our table was there with the right fields… But before we do that, let's add some logging so that next time we run a migration we'll see more information.

We'll open up `lib/projects.rb` and change the container configuration to this:

lib/projects.rb

```
1 DB = ROM.container(:sql, 'postgres://localhost/projects_dev') do |config|
2   config.gateways[:default].use_logger(Logger.new($stdout))
3 end
```

This tells ROM that we want to output any log messages to `$stdout`. This will make it so that next time we run a migration, we'll see some more information about that.

Rather than generating an entirely new migration, we can re-run the first migration.To do this, we can run this command:

```
rake db:reset
```

This time, we'll see some output from this command. We'll see that the `projects` table is dropped:

```
I, [<timestamp>]  INFO -- : (0.037019s) DROP TABLE "projects"
```

And then that the table is created:

```
I, [<timestamp>]  INFO -- : (0.044157s) CREATE TABLE "projects" (
  "id" integer GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
  "name" text
)
```

Great! Now we're able to see exactly what ROM is doing. This logging will apply to more than just migrations, as we'll see in a moment.

We have now built our database and added a table to it. In order to talk to it, we're going to need to create a *relation* class.

Interacting through a relation

In order to run queries against this new `projects` table, we're going to need to create a *relation* class. A relation class manages the relationship between Ruby and the table, allowing us to run queries on the table and get data back in Ruby form to work with it.

Let's create this new class now:

lib/projects/relations/projects.rb

```ruby
module Projects
  module Relations
    class Projects < ROM::Relation[:sql]
      schema(:projects, infer: true)
    end
  end
end
```

This class is told to inherit from `ROM::Relation[:sql]`, which adds behavior to this relation class for working with SQL tables. The `schema` line inside this class tells the relation which table in our database we're going to work with, and the `infer` option here tells the relation that we want to automatically infer the shape of the schema from what's in the database.

The alternative to inferring the schema here is to define it by hand. Here's an example of what that would look like:

```
schema(:projects) do
  attribute :id, Types::Serial
  attribute :name, Types::String
end
```

This could be useful if you wanted to hide certain fields from the relation class, but since we don't have any fields we want to hide, we're going to infer them.

To use this relation, we first must do a couple of things. We're going to need to require the file in `lib/projects.rb` and to tell our application's ROM container about it:

lib/projects.rb

```
1  require 'rom'
2  require 'rom-sql'
3
4  require_relative 'projects/relations/projects'
5
6  module Projects
7    DB = ROM.container(:sql, 'postgres://localhost/projects_dev') do |config|
8      config.gateways[:default].use_logger(Logger.new($stdout))
9
10     config.register_relation(Projects::Relations::Projects)
11   end
12 end
```

We need to require the `projects/relations/projects` file so that the `Projects::Relations::Projec` class will be loaded. We then need to register this relation with our application's ROM container so that we are able to use our relation to access our database's tables.

Having to require and register each relation becomes cumbersome after a while, and so ROM provides us a shorter way to do this:

lib/projects.rb

```
require 'rom'
require 'rom-sql'

module Projects
  DB = ROM.container(:sql, 'postgres://localhost/projects_dev') do |config|
    config.gateways[:default].use_logger(Logger.new($stdout))

    config.auto_registration('./lib/projects')
  end
end
```

This will automatically register all relations under the `lib/projects` directory of our project. Now we will not need to call `register_relation` at all!

The second thing that we need to do in order to use our relation is to create a file that starts an IRB session, so that we're able to easily play around with this new class.

Let's create this new file at `bin/console`:

bin/console

```
#!/usr/bin/env ruby

$:.unshift(File.expand_path('../lib', __dir__))

require 'projects'

require 'irb'
IRB.start
```

In this file, we start it with a shebang[19] that tells the program to execute `ruby`. On the second line, we add the `lib` directory to the load path. This means that we're then able to run `require 'projects'` on the third line to require the file that defines everything for our application. On the fourth and fifth lines, we require and start IRB.

Before we can run this program, we have to first make it executable:

```
chmod +x bin/console
```

We can now run this console and use it to interact with our relation. To start it, run this:

```
bin/console
```

The output will include a few confusing looking SQL queries. These are the relation inferring the shape of the `projects` table. You can ignore these. The last of these queries is a check from the Sequel gem, which is checking that the `projects` table actually exists.

Once we've run this command, we should have an IRB prompt:

[19] https://en.wikipedia.org/wiki/Shebang_(Unix)

```
irb(main):001:0>
```

Let's start working with our relation. We'll need to retrieve it first, which we can do with this code:

```
projects = Projects::DB.relations[:projects]
```

This code uses our application's ROM container to fetch an instance of the `Projects::Relations::Project` class. This was made possible because we called `register_relation` inside of the container's configuration. The `register_relation` command will register the relation with the name of the schema that the relation uses, which is `:projects` in this case.

Now that we have the relation, we can use it to insert new records into our `projects` table. For this, we have the `insert` command at our disposal:

```
projects.insert(name: "Ticketee")
# => 1
```

This will return the number `1` because this is the ID (and primary key) of the record that we've just inserted into this table. If we insert another record, we'll see `2`. And if we insert one after that, `3`. And so on.

If we want to find a record by this ID, we can use the `by_pk` method, combined with the `one` method:

```
projects.by_pk(1).one
# => {:id=>1, :name=>"Ticketee"}
```

The `by_pk` method acts as a scope, restricting the query to only records that have that particular primary key. The `one` method makes this query only return one record.

What we get back from this call is a Hash representing the data within that record.

If we wanted to find multiple records matching particular conditions, we could use `where` just like in an Active Record model:

```
projects.where(name: "Ticketee").to_a
```

The `to_a` on the end here will run the query and return us the results:

```
[{:id=>1, :name=>"Ticketee"}]
```

Relations also provide methods to delete and update records:

```
projects.by_pk(1).update(name: "Ticketee v2")
# => 1
projects.by_pk(1).delete
# => 1
```

Let's add that record back that we just deleted:

```
projects.insert(name: "Ticketee v2")
# => 2
```

So it appears that relations allow us to manipulate the data within our databases just fine. But there are some pitfalls.

The first pitfall is: if we're going to call methods like `where` on our relation and then pass that data back up to something like a controller, there's nothing that stops that controller from calling further methods – like more `where` queries – on that. This would mean that controllers would be capable of more than collecting data. This sort of querying should be kept as isolated as possible, to as few classes as possible. This will make it easier to manage these database calls later on when our application develops into something larger.

The second pitfall is: when we get the data back from the relation, we're only seeing pure Ruby hashes. But what if we wanted to add some special behaviour

on these records? If we had a record from a table called `people` and that record had both a `first_name` and a `last_name` field we wanted to combine... Well, we don't have a place for that logic at the moment. We need to create one of those.

To avoid these pitfalls, we need to create and use a *repository*.

Repository in the middle

A *repository* in a ROM application provides a clear separation between `Relation` classes and our application. A repository prevents the two pitfalls that were mentioned at the end of the last section. Think of them like a middleman, or a gatekeeper between your application and your database.

Let's go ahead and create a repository for our `projects` now by creating this new file and directory:

lib/projects/repositories/projects.rb

```
1 module Projects
2   module Repositories
3     class Projects < ROM::Repository[:projects]
4
5     end
6   end
7 end
```

This repository inherits from `ROM::Repository[:projects]`, which links this repository to an instance of the `Projects::Relations::Projects` class, setting that relation to be the "root" relation for this repository. A repository can work with more than one relation, but it only has a single relation as its "root". We'll cover this in more detail later on in the book.

We need to require this file in `lib/projects.rb`, because repositories aren't included in the auto registration features for ROM.

lib/projects.rb

```
require 'rom'
require 'rom-sql'

require_relative 'projects/repositories/projects'

module Projects
  DB = ROM.container(:sql, 'postgres://localhost/projects_dev') do |config|
    config.gateways[:default].use_logger(Logger.new($stdout))

    config.auto_registration('./lib/projects')
  end
end
```

Let's look at how to use this repository in `bin/console`. We'll have to restart `bin/console` first, but once that's done, we can initialize our repository:

```
repo = Projects::Repositories::Projects.new(Projects::DB)
```

Then we can call out to the `projects` relation from this repository:

```
repo.projects.to_a
=> [#<ROM::Struct::Project id=2 name="Ticketee v2">]
```

This line will fetch all of the records in the `projects` table. So far there's only one of those and so fetching them will be very quick. This chain of methods has returned not an array of Hashes, but rather an array of instances of an auto-generated class called `ROM::Struct::Project`.

By accessing the relation through the repository, we're now getting instances of a `ROM::Struct` subclass back, rather than Hashes. This is one of the advantages of using a repository over using a relation.

This means we will be able to access the field values with method calls (like `record.id`) instead of accessing them through their Hash keys (like `record[:id]`).

This works towards solving the second pitfall, but doesn't quite get there. What we want are subclasses of our own application's struct classe instead, which will allow us to define special behaviour on records.

We can look at fixing this by using a feature of ROM repositories called `struct_namespace`:

lib/projects/repositories/projects.rb

```
module Projects
  module Repositories
    class Projects < ROM::Repository[:projects]
      struct_namespace ::Projects
    end
  end
end
```

The `struct_namespace` method call here will tell the repository that when this repository gets back data from the relation, the class for this data lives under the `Projects` namespace. We're using the double colon prefix here for the class so that we use the top-level `Projects` constant, and not the `Projects::Repositories::Projects` class.

If we restart `bin/console` again and re-initialize our repository, we'll see this in action:

```
repo = Projects::Repositories::Projects.new(Projects::DB)
repo.projects.to_a
# => [#<Projects::Project id=2 name="Ticketee v2">]
```

We're now seeing instances of a `Projects::Project` class being returned. But hang on a moment, that class doesn't exist. So what's happening here? Well, ROM is automatically creating that class for us as a way of helping out.

If we don't want this behaviour exactly and instead wanted our own class, then we can define our own class. Let's go ahead and do that now:

lib/projects/project.rb

```
module Projects
  class Project < ROM::Struct
    def name_reversed
      name.reverse
    end
  end
end
```

This class defines a single method called `name_reversed` which reverses the name of the project. This is what we'll use to demonstrate that the repository is really returning `Projects::Project` objects. This `name_reversed` method is only present in this class, and so it's a good test.

Now that we've defined this class, we'll have to require it in `lib/projects.rb` to make sure it is loaded:

lib/projects.rb

```
require_relative 'projects/project'
```

This class will now be the one that is used by our repository to initialize new objects that contain the data from our database. If we restart the console one more time, then we can try out this new method.

```
repo = Projects::Repositories::Projects.new(Projects::DB)
project = repo.projects.to_a.first
project.name_reversed
# => "2v eetekciT"
```

Excellent! This is now all working. We're now able to have a model class to define behaviour at the Ruby level for these database records.

But we haven't fixed our first pitfall yet; just the second. The first one was that database calls could still be done by whatever calls the repository, as we're able to demonstrate easily in our console:

```
repo = Projects::Repositories::Projects.new(Projects::DB)
projects = repo.projects.where(name: "Ticketee v2")
projects.where(id: 2)
```

We're able to call `where` on the result of this first `where` query, but that shouldn't be possible as it allows the responsibility of database querying to leak into the application.

Let's look at how we can use the repository to isolate the application from the ability to directly perform database queries.

Isolating the application and the database

The repository is supposed to act as the "middleman" between the application and the relation. We've been using the repository to jump straight to the relation and then calling methods on that relation. This is considered bad practice, because the relation and the application (or the console, in this case) are not isolated from each other.

The application should be asking the repository to perform queries on the relation. That way then, the application and relation are isolated from each other, which makes working with our application's code much easier. The repository's job is to piece together what the relation provides to do those queries. The repository then returns *materialized* data back to the application. By "materialized", I mean data that has been loaded from the database and the application would be unable to perform any more database queries.

We've been using the repository in our application's console in a way where the data is not materialized – until we call `to_a` or `one` on it. Here's some examples:

```
repo = Projects::Repositories::Projects.new(Projects::DB)

# Unmaterialized
projects = repo.projects.where(name: "Ticketee v2")
# Materialized
projects = repo.projects.where(name: "Ticketee v2").to_a

# Unmaterialized
project = repo.projects.by_pk(2)
# Materialized
project = repo.projects.by_pk(2).one
```

In the "Unmaterialized" examples above, it's possible to further chain more relation methods on the end. The data hasn't yet been fetched from the database. In the "Materialized" versions, the final method call triggers the fetching of that data from the database and prevents the chaining of any further queries. We've "materialized" the data from the database.

With that in mind, let's look at how we can make our repository perform database queries to perform a couple of actions in our application:

- Finding all projects
- Finding a project by its ID
- Creating a new project
- Updating or deleting an existing projects

We'll see by the end of this section how a repository provides a nice clear layer between our application and the relation class that talks to the database.

Finding all projects

We've previously found all projects by using our repository in `bin/console` like this:

```
repo = Projects::Repositories::Projects.new(Projects::DB)

repo.projects.to_a
```

But this requires a chain of methods that we would probably end up repeating wherever we wanted to refer to all the projects. We can make this much easier by moving this code into our repository, like this:

lib/projects/repositories/projects.rb

```
module Projects
  module Repositories
    class Projects < ROM::Repository[:projects]
      struct_namespace ::Projects

      def all
        projects.to_a
      end
    end
  end
end
```

This method means that we can call `repo.all` instead of `repo.projects.to_a` to get a list of all the projects. It also means that the repository now contains the logic for reaching out to the relation. Our application, when it calls the repository, has no knowledge of the internal API of the relation. It only knows about what the repository provides.[20]

Finding a project by its ID

Along the same lines as the `all` method, we can also move the logic to find a project by its primary key into this repository. Previously, we had to call this

[20] This is a good application of the Law of Demeter. Or, at least I think it is. The application is made to know only about the `all` method of the repository. The relation could implement this in an *incredibly* complex way, but the application knows nothing of that complexity. It only knows *all*.

code to accomplish that:

```
repo.projects.by_pk(2).one
```

We can move this code into the repository as well:

lib/projects/repositories/projects.rb

```
module Projects
  module Repositories
    class Projects < ROM::Repository[:projects]
      struct_namespace ::Projects

      def all
        projects.to_a
      end

      def by_id(id)
        projects.by_pk(id).one
      end
    end
  end
end
```

This makes it possible for us to call `repo.by_id(id)`, instead of reaching through the repository to access methods on the relation. In a Rails application the method on the model that performs this function is called `find`, but I prefer to be more specific here and name the method by what it does: finds a record *by its ID*.

Creating a new project

Our application will also use the repository to create new projects. We've previously done this in the console by calling the `insert` method on the relation. We could add a method to the repository to call that, like this:

lib/projects/repositories/projects.rb

```
module Projects
  module Repositories
    class Projects < ROM::Repository[:projects]
      struct_namespace ::Projects

      ...

      def create(attrs)
        projects.insert(attrs)
      end
    end
  end
end
```

But there is a better way than this. Because it is such a common action to create things within a repository, the `rom-sql` gem comes with a little helper that will add a method that lets us create records in our database. To use this helper, we need to put this line at the top of the class:

lib/projects/repositories/projects.rb

```
module Projects
  module Repositories
    class Projects < ROM::Repository[:projects]
      struct_namespace ::Projects
      commands :create

      ...
    end
  end
end
```

This `commands` method is called such, as it allows us to add multiple *commands*,

or "actions" that we can take on the relation. In this case, we're only wanting the command to create for now.

By calling this class method, we will now have access to a method called `create` on our repository. Let's restart `bin/console` and try it out:

```
repo = Projects::Repositories::Projects.new(Projects::DB)
repo.create(name: "Ticketee v3")
# => #<Projects::Project id=3 name="Ticketee v3">
```

When we call `create`, we can pass it a Hash. This method will then take the Hash and pass it along to that `insert` method on relations that we saw earlier. This will ultimately create a record in the database. What we see returned from this call is the Ruby representation of this record; an instance of the `Projects::Project` class.

It's quite nice that `rom-sql` provides us this little helper. The `rom-sql` gem also provides helpers for updating and deleting too, in the form of the `update` and `delete` commands too.

Updating and deleting projects

projects/repositories/projects.rb

```
1  module Projects
2    module Repositories
3      class Projects < ROM::Repository[:projects]
4        struct_namespace ::Projects
5        commands :create, update: :by_pk, delete: :by_pk
6
7        ...
8      end
9    end
10 end
```

These commands work by finding a record, and then updating or deleting it. They find this record by using the `by_pk` relation method, but we could use any other method here if we wished. That method would just need to be defined on the relation. As an example, if we wanted to use `by_permalink` instead, we would have to define that method on the relation:

projects/relations/projects.rb

```
def by_permalink(permalink)
  where(permalink: permalink)
end
```

And then we would be able to use it in our commands over in the repository:

projects/repositories/projects.rb

```
module Projects
  module Repositories
    class Projects < ROM::Repository[:projects]
      struct_namespace ::Projects
      commands :create, update: :by_permalink, delete: :by_permalink

      ...
    end
  end
end
```

Let's assume that we're sticking with the `by_pk` variant for now.

To update a record, we must call the `update` method on the repository, like this:

```
repo = Projects::Repositories::Projects.new(Projects::DB)
repo.update(2, {name: "Ticketee v2.1"})
```

This tells the repository to tell the relation to first call `by_pk(1)` to scope the query, and then to call update on that result, updating the `name` field on all matching records – only one record in this case – to be "Ticketee v2".

Deleting works much the same way. We can call it like this:

```
repo.delete(2)
```

This tells the repository to tell the relation to delete the record with the `id` of 1, again first by scoping using `by_pk` and then calling `delete` on that result.

Summary

So there you have it. A very thorough introduction to ROM, demonstrating how it separates your application and your database by splitting the related code up into three main files:

- A relation – `Projects::Relations::Projects` – that contains queries for talking to the database
- A repository – `Projects::Repositories::Projects` – that talks to the relation, providing *materialized* data back to the application
- A model – `Projects::Project` – that represents data that we get back from the database. This is also where we could put any custom business logic that we need for these records.

Now that we've been introduced to ROM, it's time to see how we would use it within a Rails application.

Rails, meet ROM

In this chapter, we're going to set up a new Rails application with ROM, instead of Active Record. We spent the last chapter getting acquainted with ROM, and so now we can dive in to exactly how ROM and Rails can work together.

The easiest way to show you how this approach is better than the Active Record one is to demonstrate it within a brand new Rails application, free from Active Record.

Here's a Rails 5.2 application I've prepared earlier: https://git.io/vhMsm[21].

> **You should clone this application down and use it as the example application for the remainder of the book.**
>
> This repo contains the *full* code for this book, but to get started here in Chapter 2 you can check out to the very first commit that contains this Rails project and follow along from there. Fortunately, there's a branch called `start` just for you. To clone and checkout to that branch, run these commands:
>
> ```
> git clone git://github.com/radar/exploding-rails-examples/
> git checkout start
> ```
>
> The Rails application will be within the directory called `ticketee` inside this repository. You can use that to follow the remainder of this book.

This application was generated with this command:

[21] https://git.io/vhMsm

```
rails _5.2.0_ new ticketee --skip-active-record
```

Aside from the modifications that the `--skip-active-record` flag does to a regular Rails application, this application has a few other modifications:

- The `Gemfile` has had the `pg`, `rom-rails` and `rom-sql` gems added, which we'll be using to interact with a PostgreSQL database.
- The `Gemfile` has also had the `rspec-rails` gem added to it. We will be using this gem to test our application in the chapters of this book. The `rails g rspec:install` command from this gem has created the files within the `spec` directory of this application.
- I've also removed the `spring` and `spring-watcher-listen` gems from the `Gemfile`. In my experience, Spring works about 95% of the time and the 5% of the time when it doesn't work is highly frustrating. I'd rather not have it at all. It just forgets to reload things *all the damn time*. Away with you, spring.
- There's a new initializer at `config/initializers/rom.rb`, which configures a default database connection for ROM to use. This was taken from the ROM documentation, "Rails Setup"[22]. This initializer loads the `DATABASE_URL` environment variable, which is configured in another file called `.env.development`. This `.env.development` file is loaded by the `dotenv-rails` gem in the `Gemfile`, but only while the application is operating in the `development` environment.
- Similarly, there's a file called `.env.test` that has a different `DATABASE_URL` in it. This is the one that will be used for when our application is running its tests.
- The `Rakefile` has this line in it: `require 'rom/sql/rake_task'`. This requires the Rake tasks for generating / running migrations from the `rom-sql` gem. We'll need these because Active Record's aren't going to be available.
- The `app/views/layouts/application.html.erb` contains a few lines loading in Bootstrap 4.0, just to make the application look better than the default.

[22] http://www.rom-rb.org/4.0/learn/getting-started/rails-setup/

- The `rails_helper.rb` file has some custom code to ensure two things: that migrations have run on the test database, and that the database is cleared before every test.

These modifications are necessary to remove Active Record completely and to set up ROM in our application. Well, apart from the Bootstrap addition. That's just putting a cleaner coat of paint on the application so that the screenshots I use later on in this book look at least half-decent.

You could follow these same steps yourself within a brand-new Rails application, but I thought it would be a good idea to have application you could clone to save you time.

To create the database for this application, there isn't a Rake task anymore (e.g. `rake db:create`), but we can instead use PostgreSQL's `createdb` command which has the added bonus of being really quick because it doesn't have to load an entire Rails application's environment first before it can run:

```
createdb ticketee_dev
```

You might be wondering where I pulled this database name from. Well, this database name must match what's configured in `.env.development`, which is this:

```
DATABASE_URL=postgres://localhost/ticketee_dev
```

This tells ROM's underlying database engine (Sequel) where it can find our database.

With our database created and configuration in place, our application should boot successfully if we run a `rails c` session:

```
Loading development environment (Rails 5.2.0)
irb(main):001:0>
```

Great! If we saw an issue here it would be because Sequel couldn't connect to our database. With the Rails console booting successfully, we're confident that our configuration is all in place.

Now let's start generating the pieces for our application.

Generating our first ROM-powered relation, migration, repository and model

ROM splits the logic for working with your database and the records returned by queries over a few separate files: relations, repositories, and models. This is a radically different approach from a vanilla Rails application where you generate *one* file (the model) and throw everything in there. But remember: that's precisely what we're trying to avoid here. We're going to explode the responsibilities of the model across a few different files.

Put another way: We're paying a large setup cost to avoid future maintenance costs. This clear separation between the layers *will* feel exactly like "hard work" now, but on a larger scale Rails application it will pay off. A good thing to remember is that the "getting started cost" is only paid once, but the "maintenance cost" is forever.

We're going to create a new relation class, a migration file, a repository class and a model class. This is the same work that we did in the previous chapter, except this time it's going to be within the context of a Rails application.

To refresh your memory, here's what each of those terms mean again:

- Relations: a class that holds database query logic

- Migration: the same kind of file you've come to know and love from working with Rails.
- Repository: a class that provides an API between your application and a relation class
- Model: a class that *only* contains information about how a particular set of data is structured (e.g. how a table row should be structured in Ruby), and any business logic for that data.

That migration will create a table called `projects`, and the other classes will be used together to interact with this table, performing the regular CRUD operations that we would do in a Rails application.

Before we can do that, we'll need to create the relation and the migration to be able to interact with this table.

Generating the relation

As a refresher: Relations are where the queries to your tables belong. Relations make talking to the database from Ruby possible in a ROM application.

To start off, we'll generate a `Projects` relation by running the generator provided by the `rom-rails` gem:

```
rails g rom:relation projects
```

Relations are pluralized in ROM, just like their matching tables are. This generator generates a file at `app/relations/projects_relation.rb`:

app/relations/projects_relation.rb

```
class ProjectsRelation < ROM::Relation[:sql]
  gateway :default

  schema(:projects, infer: true)

  # define your methods here ie:
  #
  # def all
  #   select(:id, :name).order(:id)
  # end
end
```

This class is generated with a `gateway :default` call at the top, but that isn't actually required here, as that's the default setting. That's just how `rom-rails` has chosen to generate this file. It tells the relation to use the `:default` gateway, which is specified in `config/initializers/rom.rb`:

```
ROM::Rails::Railtie.configure do |config|
  config.gateways[:default] = [:sql, ENV.fetch('DATABASE_URL')]
end
```

If we wanted to connect to a different database in this relation, we could specify another gateway in `config/initializers/rom.rb` and refer to it in the relation, like this:

```
gateway :alternative
```

In a regular Rails application, you may have seen different models connecting to different databases with the `establish_connection` method call:

```
class Project < ApplicationRecord
  establish_connection :alternative
```

These two things are the same. For now, we're just going to stick with one database because that's what you'd do in most normal Rails applications.

The `schema` line is exactly like what we saw in the previous chapter: it tells this relation to use the `projects` table as its source, and to infer what the schema looks like when it's booted.

Further down the relation class, the commented out lines of code hint at what this relation class is for. It shows that you *could* define an `all` method that fetched the `id` and `name` fields for the records in the table, and ordered those records by their `id` fields. For now, we'll leave this code commented out.

We can use relations directly in the console – but only once we've setup the table. The code to interact with that table using this relation goes like this:

```
ROM.env.relations[:projects].by_pk(1).one
```

This method chain will use the relation to try to find a record with the ID=1 (`by_pk(1)`). Queries in relations by default will return all the records that match the query in an array. But with this particular query, we only want one result, and so we tell ROM this with the `one` method.

We saw this code in the previous chapter, so it shouldn't be surprising here. I just want to refresh your memory again.

Generating the migration

For the relation to be able to interact with a table, that table is actually going to need to exist! Fortunately, `rom-sql` provides a Rake task for generating migrations, so let's create one now by running this command in our terminal:

```
rake db:create_migration[create_projects]
```

This will generate a migration in the usual place (`db/migrate`) and it will look empty:

db/migrate/<timestamp>_create_projects.rb

```
ROM::SQL.migration do
  change do
  end
end
```

It's up to us to fill this migration out, so let's do that now:

db/migrate/<timestamp>_create_projects.rb

```
ROM::SQL.migration do
  change do
    create_table :projects do
      primary_key :id
      column :name, :text, null: false
    end
  end
end
```

This migration will now create a table called `projects`. That table will have a primary key called `id`, and it will have a `name` column that's the type of `:text`, and that column's values can *never* be null thanks to the `null: false` option.

We can run this migration the traditional way:

```
rake db:migrate
```

When you see this:

```
<= db:migrate executed
```

It means that the migrations have been completed successfully. It's not the pretty output of Rails's `db:migrate`, but it does the job.

Generating the repository

The next step that we need to undertake is to generate the repository. This will be the gateway between our application and our relation class.

The intention here is to provide a cleaner veneer over database queries than what might be available from a relation. For instance, a repository may combine several methods from a Relation class to get a result. The repository helps hide such complexities.

Let's generate a repository now:

```
rails g rom:repository project
```

This generates a new file at `app/repositories/project_repository.rb`:

app/repositories/project_repository.rb

```
1 class ProjectRepository < ROM::Repository::Root
2   root :projects
3
4   commands :create, update: :by_pk, delete: :by_pk, mapper: :project
5
6   struct_namespace Ticketee
7 end
```

This Repository class will let us perform queries on our database. It defines the `root` to be `:projects`, which will use the `ProjectsRelation` that we generated just before.

The `commands` line here gives us a few commands by default: `create`, `update` and `delete`. These will allow us to do 3-out-of-4 CRUD actions to our database. We'll cover the 4th action (Read) in a moment.

We're going to change the `struct_namespace` method call here to use the constant `Projects`, rather than `Ticketee`:

app/repositories/project_repository.rb

```
1 class ProjectRepository < ROM::Repository::Root
2   root :projects
3
4   commands :create, update: :by_pk, delete: :by_pk
5
6   struct_namespace Projects
7 end
```

We're making this change because it is good practice in ROM applications to separate out your models into distinct namespaces. The classes for working with projects (and related things) in this application will go under the `Projects` namespace.

Later on in our application, we might want to create an `Accounts` namespace too, for things like accounts and user information. This namespace separation helps developers new to our application understand what the different parts of our application are. Looking at a 65-file model directory can be a bit daunting, even for experienced developers! Separating these models into namespaces makes it less daunting.

This approach will lead to a more structured `app/models` directory, which may end up looking like this:

```
app/models
├── accounts
│   └── account.rb
│   └── user.rb
├── projects
│   └── project.rb
│   └── ticket.rb
```

So this `struct_namespace` method tells ROM that we're going to have some struct classes living underneath the `Projects` namespace, and that this repository class should try to initialize objects using a class called `Projects::Project`. ROM infers the class name from the repository's name.

Creating a model class

The final piece of this puzzle is the `Projects::Project` class itself. This class will be used to represent `Projects::Project` instances in your application, much like how a traditional Rails model is used.

There isn't a generator for this class (because Active Record is gone and ROM doesn't provide one), so we must create it ourselves. Thankfully, the process is easy.

We might be tempted here to reach for `ActiveModel::Model` here, as its supposed to turn regular classes into ones that will be compatible with your Rails app's forms, and allow for calls like `Projects::Project.new(name: "New Project")`.

While this is true, it also adds in some class-level validation and callback methods, and this means that if we use `ActiveModel::Model` inside our class, it will still fall prey to combining validation, business logic and data modelling in the one class. This is the sort of thing that we're trying to avoid by not using Active Record in the first place. We're very intentionally here trying to keep our validation, business logic and data modelling separate to make our code very

easy to understand.

It's for this reason that we're going to use another feature from ROM called `ROM::Struct`. The objects of this class are designed to be very light *and* immutable objects. The immutability means that you cannot re-set an attribute once it has been set during the initialization. For instance, in a Rails model you are able to do this:

```
project = Projects::Project.new(name: "A name")
project.name = "A new name"
```

But in `ROM::Struct` instances, there are no attribute writers and so you can't do this in your code at all. Instead, you would need to create a new instance with the same attributes. This prevents us from potentially changing model attributes on the fly, which may lead to data being different between the database and what the user sees. With immutability, we prevent this from happening.

The first thing we'll do here is to create a class that will act as a root for all our application's models. This class will be similar to `ApplicationRecord`, in that it defines logic that is common to *all* models in the application. We'll define this new model like this:

app/models/application_model.rb

```
1  class ApplicationModel < ROM::Struct
2    def self.inherited(klass)
3      super
4
5      klass.extend ActiveModel::Naming
6      klass.include ActiveModel::Conversion
7
8      klass.transform_types { |t| t.meta(omittable: true) }
9    end
10
11   def persisted?
```

```
12      respond_to?(:id) && id.present?
13    end
14  end
```

This class does quite a few things. The first thing is that it inherits from `ROM::Struct`. By inheriting from `ROM::Struct`, this model acts in a similar fashion to the Active Record models we know and... Well, the ones we know.

`ROM::Struct` allows us to create new instances of this model, just like we would with a traditional model. For example, we will still be able to run this code in `rails console` to get a new `Projects::Project` instance with a `name` attribute set to the specified value:

```
Projects::Project.new(name: "Test Project")
```

This class defines an `inherited` method, which is automatically called whenever another class inherits from this one. This method initially calls `super`, which will call `ROM::Struct`'s own `inherited` method.

Immediately after the `super`, we extend the class with the `ActiveModel::Naming` module. This adds methods to the instances of any model, and you can read about what methods that provides in the ActiveModel::Naming[23] API documentation. Particularly helpful from this particular module is the `model_name` method, which is used in polymorphic URL routing (i.e. `redirect_to project`) – the kind of routing that `form_for(@project)` uses to determine where to submit the form to. Very helpful!

The `ActiveModel::Conversion` module after that adds in methods like `to_model` and `to_param`. These methods are used in Rails also for routing and forms. Also helpful! This also adds a `to_partial_path` method, which will generate a path like `"projects/project"`, so you could use code such as this in your view to render a project partial:

[23] http://api.rubyonrails.org/classes/ActiveModel/Naming.html

```
<%= render @project %>
```

This is the same as writing:

```
<%= render partial: "projects/project", project: project %>
```

You can read about ActiveModel::Conversion[24] in the Rails API docs too.

> **Avoiding throwing the baby out with the bath water** Both `ActiveModel::Naming` and `ActiveModel::Conversion` provide helpful features that we often rely on within a Rails application. Just because we're using the cool, new, shiny tech with ROM doesn't mean that we have to completely abandon every single Rails principle.
>
> There *are* some good parts still within Rails applications. These are two small examples. There will be more througout this book.

It's worth mentioning that both of these modules, along with `ActiveModel::Validations` and `ActiveModel::Callbacks` would've been included into our class anyway if we were to use `ActiveModel::Model`.[25] By doing it this way instead, we pick-and-choose the modules that we want and don't have callback and validation logic included along with it.

While the additions of these two modules to all our models does make it so that our model is responsible for knowing more than just our regular run-of-the-mill business logic, this is a necessary thing to do to ensure that these model instances are compatible with Rails and can be used just like regular model instances within a Rails app. We are only getting the barest bones of what we need to make this model function within a Rails application.

We *could* avoid doing this by writing some of this logic ourselves but the code might seem a little too alien from the Rails that we're used to, so I am suggesting

[24] http://api.rubyonrails.org/classes/ActiveModel/Conversion.html
[25] You can see the code in `ActiveModel::Model` here if you're interested: https://git.io/vhMso.

this little bit of extra code to save some time re-implementing what is already easily available to us.

After these two `ActiveModel` lines is this line:

app/models/application_model.rb

```
1 klass.transform_types { |t| t.meta(omittable: true) }
```

This method comes from a parent class of `ROM::Struct` called `Dry::Struct`. Subclasses of `Dry::Struct` and `ROM::Struct` require all attributes to be specified during initialization of their objects. Wow that's a lot of jargon. Ok, how about an example? This is what you will need to do when initializing classes without this `transform_types` line:

```
Projects::Project.new(id: nil, name: nil, description: nil)
```

What this `transform_types` line tells `Dry::Struct` to allow is for us to omit *all* attributes when initializing objects, like this:

```
Projects::Project.new
```

Those attributes will not be defined on the object at all. This sort of initialization is exactly how we would initialize a new object of this class within a place like the `new` action of `ProjectsController` – eventually. We haven't gotten that far yet. But we need this `transform_types` line to allow for usecases like that.

The only special bit of logic required here that we have to hand-roll in the `ApplicationModel` class is the `persisted?` method, which is used by Rails helpers like `form_for` / `form_with` to determine if the form should make a `POST` request to the `create` action or a `PUT` request to the `update` action in a controller. You might ask how that all works, so here's a separate blog post explaining polymorphic routes in Rails[26] for your consumption / enjoyment.

[26] http://ryanbigg.com/2012/03/polymorphic-routes

The short version is this: if the record has been persisted to the database then it will have an `id`. This makes `persisted?` return `true`, and `form_for` will then send its form submission to a route like `PUT /projects/1`. If the record isn't in the database then it won't have an `id`, and therefore `persisted?` will return `false`. The `form_for` will then submit to the `create` action of the resource, using a route like `POST /projects`. This is a pretty great feature of Rails: that we can use the same form for two different actions. So we'll use this `persisted?` method to make our model play nice with Rails' form helpers.

OK, we've spent a lot of time talking about our model but we haven't created it yet. Let's go ahead and create a new `Projects::Project` model at `app/models/projects/project.rb` and make that model inherit from `ApplicationModel`:

app/models/projects/project.rb

```
1 module Projects
2   class Project < ApplicationModel
3   end
4 end
```

Remember, we're namespacing this model underneath the `Projects` namespace because our `ProjectRepository` is going to be expecting this class to be there, thanks to its `struct_namespace` definition.

In this model we define the model-specific attributes. In this case, our `Projects::Project` model will only just have an id (from `ApplicationModel`), and it will have the rest of its attributes automatically inferred from the database. This happens automatically due to how we've setup the `schema` call in the `ProjectsRelation` class:

app/relations/projects_relation.rb

```
class ProjectsRelation < ROM::Relation[:sql]
  gateway :default

  schema(:projects, infer: true)

  ...
end
```

Notice how the `Projects::Project` class here contains *no* information at all about how this entity is validated or even persisted to the database. It's a plain-ol'-ruby class that is accentuated with `ROM::Struct`. It will be used to represent records from the database and for no other purpose. It has a single responsibility.

We've now created a relation, a repository, and a model. We've *exploded* our traditional Rails model into 3 distinct files with 3 distinct responsibilities:

- Relation: contains logic for making database queries
- Repository: Talks to relation, asks it to do the queries on the database
- Model: A bare-bones representation of `Projects::Project` records within our application, which uses `ROM::Struct`, accentuated with some hand-picked pieces of `ActiveModel`.

There's a clear *single responsibility* for each of these classes that is sorely lacking in a traditional Rails model. Exploding our Rails model into these 3 parts makes it so much easier to reason about where particular logic goes already, leading to smaller and easier-to-reason-about classes.

We've spent a lot more time setting up the different parts for our model than we would have in a regular application. But time spent now will save time later, because each responsibility of that model is now separated into a different class.

Now that we have all our parts setup, let's actually do something with them.

Creating a record

We'll boot up a `rails console` session again and this time we'll create a new project using our repository:

```
repo = ProjectRepository.new(ROM.env)
project = repo.create(name: "Test Project")
```

We must initialize the repository with `ROM.env` here, as that object contains configuration data about how to connect to our database. This is the same type of object as `Projects::DB` was in the previous chapter – a ROM container object. Go ahead and peek at what `ROM.env` is in the console if you would like:

```
ROM.env
=> #<ROM::Container gateways...>
```

Once we've initialized the repository, we can then call methods from that class. Now is a perfect time to remember this: the repository is the class that bridges the divide between our application's code and the database. The repository will talk to the `ProjectsRelation` class to perform queries, and that `ProjectsRelation` class will pass data back to the repository, which will then pass it back to our application.

We call the `create` command on the `repo`, passing it attributes just like we would with `Model.create` within a regular Rails app. The difference here in our new app is that model knows nothing about the database. The relation knows about the database, and the repository knows about the relation.

This `create` method for our repository is defined by the `commands` line in the `ProjectRepository` class:

app/models/project_repository.rb

```
1 class ProjectRepository < ROM::Repository::Root
2   root :projects
3
4   commands :create, update: :by_pk, delete: :by_pk
5   ...
```

The return value from `create` is a `Projects::Project` instance:

```
=> #<Projects::Project id=1 name="Test Project">
```

But how does the `create` method know to return a `Projects::Project` instance? When we configured our `ProjectRepository` class earlier, we used `struct_namespace`:

app/repositories/project_repository.rb

```
1 struct_namespace Projects
```

This configuration tells the repository that our application will provide structs for any data from the repository, and those structs are going to live underneath the module namespace of `Projects`. The `Project` part is inferred from the name of the repository.

This `Projects::Project` instance is a very lightweight one. Some methods that it can respond to are `id`, `name` and `persisted?`:

```
>> project.id
=> 1
>> project.name
=> "Test Project"
>> project.persisted?
=> true
>> project.to_partial_path
=> "projects/projects/project"
```

You might run into some autoloading issues

There's been some reported cases that the above code has failed to work for some people. This is due to some autoloading weirdness associated with the gem called `rom-mapper` (and Rails is probably to blame somehow in there too...)

The main issue is that `rom-mapper` does not appear to be loading the `Projects::Project` class at all, before deciding to build its own version of this class. This imitation class doesn't have the `persisted?` or `to_partial_path` methods on it, and so the above code will fail.

I have a pull request open for the `rom` repository (PR #496[27]) which will probably fix this issue. So if you're running into this issue, you can fix it by adding this line to your `Gemfile`:

```
gem 'rom-mapper', github: 'radar/rom', branch: 'build-once'
```

Please let me know if this fixes your issue by commenting on the PR.

The `id` and `name` methods are automatically added to the class when we fetch the record, due to a ROM feature called *auto struct building*. This feature will inspect the data returned from an initial database query and add attributes that match to the class that represents the model. If at any time we want to see what

[27] https://github.com/rom-rb/rom/pull/496

these attributes are, we can call either `Projects::Project.schema` or, if we only care about the names, `Projects::Project.attribute_names`.

I want to mention something about the `persisted?` method here I should have mentioned earlier. The `persisted?` method must check if the method responds to an `id` method, as this attribute isn't guaranteed to exist. For instance, if we initialize a model by calling something like `Projects::Project.new`, it won't have any attributes by default. We can try this out in the console and see for ourselves:

```
>> new_project = Projects::Project.new
=> #<Projects::Project>
>> new_project.id
irb(main):003:0> new_project.id
Traceback (most recent call last):
      1: from (irb):3
ROM::Struct::MissingAttribute (ROM::Struct::MissingAttribute)
```

This is because there has been no database call that returns data that would setup the class with the right attributes, and so the class has *no* attributes. We would only typically initialize an object like this in the `new` action of a Rails controller, and it's not important for the attributes to be present there, and so this is OK to do. All that matters is that the attributes will be there when these objects are initialized after calls have been made to the database.

The `persisted?` method comes from `ApplicationModel` where it is defined explicitly. The `to_partial_path` method comes from `ActiveModel::Conversion`, which we include inside `ApplicationModel`. Seems like everything is working nicely together!

The object (and its class) knows nothing about how it was brought into being. The class knows only the bare essentials that it needs to know in order to accomplish its job. It can't execute any more database queries. That's someone

else's job.

OK, so this is well and good but how do we tie this to our Rails application? How do we make it so that people can create projects using ROM through the app? Well, let's take a look at that now.

Connecting a Rails controller with ROM

So far, we've built a good replacement for what would normally be a `Project` model within Rails. But this is off in its own part of the application now, disconnected from everything else. It's time that we connected these pieces to our application and made them do something useful.

Some super-eager people who love the bleeding edge of Ruby progressiveism would have you ditch everything Rails entirely and to build an application using *all* of the new shiny tools provided by rom-rb (and its "cousin", dry- rb, which we'll get to), but I think Rails has some good parts too, namely things like its router and views – and also those `ActiveModel` modules we used in the last chapter.[28]

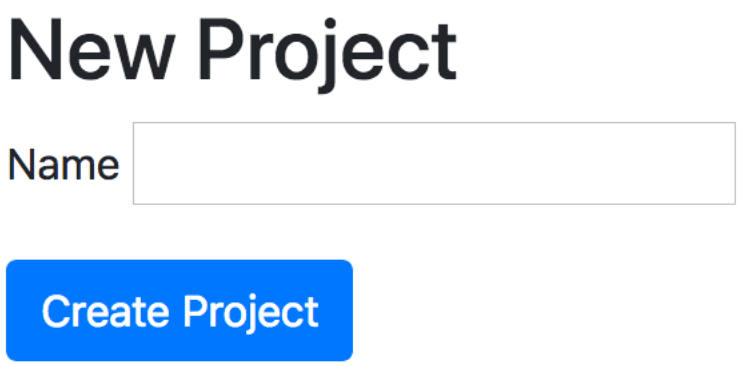
Controllers in Rails can be great too, just as long as the logic in those controllers is kept to a minimum of handling incoming requests and outgoing responses. A controller, in my opinion, should only be responsible for connecting business logic with requests and responses. It should not contain any business logic.

In this section, we're going to use those traditional parts of Rails – the router, the controllers and the views – in conjunction with the ROM code that we've just written. We're going to write a skinny controller to interact with our skinny ROM code. It's going to be sweet.

We're going to start by creating a form that lets users of this application create

[28] This seems like a good point to mention that there's a great alternative to Rails' routing out there called Roda. I don't use it at all in this book, but it's good to know that it exists. The super-eager bleeding-edge people love to use this as an alternative to the Rails router.

projects. It's going to be a traditional Rails form:



New Project

The only thing special about this form is that it's going to submit data to a controller action, and that action is going to insert data into a database using ROM instead of Active Record.

We can start out this part by writing a Capybara feature test using RSpec to ensure that our code works:

spec/features/projects/creating_spec.rb

```
require "rails_helper"

RSpec.feature "Users can create new projects" do
  scenario "with valid attributes" do
    visit "/"

    click_link "New Project"

    fill_in "Name", with: "Sublime Text 3"
    click_button "Create Project"

    expect(page).to have_content "Project has been created."
  end
end
```

I won't walk through each step of this – Rails 4 in Action[29] does that already. Instead, I'll assume you know what this test does; or at least you can read it well enough to understand it.

Let's setup the routes for all of this:

config/routes.rb

```
Rails.application.routes.draw do
  root to: "projects/projects#index"

  namespace :projects do
    resources :projects
  end
end
```

Just like how we've namespaced our models, we're also going to namespace our controllers to match. Speaking of controllers, now that we have a route we should setup the matching controller:

app/controllers/projects/projects_controller.rb

```
module Projects
  class ProjectsController < ApplicationController
    def new
      @project = Projects::Project.new
    end

    def create
      repo.create(project_params)
      flash[:notice] = "Project has been created."
      redirect_to action: :index
    end

    private
```

[29] https://manning.com/books/rails-4-in-action

```
14
15      def project_params
16        params.require(:project).permit(:name).to_h.symbolize_keys
17      end
18
19      def repo
20        ProjectRepository.new(rom)
21      end
22    end
23  end
```

Important to note here is the `repo` method right at the end which instantiates a new `ProjectRepository` instance, just as we've done in the console in the past. Except this time we're using a method called `rom`, as that is provided to us automatically by the `rom-rails` gem. We could use `ROM.env` to get the same thing, but it's four extra characters. We're lazy, so we'll just use `rom` in controllers.

The `create` action in this controller looks almost *exactly* like a normal `create` action, but rather than calling `create` on the model class (i.e. `Project.create`) it calls it on the repo instead: `repo.create`.

Also important to note that there's a `.to_h` and `symbolize_keys` call on the end of the code inside `project_params`. This is because the `create` method from ROM expects the keys to be symbols, and not the `ActionController::Parameters` default, which is strings.

Next up, we'll have to define a couple of views for our application.

We'll need a view to present the test with the "New Project" link. Let's put this code in `app/views/projects/index.html.erb` to accomplish that goal for now:

app/views/projects/projects/index.html.erb

```
<h2>Projects</h2>

<%= link_to "New Project", new_projects_project_path, class: "btn btn-primary" %>
```

This page presents the "New Project" link, which is needed for the 2nd step of our test. The first step is visiting the page that renders this view. We've already got the `new` controller action, so let's move to creating the view for this action:

app/views/projects/projects/new.html.erb

```
<h2>New Project</h2>

<%= form_with(model: @project, local: true, scope: :project) do |form| %>
  <p>
    <%= form.label :name %>
    <%= form.text_field :name %>
  </p>

  <%= form.submit class: "btn btn-primary" %>
<% end %>
```

In this `form_with` tag, we're using three options: `model`, `local` and `scope`.

The `model` option tells the form what object we want to use for this form. We want to use the `@project` object from the `new` action in the controller.

The `local` option tells the form that we want to submit it without fancy AJAX features. These would require the controller to be setup slightly differently to support that kind of thing, which it's not.

The final option, `scope`, tells `form_with` what parameter name to use for the form. If we didn't specify this option, all the form's parameters would come back under a key called `projects_project`, because the model's class is called

`Projects::Project`. The `scope` option changes the parameter name back to a sensible one: `:project`.

There's only been one place where we've had to put ROM-specific code: the controller. Everything else has been vanilla Rails.

Before we run the test, we will need to setup the test database. We can do that by running the migrations against a test database. But first, we'll need to create that test database:

```
createdb ticketee_test
```

This name is from the `.env.test` file, which will be used instead of `.env.development`, as long as `Rails.env` is set to `test`. Let's set this environment variable and run the migrations against the test database in one fell swoop:

```
RAILS_ENV=test rake db:migrate
```

Once again, we'll see this output:

```
<= db:migrate executed
```

This means that our migration has run successfully. This time, it will have run successfully against the `ticketee_test` database, and not the `ticketee_dev` database.

If we run our test with `bundle exec rspec spec/features/creating_projects_spec.rb`, we'll see that it's happy:

```
1 example, 0 failures
```

According to that test, a user can create a project just the same as they've always been able to. It's just now that the project is created through ROM, and not Active Record.

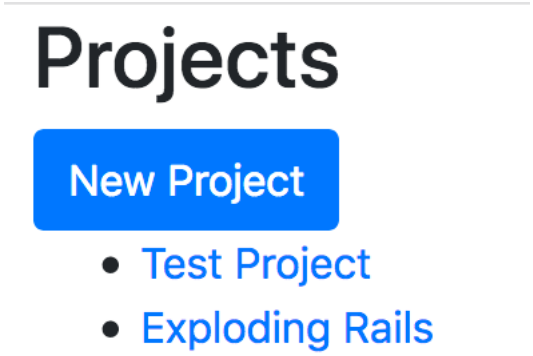
But this doesn't really demonstrate the power of rom-rb too much. The intention with this section was more to just demonstrate that it's *possible* to use ROM and Rails together very easily.

So far, we've only looked at one of the CRUD actions: Creation. Let's look at how we can use our repository now for reading this project back out of the database.

Showing a particular project

So far, we've focused *a lot* on the C part of CRUD: creating. Now is the time we're going to talk about the next part of CRUD: *reading*. In particular, we'll look at how we can implement the `index` and `show` actions for the `ProjectsController` and how we can fetch data in those actions by using the ROM classes we've built up so far.

Let's start out with the `index` action. This action should present a list of projects to a user and allow them to view more information about them:



Projects list

When a user clicks the project's name, they should be taken to that project's page. Pretty straight forward action to take within a Rails application. It shouldn't be too hard to make this work.

Let's write another feature to test both actions.

spec/features/projects/viewing_spec.rb

```
require "rails_helper"

RSpec.feature "Users can view projects" do
  let(:project_repo) { ProjectRepository.new(ROM.env) }
  let!(:project) { project_repo.create(name: "Sublime Text 3") }

  scenario "with the project details" do
    visit "/"
    click_link "Sublime Text 3"
    expect(page.current_url).to eq projects_project_url(project)
    within("h2") do
      expect(page).to have_content("Sublime Text 3")
    end
  end
end
```

In this test, we must create a project before we can view it, and so that's what we do at the start of this feature. In the scenario, when we visit the root of our application, there should be a link there with the text of our project's name. When we click that link, we should be taken to that project's URL, and on that page there should be a `<h2>` element containing the project's name again.

To make this test work, we're going to need to make the `index` action show a list of projects. And for that to happen, we will need to implement the ROM equivalent to `Model.all` on our repository.

We've been able to create records in our database by using the repository, but we haven't yet tried to *read* records out. You might think that it's as simple as calling `all` on the repository, like this:

```
repo = ProjectRepository.new(ROM.env)
repo.all
=> [<all the projects>]
```

You'd be mostly right. The only thing that we need to do is to tell the `ProjectRepository` class about this `all` method. Repositories do not get methods automatically because each repository may not need all the methods. This is wildly different to a typical Active Record model, which gets everything that `ActiveRecord::Base` provides; which is quite a lot!

In a normal Rails application, Active Record dictates that the `all` method should fetch all the records (and all the fields) from a table. But perhaps in an app that uses ROM, you don't want to do this. Perhaps you want to only fetch *some* of the records. Perhaps you only want to fetch the `id` and `name` columns. So ROM leaves it up to us to choose how we would fetch a collection of records from our table, leaving it to us to determine how many records and which fields we want to fetch. In this case, we'll just do the same thing as Active Record: fetch all the records, but only select the `id` and `name` fields.

Let's go ahead and add this `all` method to our `ProjectRepository` class now:

app/repositories/project_repository.rb

```
1  class ProjectRepository < ROM::Repository::Root
2    root :projects
3
4    commands :create, update: :by_pk, delete: :by_pk
5
6    struct_namespace Projects
7
8    def all
9      projects.all.to_a
10   end
11 end
```

The `projects` method used here refers to the relation used by this repository and is made available to the instance methods of this class by the `root` call at the top of this class.

We then put an `all` call on the end, which should call a method called `all` on the `ProjectsRelation` instance returned by `projects`. We then put the `to_a` on the end of *that* call to *materialise* the results of our query. Remember: relation methods always return a relation object until we materialise them with a method like `to_a` or `one`; and that's always what we should be doing in a repository! The repository is the only place where we should be talking to our relation and telling it to run database queries.

The `ProjectsRelation` class doesn't have this `all` method defined. If we try to call `repo.all` now, it will fail:

```
repo = ProjectRepository.new(ROM.env)
repo.all
irb(main):002:0> repo.all
Traceback (most recent call last):
        2: from (irb):2
        1: from app/repositories/project_repository.rb:7:in `all'
NoMethodError (undefined method `all' for #<ProjectsRelation:...>)
```

This is a good time to remind you that the repository is simply the bridge between our application and the `ProjectsRelation` class, with the `ProjectsRelation` class ultimately deciding how to make calls to the database. When those calls are made, the repository takes the data from the relation and presents it through these methods like `create` and `all`.

Let's add the `all` method to `ProjectsRelation` now. In fact, there's a method called `all` already in that file, but commented out:

app/relations/projects_relation.rb

```
class ProjectsRelation < ROM::Relation[:sql]
  gateway :default

  schema(:projects, infer: true)

  # define your methods here ie:
  #
  # def all
  #   select(:id, :name).order(:id)
  # end
end
```

That looks perfect! Let's uncomment that:

app/relations/projects_relation.rb

```
class ProjectsRelation < ROM::Relation[:sql]
  gateway :default

  schema(:projects, infer: true)

  def all
    select(:id, :name).order(:id)
  end
end
```

This uses the `rom-sql` provided methods to select just the `id` and `name` values for these records and to order these records by their `id` field. We're only returning `id` and `name` here for the fields as that's all our model instance needs. We're not limiting the result set here, and so we will really see *all* projects returned by this method.

The equivalent with Active Record would be:

```
Model.select(:id, :name).order(:id)
```

It's pretty close. The major difference is that Active Record calls the methods on the model, but ROM calls them on the relation instead. Models in ROM-backed applications stop being the go-to for absolutely everything, as we're quickly seeing!

Let's try out this `ProjectRepository#all` method in a new console now:

```
repo = ProjectRepository.new(ROM.env)
repo.all
 # => [#<Projects::Project id=1 name="Test Project">]
```

Ok, it looks like our `ProjectRepository#all` method is now working and fetching records from the database successfully. Or should I say, "record" because we've only created one!

It's time to use the repository's `all` method in the controller. Let's go into `Projects::ProjectsController` and define an `index` action:

app/controllers/projects/projects_controller.rb

```
1 def index
2   @projects = repo.all
3 end
```

This is calling the `repo` method defined at the bottom of this controller:

```
def repo
  ProjectRepository.new(ROM.env)
end
```

This code isn't all that different from what would regularly go in an `index` action:

```
def index
  @projects = Project.all
end
```

The only difference is that we're using the repository and not the model to make this call. Isn't this separation nice? The model still has no idea how our database works.

Next up, in the template for the `index` action we can list these projects:

app/views/projects/projects/index.html.erb

```
<h2>Projects</h2>

<%= link_to "New Project", new_projects_project_path, class: "btn btn-primary" %>

<ul>
  <% @projects.each do |project| %>
    <li><%= link_to project.name, project %></li>
  <% end %>
</ul>
```

The `link_to` here will generate a link like `/projects/projects/1` and that will go to the `show` action. We'll need to define this `show` action in `Projects::ProjectsController`:

app/controllers/projects/projects_controller.rb

```
def show
  @project = repo.by_id(params[:id])
end
```

This `by_id` method works identically to the `find` method we know from a normal model: it will return an instance of that model. The only difference here is that we're calling the `by_id` method on the repo, rather than on the model class.

The `ProjectRepository` does not have any method called `by_id`, so we will need to add one. Let's open that class and add the method:

app/repositories/project_repository.rb

```
def by_id(id)
  projects.by_pk(id).one!
end
```

This method uses the in-built `ROM::Relation` methods `by_pk` and `one!` to fetch a record with the given "pk" – primary key – and then to enforce that this method should only return the first matching record. If the search for a record returns anything but one result we'll see a `ROM::TupleCountMismatchError` exception. This exception gets raised if the `one!` query does not return *exactly* `1` record. This is different from `ActiveRecord::RecordNotFound`, which will only be raised when *no* records are found by `Model.find`. This `ActiveRecord::RecordNotFound` exception would not be raised if the query returned more than one record.

With this method defined in our repository and relation, and also used in the `show` action, the only thing left to do is to write the template for the `show` action:

app/views/projects/projects/show.html.erb

```
<h2><%= @project.name %></h2>
```

If we run our tests now with `bundle exec rspec`, we'll see that they both pass:

```
2 examples, 0 failures
```

Users can now create and view projects within our application, and our controller code isn't radically different from what we would normally have in a Rails application.

The difference with this code is that we've clearly separated out the responsibilities of our application. Code that talks to the database lives in the `ProjectsRelation` class. The application talks to this code by way of the `ProjectRepository` class. The logic we have defined isn't overly complex, but it is very clearly separated.

There is still no code in our model that knows about database queries. The controller is merely a conduit to the repository and the repository merely a conduit to the relation. The controller, repository, relation and model code are still neat and tidy.

But we haven't really done much to make a mess. Our code so far has been pretty straight-forward. Let's change that now by introducing another bit of complexity to our `create` action: a validation.

Validating projects during creation

In this chapter we're going to make our code messy by adding in a validation. And then we're going to clean it up. This will serve as another great example of how rom-rb and dry-rb lead to cleaner Rails applications.

A validation sounds innocent enough. After all, in Rails it's usually just a single line in the model:

```
validates :name, presence: true
```

But then validations tend to multiply on complex models. A single validation becomes five. Some validations have complex logic around their validation rules. Some validations run before create, others run before update, and others still run during create *and* update. Suddenly, the model becomes responsible for knowing how the data it is supposed to represent is validated when that data enters the database.

Sometimes, you might not want a validation to run. Maybe you're testing an unrelated attribute in a separate unit test? Or maybe you just want a small representation of the model, without all the "fluff". Whatever the reason, sometimes validations get in the way.

In this chapter we'll add a validation, and it *won't* get in the way of what our model does.

A small interlude about that contributors method

In a regular Rails app – one that uses Active Record – you might wish to test that `Project#contributors` method I mentioned near the start of this guide:

```
context "contributors" do
  let(:project) { Project.create(name: "Test Project") }

  context "when the project has tickets" do
    let(:ryan) { User.create(name: "Ryan") }

    before do
      project.tickets.create!(title: "Test Ticket", user: ryan)
    end

    it "lists the contributors" do
      expect(project.contributors).to include(ryan)
    end
  end
end
```

This unit test for the `Project` model knows quite a lot. It knows how to create projects and it knows how to create tickets and it knows how to create users. It's more of an integration test than a unit test! If any of these models had validations that were added that required other attributes to be present, then the `create` calls here will need to be updated with those fields. Adding something to the `Project`, `Ticket` or `User` model would require this spec to change. This spec has more than one reason to change.

Of course, we could write the spec better and stub out these other classes:

```
context "contributors" do
  let(:project) { Project.create(name: "Test Project") }

  context "when the project has tickets" do
    let(:ryan) { double(User) }
    let(:tickets) { [double(Ticket, user: ryan)] }

    before do
      allow(project).to receive(:tickets) { tickets }
    end

    it "lists the contributors" do
      expect(project.contributors).to include(ryan)
    end
  end
end
```

After all, this is a unit test of a feature of the `Project` model and it shouldn't really be involving other classes. With our mock safely in place, our test will work. That is, until we realize the performance issues caused by the N+1 query in our model:

```
def contributors
  tickets.map(&:user).uniq
end
```

So then we'll add that `includes` statement:

```
def contributors
  tickets.includes(:user).map(&:user).uniq
end
```

But then this breaks the mock in our test: we would need to stub out the `includes` method as well. This seems like a step too difficult because our test knows too

much about the implementation of the `contributors` method, and so we go back to our original test implementation: creating items in the database:

```
context "contributors" do
  let(:project) { Project.create(name: "Test Project") }

  context "when the project has tickets" do
    let(:ryan) { User.create(name: "Ryan") }

    before do
      project.tickets.create!(title: "Test Ticket", user: ryan)
    end

    it "lists the contributors" do
      expect(project.contributors).to include(ryan)
    end
  end
end
```

But then again we run into the issue that if we add validations for these models then we'll need to update the `create` calls here to specify the new fields. We're back to where we started. We just can't win! Active Record has trapped us into writing poor code for our tests, just so that those tests are compatible with Active Record's way of doing things.

We could switch our test to not using instances that are in the database, instead relying only on initialized instances:

```
context "contributors" do
  let(:project) { Project.new(name: "Test Project") }

  context "when the project has tickets" do
    let(:ryan) { User.new(name: "Ryan") }

    before do
      project.tickets.build(title: "Test Ticket", user: ryan)
    end

    it "lists the contributors" do
      expect(project.contributors).to include(ryan)
    end
  end
end
```

But then the `includes` statement means that Active Record will try to execute a database call anyway, and so this attempt won't work. We *must* use objects that have been persisted to the database in this particular test.

This is one small example of how Active Record encourages bad application decisions. It should absolutely be possible to test a small bit of business logic such as this `contributors` method *without* having database calls mixed up in it. By the stage the `contributors` method is called, the `Project` instance should have all the data it needs to do its work.

In the ROM world, validation is kept separate from the model and model classes know nothing about how their data is validated. Model classes only know what attributes to expect. Validating data is someone else's responsibility. We can have queries that work quickly to get the contributors for our projects *and* we can write nice unit tests for them *and* those unit tests don't have to speak to a database.

Separating out the validation logic into its own class not only makes sense, but

it makes it easy to test the validation logic in isolation from the business logic of the model and vice versa. Our `contributors` method would then be back to its simplest version:

```
def contributors
  tickets.map(&:user).uniq
end
```

We will eventually get to actually writing this method in our project and making it work. But for now, we'll focus on adding a validation for projects to our application.

Creating a validator

When creating projects in our application, we want to enforce that those projects always have names on them. Because we're not using Active Record, we can't simply put a line in the model and be done with it. Instead, we're going to intentionally create a separate class that is responsible for handling validations for our projects.

To begin with, we'll add `dry-validation` to our `Gemfile`:

```
gem 'dry-validation', '~> 0.12.0'
```

And we'll run `bundle install` to install this gem.

We're going to create this validation class in a moment, but let's write a test for that class first, just to see how easy it is to test the validation. `dry-validation` calls the classes that it uses for validations *schemas*, and so we're going to follow the same pattern here. We'll put the code for this test at `spec/schemas/projects/project_schema_spec.rb`:

spec/schemas/projects/project_schema.rb

```
require 'rails_helper'

describe Projects::ProjectSchema do
  let(:result) { subject.call(attributes) }

  context "when a name is specified" do
    let(:attributes) { { name: "Test Project" } }

    it "is valid" do
      expect(result).to be_success
    end
  end

  context "when a name is not specified" do
    let(:attributes) { { } }

    it "is invalid" do
      expect(result).to be_failure
      expect(result.errors[:name]).to eq(["is missing"])
    end
  end
end
```

This test is pretty easy to read and so it does not require much explanation. It's testing that when the schema receives attributes containing `name` that it works, and when those attributes do not contain `name` it fails.

Running this test now will be pretty ineffectual because our `Projects::ProjectSchema` constant doesn't exist, so let's go ahead and create it now in `app/schemas/projects/project_schema.rb`:

app/schemas/projects/project_schema.rb

```
require 'dry-validation'

module Projects
  ProjectSchema = Dry::Validation.Schema do
    required(:name).filled
  end
end
```

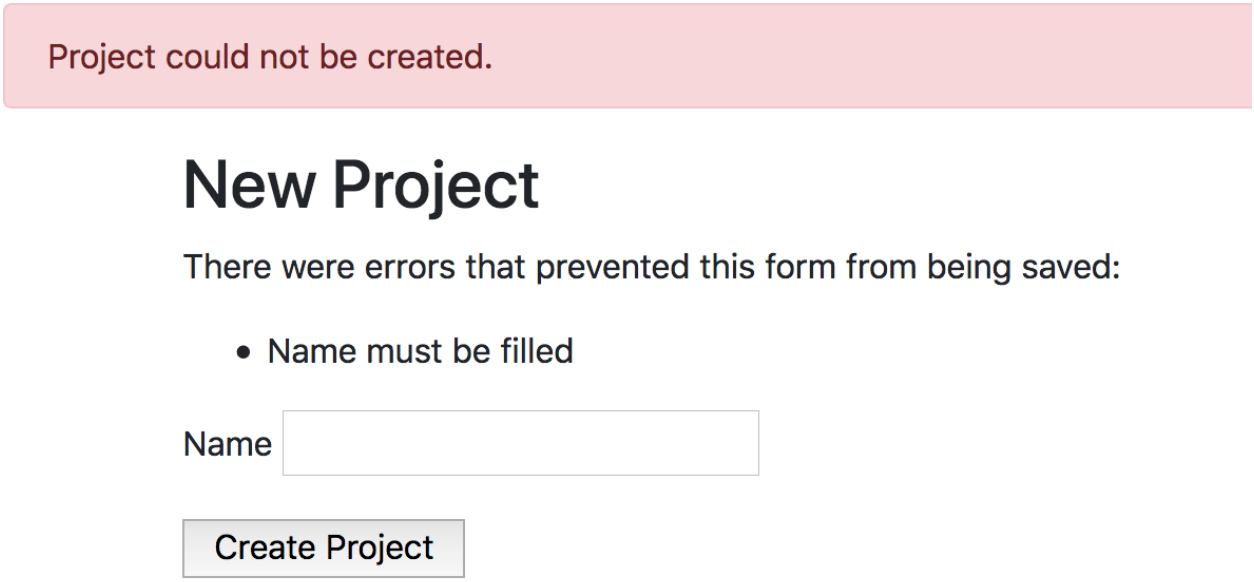
This code defines a constant called `ProjectSchema`, which defines the validation schema for projects. It has one rule: that `name` is required and it must be filled in; i.e. it cannot be `nil` or a blank value. By having this validation in its own class, we can use it or test it in complete isolation from any other part of the application.

Running the schema's specs now will show it passing:

```
2 examples, 0 failures
```

This asserts that our schema can validate the Hash that we're using in the test. If the `name` field has been left blank then the validation will fail.

The next step is to connect the controller and this validation schema together. We want it to be the case that if someone does not enter a project name that it re-renders the `new` form and shows the error messages to the user:



Project failed to save

We can assert that this behaviour happens by writing another test for this in `spec/features/projecs/creating_spec.rb`, inside the `feature` block that already exists in that file:

spec/features/projects/creating_spec.rb

```
1  scenario "with no name specified" do
2    visit "/"
3
4    click_link "New Project"
5
6    click_button "Create Project"
7
8    expect(page).to have_content "Project could not be created."
9
10   within("h2") do
11     expect(page).to have_content("New Project")
12   end
13
14   expect(page).to have_content("There were errors that prevented this form from
```

```
15    being saved")
16    expect(page).to have_content("Name must be filled")
17  end
```

This test asserts that when a user goes to the new project page and submits the form without filling in the name field, that they're sent back to that form and told that the name is missing.

This test will fail at the moment when we run it because we're *always* creating projects in the `create` action, regardless of whether or not the project's parameters are valid or not. Here's the code from that action:

app/controllers/projects/projects_controller.rb

```
def create
  repo.create(project_params)
  flash[:notice] = "Project has been created."
  redirect_to action: :index
end
```

And here's the failure from the test currently:

```
1) Users can create new projects with no name specified
 Failure/Error: expect(page).to have_content "Project could not be created."
 expected to find text "Project could not be created."
   in "Project has been created. Projects New Project"
```

So we can see here that the `create` action is still setting the "Project has been created." flash message (as it appears in the output), and so the project would still be created by this action. Unsurprising, given that we haven't changed anything to do with this action yet.

There isn't a case here where `create` doesn't create a project if the name is missing. To do that, we'll need to use our new `ProjectSchema` validation schema. We'll change our `create` action to this:

app/controllers/projects/projects_controller.rb

```
def create
  validation = ProjectSchema.(project_params)
  if validation.success?
    repo.create(project_params)
    flash[:notice] = "Project has been created."
    redirect_to action: :index
  else
    @project = Projects::Project.new(project_params)
    @errors = validation.errors
    flash.now[:alert] = "Project could not be created."
    render :new
  end
end
```

We're using our `Projects::ProjectSchema` to validate if the project parameters are valid in this case. If they are, then we go down the path of creating the project and redirecting the user back to the `index` action. This is the same flow that we previously had in this controller action.

A new part of this flow is that if the parameters aren't valid, then we initialize a new `Projects::Project` instance using whatever parameters were passed in from the form – so the form is repopulated with its correct values, if any were specified. We also show the "Project could not be created." message, and render the `new` template once more. The new template will then display these error messages from the `@errors` object.

We'll need to make one more change to our application before our test will pass.

We'll need to display the errors from the `@errors` result in the `app/views/projects/projects/new` view, which is something that our test also checks for. We can do this by changing the view to this:

app/views/projects/projects/new.html.erb

```
<h2>New Project</h2>

<%= form_with(model: @project, local: true, scope: :project) do |form| %>
  <% if @errors.present? %>
    <p>
      There were errors that prevented this form from being saved:
    </p>

    <ul>
      <% @errors.map do |key, errors| %>
        <% errors.each do |error| %>
          <li><%= key.to_s.capitalize %> <%= error %></li>
        <% end %>
      <% end %>
    </ul>
  <% end %>
  <p>
    <%= form.label :name %>
    <%= form.text_field :name %>
  </p>

  <%= form.submit class: "btn btn-primary" %>
<% end %>
```

This view now detects if there are any `@errors` and if there are it will go through the error messages from that validation and present them. This code for checking for errors is generic enough that we can consider moving it out to a partial just to tidy up our form. We may wish to use it in other forms later:

app/views/shared/_error_messages.html.erb

```
<% if errors.present? %>
  <p>
    There were errors that prevented this form from being saved:
  </p>

  <ul>
    <% errors.map do |key, messages| %>
      <% messages.each do |error| %>
        <li><%= key.to_s.capitalize %> <%= error %></li>
      <% end %>
    <% end %>
  </ul>
<% end %>
```

Then we can use this partial in the form like this:

app/views/projects/projects/new.html.erb

```
<h2>New Project</h2>

<%= form_with(model: @project, local: true, scope: :project) do |form| %>
  <%= render "shared/error_messages", errors: @errors %>

  <p>
    <%= form.label :name %>
    <%= form.text_field :name %>
  </p>

  <%= form.submit class: "btn btn-primary" %>
<% end %>
```

It *feels* like all of the parts are configured correctly, so let's run this test and see if that is true. We'll run `bundle exec rspec spec/features/projects/creating_spec.rb` and see if it passes:

```
2 examples, 0 failures
```

Excellent! Our controller now uses the `Projects::ProjectSchema` to validate the project's parameters before the project is created in the database. If the project is invalid, errors are shown to the user telling them what to do. And all this is done with code that is neatly separated. We still haven't had to add anything extra to our `Projects::Project` model. It's still devoid of validation or persistence logic.

However, our controller is getting a little messy. It's intertwining the validation and persistence logic with the response logic. It's starting to feel very Rails-y, and I mean that in a bad way.

We can do better than this by separating out these three distinct responsibilities even further.

Let’s talk about service objects

The `create` action in `Projects::ProjectsController` is now messy:

app/controllers/projects_controller.rb

```
def create
  validation = Projects::ProjectSchema.(project_params)
  if validation.success?
    project = repo.create(project_params)
    flash[:notice] = "Project has been created."
    redirect_to project
  else
    @project = Projects::Project.new(project_params)
    @errors = validation.errors
    flash.now[:alert] = "Project could not be created."
    render :new
  end
end
```

This action muddles together validation, persistence and response logic. If we were to color in the different types of logic in this action, it would look like this:

app/controllers/projects_controller.rb

```
def create
  validation = ProjectSchema.(project_params)
  if validation.success?
    repo.create(project_params)
    flash[:notice] = "Project has been created."
    redirect_to projects_path
  else
    @project = Ticketee::Project.new(project_params)
    @errors = validation.errors
    flash.now[:alert] = "Project could not be created."
    render :new
  end
end
```

Color coded action

There should be a better way to do this than to just put everything into the one action. We could move this code out somewhere, but where?

Normally when moving code out of a controller to separate the different logic parts, you would move some parts out to a service object. It's "community tradition" to move these parts out to *service objects* to make it easier to test those parts in a way that's completely isolated from the Rails request / response cycle. It's an attempt to let the controller handle the request / response cycle only. Single responsibility and all that.

However, most of the time this happens it is simply cutting the code from the controller and pasting it into another class. This code ends up in a new directory called `app/services`, and the guidelines for organizing this directory differ wildly from application to application. Unlike controllers, models and views, there is no one standard community consensus on how a service object is constructed,

what its structure looks like, or even where it is stored within a Rails application.

Here's what might happen with a service object that we create from the code that we had in the `create` action of `Projects::ProjectsController`, perhaps putting this code in `app/services/projects/create.rb`:

app/services/projects/create.rb

```
class Projects::Create
  def self.call(params)
    @validation = ProjectSchema.(params)
    if @validation.success?
      project = repo.create(project_params)
      [true, {project: project}]
    else
      @project = Projects::Project.new(project_params)
      [false, {project: project, errors: @validation.errors}]
    end
  end
end
```

This definitely isn't following the Rails "convention over configuration" mantra. In fact, it reminds me a little of the custom code that I might have written in a PHP application circa-2006. A good service object should follow some sort of convention or design pattern, but this feels like it is just following the direction the wind is blowing in.

If we had to add another responsibility to this service such as one that made the success path send emails, we'd just cram it on in there:

app/services/projects/create.rb

```
class Projects::Create
  def self.call(params)
    @validation = ProjectSchema.(params)
    if @validation.success?
      project = repo.create(project_params)
      send_emails(project)
      [true, {project: project}]
    else
      [false, {errors: @validation.errors}]
    end
  end

  def self.send_emails(project)
    # code to send emails goes here
  end
end
```

The complexity increases. Adding another responsibility to this class only made it *worse*. Nevertheless, let's persist with this for a moment.

This class would then be used in a controller like this:

```
def create
  success, result = Projects::Create.call(project_params)
  if success
    flash[:notice] = "Project has been created successfully"
    redirect_to result[:project]
  else
    @project = Projects::Project.new(project_params)
    @errors = result[:errors]
    flash[:alert] = "Project could not be created"
    render :new
  end
end
```

Suddenly, the whole combination of the service object and controller code isn't looking that pretty. The service object muddles together validation, persistence, sending emails and returning a result all in the one method. Testing the individual parts will be difficult due to the design of this method. If this service object is to grow, it will only get messier and messier.

What would be better is something that would allow is to explode the call method into a chain of connected methods. The methods should follow this order:

1. Validate the parameters for a new project
2. Persist the record to the database
3. Send emails
4. Return a valid result

However, if step #1 failed then we would not expect step #2 and step #3 to run. Similarly, if step #2 failed then we would not expect step #3 to run, and so on.

We could try writing this logic ourselves using convoluted `if` statements, but there is a better way: the `dry-transaction` gem. This gem will be the focus of this chapter.

Here's an example of the above service object, written in a `dry-transaction` way:

```ruby
require "dry/transaction"

module Projects
  class Create
    include Dry::Transaction

    step :validate
    step :persist
    step :send_emails

    def validate(input)
      validation = ProjectSchema.(input)
      if validation.success?
        Success(input)
      else
        Failure(validation.errors)
      end
    end

    def persist(input)
      project = project_repo.create(input)

      Success(project)
    end

    def send_emails(project)
      # code to send emails

      Success(project)
    end

    private

    def project_repo
```

```
      ProjectRepository.new(ROM.env)
    end
  end
end
```

This class represents a *transaction* a user undertakes with your application, hence the name `dry-transaction`. It's still a service object, but it's a *cleaner* service object.

In fact, let's stop calling them "service objects" and start referring to them as "transactions". After all, these classes represent the logic your application runs through when someone interacts (or transacts) with your application. "Service object" is a term too-diluted by the various uses in a Rails application. "Transaction" fits so much better[30].

This example transaction class separates out each step into this own clearly defined method, and we define the order of these steps at the top of the class by using the `step` method:

```
step :validate
step :persist
step :send_emails
```

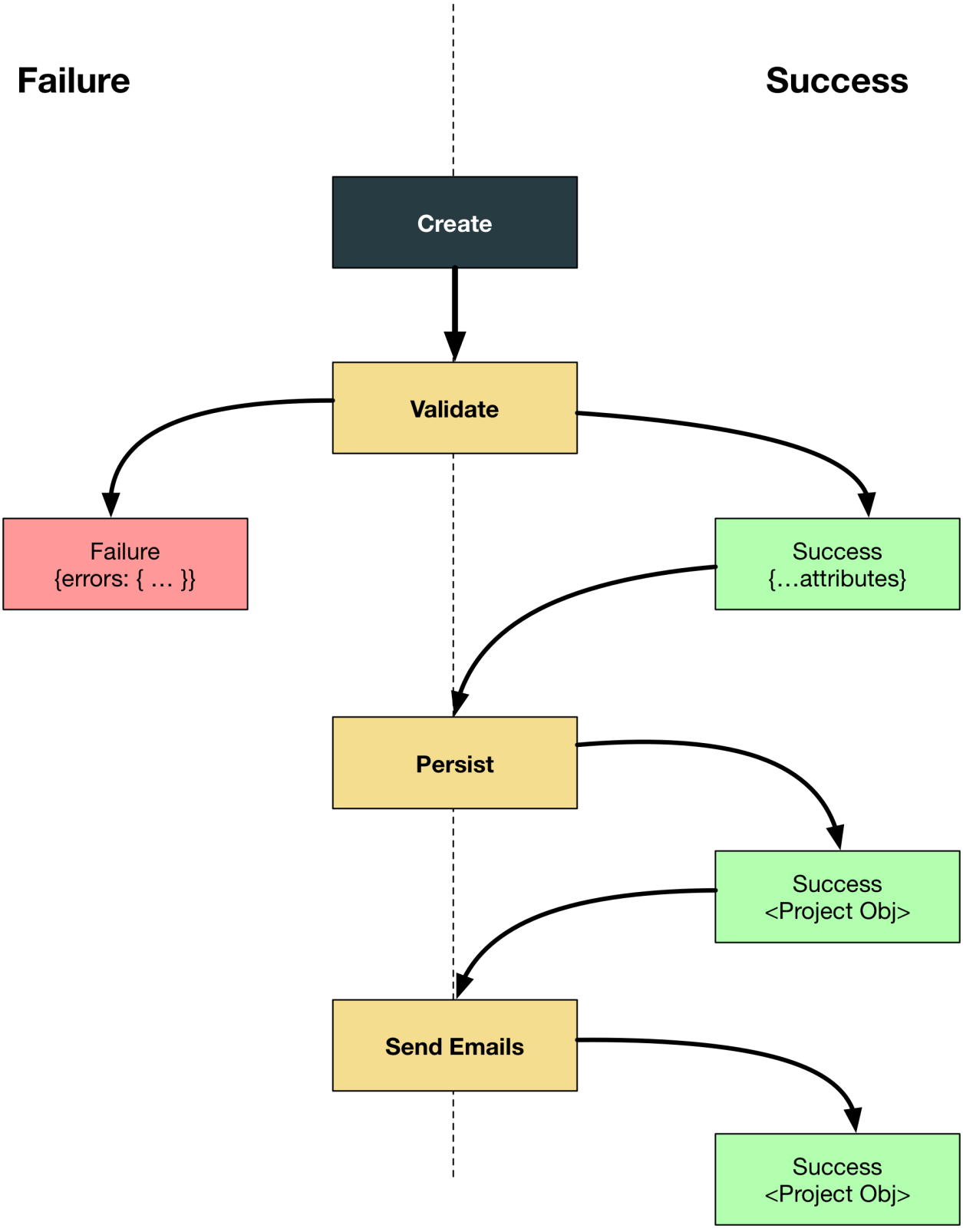
This will mean that when our transaction runs, it will run the `validate` method first, then the `persist` and the `send_emails`.

The `Success` or `Failure` constants used inside these steps come from another `dry-rb` gem called `dry-monads` (don't let the name scare you), and they indicate

[30] Although this term "Transaction" is used prominently in databases too. For instance, here's PostgreSQL's tutorial for its transactions: https://www.postgresql.org/docs/10/static/tutorial-transactions.html. When talking about the word "transaction" when we're using dry-transaction, it might be helpful to talk about them as "*application* transactions" and "*database* transactions", just to be that little bit more explicit about exactly what kind of transaction we're talking about. (Thanks to David Carlin for raising this point with me during his reading of this book.)

if the method is successful or not. If the `validate` step returns a `Failure` then the `persist` step will not be called; `dry-transaction` handles that for us automatically.

The `Success` result gets passed to the next step, but `Failure` stops the code in its tracks.



Success or failure

Unlike our service object from before, if we added another step in here it would simply mean another line in the `step` definitions, and another method in the class. It is then very clear the order the methods run in and the new method would likely be simple too.

Further to this, if you wanted to test what `validate` did when it received certain parameters, you can now do that. If you wanted to test the `persist` step, you can now do that too. Previously, it was impossible to test each step of our `Projects::Create` service because all the logic was combined into one method called `call`. To achieve this same goal of testability and separate-ness in our old `Projects::Create` class without the use of the `dry-transaction` or `dry-monads` gem, we would need to write our code like this:

```
class Projects::Create
  def call(params)
    [success, result] = validate(params)
    return result unless success

    [success, result] = persist(result)
    return result unless success

    send_emails(project)
  end

  def validate(params)
    project = Projects::Project.new(params)
    if project.valid?
      [true, project]
    else
      [false, project.errors]
    end
  end

  def persist(project)
```

```
    project.save
    [true, project]
  end

  def send_emails(project)
    # code to send emails

    [true, project]
  end
end
```

This code is gross and really hard to mentally parse. Your eyeballs dart up and down the code to try to figure out what it is doing. It doesn't follow any convention or any common design pattern. It doesn't even come close to the cleanliness of the `dry-transaction` code.

It's for this reason I think that `dry-transaction` should be used as a pattern for <s>service objects</s>, sorry I mean *transactions* in Rails applications. The cleanliness of the class is definitely worth it. The testability of each steps inside the transaction is dead-simple too.

In this book, we'll be using `dry-transaction` to build more service objects like these. We'll end up with some really slick looking code in our controllers:

```
def create
  action = Projects::Create.new
  action.(project_params) do |result|
    result.success do |project|
      flash[:notice] = "Project has been created."
      redirect_to project
    end

    result.failure :validate do |errors|
      @project = Projects::Project.new(project_params)
      @errors = errors
      flash[:alert] = "Project could not be created."
      render :new
    end
  end
end
```

This code is not too far off from the service object code from before, but under the hood it will use `dry-transaction` to abstract the logic out of the controller, away from the request / response cycle and make our application code that much cleaner. We'll cover what this code does when we get to the real example. For now, I wanted to just show the "magic trick" before explaining how it works.

Building our first transaction class

OK, enough talking about it. Let's actually do it! The first step is to add `dry-transaction` to our `Gemfile`:

```
gem 'dry-transaction', '~> 0.13.0'
```

And then to run `bundle install` to install this gem.

We'll build this class up piece-by-piece, just to demonstrate how easy it is to test each part individually. As we saw before, this class will have two distinct

steps: `validate` and `persist`. The `validate` step will wrap all the logic for validation and all the `persist` step will wrap all the logic for persisting the record to the database.

Let's start by writing some tests for the `validate` step:

spec/transactions/projects/create_spec.rb

```
1  require "rails_helper"
2
3  describe Projects::Create do
4    context "validate" do
5      context "when validation is successful" do
6        it "returns the project parameters" do
7          input = { name: "Test Project" }
8          result = subject.validate(input)
9          expect(result).to be_success
10         expect(result.value).to eq(input)
11       end
12     end
13   end
14 end
```

This test asserts that when the `validate` method is called on our `Projects::Create` transaction class that it is successful, and the return value from the result is the input parameters.

Let's setup our `Projects::Create` class now and add in some code to make this test pass:

app/transactions/projects/create.rb

```
require 'dry-transaction'

module Projects
  class Create
    include Dry::Transaction

    step :validate

    def validate(input)
      validation = ProjectSchema.(input)
      if validation.success?
        Success(input)
      end
    end
  end
end
```

We've placed this class at `app/transactions`, because this class represents a *transaction* that a user can undertake within our application. The main type of object that this transaction focuses on is a project, we call the subdirectory inside `app/transactions` "projects". And `create.rb` should be pretty obvious because this particular transaction will create a project within our application.

This directory structure lends itself to helping others understand how our application is used. They can look through the list of files and see the transactions that a user could perform in our application.

In this class, we define our first step: `validate`. Inside that step, the only thing we do is call out the `ProjectSchema` and determine if the input for this step – the parameters coming from the controller – would be valid or invalid according to that schema. If they're valid, we return `Success(input)`. We don't have a test for invalid parameters yet, so we don't have any code for that either.

Running the test with `bundle exec rspec spec/transactions/projects/create_spec.rb` will show that it works:

```
1 example, 0 failures
```

The code for our test and the code for this transaction is dead simple. There's no muddling of validation, persistence and business concerns. It's all about the validation here.

Let's write another test, this time for the failure case of the validation:

spec/transactions/projects/create_spec.rb

```
context "when validation fails" do
  it "returns validation errors" do
    input = { }
    result = subject.validate(input)
    expect(result).to be_failure
    expect(result.value).to eq({name: ["is missing"]})
  end
end
```

For this test to pass, we'll need to handle the validation failing in our transaction class. We'll change the `Projects::Create#validate` method to this:

app/transactions/projects/create.rb

```
def validate(input)
  validation = ProjectSchema.(input)
  if validation.success?
    Success(input)
  else
    Failure(validation.errors)
  end
end
```

In the case where the validation is unsuccessful, we're returning `Failure` result and that result contains the validation's errors. We're returning the errors rather than the input as the controller will use these to show the user which attributes were wrong according to the schema. Also, it's worth remembering here that by returning `Failure` rather than `Success`, the transaction will not process any further steps. The result of the transaction will be the `Failure(validation.errors)` result.

When we run that test again, we'll see that it passes:

```
2 examples, 0 failures
```

We've now written the first of the two steps that we need, so let's now look at the 2nd step. This step is easier to test as it can only be successful. Let's write this test now:

spec/transactions/projects/create_spec.rb

```
context "persist" do
  let(:project_repo) { double(ProjectRepository) }
  before { allow(subject).to receive(:project_repo) { project_repo } }

  it "step is successful" do
    project = double(:project)
    expect(project_repo).to receive(:create).with(name: "Test Project") { project }
    subject.persist({name: "Test Project"})
  end
end
```

In this test, we're using a double for the repo because we don't want to make this test depend on a real database. It's enough to stub out that particular dependency. The test only cares that there's *something* called `repo` and that the `repo` receives a `create` method with the attributes when `persist` is called.

Let's write the code to make this test pass, putting these new methods directly underneath `validate`:

app/transactions/projects/create.rb

```
def persist(input)
  project = project_repo.create(input)

  Success(project)
end

private

def project_repo
  ProjectRepository.new(ROM.env)
end
```

The `persist` method only needs to call `create` on `repo`. The result of that call gets wrapped in `Success` and returned from this step.

We've defined the `repo` method here also because our class needs the repository in order to create the project.

To make our `Projects::Create` use this step, we'll need to specify a `step` for it near the top of the class. This `persist` should run after the `validate` step, so we'll put the two steps in this order:

```
step :validate
step :persist
```

Running the test again with `bundle exec rspec spec/transactions/projects/create_spec.rb` will show all three examples passing:

```
3 examples, 0 failures
```

This `Projects::Create` class encapsulates all the steps for creating a project within our application. It's like a traditional service object, but with the clear separation between the individual methods in this class it makes it much easier to read and understand. This `Projects::Create` class knows nothing about how it is used; it could be used as a part of a gem's CLI tool, or as a part of a Rails request. It doesn't care one way or the other.

Before we continue, I should mention that there is a shorter way to write the `persist` step in this class. We can do this by using the `map` step class method for `persist`, like this:

app/transactions/projects/create.rb

```
1 step :validate
2 map :persist
```

By using `map`, we're telling the transaction class that we care about the output of `validate` and that we should bail if it fails, but for `persist` there's no failure case. The `map` method here instead will automatically wrap the result of the `persist`

method in a `Success` case. This effectively implements what we have already, but in less code. By using `map`, it means we can shorten the code in `persist` to this:

app/transactions/projects/create.rb

```
def persist(input)
  project_repo.create(input)
end
```

If we run our tests yet again, we'll see that everything is still working:

```
3 examples, 0 failures
```

We've tested the individual steps of this transaction class, but what if we wanted to test that the whole thing worked, end-to-end? Well, we can certainly write a test for that too, if we wished:

spec/transactions/projects/create_spec.rb

```
context "the whole thing" do
  let(:project_repo) { double(ProjectRepository) }
  before { allow(subject).to receive(:project_repo) { project_repo } }

  context "when valid parameters are given" do
    let(:input) { { name: "Test Project" } }

    it "creates a project" do
      expect(project_repo).to receive(:create).with(input)
      result = subject.(input)
      expect(result).to be_success
    end
  end

  context "when invalid parameters are given" do
    let(:input) { { } }
```

```
17
18      it "fails to create a project" do
19        expect(project_repo).not_to receive(:create).with(input)
20        result = subject.(input)
21        expect(result).to be_a_failure
22      end
23    end
24  end
```

We're testing the two paths that the input can take through our transaction here:

1. The input is *valid* – according to the `validate` step – and so the project is created by the `persist` step.
2. The input is *invalid* – according to the `validate` step – and so the project is *not* created.

If we run these tests, we'll see that they already pass thanks to the work we've done previously:

```
5 examples, 0 failures
```

These are some pretty robust tests on our `Projects::Create` transaction class. We've now been able to test each step of our transaction individually, as well as being able to test the whole. This is more than you can say for a lot of regular service objects!

Integrating the Projects::Create class with the controller

The `Projects::Create` class is now fully formed and we can now use this in the `create` action for `Projects::ProjectsController`. Let's turn that `create` action into

this:

app/controllers/projects/projects_controller.rb

```
def create
  transaction = Projects::Create.new
  transaction.(project_params) do |result|
    result.success do |project|
      flash[:notice] = "Project has been created."
      redirect_to project
    end

    result.failure :validate do |errors|
      @project = Projects::Project.new(project_params)
      @errors = errors
      flash[:alert] = "Project could not be created."
      render :new
    end
  end
end
```

This controller action now calls out to the transaction and the transaction handles all the responsibility of validating and persisting the project. The controller doesn't know how that validation or persistence happens. What the controller knows now is that the transaction can be successful or it can fail at the `validate` step.

If the transaction is successful, then the user will see the "Project has been created." message and be redirected to the `show` action for that new project.

If the transaction fails, but only if it fails during the validate step, then the user will see the form once again and be shown the errors from the validation step.

Writing the action in this way means that we are given the opportunity to handle different failure cases individually. For instance, if we had another step where

this could fail, we could present the user with a more specific error message for that error or perhaps notify our bug tracking app about that particular path.

With this change to our controller's action we've simplified the controller code massively. But do the tests still work? Let's run them with `bundle exec rspec` to find out:

```
10 examples, 0 failures
```

All of our application still works, and that includes the two feature tests in `spec/features/projects/creating_spec.rb` which test how the user interacts with our application. These tests act as integration tests, ensuring that all our different parts from the controller down to the relation work together in harmony.

If at any point we detect a bug in what we've done, we *could* write a regression test as a feature test to cover that edgecase. But we could also write a regression test as a unit test to cover that too, since the bug is very likely to be in one of the small classes or methods we've defined. By exploding the responsibilities of our Rails application into these smaller classes, we've made it so much easier to reason about what each individual part is doing.

Refactoring the transaction class one step further

Before we round out this chapter, I'd like to take the time to clean up the `Projects::Create` class a little more. We've spent this chapter so far moving the logic from the create action of `ProjectsController` out to `Projects::Create` class, and our `create` action is all the better for it. But our `Projects::Create` class isn't quite perfect yet. The `Projects::Create` class knows too much.

It knows how to validate and persist the input, and those things are fine. What could be better is that this class knows how to initialize an instance of the `ProjectRepository` class. This is a step too far, in my opinion. This class

should know how to validate and persist objects, but not how to initialize that persistence layer as well. That should be the responsibility of something "higher" up, to give this class fewer reasons to change. We can probably move this responsibility elsewhere.

To do this, we can use *dependency injection*.

Introducing dependency injection

The [Wikipedia description for Dependency Injection] is a good one:

> In software engineering, dependency injection is a technique whereby one object (or static method) supplies the dependencies of another object. A dependency is an object that can be used (a service). An injection is the passing of a dependency to a dependent object (a client) that would use it. The service is made part of the client's state.[1] Passing the service to the client, rather than allowing a client to build or find the service, is the fundamental requirement of the pattern.

In more concrete terms that are relevant to our application, in order to make `Projects::Create` aware of a `ProjectRepository` instance without that class initializing one itself, we can use dependency injection.

A common way to do this would be to define an `initialize` method at the top of the class, like this:

app/transactions/projects/create.rb

```
def initialize(project_repo:)
  @project_repo = project_repo
end
```

And then in our controller, when we initialize the `Projects::Create` class we can pass it the dependency then and there:

app/controllers/projects/projects_controller.rb

```
def create
  transaction = Projects::Create.new(project_repo: ProjectRepository.new(ROM.env))
```

But then this runs into the same problem as before: the `ProjectsController` knows how to initialize this repository. And what happens if we need *other* things that use this repository, and then what if those ways of initializing that repository change?

What we need is one central, clean way to do this.

Thankfully, the `dry-rb` gems provide another *two* gems to help us along this path: `dry-container`[31] and `dry-auto_inject`[32]. These gems make it possible for us to inject dependencies into our classes, and also to have a central place to track what those dependencies are.

To get started, we will add these gems to our Gemfile:

Gemfile

```
gem 'dry-container', '0.6.0'
gem 'dry-auto_inject', '0.4.6'
```

Then we'll run `bundle install` to install these gems.

[31] http://dry-rb.org/gems/dry-container
[32] http://dry-rb.org/gems/dry-auto_inject/

To get started, we will need to create a container. This container will be used to register our application's dependencies. We'll create a new file in `config/initializers/contain` for this purpose:

config/initializers/container.rb

```
module Ticketee
  class Container
    extend Dry::Container::Mixin
  end
end
```

The `Dry::Container::Mixin` mixin here adds two methods to our `Ticketee::Container` class, `register` and `resolve`. We can register things inside this container with this code:

config/initializers/container.rb

```
module Ticketee
  class Container
    extend Dry::Container::Mixin
  end

  Container.register(:project_repo, -> { ProjectRepository.new(ROM.env) })
end
```

Once we've registered something with this container, we can use `resolve` to get it. Try this out in a `rails console` session:

```
Ticketee::Container.resolve(:project_repo)
=> #<ProjectRepository struct_namespace=Projects auto_struct=true>
```

We could use this to resolve a `ProjectRepository` instance in the `Projects::ProjectsController`, but having this code in our controller is going to look a bit long:

app/controllers/projects/projects_controller.rb

```
def create
  transaction = Projects::Create.new(project_repo: Ticketee::Container.resolve(:proj\
ect_repo))
```

So far, we've only used the `dry-container` gem. We can turn this line in our controller back into its simplest version by using the other gem we installed, `dry-auto_inject`. It requires a little bit more setup in our new initializer:

config/initializers/container.rb

```
module Ticketee
  class Container
    extend Dry::Container::Mixin
  end

  Container.register(:project_repo, -> { ProjectRepository.new(ROM.env) })

  Import = Dry::AutoInject(Container)
end
```

By defining the `Import` constant here, we make use of the `Dry::AutoInject` class. Well, part of the way at least. To really show what this accomplishes, we need to make a use of the `Import` class. Let's do that now in our `Projects::Create` class, like this:

app/transactions/projects/create.rb

```
module Projects
  class Create
    include Dry::Transaction
    include Ticketee::Import["project_repo"]
```

And now we can get rid of the `project_repo` method at the bottom of the class altogether, because it will be defined by this `include` statement instead.'

We're now injecting the `project_repo` dependency into this class by way of the `include Ticketee::Import["project_repo"]` line at the top of the class. This makes a method called `project_repo` available to instance methods of our class.

Now that we're injecting this dependency into the `Projects::Create` class, we can revert our `Projects::ProjectsController`'s `create` method back to its original state:

app/controllers/projects/projects_controller.rb

```
def create
  transaction = Projects::Create.new
```

The controller absolutely has no knowledge of the `project_repo` dependency. That's for the `Projects::Create` class and `Ticketee::Container` to know.

There's one final change to make. Our test for this class is still stubbing a method called `project_repo`. Rather than stubbing this method, we can inject it as a dependency thanks to the changes we've just made. We will remove this line:

spec/transactions/projects/create_spec.rb

```
1 before { allow(subject).to receive(:project_repo) { repo } }
```

And replace it with:

```
subject { described_class.new(project_repo: project_repo) }
```

This works because the `include Ticketee::Import[:project_repo]` method overrides the `initialize` method of this class to take keyword arguments that match what has been imported. This means that in our tests we can choose to inject a different dependency than the one that would be automatically injected.

If we run our tests once again, we'll see they're all still passing:

```
10 examples, 0 failures
```

We've now used dependency injection to abstract the responsibility of initializing a new `ProjectRepository` instance from the `Projects::Create` class. This class now only has the barest responsibilities it needs to have. In addition to this, if we were to change somehow how our `ProjectRepository` was initialized, we will now only ever need to do that in one place.

Now that we've got a couple of features for creating and viewing projects established in our application, let's circle back around to that `Project#contributors` method.

Adding tickets and users

In this chapter, we'll add two more models to our application: `Ticket` and `User`. We'll use these models to implement our `contributors` method on the `Project` class. Finally!

We've talked about having a `Project` model and a `Ticket` model in our Rails application ever since that `contributors` method was shown for the first time:

```
class Project < ApplicationRecord
  has_many :tickets

  def contributors
    tickets.map(&:user).uniq
  end
end
```

But we've spent a lot of time on just the `Project` model, and its related transaction and controller classes.

So let's come back to this `Project#contributors` method in this chapter now, and finish what we started. We're going to spend this entire chapter on making it possible to fetch contributors for a particular project.

Let's talk first about how this would work in a regular Rails application. For this `tickets` method to work in the `Project` model, we need the `has_many :tickets` method defined at the top. We will also need a `Ticket` model.

That `Ticket` model would be responsible for fetching rows from a `tickets` table, validating them and any business logic involving a ticket. In a normal Rails application, we could generate it with `rails g model`, but we have no such luck here in our ROM-powered application.

In our ROM-powered application to define such an association and to work with tickets we'll have to generate a few things:

- A repository class to provide methods to our application that can be used to perform CRUD operations on tickets.
- A Relation class to wrap the logic of talking to our `tickets` table.
- A migration to create this `tickets` table.
- A Model class to represent the data of each ticket.

While we're at it, we'll also generate the same things for a `User` model, which is what will be returned by that `contributors` method.

Once we have these things, we will look at how we can add a method to our application that will allow us to fetch the contributors for a project. This method will return a list of users who have filed at least one ticket on a project. And this method isn't going to go in the `Project` model, or even the `ProjectRepository`. Read on to find out where it goes.

Let's start by generating the few files that we need.

Creating a tickets repository, relation, model and migration

The first thing we'll generate is the repository for tickets. We'll need to generate this so that we can create tickets within our application. We could make database calls to do the same thing, but because we've been using Ruby so far, let's keep using Ruby.

We'll generate the repository with this:

```
rails g rom:repository ticket
```

This generates the following class:

app/repositories/ticket_repository.rb

```
class TicketRepository < ROM::Repository::Root
  root :tickets

  commands :create, update: :by_pk, delete: :by_pk

  struct_namespace Ticketee
end
```

We'll need to change this repository to use the `Projects` namespace, by changing the `struct_namespace`, just like our `ProjectRepository` does:

app/repositories/ticket_repository.rb

```
class TicketRepository < ROM::Repository::Root
  root :tickets

  commands :create, update: :by_pk, delete: :by_pk

  struct_namespace Projects
end
```

The `struct_namespace` call here will make the repository try to find a class called `Projects::Ticket` to use to represent data returned from this repository. We'll get around to creating that in a short while. We're putting tickets under this namespace because tickets will only ever be referenced within the context of a project.

For this repository to be able to interact with the database, we'll need to create a relation, which we can do with this command:

```
rails g rom:relation tickets
```

This generate the relation class at `app/relations/tickets_relation.rb`:

app/relations/tickets_relation.rb

```
1  class TicketsRelation < ROM::Relation[:sql]
2    gateway :default
3
4    schema(:tickets, infer: true)
5
6    # define your methods here ie:
7    #
8    # def all
9    #   select(:id, :name).order(:id)
10   # end
11 end
```

Next, up we'll generate the migration:

```
rake db:create_migration[create_tickets]
```

We'll need to populate this migration with data about our `tickets` table. For now, we will just have a few fields: `title`, `comment` and `project_id`. Let's update that migration file now:

db/migrate/[timestamp]_create_tickets.rb

```
ROM::SQL.migration do
  change do
    create_table :tickets do
      primary_key :id
      String :title
      String :comment
      foreign_key :project_id, :projects
    end

    add_index :tickets, :project_id
  end
end
```

The `foreign_key` method here makes this `project_id` field an integer, but also enforces that the `project_id` *must* contain an ID from the `projects` table; also known as a *foreign key constraint*. This is just to ensure that there's no chance of tickets being created in the `tickets` table without them linking to a project.

The `add_index` line does exactly what it does in a Rails application: it tells the database to add an index on the `tickets` table for the `project_id` column. This will make looking up groups of tickets by their `project_id` much faster.

Let's run this migration with these commands:

```
rake db:migrate
RAILS_ENV=test rake db:migrate
```

Now that we have the migration, we've got one more step to go: the model. We'll manually create this model class by creating a new file at `app/models/projects/ticket.rb`:

app/models/projects/ticket.rb

```
module Projects
  class Ticket < ApplicationModel
  end
end
```

We've placed this model underneath the `Projects` namespace because both projects and tickets belong in the "projects" feature of our application. At this point, it doesn't make much sense to have both a `Projects` and `Tickets` namespace, as tickets will only ever be viewed within the context of a particular project. Therefore, the `Projects` namespace makes the most sense.

Next, we should make sure all of this works together. The best way to do that would be to create a new ticket through our Rails console, as that will test the repository, relation, mapper and model all together. We'll try and associate this ticket with a project too. Let's open up `rails console` and try this now:

```
project_repo = ProjectRepository.new(ROM.env)
ticket_repo = TicketRepository.new(ROM.env)

project = project_repo.create(name: "Exploding Rails")
ticket = ticket_repo.create(
  title: "First errata",
  comment: "I found a boog",
  project_id: project.id
)
```

This should show us our new ticket:

```
=> #<Projects::Ticket id=1 title="First errata" comment="I found a boog" project_id=\
1>
```

Great! Everything is working together nicely.

But in order to list the contributors for the project we're going to need to add one more model: a `User` model. So let's quickly go through the steps we've just gone through, but this time we'll setup all the things for a `User` model as well.

Creating all the parts for users

We're now going to quickly create all the necessary parts for the `User` model in our application. Let's start with the repository, because again we're going to use Ruby to create users in our application.

```
rails g rom:repository user
```

This generates the following class:

app/repositories/user_repository.rb

```
1 class UserRepository < ROM::Repository::Root
2   root :users
3
4   commands :create, update: :by_pk, delete: :by_pk
5
6         struct_namespace Ticketee
7 end
```

And we'll change the `struct_namespace` too, but this time we'll call it `Users`:

app/repositories/user_repository.rb

```
class UserRepository < ROM::Repository::Root
  root :users

  commands :create, update: :by_pk, delete: :by_pk

  struct_namespace Users
end
```

The `User` model is going to exist in a separate namespace to `Project` and `Ticket`, called `Users`. Anything to do with users will go into this namespace, such as `Users::UsersController`.[33]

"account", which was the "parent" of projects. An account would have many projects, and each account would not be able to see another account's projects. In this setup, it might make sense to group the `Account` and `User` underneath a namespace called `Accounts`, since controllers under that namespace are managing things to do with an account. Since we do not have an `Account` model in our application yet, the `Users` namespace makes more sense.

Next up, we'll need the relation:

```
rails g rom:relation users
```

This generate the relation class at `app/relations/users_relation.rb`:

[33] In a larger application, we might have something called an

app/relations/users_relation.rb

```
class UsersRelation < ROM::Relation[:sql]
  gateway :default

  schema(:users, infer: true)

  # define your methods here ie:
  #
  # def all
  #   select(:id, :name).order(:id)
  # end
end
```

Then, we'll generate the migration:

```
rake db:create_migration[create_users]
```

In this migration, we want to add in some fields for identifying users. We'll just add a `username` field now. We'll also add the `user_id` field to the `tickets` table while we're inside this migration:

db/migrate/[timestamp]_create_users.rb

```
ROM::SQL.migration do
  change do
    create_table :users do
      primary_key :id
      String :username
    end

    alter_table :tickets do
      add_foreign_key :user_id, :users
    end
  end
end
```

We're using `add_foreign_key` here to ensure that tickets can't be created without a linked user record existing. This is similar to the other foreign key that we setup for this table: `project_id`. Just like how the database would refuse to create a ticket without a linked project, this new one ensures that tickets couldn't be created without a linked user record. For a ticket to be created after this migration runs, both the associated project and user must exist first.

Let's run this migration with these commands:

```
rake db:migrate
RAILS_ENV=test rake db:migrate
```

Now that we have the migration, we only need to generate the model. Let's create a new file:

app/models/users/user.rb

```
1 module Users
2   class User < ApplicationModel
3   end
4 end
```

This is much the same as our models of the past: it inherits from `ApplicationModel`. We've put this inside the `Users` namespace to keep this model separate from the `Project` and `Ticket` models, because the `User` model will often be used separately from those models.

Why not Devise?

You might be thinking at this point: why hasn't Ryan used Devise to generate the User model? Isn't that the cool thing to do in Rails applications?

Well, yes it is. We *could* use Devise for this part and show the way to integrate Devise and ROM together. But I think it'd distract from the goal here of showing off the `contributors` method.

> I will leave it to the reader to attempt this instead.

Next, we should make sure all of this works together. The best way to do that would be to create a new user through our Rails console, just as we did after we created all the things for tickets. Let's give it a go:

```
user_repo = UserRepository.new(ROM.env)
user_repo.create(username: "radar")
```

If we see this then we know that it's working:

```
=> #<Users::User id=1 username="radar">
```

The user has been created in the database and a `Users::User` model instance is returned. While we're in the console, let's associate this new user with the ticket we just created by running this code in our console:

```
ticket_repo = TicketRepository.new(ROM.env)
ticket_repo.update(1, user_id: 1)
```

Our ticket will now "belong to" this new user. We'll be using this association in a short while to work out our version of the `contributors` method.

The contributors method

To refresh your memory, we want our `contributors` method to act like this:

app/models/project.rb

```
class Project < ApplicationModel
        has_many :tickets

        def contributors
                tickets.map(&:user).uniq
        end
end
```

We want this method to fetch all the users who have contributed to this project and by "contributed", we mean that they should have created at least one ticket within that project.

A quick re-cap of how it's done in vanilla Rails-land

For this `contributors` method to work in vanilla Rails-land, we would need to setup some associations on our models. In the `Project` model we would need to have a `tickets` association (as seen in the above example):

app/models/project.rb

```
class Project < ApplicationRecord
  has_many :tickets
end
```

And in the `Ticket` model we would need to have a `user` association:

app/models/ticket.rb

```
class Ticket < ApplicationRecord
  belongs_to :user
end
```

By calling `has_many` and `belongs_to` like this, it defines the `Project#tickets` and `Ticket#user` methods that we use in the `Project#contributors` method. To get the same thing to happen within our non-vanilla-Rails application models, we would also need to define a `tickets` method on the `Project` model and a `user` method on the `Ticket` model.

But how do we get the data from the database into those methods? We can't simply rely on those methods making database calls for us whenever we call them, like we can do with Active Record. Instead, the data would have to be populated into those methods at the same time as the other attributes. For instance, when we're in a situation where we want to display a project's contributors, we would need to load the project's information from the `projects` table *as well as* any related tickets from the `tickets` table. Then, when we load those tickets we would also need to load their related users too.

In a vanilla Rails application, to load all the data for a project, its related tickets and their related users all in one fell swoop, we would write a query like this:

```
Project.includes(tickets: :user).find(1)
```

This would do three queries:

```
Project Load (0.1ms)  SELECT  "projects".* FROM "projects" WHERE "projects"."id" = ?\
 LIMIT ?  [["id", 1], ["LIMIT", 1]]
Ticket Load (0.1ms)  SELECT "tickets".* FROM "tickets" WHERE "tickets"."project_id" \
= 1
User Load (0.2ms)  SELECT "users".* FROM "users" WHERE "users"."id" IN (1, 5)
```

It does one query to fetch the project, one query to fetch all the tickets related to that project, and then one query to fetch all the users for the tickets. This is that *eager loading* we've talked about before, where we're pre-emptively loading the data because we know we'll need it later.

We could also accomplish the same thing, but with `joins`:

```
Project.joins(tickets: :user)
```

This does one query instead of three:

```
SELECT  "projects".* FROM "projects"
INNER JOIN "tickets" ON "tickets"."project_id" = "projects"."id"
INNER JOIN "users" ON "users"."id" = "tickets"."user_id"
WHERE "projects"."id" = ? LIMIT ?
```

I won't debate the performance merits of `includes` over `joins` or vice versa here. My point here is that both queries give you the same result: a `Project` instance, which has a `tickets` method that *won't* call out to the database because its data is already loaded. And also because we've included the `:user` association here, calling `user` on a `Ticket` instance won't make a database call either.

OK great so we know how to solve this within a vanilla Rails application but this is *not* a vanilla Rails application we're in. It's a ROM-backed application and we need to reproduce this behaviour that we see with Active Record in ROM instead.

Building the ROM version of the contributors method

Inside our Rails application, we're fetching the users by going from the `Project` model, to the `Ticket` model to the `User` model, to end up with an array of `User` instances. But this feels backwards: the target table that we want to query is the `users` table, not the `projects` table, and so shouldn't the query be on the user side of things, and not the project side of things?

The simple English explanation of this is that we want "all users who have created tickets on a particular project". So based on that, it seems like it really makes sense to run a query on the `users` table to get the data we want. Let's investigate this avenue now.

We want to come up with a Ruby way to fetch all the users for our projects, but in order to get there it's a good idea to try experimenting with an SQL query first. Often, when working with `rom-sql`, the Ruby code isn't all that different looking from the SQL query. Once we've worked out a way to do it in SQL, we can easily convert that query into Ruby code.

So let's try it! A way to express this query in SQL would be this:

```
SELECT DISTINCT(users.id) FROM users
LEFT JOIN tickets on tickets.user_id = users.id
WHERE tickets.project_id = 1;
```

Let's break this query down.

We only want users to be returned *once* within our query, regardless of if they've filed one or more tickets on the project. To do this, we `SELECT DISTINCT(users.id) from users`: we select all the distinct `id` values from the `users` table returned in this query.

Next, we want to only return those users who've filed tickets on a particular project. But we don't have that information on the `users` table. The only link

the `users` table has to the `projects` table is through the `tickets` table. To include the `tickets` table data in this query, we issue a `LEFT JOIN`, joining the `tickets` table and the `users` table together in the same query.

The final line pulls only those `tickets` records that have a `project_id` equal to `1`.

And now how do we express this in Ruby code? Well, we can use Ruby methods that have the same name as the components of our SQL query. Let's try this code in a `rails console` session:

```
user_repo = UserRepository.new(ROM.env)
query = user_repo.users.distinct(:id)
query = query.left_join(:tickets, user_id: :id)
query = query.where(project_id: 1)
query.to_a
```

We could all of these methods on the same line, but I have split it out here for readability. On the first line, we initialize a new user repository. This is because we want to do the query on the relation that is linked to this repository, the `users` relation. We then get to that relation by calling `user_repo.users` on the second line, and then call `distinct` in order to make the query only return distinct responses. We can see that this is already working by the SQL query that shows up in our console after this line is entered:

```
=> #<UsersRelation
      name=ROM::Relation::Name(users)
      dataset=#<Sequel::Postgres::Dataset:
        "
          SELECT DISTINCT ON (\"id\") \"users\".\"id\", \"users\".\"username\"
          FROM \"users\"
          ORDER BY \"users\".\"id\"
          "
      >
    >
```

This SQL query is showing us what query would run if we were to *materialize* the results of this query, either by calling `to_a` or `one`. We will do that eventually, but for now we have some more method chaining to do on this query.

The next thing we do is to add the `LEFT JOIN` part of this query, which we can do with the `left_join` method on the relation. The arguments that we pass this method are the table name, and then a Hash where the key is the field we want to join to in that table, and the value is the field we want to join to from our original table. This turns our query into:

```
=> #<UsersRelation
      name=ROM::Relation::Name(users)
      dataset=#<Sequel::Postgres::Dataset:
        "
          SELECT DISTINCT ON (\"id\") \"users\".\"id\", \"users\".\"username\"
          FROM \"users\"
          LEFT JOIN \"tickets\" ON (\"tickets\".\"user_id\" = \"users\".\"id\")
          ORDER BY \"users\".\"id\"
          "
      >
    >
```

The final part of our query scopes the query to only match the tickets from the

right project. We add this part to our query by using the `where` relation method, passing it the `project_id` field name and then the value that we want this field to be on matching records.

This turns our query into its final result:

```
=> #<UsersRelation
      name=ROM::Relation::Name(users)
      dataset=#<Sequel::Postgres::Dataset:
        "
          SELECT DISTINCT ON (\"id\") \"users\".\"id\", \"users\".\"username\"
          FROM \"users\"
          LEFT JOIN \"tickets\" ON (\"tickets\".\"user_id\" = \"users\".\"id\")
          WHERE "project_id" = 1
          ORDER BY \"users\".\"id\"
          "
      >
    >
```

This looks close enough to our SQL query. In the original version we didn't select as many fields, but this should work even better. We do want those extra fields in this case, as we will want to display the usernames of the users that match this query.

The final line for our query is to *materialize* it, which will run the query and return us an array of `Users::User` instances:

```
=> [#<Users::User id=1 username="radar">]
```

There's only one that matches so far, and so only one will be returned. That's OK. If you'd like to, you could try to see what happens if you create more tickets with different users.

We now have a way to fetch all the contributors for a project, but reaching through the `UserRepository` instance to the `UsersRelation` instance is messy. We

should move this code into `UserRepository`.

Before we do that, we should write a test to make sure that this feature of the `UserRepository` works. We'll write this test in `spec/repositories/user_repository_spec.rb`:

spec/repositories/user_repository_spec.rb

```
require 'rails_helper'

describe UserRepository do
  let(:project_repo) { ProjectRepository.new(ROM.env) }
  let(:ticket_repo) { TicketRepository.new(ROM.env) }
  let(:user_repo) { described_class.new(ROM.env) }

  let!(:project_1) { project_repo.create(name: "Ticketee") }
  let!(:project_2) { project_repo.create(name: "Ticketee v2") }

  let!(:user_1) { user_repo.create(username: "annie") }
  let!(:user_2) { user_repo.create(username: "bob") }
  let!(:user_3) { user_repo.create(username: "charlie") }

  let!(:ticket_1) do
    ticket_repo.create(
      title: "First bug",
      comment: "This is the first bug",
      user_id: user_1.id,
      project_id: project_1.id,
    )
  end

  let!(:ticket_2) do
    ticket_repo.create(
      title: "Second bug",
      comment: "This is the second bug",
      user_id: user_1.id,
```

```
29        project_id: project_1.id,
30      )
31    end
32
33    let!(:ticket_3) do
34      ticket_repo.create(
35        title: "Third bug",
36        comment: "This is the third bug",
37        user_id: user_2.id,
38        project_id: project_1.id,
39      )
40    end
41
42    let!(:ticket_4) do
43      ticket_repo.create(
44        title: "First bug, 2nd project",
45        comment: "This is the first bug in the 2nd project",
46        user_id: user_3.id,
47        project_id: project_2.id,
48      )
49    end
50
51    context "contributors_for_project" do
52      it "finds contributors for the project" do
53        contributors = user_repo.contributors_for_project(project_1.id)
54        expect(contributors.count).to eq(2)
55        expect(contributors).to include(user_1)
56        expect(contributors).to include(user_2)
57        expect(contributors).not_to include(user_3)
58      end
59    end
60  end
```

This test is pretty extensive. It starts out by creating two projects, three users

and four tickets. What we want this test to assert is a few things:

1. When a project has multiple tickets from the same user, that does not mean that the user appears multiple times in the contributors list. In fact, they should only appear once. This is why the first 3 tickets exist.
2. When we ask the `contributors_for_project` method to do its thing, we only expect it to show us the contributors from the project that we asked for. This is why the 2nd project, 3rd user and 4th ticket exist. They shouldn't be included in this query.

Database calls in a test? Didn't you say that was bad, Ryan?

You might be reading this book because you watched my Exploding Rails[a] talk on YouTube. In this talk, I cover a lot of what I talk about in this book: what I think the problems are with Active Record and how Rails applications could be better architected for better future maintainability.

At about the 7 minute mark, I start talking about how Active Record kind of leads you into your only option being to write database calls to test a `contributors` method on a `Project` model, due to that `contributors` method using some Active Record querying methods.

But in this test, we're doing database calls in order to test the `contributors_for_project` method. Exactly the same thing, right?

Well, almost. This test that we've just written is less likely to change than an equivalent Active Record one. For instance, there's no callbacks or validations on our repositories that, if altered, might also alter the behaviour of our test.

In this architecture, we still have to create *some* records in the database to test database queries – like `contributors_for_project` – but those tests will be less likely to change due to external interference from model validations, callbacks

or other gremlins.

[a] https://youtu.be/04Kq_9scT1E

If we run this test with `bundle exec rspec spec/repositories/user_repository_spec.rb`, we'll see that the `contributors_for_project` method is missing from our `UserRepository`:

```
undefined method `contributors_for_project' for #<UserRepository ...>
```

Let's add this method to our repository now. It will do the same query that we used in the console earlier:

app/repositories/user_repository.rb

```
1  class UserRepository < ROM::Repository::Root
2    root :users
3
4    commands :create, update: :by_pk, delete: :by_pk
5
6    struct_namespace Users
7
8    def contributors_for_project(project_id)
9      users.contributors_for_project(project_id).to_a
10   end
11 end
```

This method in our repository calls the same method on `users`, which is a `UsersRelation` object. We'll need to define the actual "brains" of this query in the relation:

app/relations/users_relation.rb

```
class UsersRelation < ROM::Relation[:sql]
  gateway :default

  schema(:users, infer: true)

  def contributors_for_project(project_id)
    distinct(:id)
      .left_join(:tickets, user_id: :id)
      .where(project_id: project_id)
  end
end
```

This code belongs in the relation and not in the repository because the relation is where we put our database query code. The repository's job is to call out to this code that constructs a query and then to *materialize* that data by calling `to_a` on that query.

When we run `bundle exec spec/repositories/user_repository_spec.rb` again, the test will now be passing:

```
1 example, 0 failures
```

Great! Our `contributors_for_project` method on the `UserRepository` works as we want it to. We've accomplished this very differently from how we would normally do it within a Rails application. There, we would (most likely) use a `has_many :through` association to reach through the `tickets` table to reach the `users` table. In what we've just done, we've gone the other way and told ROM to fetch all the users that have tickets in a particular project.

This has an added benefit that the `ProjectRepository` knows nothing about fetching users. That repository should only know about projects. The repository that should know about users is the `UserRepository`, and so that's where our code resides.

We've got this method, but we're not using it anywhere in particular within our Rails application. You can use your imagination for where that method might go. One good place would be in the `show` action of the `ProjectsController`, to display the people who have contributed to a particular project. You might even want to change the method to only show the top 10 contributors to the project.

Have a go at it and see what you can come up with.

Epilogue

So there you have it, a ROM-powered Rails application. Well, at least the beginnings of one. There's still a lot that could be done with this application. For some ideas, be sure to check out the "Homework" chapter, which is after this one.

This has hopefully been a good demonstration about how to organise a Rails application in a better fashion than what's given to us by default. The separation between classes makes this application a little more tedious to setup than a traditional Rails application, but really lends itself to future maintenance. The logic for working with databases is no longer constrained in a single file. They're spread across repositories, relations, and models.

The validations aren't placed in the model either, they live in their own special file now. If we had some complex validation logic, it would only "pollute" that file, and not the model. The model should be kept solely for business logic.

I especially like that when a view receives a `Projects::Project` instance, it is absolutely *unable* to do more database queries after that point. Quite a few times in my Rails experience I've been bitten by views performing N+1 queries, mostly by accident. This sort of thing is actually impossible with ROM – both the accessing the database in the view and N+1 queries – and that will lead to an application with fewer surprises than a common Rails application. It also means that a view behaves as it well-and-truly should: a dumb receptacle to merge together some data and some HTML / JSON. In my view (heh), a view should work with incredibly simple objects and that way then a view becomes very easily testable in complete isolation from the database – if that's what we wished to do.

Then there's my favourite part: the code in `app/transactions/projects/create.rb`. This is so much more of a cleaner approach to "service objects", which I feel are nebulously defined, with no two Ruby developers agreeing on how to properly structure one.

When I originally saw `dry-transaction`, I was blown away with how clean it was. It *just made sense*. Having the logic for a business transaction all nicely encapsulated in the one class *and* having each method easily testable in isolation is a massive win. If anything from the ROM ecosystem was to be adopted as a new standard Rails best practice, I would choose this. It's leaps-and-bounds better than anything else out there. Rails sorely needs a "chosen" pattern for service objects for exactly the same reasons it needed controllers, models and views. It provides order to what is otherwise chaos. But that might just be wishful thinking on my part.

Tidbits

And now for a few more extra tidbits. I couldn't think of where to put them, so I'm going to put them here until they find a better home.

Did you catch this?

So in the book I talked *a lot* about the `Project#contributors` method. But what I didn't mention is that there's a massively simpler way to write that code in a vanilla Rails application. It goes like this:

app/models/project.rb

```
class Project < ApplicationRecord
  has_many :tickets
  has_many :contributors, through: :tickets, source: :user
end
```

Yes, the versatile `has_many :through` association type. This would make our contributors as accessible as this:

```
project = Project.first
project.contributors.distinct
```

This was a willful omission on my part. I do wonder if people picked up on this *before* reading this section or if they just followed along with the book? (You can email me and let me know either way: me@ryanbigg.com)

The point of this book's way of writing the `contributors` method was to have a teeny-tiny illustration of how easy it is to intermix Active Record with business logic. And I think I pulled it off.

As I said (much) earlier in this book, Active Record's database queries shouldn't be so easy to reach for. The database and business logic for your application should be kept apart to ensure a good separation of concerns.

So bonus points for you if you happened to pick up on this earlier.

Working out how to do a `has_many :through` association in ROM is part of the "homework" of this book, so look out for it in the next chapter.

A radical architecture change

When pairing with Seb on his project, we started structuring things the "Rails way", having directories named after the *type* of the objects that they contained. We had a "validators" directory which contained, you guessed it, validators. I'm

sure Seb pointed out that this felt wrong about mid-way through our sessions. He also mentioned Uncle Bob's The Lost Architecture[34] talk wherein Uncle Bob talks about the Entity-Boundary-Interactor pattern and other things. Uncle Bob explains early on in that talk that a Rails application gives you no sense of how a Rails application is structured, other than it's structured *like a Rails application*. It doesn't tell you want functions the application has, specifically. He also talks at great length about how the web should be a dumb delivery mechanism for the application, and I very strongly agree with him on that topic.

So when Seb and I worked on his application, we ended up creating directories like `lib/app_name/category` and `lib/app_name/transaction`, and placing anything related to categories or transactions under there. That way, it gave us a clearer sense of what functionality the application had, rather than "oh this is just another Rails app". If I wanted to know that these days, I could just take a look at the `Gemfile`, or look for another common file, like `config/application.rb`.

Why does a Rails application need to be structured *like a Rails application* at all?

Why does our application's logic need to be tied so closely to the request / response pattern? And taking that a step further: why does it need to be tied to whatever a database does? Abstracting those two things away (as I've done in this book) leads to a much cleaner architecture... even though the book does not mention the Entity-Boundary-Interactor pattern at all.

I think that we could've approached this application by putting all the ROM and dry-rb code in the `lib` directory and treating it as if it were an external gem dependency. This is similar to how Seb and I are writing our application. I'd imagine a more fleshed-out example might look like this:

[34] https://www.youtube.com/watch?v=WpkDN78P884

```
lib
└── ticketee
    └── projects
        ├── model.rb
        ├── relation.rb
        ├── repo.rb
        └── transactions
            └── create.rb
```

The benefit of this is that the application is structured in a way that makes sense to humans: there's a `projects` directory, so that indicates that this application does *something* with projects. Underneath that directory, there is a `transactions` directory which contains files that describe actions that we can do with this "projects" thing. All the other files are an implementation detail.

We would then choose to interact with this application like this:

```
action = Ticketee::Projects::Create.new
action.call(project_params)
...
```

Doesn't that just make more sense?

The Phoenix Approach

I did say in the introduction that I had finished talking about Phoenix "for now". Well, here I am again about to talk (or write) about Phoenix.

Phoenix[35] – arguably a more "modern" web framework – separates out applications into two major parts: your business logic and your web logic. Here's a brand new Phoenix application's directory structure:

[35] http://phoenixframework.org/

```
lib
├── store
│   ├── application.ex
│   └── repo.ex
├── store.ex
├── store_web
│   ├── channels
│   ├── controllers
│   ├── endpoint.ex
│   ├── gettext.ex
│   ├── router.ex
│   ├── templates
│   └── views
└── store_web.ex
```

The "web" part of the application is automatically kept separate from the "main" part of the application.

I really think that Rails applications should follow this sort of design principle too. It would mean that there is a clearer boundary from the outset for each Rails application between the main application logic and the request / response cycle.

Phoenix also has the concept of "contexts"[36], splitting the application up into logical chunks and then combining them to build the whole. I think Rails engines are an attempt in this direction, and maybe we could all benefit more from using engines like that in our application. I've tried to copy the idea of Phoenix contexts here by namespacing things like `Projects::ProjectsController` and `Users::User`. If you build a Phoenix app, you will most definitely see where I have gained inspiration from.

Phoenix has certainly "borrowed" a lot of good ideas from Rails. My hope is that in the next few years, Rails would aim to do the same right back.

[36] https://hexdocs.pm/phoenix/contexts.html

The Hanami approach

It'd be remiss of me not to mention Hanami[37], a Ruby web framework that aims to be a "modern web framework for Ruby". This essentially means, to me at least, that it has learned many lessons from over a decade of Rails application development.

I've played around with Hanami a little bit and I really like what I see. The application's structure is along the same lines of the two examples I've given above, with a few changes. I put my thoughts about Hanami into a blog post[38] and I'd suggest you read it if you're interested in learning more about Hanami.

I could've certainly made this book a book about Hanami, but instead I wanted to demonstrate that you *can* use the new shiny tools (that's dry-rb and rom- rb) without rewriting your entire application in a whole new framework.

I've begun writing my own Hanami application[39] using the latest cool, new, shiny tools. It uses dry-rb, rom-rb, and many other cool new technologies. Check out the application's README for more info.

[37] http://hanamirb.org/
[38] https://ryanbigg.com/2018/03/my-thoughts-on-hanami
[39] http://github.com/radar/twist-v2

Homework

I could spend quite a long time writing this book, fleshing out the application much in the same way as the Rails 4 in Action[40] application was done. That book took me a year and a half to write both times I did it. I am older and (supposedly) wiser, and so I aim to not spend so long writing books any more.

Instead, wise reader, I leave you with some homework to do. Take this example application and see if you can expand upon it. Maybe it will give you a better appreciation for ROM by doing some of the work yourself.

Finish the CRUD actions for the ProjectsController

The `ProjectsController` does only two parts of CRUD: Creating and Reading. Can you make it update and delete as well?

Add a TicketsController

This controller should allow users to create, read, update and delete tickets within the context of a project: for example, the request for a `create` request should be `POST /projects/1/tickets`.

Bonus points for finding a way to associate tickets with their projects without explicitly setting `project_id` and setting `project` instead. In this book I haven't shown how to do associations with ROM, but I am sure you will be able to figure it out.

[40] https://manning.com/books/rails-4-in-action

Add Devise to the application

Add Devise to the application, but re-work its `DatabaseAuthenticatable` module to work with ROM.